

Architettura degli Elaboratori – gruppo 1

Introduzione al corso

Orario e svolgimento delle lezioni

Corso

Giorno	Ora	Aula
Lunedì	8:30 – 10:30	T5
Mercoledì	14:00 – 16:00	T5
Venerdì	11:00 – 13:00	T5

Vedere anche <http://informatica.dieti.unina.it/index.php/it/servizi-per-gli-studenti/orario-lezioni/laurea-triennale>

Le lezioni si svolgeranno in modalità mista, in presenza tramite prenotazione e in dad via MTeams:

- Nome team: AdE - gruppo 1
- Codice: 9i11ocv

Ricevimento

Giorno	Ora	Studio
Lunedì	11:00 – 13:00	D.I.E.T.I. (via Claudio 21, Edificio 1, 2p)

In casi opportunamente motivati il ricevimento si può svolgere anche via MS Teams.

In entrambi i casi contattarmi via mail per prendere un appuntamento

AdE: scheda del corso

<https://informatica.dieti.unina.it/index.php/it/laurea-triennale/insegnamenti/41-laurea-triennale/insegnamenti-lt/13-architettura-degli-elaboratori-i>

La scheda indica:

- CFU assegnati al corso
- durata in ore ed eventuale divisione in moduli
- semestre di svolgimento
- obiettivi formativi
- contenuti e prerequisiti
- modalità didattiche
- modalità di esame

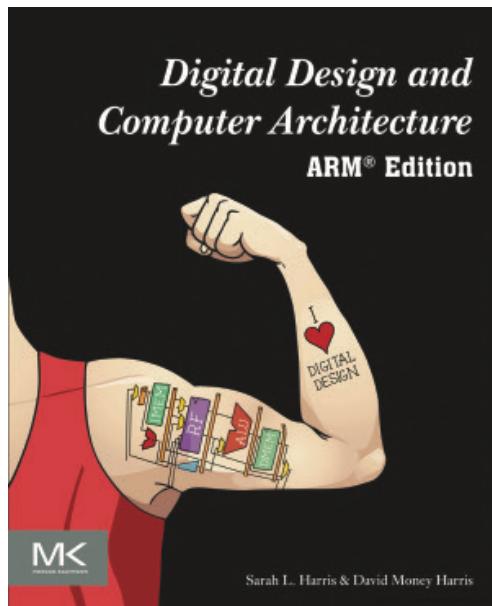
Architettura degli elaboratori	
Titolo insegnamento in inglese: Computer Architecture	Lingua Italiano
Insegnamento: Architettura degli elaboratori	Anno di corso: I
CFU: 9	SSD: INF/01
Ore di lezione: 72	Semestre: 2
Modulo:	Codice:
Obiettivi formativi:	
Conoscere e applicare le principali costituenti digitali del circuito. Saper interpretare e manipolare espressivo dell'algebra di Boole. Saper tradurre un'espressione booleana in circuito combinatorio e viceversa. Saper sintetizzare espressioni booleane. Conoscere le macchine di Moore e Mealy. Conoscere la struttura dei principali circuiti logici implementati e dalla ALU.	
Conoscere l'architettura dei microprocessori basati sul paradigma ARM. Saper realizzare programmi in linguaggio assembly di un processore ARM. Conoscere le principali architetture di memoria, incluse le memorie cache e la memoria virtuale.	
Contenuti:	
Rappresentazioni digitali dei dati. Operazioni aritmetiche e overflow. Algebra di Boole, funzioni booleane, circuiti combinatori e porte logiche. Minimizzazione di funzioni booleane. Multiplexer e decoder. Elementi di timing. Circuiti sequenziali: elementi: latch e flip-flop. Macchine di Mealy e Moore: analisi e sintesi. Circuiti addizionatori e ALU. Architettura ARM: elementi hardware, formato istruzione, architettura interna. Programmazione in assembly ARM. Connessioni con i costrutti del linguaggio C. Introduzione alle memorie cache. Analisi delle prestazioni di sistemi con cache. Introduzione al concetto di memoria virtuale. Traduzione degli indirizzi. Architetture a ciclo singola, a ciclo multiplo e basate su pipeline.	
Prerequisiti: Conoscenze di algebra elementare, Insiemi numerici, Insiematica, Logica elementare	
Modalità didattiche: Lezioni frontali.	
Materiale didattico:	
• D. Harris e S. Harris, Digital Design and Computer Architecture: ARM Edition, Morgan Kaufmann 2015 • Inoltre, trasparenze delle lezioni	
Modalità di esame:	
Esame si articola in prova	Scritta e orale
In caso di prova scritta i quesiti sono	A risposta multipla X A risposta libera X Esercizi numerici
Altro	
Docente (codice A-Q): Scuro Luigi	Docente (codice H-Z): Aloisio Alberto

Obiettivi del corso

- Introdurre le principali strutture logiche e i componenti digitali che consentono l'elaborazione dei dati:
 - Rappresentazione dell'informazione: interi unsigned/signed, reali (float), etc.
 - Algebra di Boole e circuiti combinatori: comparatore, sommatore, multiplexer, etc.
 - Circuiti sequenziali: flip-flop, registri, automi finiti etc.
 - Componenti di base di un elaboratore elettronico: memoria (S/D)RAM, ALU, etc.
- Descrivere la struttura di un sistema di elaborazione con particolare riferimento all'architettura ARM:
 - Microarchitettura: formato delle istruzioni, elementi hardware, single/multiple cycle datapath, etc.
 - Memorie cache e memorie virtuali
 - Linguaggio Assembly: data-processing instructions, conditional-loop statements, function calls

Testo adottato (<https://bit.ly/2Ele1OY>)

Harris&Harris, *Digital Design and Computer Architecture ARM Edition*, Elsevier



Dalla Prefazione:

This book is unique in its treatment in that it presents digital logic design from the perspective of computer architecture, starting at the beginning with 1's and 0's, and leading through the design of a microprocessor.

We believe that building a microprocessor is a special rite of passage for engineering and computer science students. The inner workings of a processor seem almost magical to the uninitiated, yet prove to be straightforward when carefully explained. Digital design in itself is a powerful and exciting subject. Assembly language programming unveils the inner language spoken by the processor. Microarchitecture is the link that brings it all together.

Materiale didattico ausiliario

- Dispense del corso:
 - <https://www.docenti.unina.it/LUIGI.SAURO>
 - Materiale didattico -> Architettura degli elaboratori
- Dispense integrative:
 - <https://www.docenti.unina.it/NICOLA.MAZZOCCA>
 - Materiale didattico -> CALCOLATORI ELETTRONICI I

il materiale didattico ausiliario (lucidi, registrazioni del corso, etc) non deve considerarsi sostitutivo del testo adottato che rimane la fonte principale per poter sostenere al meglio l'esame

Indice e Pianificazione del Corso

Indice	
1 From Zero to One	
2 Combinational Logic Design	
3 Sequential Logic Design	
4 Hardware Description Languages	
5 Digital Building Blocks	
6 Architecture	
7 Microarchitecture	
3 8 Memory Systems	487
55 9 IO Systems	531
109 Appendix A Digital System Implementation	533
173 Appendix B ARM Instructions	535
239 Appendix C C Programming	542
295 Index	543
385 Copyright	

5 CFU

3 CFU

- 9 CFU, suddivisi in:
 - 5 CFU: logica combinatoria e sequenziale + elementi ed architetture circuitali di base
 - 3 CFU: architettura dei processori ARM, memorie cache e memorie virtuali
 - 1 CFU: attività di laboratorio (ARM ↔ C)

Pagina docente

- <https://www.docenti.unina.it/luigi.sauro>
- Nella sezione *Avvisi* della pagina docenti troverete durante il corso diverse tipologie di informazioni:
 - Possibili assenze
 - Appelli
 - Esito esami
 - Altre comunicazioni
- E' bene controllare spesso

Vi sono anche avvisi di altri corsi da me svolti (*Intelligent Web*)

Appelli d'esame

- Si applica il Regolamento della Scuola Politecnica e delle Scienze di Base
- 7 appelli durante l'anno accademico che cadono *orientativamente* nei mesi di giugno, luglio, settembre, ottobre, gennaio, febbraio e marzo.
- Fra due appelli devono intercorrere almeno 15 giorni

5.1 Periodi didattici e periodi di esami.

Le Strutture Didattiche individuano, coerentemente con l'organizzazione didattica semestrale dei Corsi di Studio, i periodi didattici (I e II periodo didattico) ed i periodi di esami (I periodo di esami: di norma tra la fine del primo periodo didattico e l'inizio del secondo; II periodo di esami: di norma tra la fine del secondo periodo didattico e l'inizio del periodo di vacanza accademica estiva; III periodo di esami: di norma tra la fine del periodo di vacanza accademica estiva ed il 30 settembre).

Fatto salvo quanto stabilito al successivo punto 5.3, gli studenti iscritti in corso agli anni diversi dall'ultimo non possono sostenere esami nel corso dei periodi didattici. Gli studenti iscritti all'ultimo anno del percorso normale di studi possono sostenere esami in debito a partire dalla conclusione dei corsi del I periodo didattico, anche al di fuori dei "periodi di esami" sopra indicati, seguendo la programmazione delle sedute di esame stabilita dalle Strutture Didattiche di concerto con i docenti.

Gli studenti iscritti fuori corso possono sostenere esami durante tutto l'anno, secondo la programmazione delle sedute di esame stabilita dalle Strutture Didattiche di concerto con i docenti.

5.3 Numero di appelli di esame e loro distribuzione.

Per tutti gli insegnamenti curriculari che costituiscono il prospetto della Didattica Programmata del Corso di Studi e per gli studenti iscritti in corso è previsto un **numero minimo di appelli**, tra i quali devono intercorrere almeno 15 giorni solari, così articolato:

- due appelli nel primo periodo di esami;
- due appelli nel secondo periodo di esami;
- un appello nel terzo periodo di esami;
- un appello straordinario per il recupero degli esami in debito nel mese di ottobre;
- un appello straordinario per il recupero degli esami in debito nel mese di marzo.

5.4 Ripetizione di un esame.

Nell'ambito della disciplina generale stabilita dal Regolamento Didattico di Ateneo*, si dispone che gli studenti possano sostenere un esame non superato senza alcuna limitazione, purché tra l'appello dell'esame sostenuto e il successivo siano trascorsi almeno 15 giorni solari.

5.6 Raccomandazioni e linee di indirizzo

Per i corsi tenuti nel I periodo didattico e nell'ambito del coordinamento trasversale potranno essere previsti "pre-appelli" immediatamente successivi alla fine del corso, nel quadro di una bilanciata collocazione complessiva degli appelli nel I periodo di esame.

Per gli esami che prevedono più prove (ad es. scritto e orale) si raccomanda fortemente di contenere l'intervallo temporale intercorrente tra le stesse al minimo compatibile con le normali operazioni di correzione degli elaborati. Si raccomanda inoltre di curare la tempestiva trasmissione dei verbali alle Segreterie Studenti competenti.

Modalità di esame

- L'esame consiste in una prova scritta e un colloquio orale
- Accedono alla prova orale gli studenti che hanno raggiunto o superato nella prova scritta la soglia di ammissione di 18/30
- Il superamento della prova scritta in un appello permette l'accesso all'orale solo ed esclusivamente nello stesso appello
- Studiare **durante il corso** vi permette di assimilare per tempo e in maniera graduale le nozioni necessarie per sostenere al meglio l'esame
- I programmi dei tre gruppi sono allineati
- Per ragioni amministrative non è possibile svolgere l'esame in un gruppo diverso da quello di appartenenza
 - Il gruppo 1 corrisponde ai cognomi: A-DE

Prova scritta

- La prova scritta è suddivisa in una parte relativa agli argomenti trattati nei primi 5 crediti ed una parte relativa ai rimanenti 3 crediti.
- Il punteggio finale è dato dalla somma dei punteggi di ogni singolo esercizio
- Un esercizio è svolto correttamente *se e solo se* il suo risultato è corretto

Contatti

- Email: luigi.sauro@unina.it
 - La mail è utilizzata per informazioni di carattere organizzativo, non per dubbi riguardanti argomenti del corso
 - Per dubbi riguardo gli argomenti del corso c'è il ricevimento

La scheda di valutazione del Corso

- Sarete chiamati a dare una valutazione sul corso
 - Per noi è importante ricevere feedback su come migliorare l'organizzazione del corso



Pesi e Medie

Valori medi compresi tra -1.5 e +1.5

Quesito	Risultati docente-insegnamento (frequenza: tra il 51% e il 70%)					Dati di confronto II periodo: quartili Ateneo 2017/18					
	Risposte 2017/18					Media 2017/18	Media 2016/17	(mediana)	1°	2°	3°
	decisamente no	più no che si	più si che no	decisamente si							
	n1	n2	n3	n4							

$$m_1 = n_1 \times (-1.5)$$

$$m_2 = n_2 \times (-0.5)$$

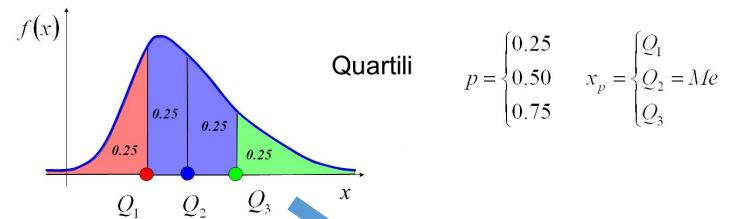
$$m_3 = n_3 \times (+0.5)$$

$$m_4 = n_4 \times (+1.5)$$

Legenda:

- Questionario: la rilevazione avviene tramite procedura online di Ateneo (dall'a.a. 2015/16). Il questionario è compilabile nella versione in lingua inglese per i corsi di studio internazionali. Le schede dell'a.a. 2017/18 sono riferite agli studenti che hanno risposto di aver seguito l'insegnamento nell'a.a. corrente.
- Media 2017/18: Decisamente no = -1,5; Più no che sì = -0,5; Più sì che no = +0,5; Decisamente sì = +1,5. La media aritmetica è calcolata solo se ci sono almeno 5 moduli compilati.
- Dati di confronto: per ciascun quesito sono riportati i valori che dividono in quartili la distribuzione delle medie di tutti gli insegnamenti valutati nel periodo considerato in Ateneo. I valori riferiti al precedente a.a. 2016/17 non sono stati riportati in quanto gli studenti i rispondenti considerati per l'a.a. 2017/18 non sono confrontabili.
- Suggerimenti: la risposta prevede la scelta multipla, è possibile indicare più di un suggerimento.

Quartili



Quesito	Risultati docente-insegnamento (frequenza: tra il 51% e il 70%)				Dati di confronto II periodo: quartili Ateneo 2017/18			
	Risposte 2017/18				Media 2017/18	Media 2016/17	1°	2° (mediana)
	decisamente no	più no che si	più si che no	decisamente si				

- I quartili sono i valori in corrispondenza dei quali una distribuzione viene suddivisa in quattro parti uguali:
 - Nel 1° quartile Q1 si colloca il peggior 25% dei casi
 - Nel 2° quartile Q2 si colloca la fascia 25% -50% dei casi
 - Nel 3° quartile Q3 si colloca la fascia 50 % - 75% dei casi
 - Nel 4° quartile Q4 si colloca il miglior 25% dei casi

Aule, Laboratori, Biblioteche - risposte

Quesito	Risultati docente-insegnamento (frequenza: tra il 51% e il 70%)					Dati di confronto II periodo: quartili Ateneo 2017/18			
	Risposte 2017/18				Media 2017/18	Media 2016/17	1°	(mediana)	3°
	decisamente no	più no che si	più si che no	decisamente si					
<i>q.1 Le aule dove si svolgono le lezioni sono adeguate?</i>		13	30	9	0,42		0,10	0,39	0,67
<i>q.2 I laboratori e le attrezzature per le attività didattiche integrative sono adeguati? (se non previste rispondere 'Non Previsto')</i>	1	7	26	4	0,37		0,00	0,33	0,63
<i>q.3 I servizi bibliotecari (prestito, consultazione, orari di apertura) di cui ha usufruito sono adegua</i> ti?		3	8	12	0,89		0,25	0,50	0,72

Insegnamento - risposte

Quesito	Risultati docente-insegnamento (frequenza: tra il 51% e il 70%)					Dati di confronto II periodo: quartili Ateneo 2017/18			
	Risposte 2017/18				Media 2017/18	Media 2016/17	1°	(mediana)	3°
	decisamente no	più no che si	più si che no	decisamente si					
<i>q.4 Sono state fornite spiegazioni chiare su programma e obiettivi dell'insegnamento?</i>	1	2	8	41	1,21		0,58	0,83	1,06
<i>q.5 L'insegnamento è stato svolto in maniera coerente con quanto dichiarato sul sito Web del corso di studio?</i>		3	10	39	1,19		0,68	0,90	1,10
<i>q.6 Le attività didattiche integrative (esercitazioni, tutorati, laboratori, etc.) sono utili all'apprendimento della materia?</i>	2	4	22	24	0,81		0,44	0,68	0,90

Esami e Organizzazione - risposte

<i>q.7 Le modalità di esame sono state definite in modo chiaro?</i>	3	3	12	34	0,98	0,59	0,83	1,06
<i>q.8 Il carico di studio dell'insegnamento è proporzionato ai crediti assegnati?</i>		4	23	25	0,90	0,30	0,62	0,86
<i>q.9 L'insieme degli insegnamenti previsti nel periodo di riferimento (bimestre, trimestre, semestre, etc.) è accettabile?</i>	4	14	16	18	0,42	0,13	0,36	0,58
<i>q.10 L'organizzazione complessiva (orario, esami, intermedi e finali) degli insegnamenti previsti nel periodo di riferimento (bimestre, trimestre, semestre, etc.) è accettabile?</i>	3	4	25	20	0,69	0,10	0,32	0,50
<i>q.11 Le conoscenze preliminari possedute sono risultate sufficienti per la comprensione degli argomenti previsti nel programma d'esame?</i>	2	7	21	22	0,71	0,36	0,59	0,80
<i>q.12 E' complessivamente soddisfatto di questo insegnamento?</i>	1	5	12	34	1,02	0,50	0,79	1,00

Efficacia Questionario, Interesse - risposte

Quesito	Risultati docente-insegnamento (frequenza: tra il 51% e il 70%)					Dati di confronto II periodo: quartili Ateneo 2017/18			
	Risposte 2017/18				Media 2017/18	Media 2016/17	1°	(mediana)	3°
	decisamente no	più no che si	più si che no	decisamente si					
<i>q.13 Sono stati presentati in modo esauriente il processo e i fini della valutazione?</i>		4	19	29	0,98		0,50	0,69	0,90
<i>q.14 Lei ha percepito l'efficacia del questionario ai fini del miglioramento della didattica?</i>	1	14	20	17	0,52		0,13	0,33	0,50
<i>q.16 E' interessato/a agli argomenti trattati nell'insegnamento?</i>	1	3	20	28	0,94		0,70	0,91	1,13

Docente -1/2 - risposte

Quesito	Risultati docente-insegnamento (frequenza: tra il 51% e il 70%)					Dati di confronto II periodo: quartili Ateneo 2017/18			
	Risposte 2017/18				Media 2017/18	Media 2016/17	1°	(mediana)	3°
	decisamente no	più no che si	più si che no	decisamente si					
<i>q.17 Il docente espone gli argomenti in modo chiaro?</i>	1	7	15	29	0,88		0,61	0,91	1,16
<i>q.18 Il docente stimola/motiva l'interesse verso la disciplina?</i>	1	8	16	27	0,83		0,50	0,83	1,10
<i>q.19 Gli orari di svolgimento di lezioni, esercitazioni e altre eventuali attività didattiche sono rispettati?</i>			10	42	1,31		0,80	1,00	1,18
<i>q.20 Il docente è reperibile per chiarimenti e spiegazioni?</i>	1	4	22	25	0,87		0,75	0,98	1,17

Docente -2/2 - risposte

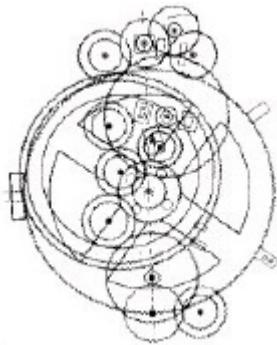
Quesito	Risultati docente-insegnamento (frequenza: tra il 51% e il 70%)					Dati di confronto II periodo: quartili Ateneo 2017/18			
	Risposte 2017/18				Media 2017/18	Media 2016/17	1°	(mediana)	3°
	decisamente no	più no che si	più si che no	decisamente si					
<i>q.21 Il materiale didattico (indicato e disponibile) è adeguato per lo studio della materia?</i>	2	7	23	20	0,67		0,50	0,75	0,97
<i>q.22 Il docente è attento ai problemi che gli vengono segnalati?</i>	1	3	18	30	0,98		0,70	0,95	1,15
<i>q.23 Il docente titolare dell'insegnamento svolge o comunque presiede la maggior parte delle lezioni?</i>			4	48	1,42		0,94	1,20	1,36

Storia degli elaboratori

- Alcuni cenni
 - se volete qualche altra informazione <http://www.computerhistory.org/timeline/>
- Chiaramente è fortemente legata alla storia dell'Informatica, ma non coincide con essa!
 - Così come la storia degli strumenti musicali non coincide con la storia della musica

Strumenti di calcolo

Fin dall'antichità l'uomo ha sviluppato strumenti che l'aiutassero nello svolgimento di calcoli matematici. Un tipico esempio è l'**abaco** (3000 a.C.), probabilmente originario della Cina, aiuta ad eseguire le quattro operazioni (somma, sottrazione, prodotto e divisione intera).



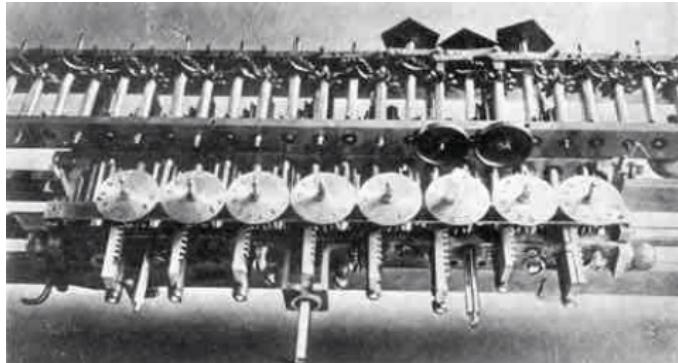
La macchina di Antikythera (150 - 100 a.C.) è un sofisticato planetario ritrovato in una isola greca. Esso è mosso da ruote dentate, che serviva per calcolare il sorgere del sole, le fasi lunari, i movimenti dei 5 pianeti allora conosciuti, gli equinozi, i mesi e i giorni della settimana.

<https://www.youtube.com/watch?v=8KYLoDW7Gic>

Pascalina

L'abaco è un strumento che aiuta ad eseguire calcoli, ma di per sé non li svolge in maniera autonoma.

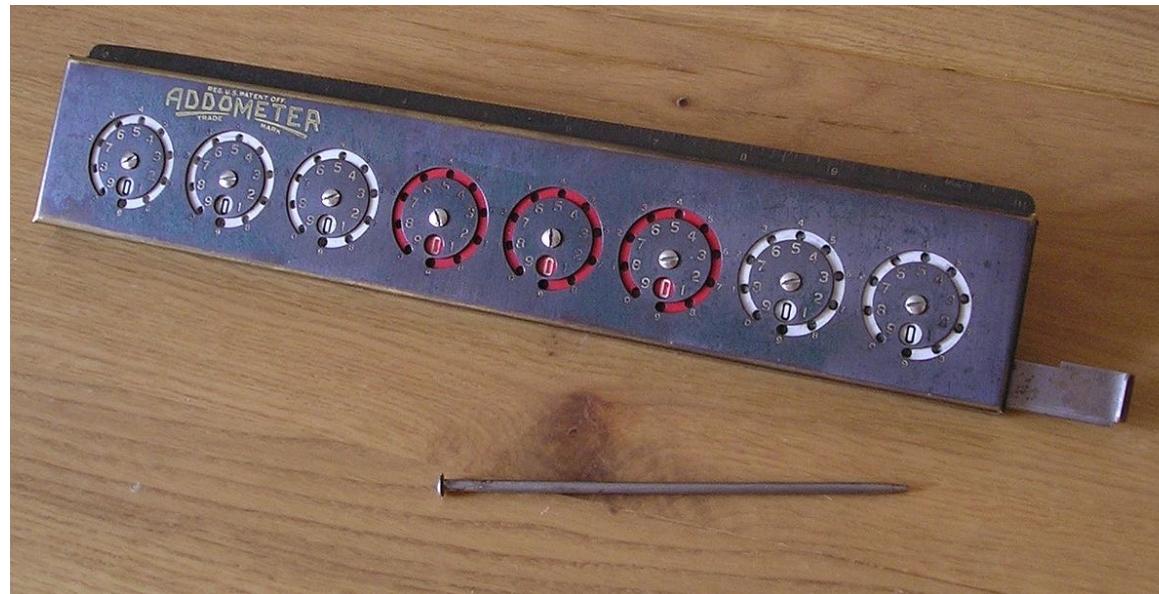
Uno dei primi esempi di macchina aritmetica fu la **pascalina** (1640). Progettata dal filosofo Blaise Pascal, era in grado mediante un sistema di rotori di operare i riporti di una somma. Quindi, a differenza dell'abaco, la pascalina poteva eseguire autonomamente la somma di due numeri di massimo dodici cifre.



La **Stepped Reckoner** (1673) è una macchina per addizioni, sottrazioni, moltiplicazioni, divisioni e radice quadrata progettata dal filosofo Gottfried Wilhelm von Leibniz.

L'addometer

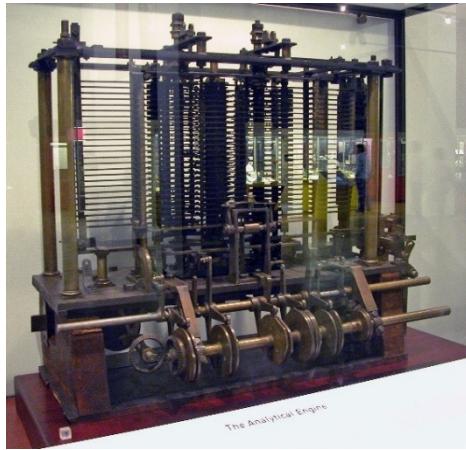
Intorno agli anni sessanta si vendeva ancora un addizionatore meccanico del tutto analogo alla pascalina.



6\$ su eBay

La macchina analitica

La **Analithycal Engine** (1824) è una macchina *programmabile* progettata dal matematico inglese Charles Babbage. Questo viene considerato il primo tentativo di realizzare un calcolatore moderno.



Allo sviluppo della macchina analitica contribuì anche la contessa Ada Lovelace (1815-1852). In una delle sue note è descritto un algoritmo per calcolare i numeri di Bernulli. Queste note sono considerate il primo programma della storia.

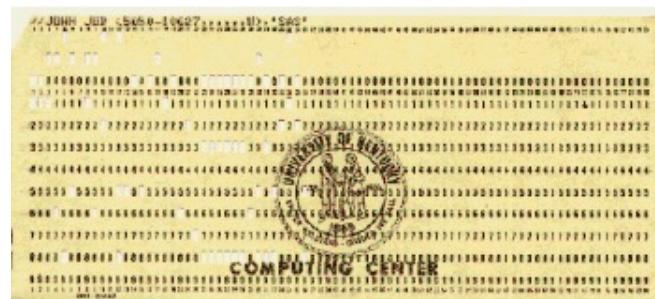
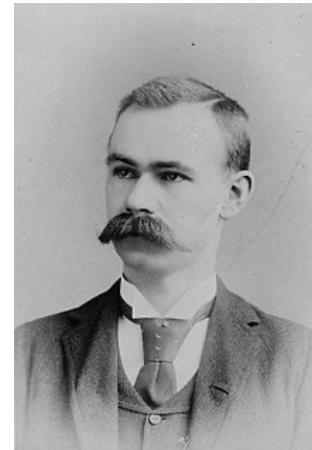
La macchina tabulatrice

Il censimento degli Stati Uniti nel 1880 durò nel suo complesso circa sette anni. Di fatto i dati, una volta disponibili era già obsoleti.

Per ovviare a questo problema, Herman Hollerith realizzò una macchina tabulatrice per il censimento del 1890. Grazie ad essa, il censimento durò solo qualche mese.

Ispirandosi ai biglietti ferroviari dell'epoca, questa macchina utilizzava schede perforate per immagazzinare le informazioni.

Hollerith fondò la Tabulating Machine Company che nel 1924 diventò la **IBM**.

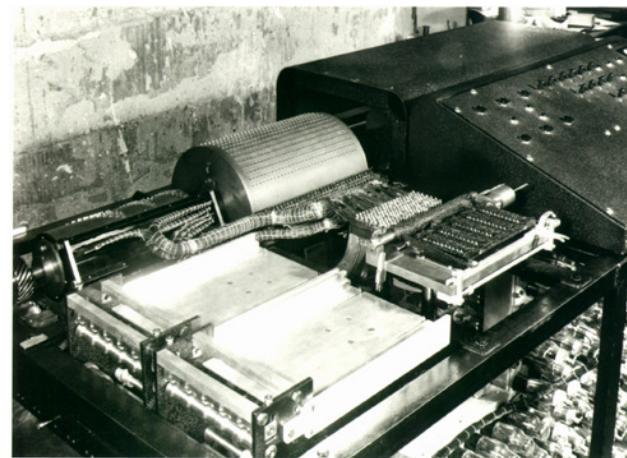


ABC

Il **Atanasoff Berry Computer** (1939) fu il primo computer elettronico digitale.

Sviluppato da John Atanasoff usava valvole termoioniche, antesignane dei futuri transistor, che si vedono ancora in qualche impianto stereo ad alta fedeltà.

L'informazione era rappresentata in codice **binario**.

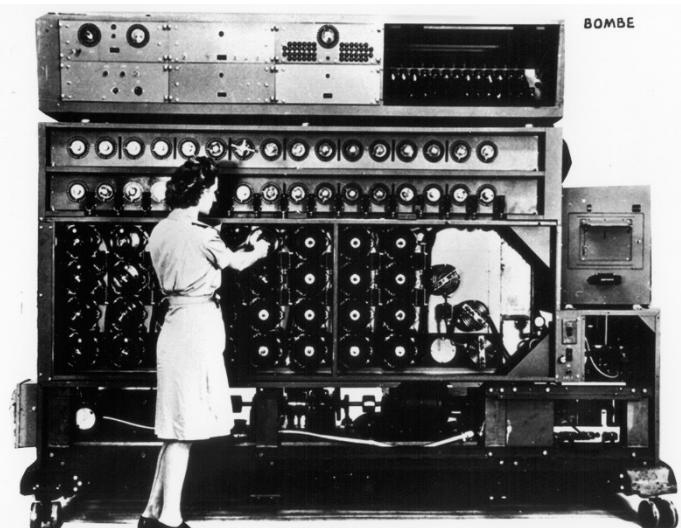


La «Bomba»

Nel 1941 viene completato in Inghilterra il primo modello della “Bomba”, una macchina elettromeccanica il cui scopo era quello di decrittare il codice nazista ENIGMA.

L'Informatica viene impiegata per risolvere un problema “difficile” e di grande importanza strategica.

A capo del gruppo di matematici ed enigmisti che realizzò la Bomba c'era uno dei padri fondatori della scienza informatica, [Alan Turing](#).

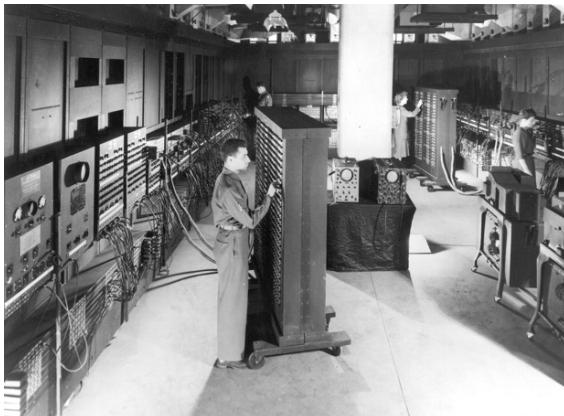


ENIAC

Nel 1946 Dr. John W. Mauchly e J. Presper Ecjert realizzarono presso l'Università della Pensilvania uno dei primi calcolatori moderni, l'**Electronic Numerical Integrator and Computer** (ENIAC).

L'ENIAC era un calcolatore *Turing-completo* che fu usato soprattutto per calcolare traiettorie balistiche per l'esercito statunitense.

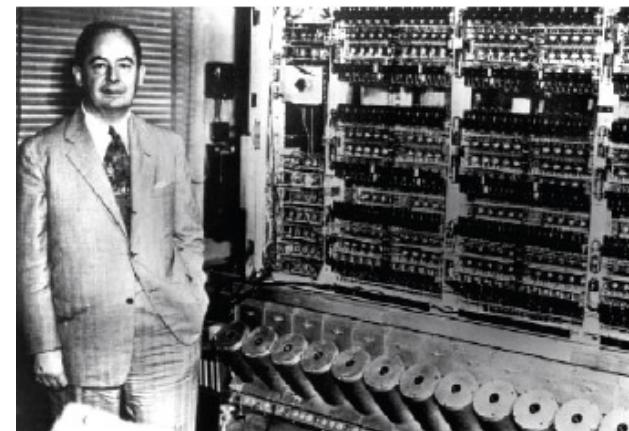
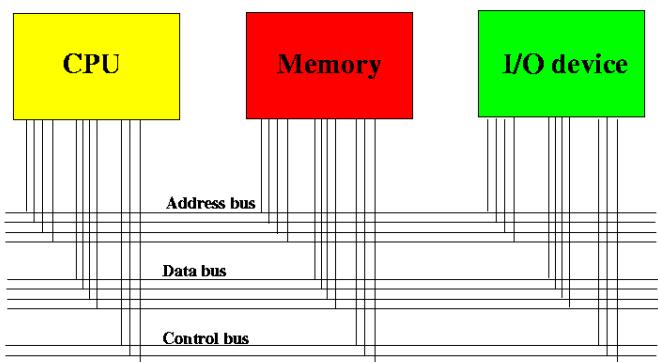
Pesava circa 27 tonnellate, occupava un area di 127 m² e conteneva 17468 valvole termoioniche.



EDVAC

L'Electronic Discrete Variable Automatic Computer (1951) fu realizzato dagli stessi progettisti dell'ENIAC.

Diversamente dall'ENIAC, in cui i programmi erano cablati, l'EDVAC è uno dei primi esempi di architettura di von Neumann.



L'intuizione del grande matematico ungherese è quella di rappresentare i programmi in codice binario, esattamente come i dati. In questo modo sia i dati che i programmi sono caricati in memoria.

L'architettura di von Neumann è il modello su cui si basano ancora oggi i nostri computer.

IBM 1401

Nel 1961 la IBM realizza il mainframe 1401. Le valvole termoioniche sono sostituite con i più stabili e piccoli transistors.

Il 1401 può essere considerato il primo successo commerciale, pochi anni dopo circa la metà dei computer in circolazione erano un 1401.

L'IBM 1401 costava 2.500\$ (circa 25.000\$ ai giorni nostri), la memoria era di pochi Kbyte.



Programma 101

Nella fiera di New York del 1965 la Olivetti presenta il primo personal computer della storia, la Programma 101.

Se si pensa che i computer di allora erano grandi come armadi, Programma 101 era una macchina strabiliante per dimensioni, prestazioni e semplicità d'uso.

44 mila esemplari venduti di cui il 90% negli Stati Uniti. Fu usata anche dalla NASA.

<https://www.youtube.com/watch?v=2RjIRKletP8>



ATARI 2600

Nel 1977 viene commercializzata la console ATARI 2600. Nasce l'industria videoludica.

CPU: 8-bit 1.19 MHz MOS Technology 6507

RAM: 128 bytes

ROM: 4K bytes



IBM 5150

Il mercato dei personal computer crebbe esponenzialmente negli anni ottanta grazie all'imponente campagna pubblicitaria della IBM.

Il primo PC della IBM era il 5150. Montava un processore Intel 8088 a 4.77 Mhz e usava il sistema operativo DOS di Microsoft.

IBM 5150 costituì anche uno dei primi “ecosistemi” di software, periferiche e altri accessori sviluppati per questa piattaforma.



Commodore64

Negli anni ottanta i computer iniziarono a popolare milioni di case grazie allo sviluppo di computer economicamente accessibili. Il maggior successo commerciale fu il Commodore 64.

Coi suoi 64 Kbyte di memoria RAM, il C64 era un ottimo computer per uso domestico (per certi versi migliore di altri computer molto più costosi). Oggi un portatile di fascia media ha una memoria circa 100.000 volte più grande.



Macintosh

Nel 1984 viene commercializzato il primo Macintosh. La macchina era fornito di 128KB di memoria Ram ma non possedeva un hard disk interno. Inoltre aveva problemi di surriscaldamento.

La vera innovazione era in realtà il sistema operativo, il primo che supportava in maniera nativa un interfaccia grafica.

Era il 1998 quando vide la luce il primo modello di l'IMac. Questo all-in-one computer segnerà la rinascita della Apple e la sua scalata economica fino a diventare il colosso che conosciamo oggi.

Aveva un processore G3 a 233 Mhz, 32MB di RAM, e una memoria di massa di 4 GB.



iPhone

Nel 2007 Apple presenta iPhone, il computer è ormai così miniaturizzato da entrare in tasca.

Subito molte altre compagnie (Samsung, Google) si lanciano nel promettente settore degli smartphone.

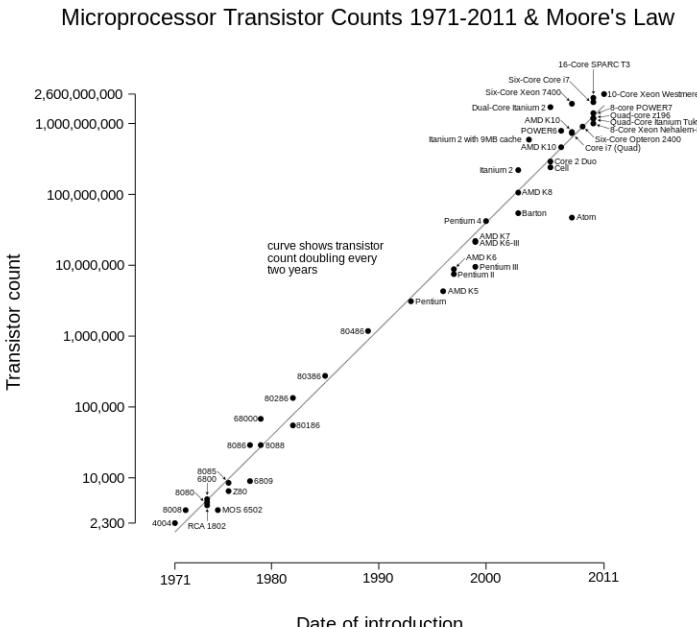
Grazie ai social networks e un vasto ecosistema di apps gli smartphone sono di fatto la tipologia di computer che utilizziamo di più durante la giornata.

Le caratteristiche peculiari degli smarthpone hanno favorito lo sviluppo dei processori ARM.



Sviluppo degli elaboratori

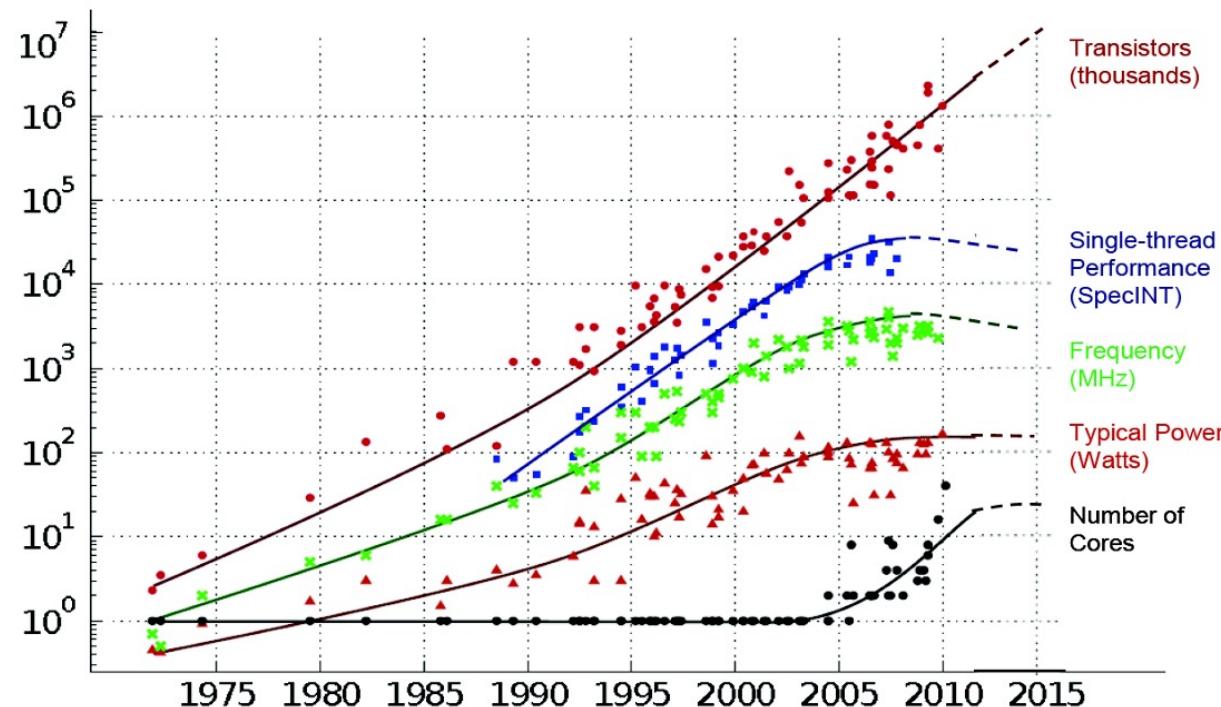
- Memoria RAM di un portatile di fascia media:
 - 256 MB (2000) 1GB (2005) 4GB (2010) 16GB (2015) 16GB (2020)
 - Fino ad oggi, le caratteristiche dei componenti hardware hanno avuto una crescita esponenziale.
 - Legge di Moore: il numero di transistor di un processore raddoppia ogni 18 mesi



Sviluppo degli elaboratori

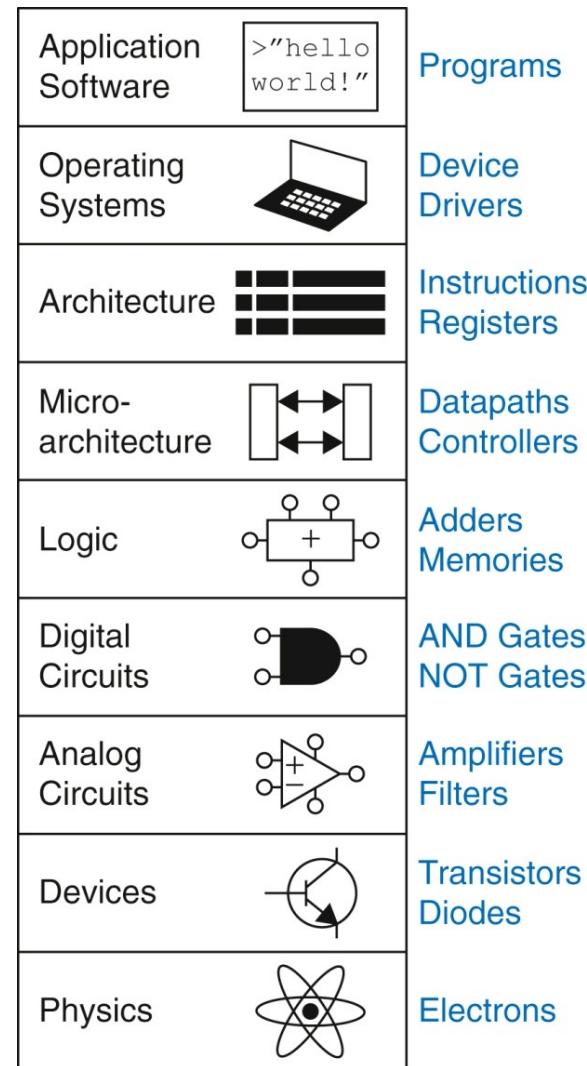
Tuttavia alcune caratteristiche hardware sembra che abbiano rallentato la loro crescita esponenziale

35 YEARS OF MICROPROCESSOR TREND DATA



Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten
Dotted line extrapolations by C. Moore

Livelli di astrazione





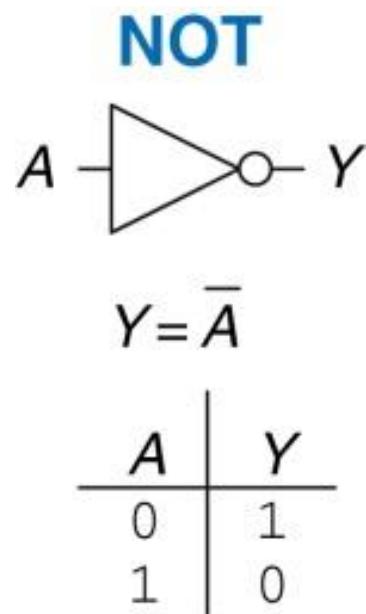
Algebra di Boole e reti combinatorie

Porte logiche

- Come accennato, un calcolatore può essere visto come un complesso sistema digitale che manipola e memorizza informazioni rappresentate in codice *binario*.
- I componenti digitali che costituiscono i mattoni fondamentali di un calcolatore sono le *porte logiche*
- Le porte logiche realizzano delle semplici operazioni che prendono uno o più input e producono un output
- Gli input sono indicati generalmente con le prime lettere dell'alfabeto A,B,C,D,... mentre gli output con le ultime X,Y,...
- Poiché gli input e gli output possono assumere in generale sia il valore 0 che 1 allora essi costituiscono delle variabili (dette variabili booleane)

Porta NOT

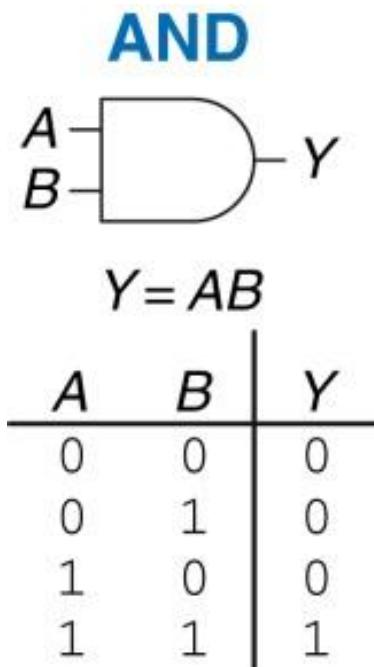
- La porta NOT restituisce in output il *complemento* dell'input:



- Y è uguale a 1 se e solo se A **non** è uguale a 1: $Y=1-A$
- Altri simboli per NOT sono $\neg A$ o $\sim A$ (usati soprattutto dai logici)

Porta AND

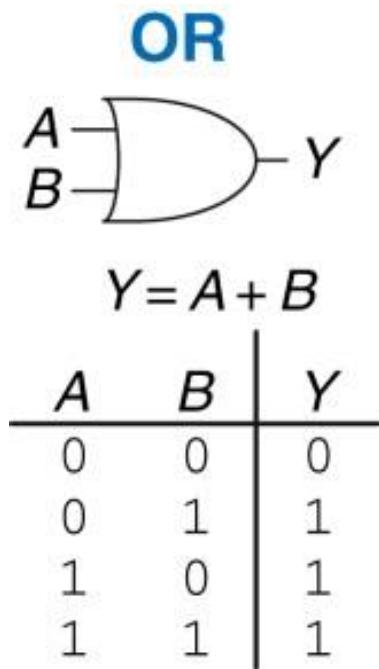
- La porta AND restituisce in output la *congiunzione* degli input:



- Y è uguale a 1 se e solo se A **e** B sono entrambi uguali a 1
- L'operatore AND è anche rappresentato con $A \cdot B$ o $A \wedge B$

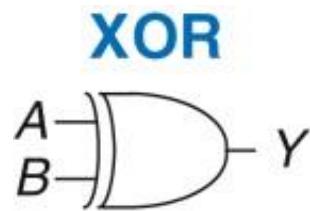
Porta OR

- La porta OR restituisce in output la *disgiunzione* degli input:

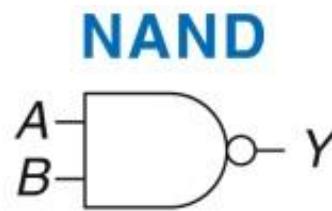


- Y è uguale a 1 se e solo se A **o** B è uguale a 1
- L'operatore OR è anche rappresentato con $A+B$ o $A \vee B$

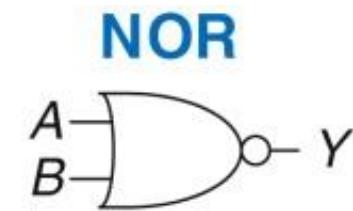
Altre porte logiche



$$Y = A \oplus B$$



$$Y = \overline{AB}$$



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

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

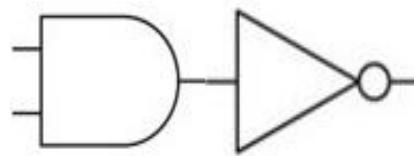
A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

- XOR: $Y = 1$ se e solo se A oppure B è uguale a 1
- NAND: Y è uguale a 1 se e solo se A o B non sono uguali a 1
- NOR: Y è uguale a 1 se e solo se A e B non sono uguali a 1

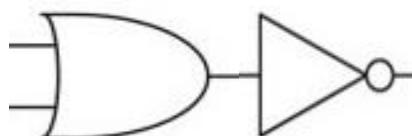
Altre porte logiche

- La porta NAND è un elemento circuitale che viene prodotto direttamente, tuttavia può essere ottenuto complementando una porta AND, ovvero mettendo *in serie* una porta AND e una NOT:



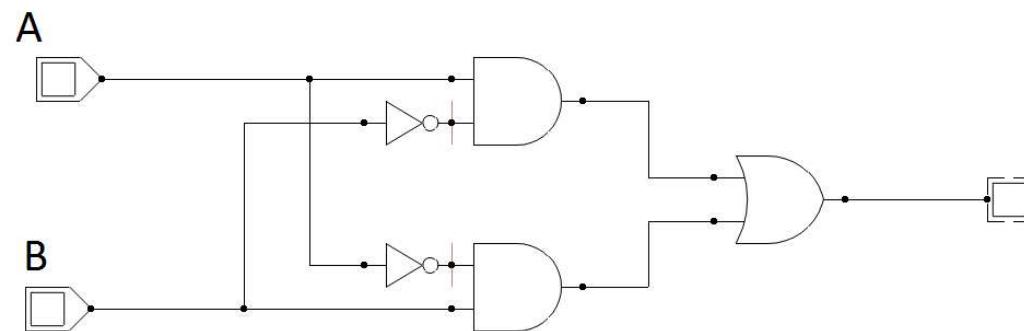
A	B	$A \cdot B$	$\overline{A \cdot B}$
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	0

- Analogamente la porta NOR si può ottenere mettendo *in serie* una porta OR e una NOT:



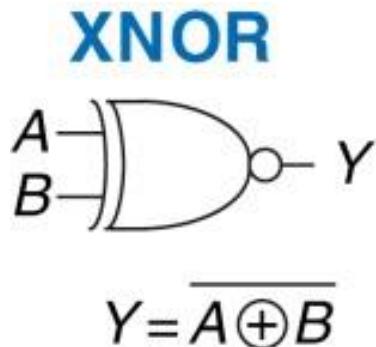
Altre porte logiche

- Anche la porta XOR può essere ottenuta mediante porte AND, OR e NOT.
 - XOR: $Y = 1$ sse $A \neq B$
 - $A \neq B$ sse $(A=1 \text{ e } B=0) \text{ o } \underbrace{(A=0 \text{ e } B=1)}$
 - $Y = (A \cdot \bar{B}) + (\bar{A} \cdot B)$



XNOR

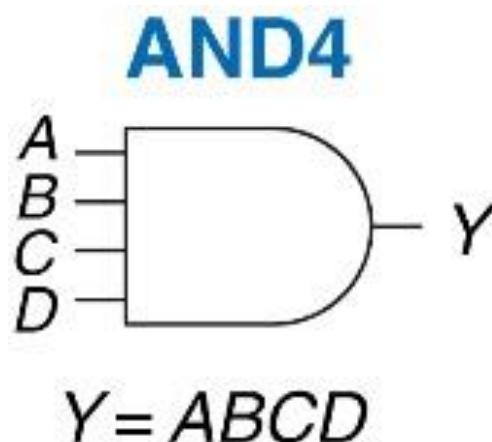
- Problema: quale è la tabella di verità della porta XNOR?



A	B	$A \oplus B$	$\overline{A \oplus B}$
0	0	0	1
0	1	1	0
1	0	1	0
1	1	0	1

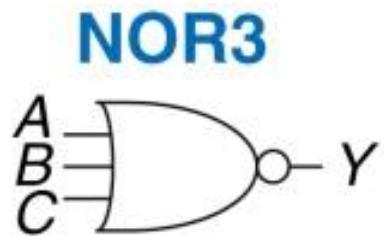
Porte logiche con più linee di input

- Le porte logiche AND, OR, NAND,... possono avere anche più di 2 linee di ingresso.



A	B	C	D	Y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

Porte logiche con più linee di input



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

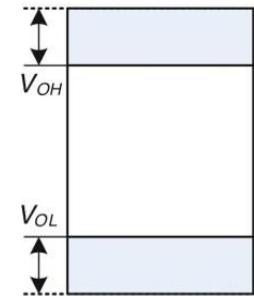
<i>A</i>	<i>B</i>	<i>C</i>	<i>Y</i>
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

Cosa c'è sotto?

- Finora abbiamo descritto le porte logiche in termini astratti in cui gli ingressi e le uscite assumono i valori binari 0 o 1.
- Chiaramente, per realizzare queste porte in concreto, questi valori devono corrispondere a una certa grandezza fisica.
- Nei calcolatori elettronici (questo laptop per intenderci) la grandezza fisica che *reifica* i valori logici 0 e 1 è il potenziale elettrico.
- Daremo per scontato che abbiate le nozioni minime di base su cosa sia il potenziale elettrico (o che almeno non mettiate le dita in una presa da 220V).
- Il valore logico 0 è rappresentato dal valore di potenziale di 0V (GRD)
- Il valore logico 1 è rappresentato da un valore di potenziale V_{DD} fornito dal generatore.
 - Fino agli anni ottanta $V_{DD} = 5V$, l'avvento di portatili, tablet e smartphone ha reso necessario operare con valori di potenziale più bassi $V_{DD} \leq 1,5V$

Cosa c'è sotto?

- Notate che il potenziale elettrico è una grandezza fisica *continua* con cui rappresentiamo dei valori discreti (0 e 1) mediante le grandezze nominali GRD e V_{DD} .
- Come ogni sistema fisico reale, un circuito digitale è soggetto a del *rumore* che può alterare i valori nominali su cui operiamo. Per questo i componenti digitali hanno delle soglie di tolleranza $GRD - V_{OL}$ e $V_{OH} - V_{DD}$.



- Per un corretto funzionamento di un circuito, il rumore non deve eccedere le soglie di tolleranza di tutti i suoi componenti.

Funzioni booleane

- Le porte logiche esaminate finora costituiscono delle specifiche *funzioni booleane*

$$f: \{0,1\}^N \rightarrow \{0,1\}$$

- Domanda: quante funzioni booleane di N variabili esistono?

$A_N A_{N-1} \dots A_1 A_0$	y
0 0 ... 0 0	y_0
0 0 ... 0 1	y_1
0 0 ... 1 0	y_2
0 0 ... 1 1	y_3
.	.
.	.
1 1 ... 1 1	y_{2^N}

2^N righe

Una generica f assegna «liberamente» valori 0 o 1 alle y_i .
Quindi il numero di funzioni booleane di N variabili è pari al numero di parole binarie di lunghezza 2^N , ovvero:

$$2^{2^N}$$

Funzioni booleane di 2 variabili

Se le variabili sono 2 allora ottengo $2^4 = 16$ possibili funzioni booleane

A B	f ₀	f ₁	f ₂	f ₃	f ₄	f ₅	f ₆	f ₇	f ₈	f ₉	f ₁₀	f ₁₁	f ₁₂	f ₁₃	f ₁₄	f ₁₅
0 0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0 1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1 0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1 1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

AND XOR OR NOR XNOR NAND

Funzioni booleane di 2 variabili

Se le variabili sono 2 allora ottengo $2^4 = 16$ possibili funzioni booleane

A B	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}
0 0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
0 1	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	1
1 0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1 1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Funzione costante
 $Y=0$

Funzione costante
 $Y=1$

Funzioni booleane di 2 variabili

Se le variabili sono 2 allora ottengo $2^4 = 16$ possibili funzioni booleane

A	B	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}
0 0		0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0 1		0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1 0		0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1 1		0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

$\gamma = A$ $\gamma = B$ $\gamma = \bar{A}$ $\gamma = \bar{B}$

Funzioni booleane di 2 variabili

Se le variabili sono 2 allora ottengo $2^4 = 16$ possibili funzioni booleane

A B	f ₀	f ₁	f ₂	f ₃	f ₄	f ₅	f ₆	f ₇	f ₈	f ₉	f ₁₀	f ₁₁	f ₁₂	f ₁₃	f ₁₄	f ₁₅
0 0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0 1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1 0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1 1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

$\gamma = A\bar{B}$ $\gamma = \bar{A}B$ $\gamma = \overline{AB} = A + \bar{B}$ B ≤ A

$\gamma = \overline{A\bar{B}} = \bar{A} + B$ A ≤ B

Funzioni booleane di 2 variabili

Se le variabili sono **2** allora ottengo $2^4 = 16$ possibili funzioni booleane

A B	f ₀	f ₁	f ₂	f ₃	f ₄	f ₅	f ₆	f ₇	f ₈	f ₉	f ₁₀	f ₁₁	f ₁₂	f ₁₃	f ₁₄	f ₁₅
0 0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
0 1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1 0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1 1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

$$Y=A$$

$$Y=\bar{B}$$

$$Y=\overline{\bar{A}B} = A + \bar{B}$$

Funzioni booleane di 2 variabili

Se le variabili sono **2** allora ottengo $2^4 = 16$ possibili funzioni booleane

A B	f ₀	f ₁	f ₂	f ₃	f ₄	f ₅	f ₆	f ₇	f ₈	f ₉	f ₁₀	f ₁₁	f ₁₂	f ₁₃	f ₁₄	f ₁₅
0 0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0 1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1 0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1 1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Y=B

Y= \bar{A}

Y= $\overline{AB} = \bar{A} + B$

Funzioni booleane di 2 variabili

- Se le variabili sono 2 allora ottengo 16 possibili funzioni booleane

A B	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}
0 0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
0 1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1 0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1 1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

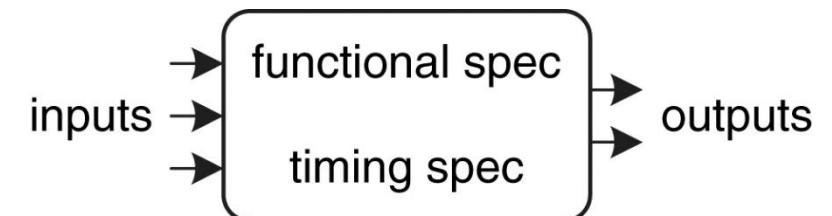
$$Y = (A \leq B) \cdot (B \leq A) \quad A \equiv B$$

$$Y = B \leq A$$

$$Y = A \leq B$$

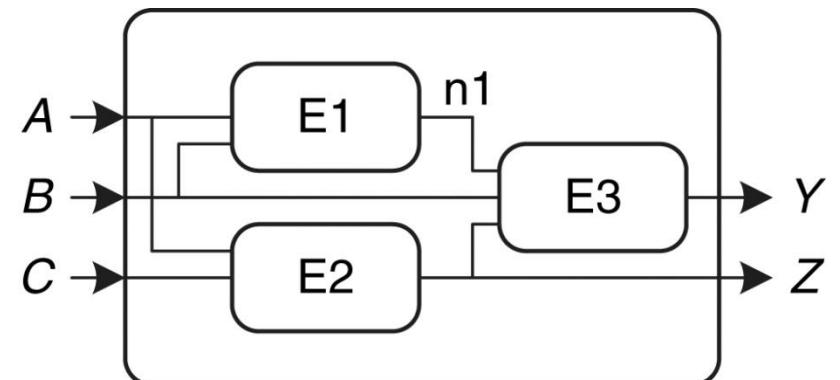
Circuiti digitali

- In generale un circuito digitale è una rete che elabora segnali discreti (rappresentati da variabili booleane). Prescindendo dalla sua configurazione interna, un circuito può essere visto come una *black-box* con
 - Uno o più input
 - Uno o più output
 - Una *specifica funzionale* che rappresenta la relazione fra input e output
 - Una *specifica temporale* che descrive il ritardo che intercorre affinché i segnali di input si propaghino nel circuito fino agli output.



Circuiti digitali

- La struttura interna di un circuito è composta da *elementi* e *nodi*.
- Un elemento è esso stesso un circuito digitale.
- Un nodo è una connessione che trasporta il segnale (e.g. filo elettrico). I nodi si distinguono in nodi di input, output e interni
 - input: riceve il segnale dal mondo esterno
 - output: riporta il segnale al mondo esterno
 - Nodo interno: connette due elementi

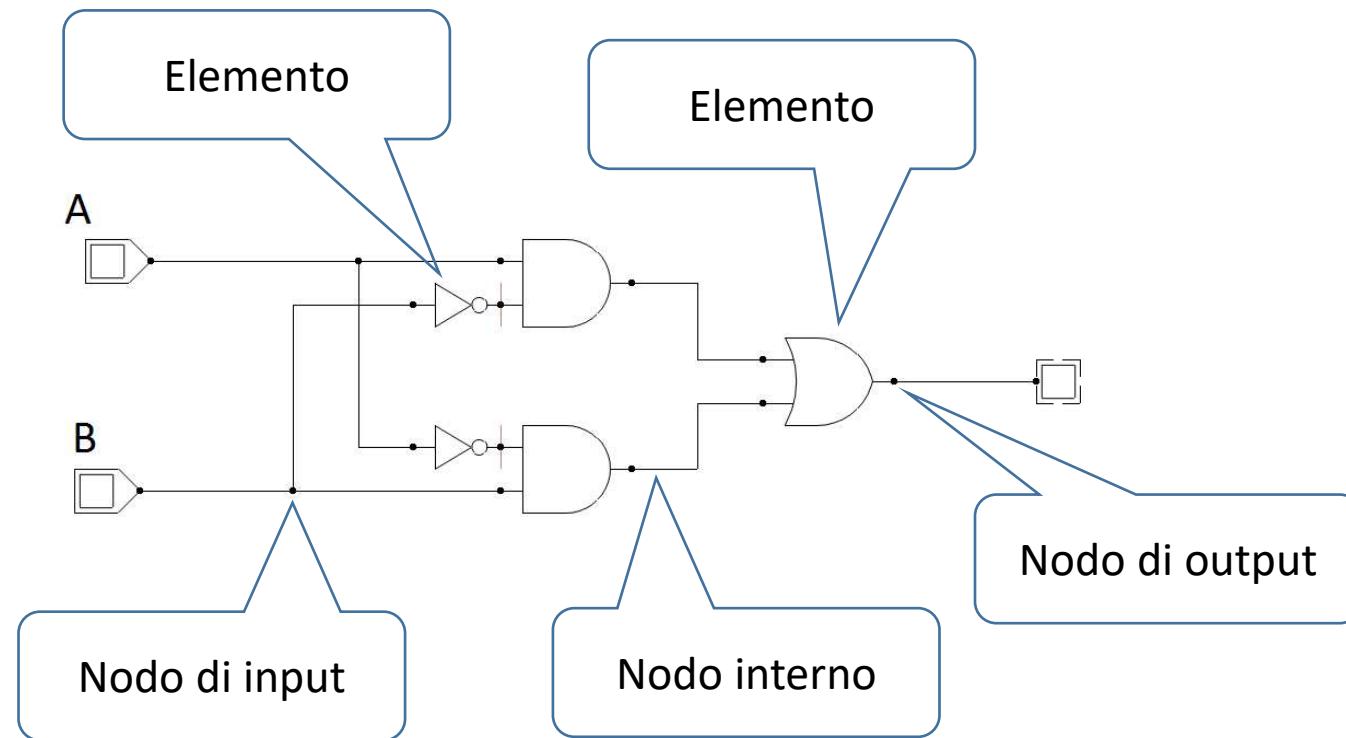


Reti combinatorie e sequenziali

- Vi sono due grandi categorie di circuiti digitali: le reti combinatorie e le reti sequenziali.
- In una rete combinatoria, i valori degli output dipendono esclusivamente dal valore corrente degli input (al netto dei ritardi di propagazione). In tal senso, le reti combinatorie si dicono memoryless, ovvero non hanno memoria della “*storia*” precedente del circuito
- In un rete sequenziale, invece, gli output dipendono non solo dal valore corrente degli input, ma anche dai valori precedenti. Si dice quindi che il circuito ha memoria.

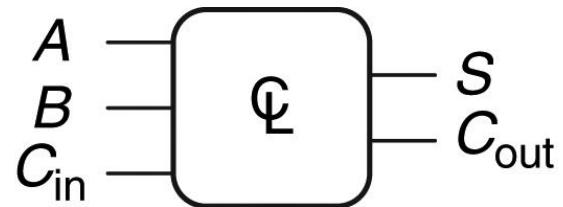
Reti combinatorie

- Le porte logiche viste finora sono un esempio di rete combinatoria.
- Un'altro esempio di rete combinatoria l'abbiamo visto in precedenza



Full adder

- Quando non siamo interessati alla struttura interna allora una rete combinatoria è descritta come segue

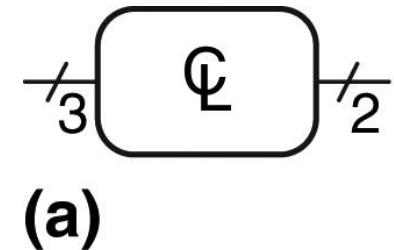


$$S = A \oplus B \oplus C_{in}$$
$$C_{out} = AB + AC_{in} + BC_{in}$$

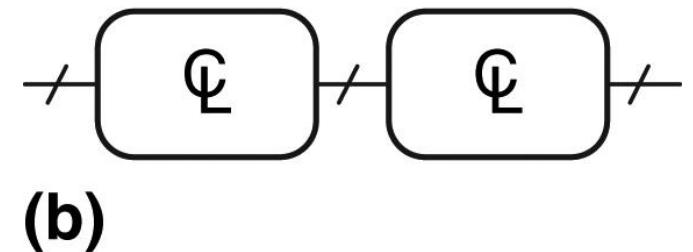
- Questa rete combinatoria rappresenta un full adder di cui parleremo in seguito.
- Come avrete già capito, vi sono molti modi differenti per realizzare le funzioni che corrispondono al output S e C_{out}

Input e output multipli

- Per semplificare la rappresentazione grafica, si usa la notazione qui di fianco per indicare ingressi ed uscite multiple.
- In figura (a) un generico circuito combinatorio con 3 input e 2 output
- Il numero di input e output può essere omesso quando chiaro dal contesto o non rilevante. In figura (b) due generiche reti combinatorie in sequenza. Notate che il numero di output della prima rete deve essere uguale al numero di input della seconda.



(a)



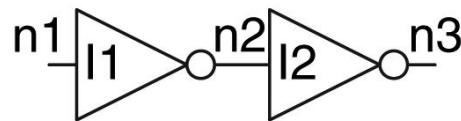
(b)

Regole di composizione di reti combinatorie

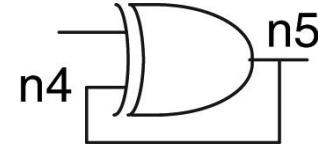
- Come abbiamo visto reti combinatorie possono essere composte per formare reti più grandi. Cosa ci garantisce che il risultato sia ancora una rete combinatoria? Ecco un insieme di regole che costituiscono delle condizioni sufficienti (ma non necessarie)
 - Ogni elemento è esso stesso una rete combinatoria
 - Ogni nodo che non è un input connette esattamente un output di un elemento
 - Il circuito non contiene cicli, ogni cammino interno alla rete visita un nodo al più una volta

Regole di composizione di reti combinatorie

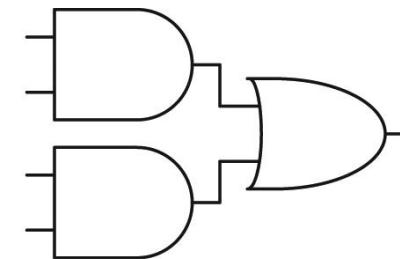
- Quali dei seguenti circuiti costituisce, secondo le regole di composizione, una rete combinatoria?



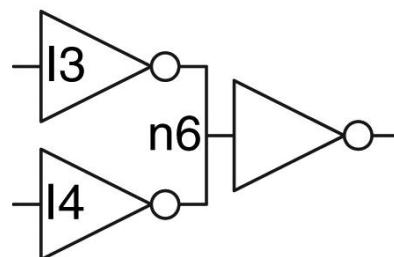
(a)



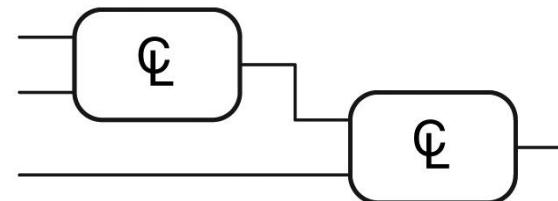
(b)



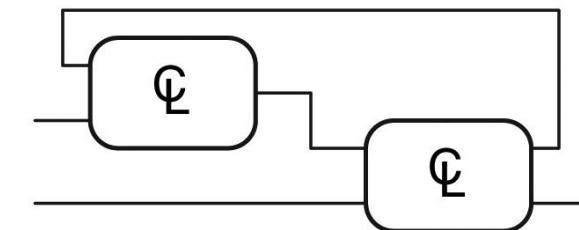
(c)



(d)



(e)



(f)

Tabella di verità e espressioni booleane

- Le specifiche funzionali di un circuito combinatorio sono descritte tramite tabelle di verità o espressioni booleane
- In generale, più espressioni booleane corrispondono alla stessa tabella di verità
 - Per esempio abbiamo visto che $\overline{A}\bar{B}$ e $\bar{A} + B$ corrispondono alla stessa tabella di verità, ovvero $\overline{A}\bar{B} = \bar{A} + B$.
- Come si ricava la tabella di verità di una data espressione?
- Viceversa, a partire da una tabella di verità, come si ricava una espressione corrispondente?

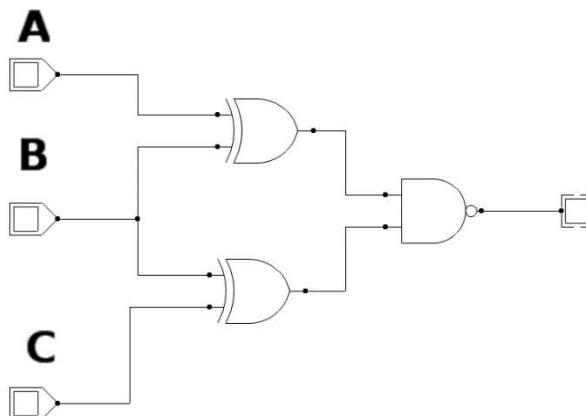
Dalle espressioni alle tavole

- Il calcolo di una tabella di verità a partire una espressione booleana non è dissimile concettualmente dal calcolo di una espressione aritmetica. Si calcolano a ritroso le tavole di verità delle sue sotto espressioni.
- Consideriamo l'espressione $(\bar{A}B) + (B \oplus C)$

A	B	C	\bar{A}	$\bar{A} \cdot B$	$B \oplus C$	$(\bar{A}B) + (B \oplus C)$
0	0	0	1	0	0	0
0	0	1	1	0	1	1
0	1	0	1	1	1	1
0	1	1	1	1	0	1
1	0	0	0	0	0	0
1	0	1	0	0	1	1
1	1	0	0	0	1	1
1	1	1	0	0	0	0

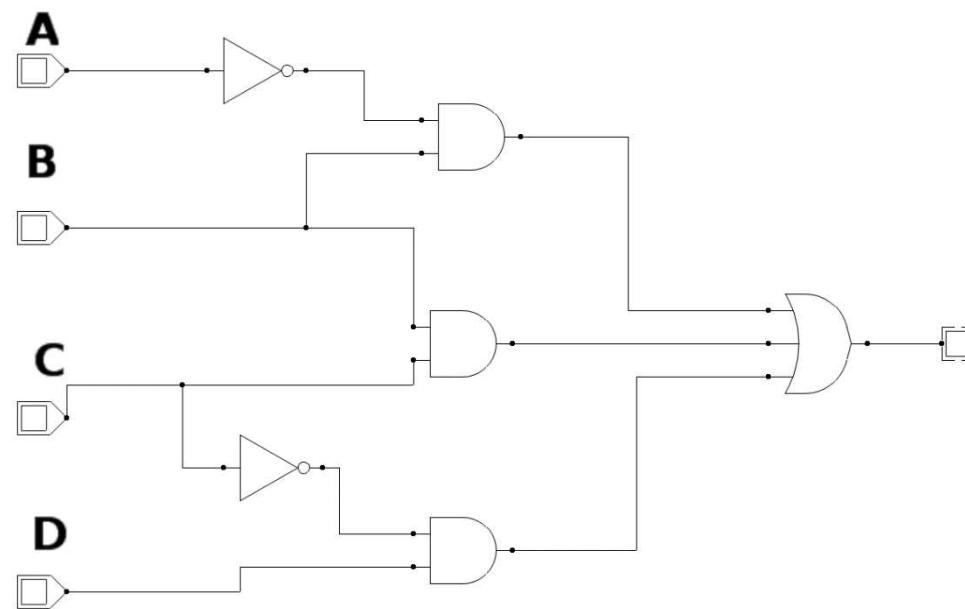
Esercizi

- Dimostrare che $(A \cdot \bar{B}) + (\bar{A} \cdot B)$ è equivalente alla porta XOR calcolandone la tabella di verità
- Esercizi del libro “Digital Design and Computer Architecture – ARM edition”
 - 1.86
 - 1.87
- Scrivere l'espressione booleana e la relativa tabella di verità della rete:



Esercizi

- Scrivere l'espressione booleana e la relativa tabella di verità della rete:



Dalle tabelle alle espressioni

- ricavare da una tabella di verità un'espressione booleana
- forma sintattica ben precisa: **SOP** (somma di prodotti)
- Definizioni preliminari:
 - Una variabile booleana A o la sua negata \bar{A} sono detti **litterali**
 - Un prodotto (AND) di litterali è detto **implicante**: $\bar{A}B$, $\bar{A}\bar{B}\bar{C}$, B sono tutti implicanti per una funzione booleana di almeno 3 variabili.
 - Dato un insieme K di variabili booleane, un **mintermine** di K è un implicante che comprende (positive o negative) tutte le variabili in K .
 - Ad esempio, se $K = \{A, B, C\}$ allora $\bar{A}\bar{B}\bar{C}$ è un mintermine per K . Invece, $\bar{A}B$ è un implicante ma non un mintermine.
 - Analogamente, un **maxtermine** di K è una somma di litterali in cui occorrono tutte le variabili in K .
 - Ad esempio $\bar{A} + \bar{B} + \bar{C}$ è un maxtermine di $K = \{A, B, C\}$.
- ordine di precedenza NOT → AND → OR
 - $\bar{A}\bar{B}\bar{C} + CB = (\bar{A}\bar{B}\bar{C}) + (CB)$

La forma SOP

- Ognuna delle 2^N righe di una tabella di verità è caratterizzata da un mintermone

A B	f ₀	f ₁	f ₂	f ₃	f ₄	f ₅	f ₆	f ₇	f ₈	f ₉	f ₁₀	f ₁₁	f ₁₂	f ₁₃	f ₁₄	f ₁₅
0 0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0 1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1 0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1 1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Y=AB Y=A \bar{B} Y= $\bar{A}B$ Y= $\bar{A}\bar{B}$

La forma SOP

- I mintermini sono enumerati rigo dopo rigo a partire da 0,1 e così via. Quindi ogni mintermine è denotato dal numero binario della configurazione di input corrispondente.

A	B	Y	minterm	minterm name
0	0	0	$\bar{A} \bar{B}$	m_0
0	1	1	$\bar{A} B$	m_1
1	0	0	$A \bar{B}$	m_2
1	1	0	$A B$	m_3

La forma SOP

- Quindi ad ogni tabella di verità corrisponde una espressione booleana ottenuta sommando tutti i mintermini per cui il valore dell'output Y è pari a 1

A	B	Y	minterm	minterm name
0	0	0	$\bar{A} \bar{B}$	m_0
0	1	1	$\bar{A} B$	m_1
1	0	0	$A \bar{B}$	m_2
1	1	1	$A B$	m_3

$$Y = \bar{A}B + AB$$

$$Y = \Sigma(1,3)$$

La forma SOP

- Consideriamo un esempio a tre variabili

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

$$Y = \bar{A}\bar{B}\bar{C} + A\bar{B}\bar{C} + A\bar{B}C$$

$$Y = \Sigma(0,4,5)$$

La forma POS

- Una forma duale per rappresentare una funzione booleana è in forma **POS** (prodotto di somme).
- Ad ogni riga di una tabella di verità corrisponde un maxtermine che è uguale a 0 solo per quella riga

A	B	f ₀	f ₁	f ₂	f ₃	f ₄	f ₅	f ₆	f ₇	f ₈	f ₉	f ₁₀	f ₁₁	f ₁₂	f ₁₃	f ₁₄	f ₁₅
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Y = A + B Y = A + \bar{B} Y = \bar{A} + B Y = \bar{A} + \bar{B}

La forma POS

- Anche i maxtermini sono enumerati come i mintermini.

A	B	Y	maxterm	maxterm name
0	0	0	$A + B$	M_0
0	1	1	$A + \bar{B}$	M_1
1	0	0	$\bar{A} + B$	M_2
1	1	1	$\bar{A} + \bar{B}$	M_3

- La forma normale POS di una funzione booleana si ottiene come prodotto dei maxtermini per cui la funzione ritorna 0
 - $Y = (A + B) \cdot (\bar{A} + B)$
 - $Y = \prod(0,2)$

La forma POS

- $Y = (A + B) \cdot (\bar{A} + B)$

A B	f ₀	f ₁	f ₂	f ₃	f ₄	f ₅	f ₆	f ₇	f ₈	f ₉	f ₁₀	f ₁₁	f ₁₂	f ₁₃	f ₁₄	f ₁₅
0 0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0 1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1 0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1 1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

$\text{Y} = (A + B) \cdot (\bar{A} + B)$

$\text{Y} = A + B$

$\text{Y} = \bar{A} + B$

Forme SOP & POS

SOP – sum-of-products

O	C	E	minterm
0	0	0	$\bar{O} \bar{C}$
0	1	0	$\bar{O} C$
1	0	1	$O \bar{C}$
1	1	0	$O C$

$$E = O\bar{C}$$

$$= \Sigma(2)$$

POS – product-of-sums

O	C	E	maxterm
0	0	0	$O + C$
0	1	0	$O + \bar{C}$
1	0	1	$\bar{O} + C$
1	1	0	$\bar{O} + \bar{C}$

$$E = (O + C)(O + \bar{C})(\bar{O} + C)$$
$$= \Pi(0, 1, 3)$$

Forme SOP & POS

- Se la tabella di verità ha pochi 1 allora la forma SOP è più succinta della forma POS
- Se la tabella di verità ha pochi 0 allora la forma POS è più succinta della forma SOP
- Nel caso in cui il numero di 0 e 1 è pressappoco lo stesso le due forme si equivalgono

Algebra di Boole

- Come abbiamo visto la medesima funzione può essere descritta da espressioni booleane distinte
- Alcune di queste possono essere più semplici di altre

A	B	Y	minterm	minterm name
0	0	0	$\bar{A} \bar{B}$	m_0
0	1	1	$\bar{A} B$	m_1
1	0	0	$A \bar{B}$	m_2
1	1	1	$A B$	m_3

$$Y = \bar{A}B + AB$$

$$Y = (A + B) \cdot (\bar{A} + B)$$

$$Y = B$$

- Come si fa con l'aritmetica, possiamo utilizzare un'algebra per semplificare le espressioni

$$\frac{1}{x} (x + xy) \Rightarrow \frac{x}{x} (1 + y) \Rightarrow (1 + y)$$

Axiomi dell'algebra di Boole

	Axiom		Dual	Name
A1	$B = 0 \text{ if } B \neq 1$		$A1' \quad B = 1 \text{ if } B \neq 0$	Binary field
A2	$\bar{0} = 1$		$A2' \quad \bar{1} = 0$	NOT
A3	$0 \bullet 0 = 0$		$A3' \quad 1 + 1 = 1$	AND/OR
A4	$1 \bullet 1 = 1$		$A4' \quad 0 + 0 = 0$	AND/OR
A5	$0 \bullet 1 = 1 \bullet 0 = 0$		$A5' \quad 1 + 0 = 0 + 1 = 1$	AND/OR

Axiomi dell'algebra di Boole

Axiom		Dual		Name
A1	$B = 0 \text{ if } B \neq 1$	A1'	$B = 1 \text{ if } B \neq 0$	Binary field
A2	$\bar{0} = 1$	A2'	$\bar{1} = 0$	NOT
A3	$0 \bullet 0 = 0$	A3'	$1 + 1 = 1$	AND/OR
A4	$1 \bullet 1 = 1$	A4'	$0 + 0 = 0$	AND/OR
A5	$0 \bullet 1 = 1 \bullet 0 = 0$	A5'	$1 + 0 = 0 + 1 = 1$	AND/OR

- A1 e A1' ci dicono che il valore di una variabile booleana può essere 0 oppure 1

Axiomi dell'algebra di Boole

Axiom		Dual		Name
A1	$B = 0 \text{ if } B \neq 1$	A1'	$B = 1 \text{ if } B \neq 0$	Binary field
A2	$\overline{0} = 1$	A2'	$\overline{1} = 0$	NOT
A3	$0 \bullet 0 = 0$	A3'	$1 + 1 = 1$	AND/OR
A4	$1 \bullet 1 = 1$	A4'	$0 + 0 = 0$	AND/OR
A5	$0 \bullet 1 = 1 \bullet 0 = 0$	A5'	$1 + 0 = 0 + 1 = 1$	AND/OR

- A1 e A1' ci dicono che il valore di una variabile booleana può essere 0 oppure 1
- A2 e A2' definiscono l'operatore NOT (di fatto questi assiomi ripropongono la tabella di verità dell'operatore)

Axiomi dell'algebra di Boole

	Axiom	Dual	Name
A1	$B = 0 \text{ if } B \neq 1$	$A1'$ $B = 1 \text{ if } B \neq 0$	Binary field
A2	$\bar{0} = 1$	$A2'$ $\bar{1} = 0$	NOT
A3	$0 \bullet 0 = 0$	$A3'$ $1 + 1 = 1$	AND/OR
A4	$1 \bullet 1 = 1$	$A4'$ $0 + 0 = 0$	AND/OR
A5	$0 \bullet 1 = 1 \bullet 0 = 0$	$A5'$ $1 + 0 = 0 + 1 = 1$	AND/OR

- A3, A4 e A5 definiscono l'operatore AND
- A3', A4' e A5' definiscono l'operatore OR
- Notate che ogni assioma «primato» si ottiene dal corrispondente non primato invertendo, da un lato, OR e AND e dall'altro gli 0 e 1. Questo è un principio generale detto principio di dualità.

Teoremi ad una variabile

	Theorem		Dual	Name
T1	$B \cdot 1 = B$	T1'	$B + 0 = B$	Identity
T2	$B \cdot 0 = 0$	T2'	$B + 1 = 1$	Null Element
T3	$B \cdot B = B$	T3'	$B + B = B$	Idempotency
T4		$\overline{\overline{B}} = B$		Involution
T5	$B \cdot \overline{B} = 0$	T5'	$B + \overline{B} = 1$	Complements

Teorema $B \cdot 1 = B$

Dimostrazione:

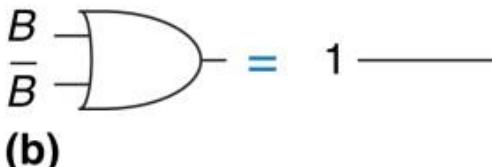
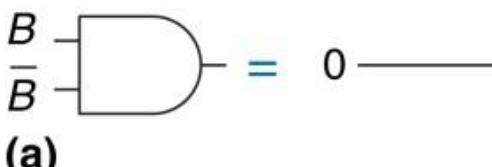
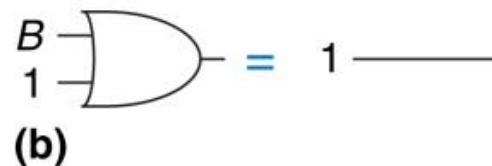
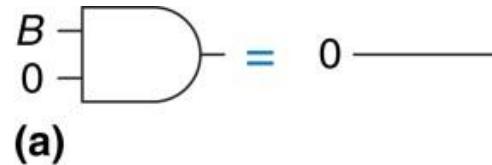
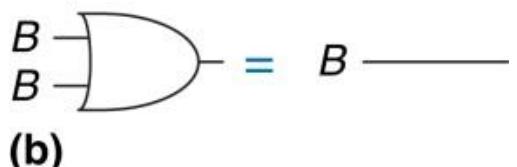
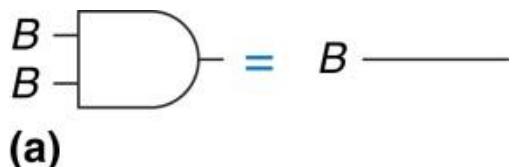
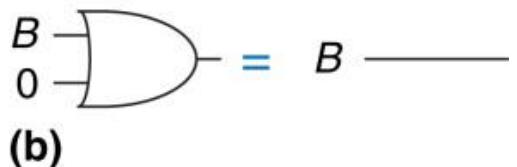
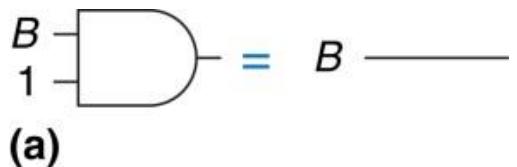
- Supponiamo $B=1$, allora $B \cdot 1 = 1 \cdot 1 = 1 = B$
- Supponiamo $B=0$, allora $B \cdot 1 = 0 \cdot 1 = 0 = B$

Teorema $B + \overline{B} = 1$

Dimostrazione:

- Supponiamo $B=1$, allora $B + \overline{B} = 1 + 0 = 1$
- Supponiamo $B=0$, allora $B + \overline{B} = 0 + 1 = 1$

Corrispondenza hardware



Teoremi più variabili

Number	Theorem	Name
T6	$B \bullet C = C \bullet B$	Commutativity
T7	$(B \bullet C) \bullet D = B \bullet (C \bullet D)$	Associativity
T8	$B \bullet (C + D) = (B \bullet C) + (B \bullet D)$	Distributivity
T9	$B \bullet (B+C) = B$	Covering
T10	$(B \bullet C) + (B \bullet \bar{C}) = B$	Combining
T11	$(B \bullet C) + (\bar{B} \bullet D) + (C \bullet D) = (B \bullet C) + (\bar{B} \bullet D)$	Consensus

Teoremi più variabili: dualità

#	Theorem	Dual	Name
T6	$B \bullet C = C \bullet B$	$B + C = C + B$	Commutativity
T7	$(B \bullet C) \bullet D = B \bullet (C \bullet D)$	$(B + C) + D = B + (C + D)$	Associativity
T8	$B \bullet (C + D) = (B \bullet C) + (B \bullet D)$	$B + (C \bullet D) = (B + C)(B + D)$	Distributivity
T9	$B \bullet (B + C) = B$	$B + (B \bullet C) = B$	Covering
T10	$(B \bullet C) + (B \bullet \bar{C}) = B$	$(B + C) \bullet (B + \bar{C}) = B$	Combining
T11	$(B \bullet C) + (\bar{B} \bullet D) + (C \bullet D) =$ $(B \bullet C) + (\bar{B} \bullet D)$	$(B + C) \bullet (\bar{B} + D) \bullet (C + D) =$ $(B + C) \bullet (\bar{B} + D)$	Consensus

Principio di dualità: $+ \leftrightarrow \cdot \quad 1 \leftrightarrow 0$

Tecniche di dimostrazione

- Vi sono diversi metodi per dimostrare l'equivalenza di due espressioni.
 - Perfect induction: se le tabelle di verità di due espressioni coincidono allora le due espressioni sono equivalenti
 - usare assiomi e teoremi precedentemente provati per manipolare le espressioni fino ad ottenere espressioni uguali
 - Sfruttare la struttura delle espressioni e verificare che le tabelle di verità coincidono solo nei casi rilevanti

Perfect induction: commutativity

Number	Theorem	Name
T6	$B \bullet C = C \bullet B$	Commutativity

B	C	BC	CB
0	0	0	0
0	1	0	0
1	0	0	0
1	1	1	1

Perfect induction: covering

Number	Theorem	Name
T9	$B \bullet (B+C) = B$	Covering

B	C	$(B+C)$	$B(B+C)$
0	0	0	0
0	1	1	0
1	0	1	1
1	1	1	1

Perfect induction: consensus

Number	Theorem	Name
T11	$(B \bullet C) + (\bar{B} \bullet D) + (C \bullet D) =$ $(B \bullet C) + (\bar{B} \bullet D)$	Consensus

B	C	D	$BC + \bar{B}D + CD$	$BC + \bar{B}D$
0	0	0	0	0
0	0	1	1	1
0	1	0	0	0
0	1	1	1	1
1	0	0	0	0
1	0	1	0	0
1	1	0	1	1
1	1	1	1	1

Limiti del perfect induction

- La tecnica del perfect induction è semplice ma «priva di intelligenza»
- Al crescere della lunghezza delle espressioni diventa sempre più laboriosa
- Al crescere delle variabili che occorrono nelle espressioni diventa estremamente più laboriosa
 - $4 \rightarrow 16$ checks
 - $5 \rightarrow 32$ checks
 - $6 \rightarrow 64$ checks
 - ...

Proof teoretico: covering

Number	Theorem	Name
T9	$B \bullet (B+C) = B$	Covering

$$\begin{aligned} B \bullet (B+C) &= B \bullet B + B \bullet C && T8: \text{Distributivity} \\ &= B + B \bullet C && T3: \text{Idempotency} \\ &= B \bullet (1 + C) && T8: \text{Distributivity} \\ &= B \bullet (1) && T2: \text{Null element} \\ &= B && T1: \text{Identity} \end{aligned}$$

Proof teoretico: combining

Number	Theorem	Name
T10	$(B \bullet C) + (B \bullet \bar{C}) = B$	Combining

$$\begin{aligned} B \bullet C + B \bullet \bar{C} &= B \bullet (C + \bar{C}) && \text{T8: Distributivity} \\ &= B \bullet (1) && \text{T5': Complements} \\ &= B && \text{T1: Identity} \end{aligned}$$

Proof teoretico: consensus

Number	Theorem	Name
T11	$\begin{aligned} (B \bullet C) + (\bar{B} \bullet D) + (C \bullet D) &= \\ (B \bullet C) + (\bar{B} \bullet D) \end{aligned}$	Consensus

$$B \bullet C + \bar{B} \bullet D + C \bullet D$$

$$= BC + \bar{B}D + (CDB + CDB)$$

$$= BC + \bar{B}D + BCD + \bar{B}CD$$

$$= BC + BCD + \bar{B}D + \bar{B}CD$$

$$= (BC + BCD) + (\bar{B}D + \bar{B}CD)$$

$$= BC + \bar{B}D$$

T10: Combining

T6: Commutativity

T6: Commutativity

T7: Associativity

T9': Covering

Per casi rilevanti

- Consideriamo T11

$$E: B \cdot C + \bar{B} \cdot D + C \cdot D$$

$$F: B \cdot C + \bar{B} \cdot D$$

Poiché $E = F + CD$, se F ritorna 1, allora E ritorna 1.

Occorre quindi dimostrare il contrario: se E ritorna 1, allora F ritorna 1.

Notate che E ritorna 1 sse (almeno) una fra $B\bar{C}$, BD , o CD ritornano 1.

Mostreremo allora che l'alternativa CD è superflua, ovvero che quando CD ritorna 1 almeno una $\bar{B}D$ e BC ritorna 1.

Supponiamo $CD = 1$, per definizione questo vuol dire che $C=1$ e $D=1$. Quindi se $B=0$, allora $\bar{B}D = 1$, altrimenti $BC=1$.

Per casi rilevanti

B	C	D	$BC + \bar{B}D + CD$	$BC + \bar{B}D$
0	0	0	0	0
0	0	1	1	1
0	1	0	0	0
0	1	1	1	1
1	0	0	0	0
1	0	1	0	0
1	1	0	1	1
1	1	1	1	1

Teoremi di De Morgan

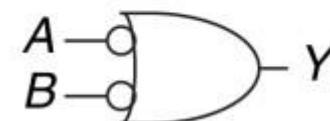
#	Theorem	Dual	Name
T12	$\overline{B_0 \cdot B_1 \cdot B_2 \dots} = \overline{B_0} + \overline{B_1} + \overline{B_2} \dots$	$\overline{B_0} + \overline{B_1} + \overline{B_2} \dots = \overline{B_0 \cdot B_1 \cdot B_2 \dots}$	DeMorgan's Theorem

- La negata di un prodotto è uguale alla somma delle negate
- La negata di una somma è uguale al prodotto delle negate

Teoremi di De Morgan

- I teoremi di De Morgan consentono di convertire un porta AND in una OR e viceversa

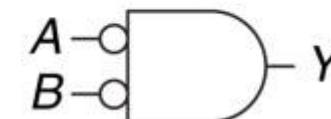
NAND



$$Y = \overline{AB} = \overline{A} + \overline{B}$$

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

NOR



$$Y = \overline{A+B} = \overline{A} \cdot \overline{B}$$

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

Esempio

$$\begin{aligned}Y &= \overline{(A+\overline{BD})\overline{C}} \\&= \overline{(A+\overline{BD})} + \overline{\overline{C}} \\&= (\overline{A} \bullet (\overline{BD})) + C \\&= (\overline{A} \bullet (BD)) + C \\&= \overline{A}BD + C\end{aligned}$$

Altri due teoremi fondamentali

- Sia E una espressione booleana che contiene una sotto espressione E_1 . Se $E_1 = E_2$, allora $E = E'$ dove E' è ottenuto sostituendo E_1 con E_2 .
 - Esempio: $BC + \bar{B}D + CD = BC + \bar{B}D$
 $(BC + \bar{B}D + CD)(FD + A) = (BC + \bar{B}D)(FD + A)$
- Se $E_1 = E_2$ allora $E'_1 = E'_2$ dove E'_1 e E'_2 sono ottenuti sostituendo uniformemente una variabile booleana A con una qualsiasi espressione E .
 - Esempio: $A\bar{B} + AB = A$
 $AD\bar{C}\bar{B} + AD\bar{C}B = AD\bar{C}$
 $(CD + A)\bar{B} + (CD + A)B = (CD + A)$
 $(CD + A)\bar{E}\bar{F} + (CD + A)EF = (CD + A)$
- Se $E = E'$ allora $\bar{E} = \bar{E}'$

Leggi di de Morgan e dualità

Alla luce del teorema precedente risulta chiaro che il principio di dualità è una conseguenza dei teoremi di De Morgan

- $B \cdot C = C \cdot B \rightarrow \overline{B} \cdot \overline{C} = \overline{C} \cdot \overline{B} \rightarrow \overline{\overline{B} \cdot \overline{C}} = \overline{\overline{C} \cdot \overline{B}} \rightarrow \overline{\overline{B}} + \overline{\overline{C}} = \overline{\overline{C}} + \overline{\overline{B}} \rightarrow B + C = C + B$
- $B \cdot 0 = 0 \rightarrow \overline{B} \cdot 0 = 0 \rightarrow \overline{\overline{B} \cdot 0} = \overline{0} \rightarrow \overline{\overline{B} + 0} = \overline{0} \rightarrow B + 1 = 1$

Teoremi di De Morgan e forme SOP/POS

I teoremi di De Morgan possono essere usati per ridurre una generica espressione E in forma SOP/POS senza «passare» per la tabella di verità

- Applica esaustivamente «De Morgan» per spingere la negazione nella struttura della formula
- Applica esaustivamente la proprietà distributiva dell'AND sull'OR (SOP)
- Applica esaustivamente la proprietà distributiva dell'OR sull'AND (POS)

Esempio

$$\begin{aligned} Y &= \overline{\overline{(ACE + \overline{D})}} + B \\ &= \overline{\overline{(ACE + \overline{D})}} \bullet \overline{B} \\ &= \overline{\overline{(ACE)} \bullet \overline{\overline{D}}} \bullet \overline{B} \\ &= ((\overline{AC} + \overline{E}) \bullet D) \bullet \overline{B} \\ &= ((AC + \overline{E}) \bullet D) \bullet \overline{B} \\ &= (ACD + D\overline{E}) \bullet \overline{B} \\ &= A\overline{B}CD + \overline{B}D\overline{E} \end{aligned}$$

Da SOP a POS

I teoremi di De Morgan consentono anche di ricavare la forma POS a partire dalla SOP (e viceversa)

A	B	Y	\bar{Y}	minterm
0	0	0	1	$\bar{A} \bar{B}$
0	1	0	1	$\bar{A} B$
1	0	1	0	$A \bar{B}$
1	1	1	0	$A B$

$$\bar{Y} = \bar{A}\bar{B} + \bar{A}B$$

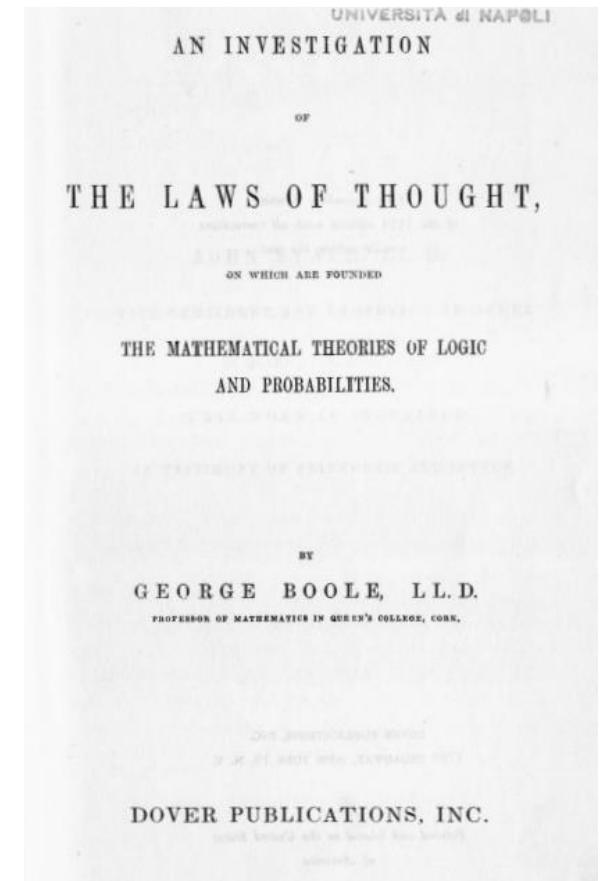
$$\bar{\bar{Y}} = \overline{\bar{A}\bar{B} + \bar{A}B} = \overline{\bar{A}\bar{B}} \cdot \overline{\bar{A}B} = (A + B) \cdot (A + \bar{B})$$

Completezza degli operatori

- Abbiamo visto che una qualsiasi funzione booleana $f:\{0,1\}^N \rightarrow \{0,1\}$ può essere espressa in forma SOP
- La forma SOP usa solo gli operatori NOT, AND, OR
- Questo vuol dire che questo insieme di operatori costituisce un insieme **completo**
- Per le leggi di de Morgan
 - $A \cdot B = \overline{\overline{A} + \overline{B}}$ (AND può essere espresso in termini di NOT e OR)
 - $A + B = \overline{\overline{A} \cdot \overline{B}}$ (OR può essere espresso in termini di NOT e AND)
- Quindi $\{\text{NOT, AND, OR}\}$ è completo ma non minimale essendo $\{\text{AND, NOT}\}$ e $\{\text{OR, NOT}\}$ anche loro completi

Un po' di storia...

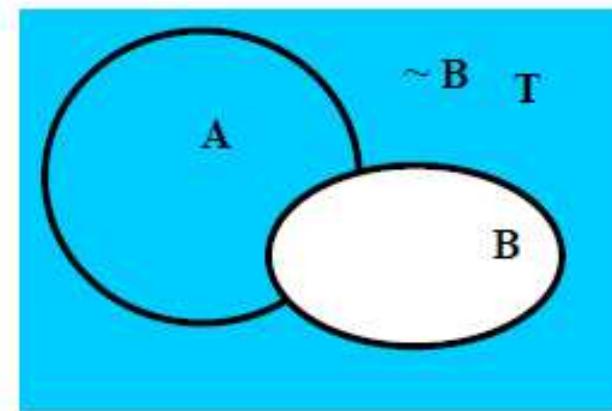
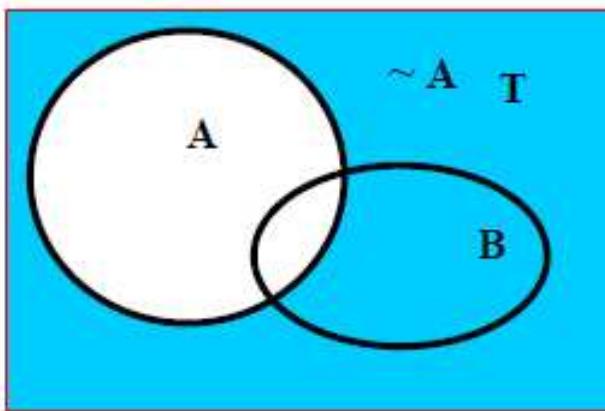
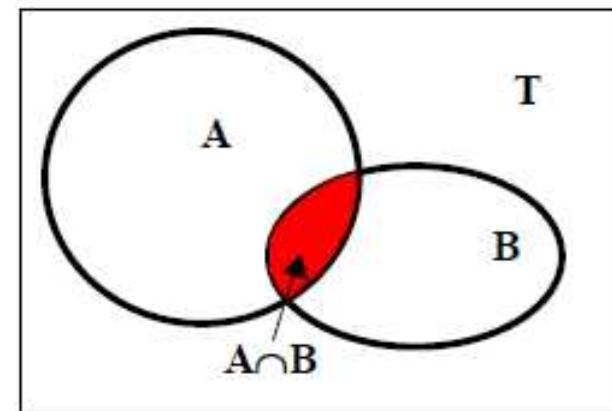
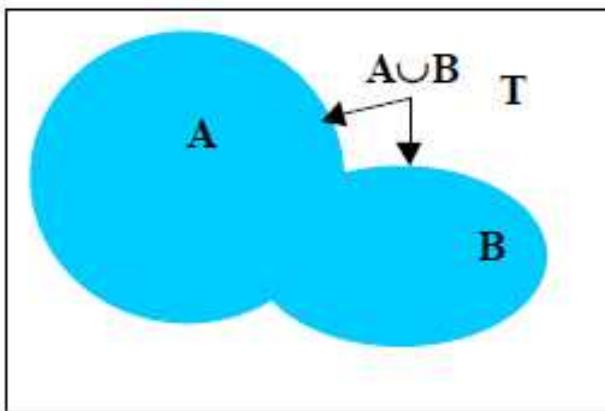
- L'algebra booleana prende nome dal matematico inglese **George Boole**
- L'oggetto di questa algebra non erano le porte logiche (che ancora non esistevano) ma *le leggi del pensiero e della deduzione*
- Infatti si può dare una interpretazione puramente logica delle espressioni booleane in cui
 - Le variabili corrispondono a delle proposizioni che possono essere **vere** (1) o **false** (0)
 - Gli operatori corrispondono alle relative preposizioni coordinate (e, o, non...)
 - «piove **e** ho l'ombrelllo» è vera sse «piove» e «ho l'ombrelllo» sono entrambe vere



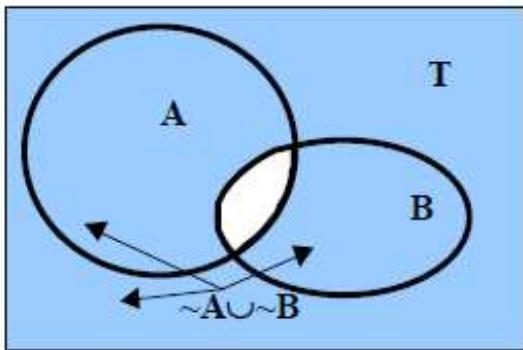
Operatori insiemistici

- Anche gli usuali operatori insiemistici di unione, intersezione e complemento formano un algebra di Boole
- T insieme universo
- $2^T = \{A \mid A \subseteq T\}$ insieme dei sottoinsiemi di T
- $\cap \rightarrow \text{AND}$
- $\cup \rightarrow \text{OR}$
- $T/ \rightarrow \text{NOT}$
- $1 \rightarrow T$
- $0 \rightarrow \emptyset$

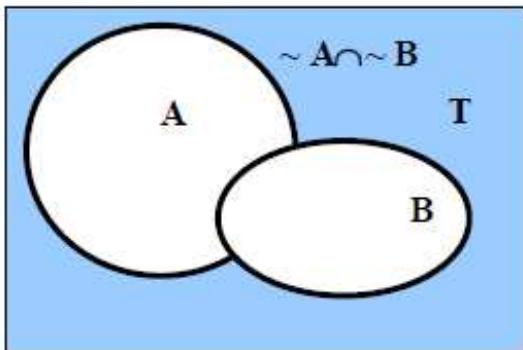
Operatori insiemistici



Operatori insiemistici



dal confronto con la figura
relativa a $A \cap B$ si ricava
 $\sim A \cup \sim B = \sim (A \cap B)$



dal confronto con la
figura relativa a $A \cup B$ si
ricava
 $\sim A \cap \sim B = \sim (A \cup B)$

Esercizi

- Sfruttando i teoremi precedenti verificare le seguenti proprietà della porta XOR

$$a \oplus a = 0$$

$$a \oplus 0 = a$$

$a \oplus 1 = \sim a$, where \sim is bit complement.

$$a \oplus \sim a = 1$$

$$a \oplus b = b \oplus a \text{ (commutativity)}$$

$$a \oplus (b \oplus c) = (a \oplus b) \oplus c \text{ (associativity)}$$

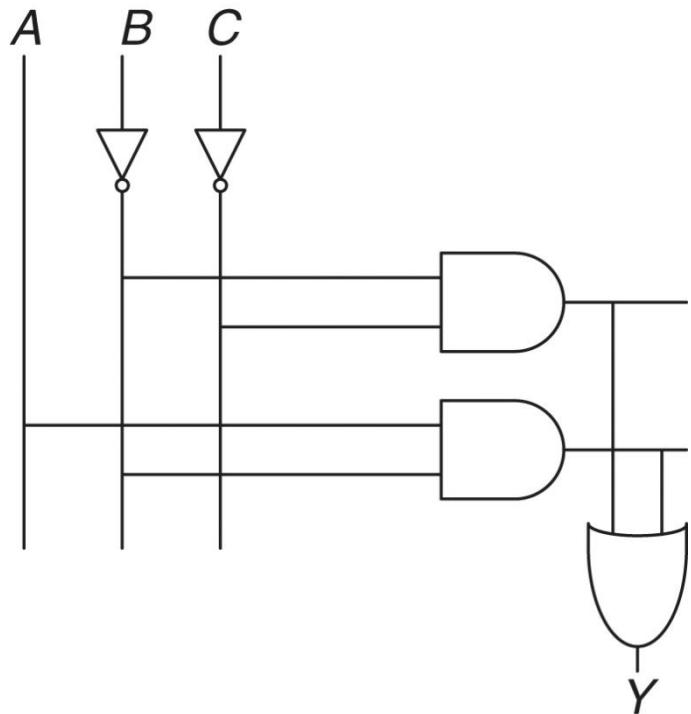
Ricordarsi che, date due espressioni booleane E e F, $E \oplus F = E\bar{F} + \bar{E}F$

Esercizi

- Esercizi del libro “Digital Design and Computer Architecture – ARM edition”
 - 2.1
 - 2.2
 - 2.3
 - 2.4
- Verificare in maniera proof-teoretica che
$$a \oplus b = (\bar{a} + \bar{b})(a + b)$$
- Dimostrare che la porta NAND è da sola un insieme completo (e minimale) di operatori booleani

Perché semplificare una espressione?

- Importanza della minimizzazione: utilizzare meno porte logiche
- Esercizio: mostrare che $\bar{A}\bar{B}\bar{C} + A\bar{B}\bar{C} + A\bar{B}C = \bar{B}\bar{C} + A\bar{B}$



$$Y = \bar{B}\bar{C} + A\bar{B}$$

Esempio 1

$$Y = A(AB + ABC)$$

$$= A(AB(1 + C))$$

T8: Distributivity

$$= A(AB(1))$$

T2': Null Element

$$= A(AB)$$

T1: Identity

$$= (AA)B$$

T7: Associativity

$$= AB$$

T3: Idempotency

Esempio 2

$$Y = \overline{ABC} + \overline{A}$$

$$= \overline{A}$$

oppure

$$= \overline{A}(BC + 1)$$

$$= \overline{A}(1)$$

$$= \overline{A}$$

T9' Covering: $X + XY = X$

T8: Distributivity

T2': Null Element

T1: Identity

Alcuni teoremi per semplificare espressioni

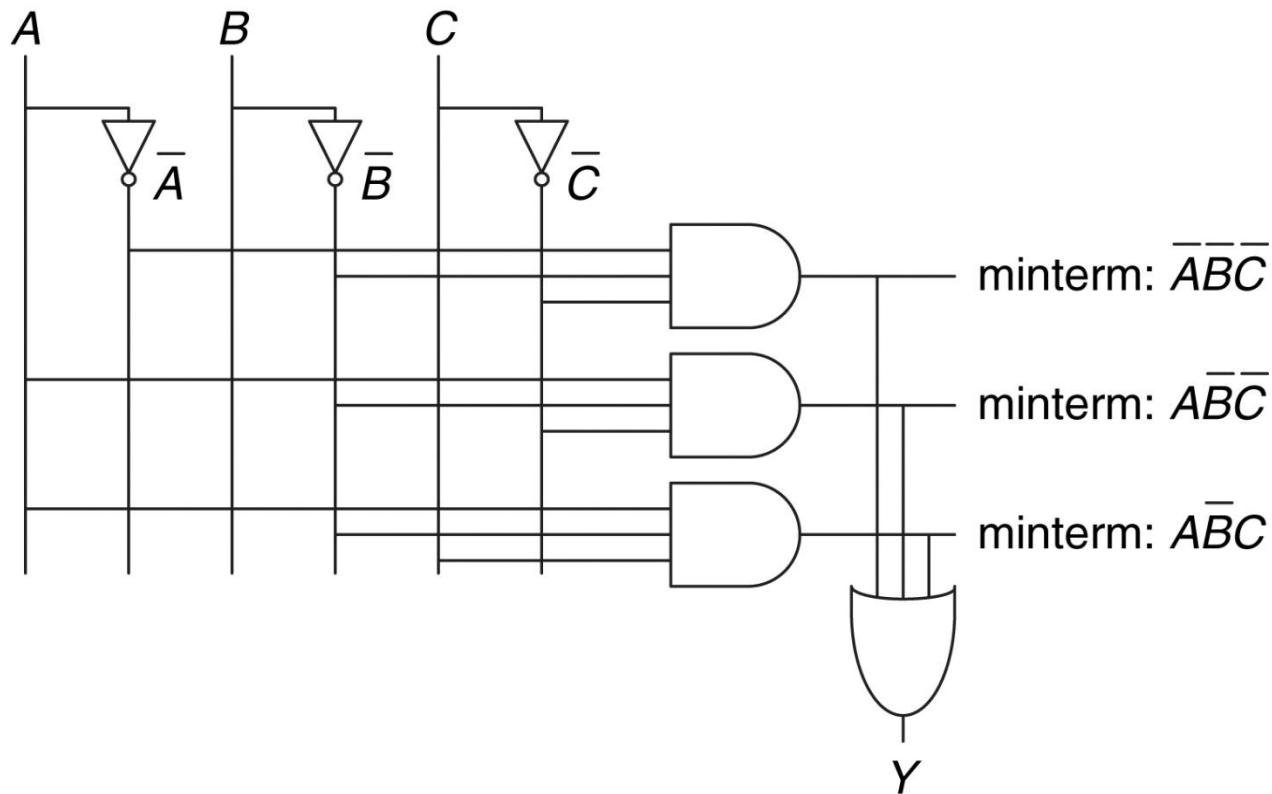
- Distributivity (T8, T8')
$$B(C+D) = BC + BD$$
$$B + CD = (B+C)(B+D)$$
- Covering (T9')
$$A + AP = A$$
- Combining (T10)
$$\overline{PA} + PA = P$$
- Expansion
$$P = \overline{PA} + PA$$
$$A = A + AP$$
- Duplication
$$A = A + A$$
- “Simplification” theorem
$$\overline{PA} + A = P + A$$
$$PA + \overline{A} = P + \overline{A}$$

«Simplification theorem»

$$\begin{aligned} PA + \bar{A} &= PA + (\bar{A} + \bar{A}P) && T9' \text{ Covering} \\ &= PA + P\bar{A} + \bar{A} && T6 \text{ Commutativity} \\ &= P(A + \bar{A}) + \bar{A} && T8 \text{ Distributivity} \\ &= P(1) + \bar{A} && T5' \text{ Complements} \\ &= P + \bar{A} && T1 \text{ Identity} \end{aligned}$$

Schemi circuituali

- Ad ogni espressione booleana corrisponde in circuito combinatorio



$$Y = \bar{A}\bar{B}\bar{C} + \bar{A}\bar{B}C + A\bar{B}C$$

Schemi circuitali SOP

- Le formule in forma SOP hanno degli schemi circuitali molto regolari:
 - Disegna una linea di input per ogni variabile che occorre positiva
 - Aggiungi delle linee con un NOT per ogni variabile che occorre negata
 - Per ogni mintermine disegna una porta AND e aggiungi in ingresso le linee che corrispondono ai relativi litterali
 - Collega tutte le uscite delle porte AND in un unico OR
- Per questo la forma SOP viene detta una logica a due livelli AND-OR

Semplificare formule SOP

- Per T10 $P\bar{B} + PB = P$ per ogni implicante P .
- Un implicante è detto *implicante primo* se non può essere combinato con altri implicanti della formula per ottenere un nuovo implicante con meno litterali.
- Una espressione SOP è minimale se tutti i suoi implicanti sono primi
- Minimizzazione: *ridurre il numero di implicanti e per ogni implicante ridurre il numero di litterali*

Semplificare formule SOP

- Nel minimizzare una SOP, può essere necessario «sdoppiare» un implicante allorché questo può essere ridotto in modi differenti.

Step	Equation	Justification
	$\bar{A}\bar{B}\bar{C} + A\bar{B}\bar{C} + A\bar{B}C$	
1	$\bar{B}\bar{C}(\bar{A} + A) + A\bar{B}C$	T8: Distributivity
2	$\bar{B}\bar{C}(1) + A\bar{B}C$	T5: Complements
3	$\bar{B}\bar{C} + A\bar{B}C$	T1: Identity

Non è minimale $A\bar{B}C$ e $A\bar{B}\bar{C}$ possono ridursi in $A\bar{B}$

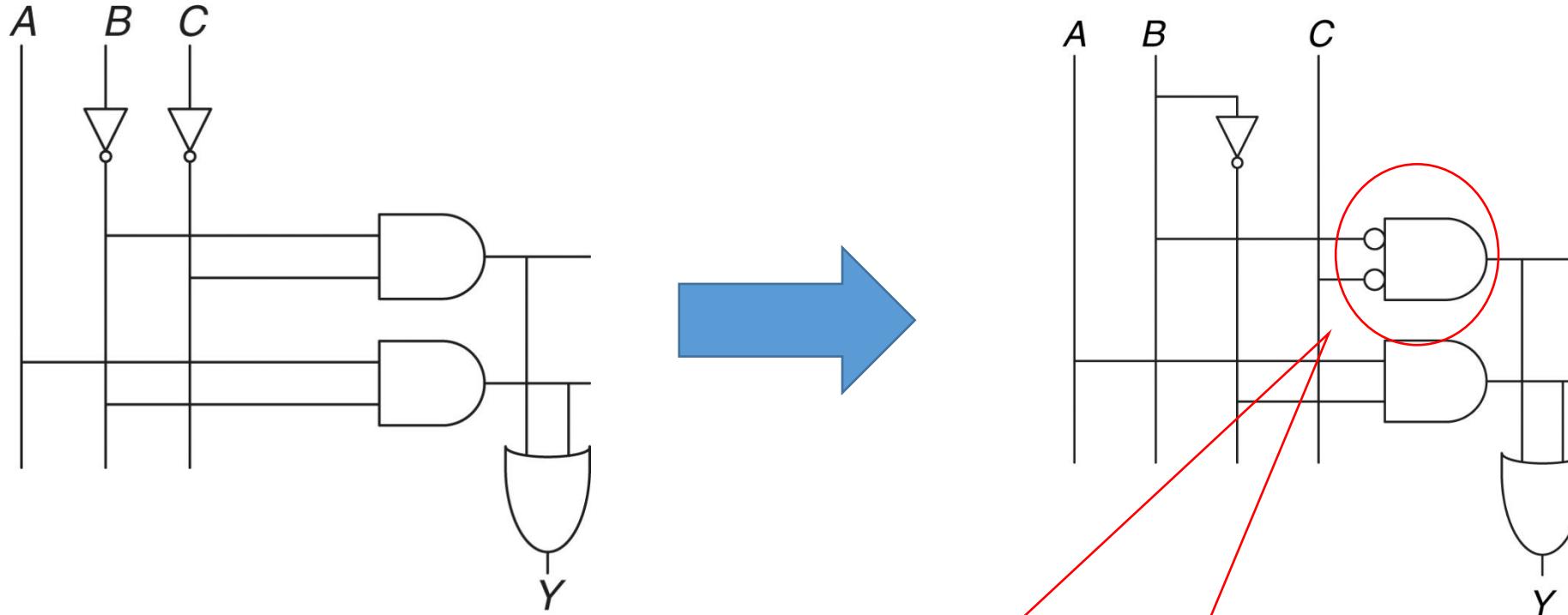
Semplificare formule SOP

Step	Equation	Justification
	$\bar{A}\bar{B}\bar{C} + A\bar{B}\bar{C} + A\bar{B}C$	
1	$\bar{A}\bar{B}\bar{C} + A\bar{B}\bar{C} + A\bar{B}\bar{C} + A\bar{B}C$	T3: Idempotency
2	$\bar{B}\bar{C}(\bar{A} + A) + A\bar{B}(\bar{C} + C)$	T8: Distributivity
3	$\bar{B}\bar{C}(1) + A\bar{B}(1)$	T5: Complements
4	$\bar{B}\bar{C} + A\bar{B}$	T1: Identity



Il numero di implicanti è lo stesso ma
 $A\bar{B}$ ha meno letterali di $A\bar{B}C$

Schemi circuitali e minimizzazione



Per De Morgan: $\bar{B}\bar{C} = \overline{B + C}$ quindi questa porta AND con gli ingressi negati può essere sostituita da un NOR. Nella tecnologia MOSFET il NOR è più veloce di AND

Semplificare le forme POS

- Come si semplifica una forma POS?
- La proprietà principale che si usa è il combining $(B+C) \cdot (B+\bar{C}) = B$
- Esempio:

$$(A + \bar{B} + \bar{C})(\bar{A} + B + \bar{C})(\bar{A} + \bar{B} + C)(\bar{A} + \bar{B} + \bar{C}) =$$

$$(A + \bar{B} + \bar{C})(\bar{A} + B + \bar{C})(\bar{A} + \bar{B} + C)(\bar{A} + \bar{B} + \bar{C})(\bar{A} + \bar{B} + \bar{C})(\bar{A} + \bar{B} + \bar{C}) =$$

$$(A + \bar{B} + \bar{C})(\bar{A} + \bar{B} + \bar{C})(\bar{A} + B + \bar{C})(\bar{A} + \bar{B} + \bar{C})(\bar{A} + \bar{B} + C)(\bar{A} + \bar{B} + \bar{C}) =$$

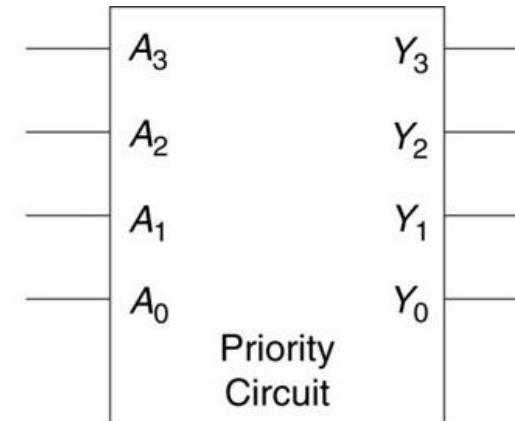
$$(B + \bar{C})(\bar{A} + B + \bar{C})(\bar{A} + \bar{B} + \bar{C})(\bar{A} + \bar{B} + C)(\bar{A} + \bar{B} + \bar{C}) =$$

$$(B + \bar{C})(\bar{A} + \bar{C})(\bar{A} + \bar{B} + C)(\bar{A} + \bar{B} + \bar{C}) =$$

$$(B + \bar{C})(\bar{A} + \bar{C})(\bar{A} + \bar{B})$$

Circuito a priorità

- I circuiti a priorità vengono utilizzati per assegnare una risorsa condivisa secondo un ordine di priorità fra chi ne fa richiesta
- A esempio posso avere 4 possibili richiedenti con priorità $3>2>1>0$
- Gli input A_0, \dots, A_3 rappresentano le richieste della risorsa
- Gli output Y_0, \dots, Y_3 rappresentano a chi assegnata la risorsa, di volta in volta uno solo di essi sarà uguale a 1

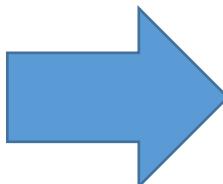


Circuito a priorità

A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	0
0	0	1	1	0	0	1	0
0	1	0	0	0	1	0	0
0	1	0	1	0	1	0	0
0	1	1	0	0	1	0	0
0	1	1	1	0	1	0	0
1	0	0	0	1	0	0	0
1	0	0	1	1	0	0	0
1	0	1	0	1	0	0	0
1	0	1	1	1	0	0	0
1	1	0	0	1	0	0	0
1	1	0	1	1	0	0	0
1	1	1	0	1	0	0	0
1	1	1	1	1	0	0	0

Don't cares

A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	0
0	0	1	1	0	0	1	0
0	1	0	0	0	1	0	0
0	1	0	1	0	1	0	0
0	1	1	0	0	1	0	0
0	1	1	1	0	1	0	0
1	0	0	0	1	0	0	0
1	0	0	1	1	0	0	0
1	0	1	0	1	0	0	0
1	0	1	1	1	0	0	0
1	1	0	0	1	0	0	0
1	1	0	1	1	0	0	0
1	1	1	0	1	0	0	0
1	1	1	1	1	0	0	0

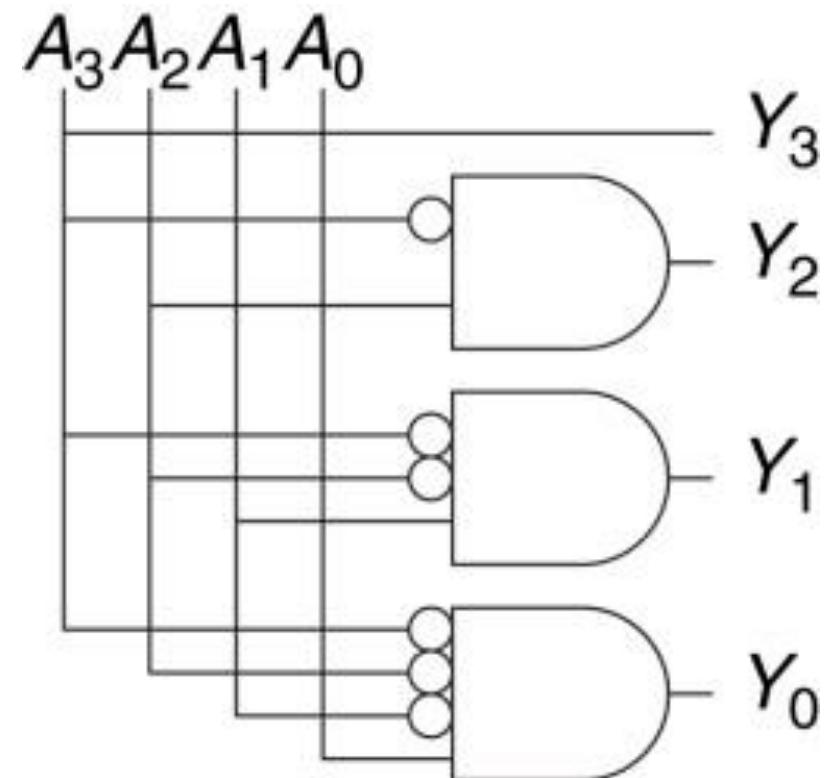


A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	X	0	0	1	0
0	1	X	X	0	1	0	0
1	X	X	X	1	0	0	0

don't cares

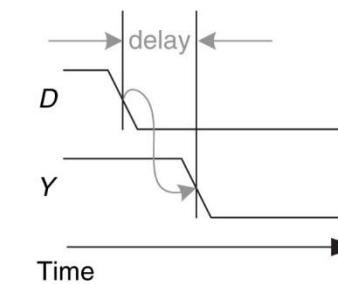
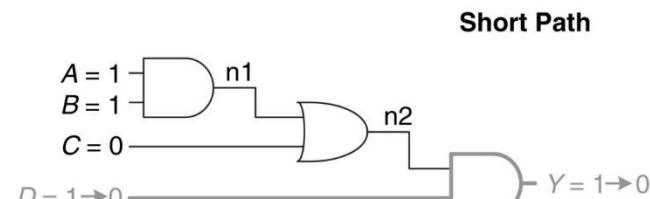
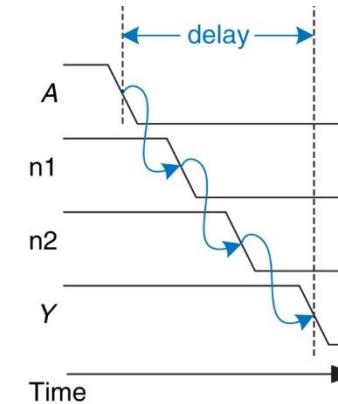
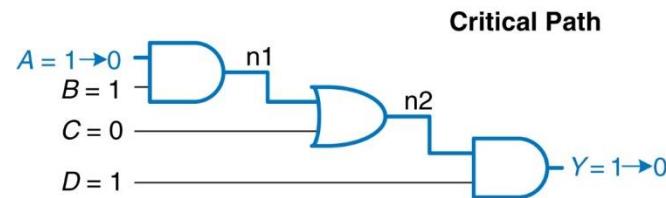
Don't cares

A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	X	0	0	1	0
0	1	X	X	0	1	0	0
1	X	X	X	1	0	0	0



Vantaggi delle SOP/POS

- La logica a 2 livelli della forma SOP presenta dei vantaggi, ad esempio, nei tempi di propagazione



Limiti delle forme SOP/POS

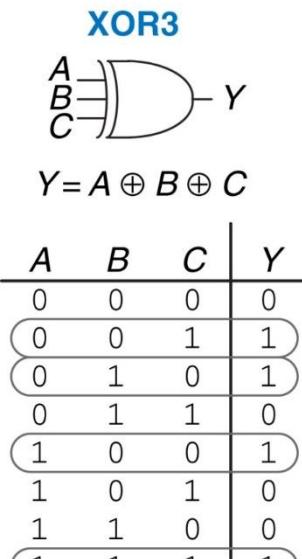
- Tuttavia alcune funzioni booleane, poste in forma SOP, sono estremamente poco succinte e quindi richiedono un numero considerevole di porte
- Presa la tabella di verità di funzione booleana di n variabili:
 - Se il numero di 1 nella colonna di output è piccolo allora la forma SOP è succinta
 - Se il numero di 0 nella colonna di output è piccolo allora la forma POS è succinta
 - Se il numero di 0 e 1 è più o meno lo stesso? Problema: avrò circa 2^{n-1} mintermini/maxtermini

Limiti delle forme SOP/POS

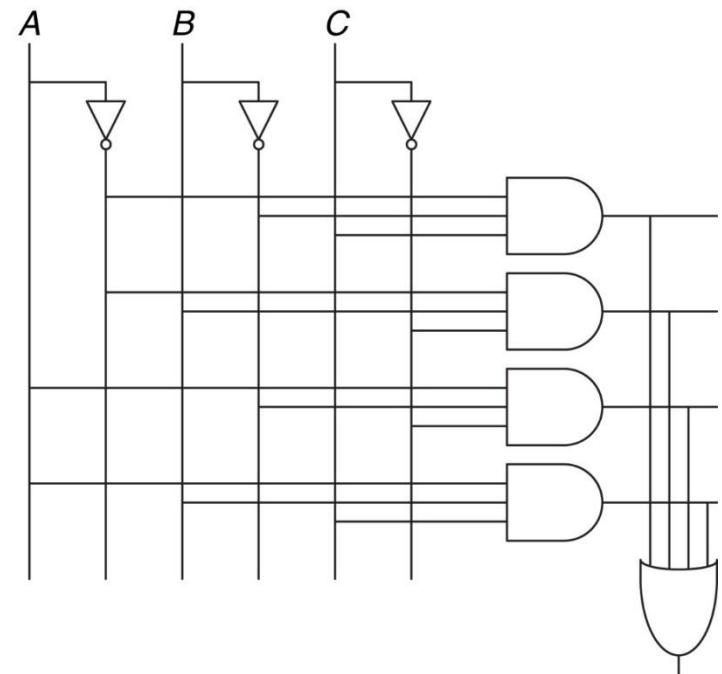
- Consideriamo ad esempio uno XOR a più variabili
- $Y=1$ sse il numero di input uguali a 1 è dispari
- XOR3 in forma SOP

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

- XOR8 richiede 128 AND8 e un OR128



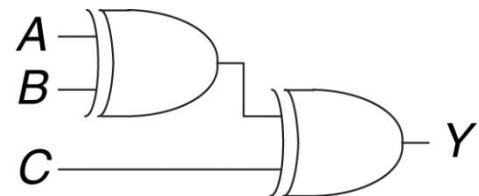
(a)



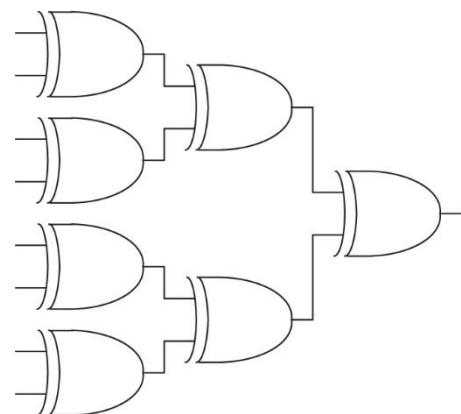
(b)

Logiche multilivello

- Per ridurre il numero di porte logiche a volte è necessario ricorrere ad una logica multilivello
- Ad esempio è facile verificare che $A \oplus B \oplus C = (A \oplus B) \oplus C$



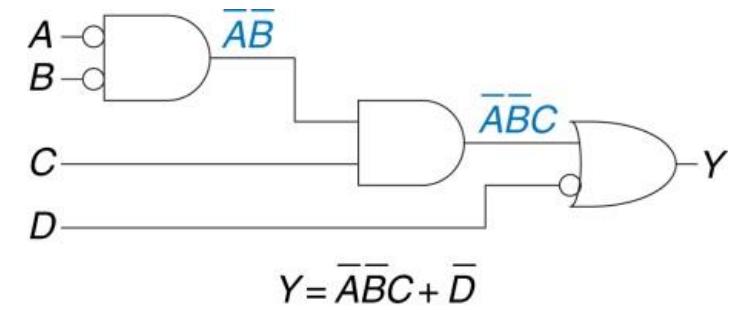
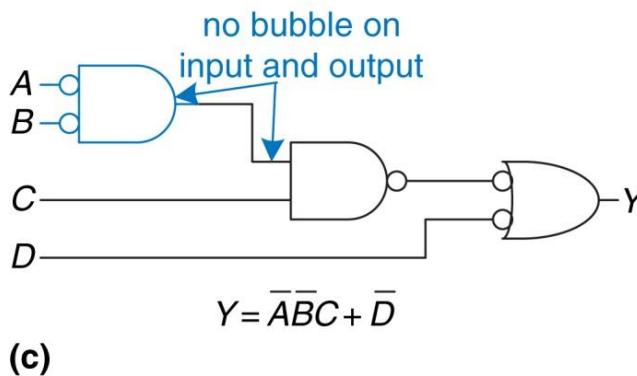
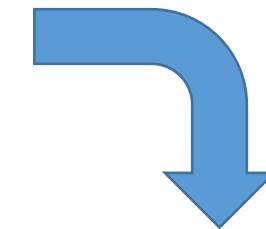
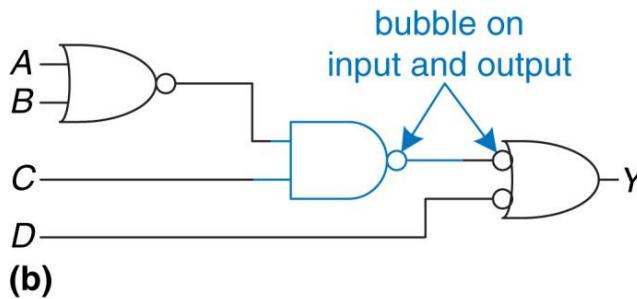
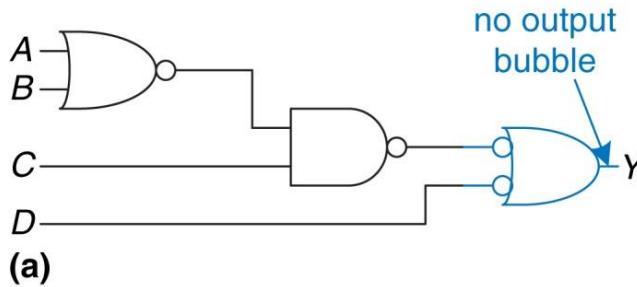
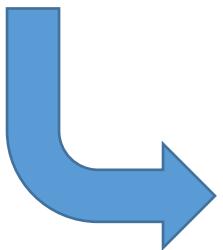
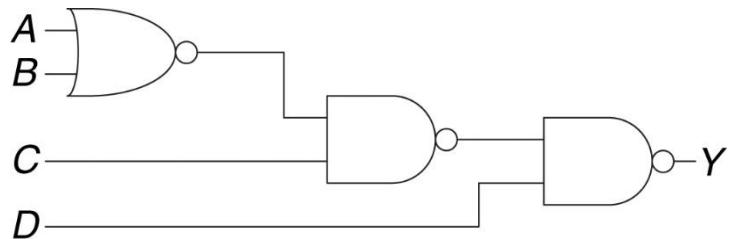
- Analogamente per XOR8:



Bubble pushing

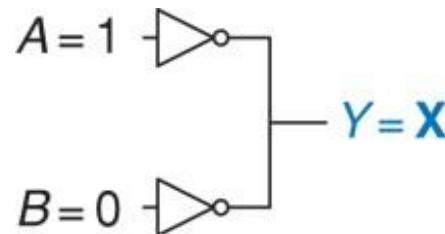
- Usando logiche multilivello e porte NAND/NOR a volte rende difficile capire quale funzione booleana un circuito realizza a causa delle molte negazioni annidate
- Per avere una espressione un po' più leggibile si possono applicare le leggi di De Morgan
- Il bubble pushing è una tecnica che applica sistematicamente le leggi di De Morgan e la legge della doppia negazione per eliminare le negazioni annidate
- Partendo dall'output Y si applicano a ritroso le leggi di De Morgan in modo che l'input e l'output di ogni nodo siano entrambi positivi o negati

Bubble pushing



Valori illegali

- Considerate il circuito in figura. Quale sarà il valore di Y?



- Il valore del potenziale è portato ad essere contemporaneamente sia 0 che V_{DD}
- Questo tipo di configurazioni vengono dette *illegali*. Il valore di Y dipenderà dalla natura fisica delle porte.
 - Stabile ma indeterminato
 - Oscillare fra i valori 0 e V_{DD}
 - Avere un valore intermedio
- Configurazioni del genere possono indurre ad una forte dissipazione che riscalda e danneggia il circuito (corto circuito)

Stato di alta impedenza

- Un nodo, oltre ad essere in uno stato 0/1, può anche essere in un stato Z *floating* o di alta impedenza (alta impedenza=no passaggio di corrente)
- Ad esempio un ingresso di una porta logica non collegato è nello stato Z
- Gli stati di alta impedenza vengono utilizzati per disconnettere una parte di un circuito dal resto. Per questo si utilizzano i tristate buffer

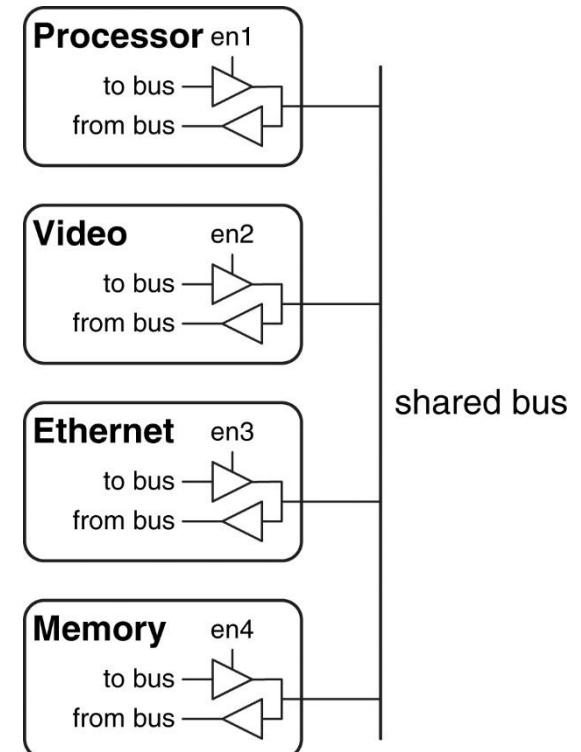
Tristate
Buffer



E	A	Y	E	A	Y
0	0	Z	0	0	0
0	1	Z	0	1	1
1	0	0	1	0	Z
1	1	1	1	1	Z

Bus condiviso

- I tristate buffers sono usati in bus che connettono diversi chip
- Nell'esempio processore, scheda video e controller ethernet devono poter comunicare con la memoria centrale.
- Tuttavia, per evitare stati illegali, solo un componente alla volta può «immettere» segnali sul bus
- Gli altri componenti quindi devono essere temporaneamente disconnessi tramite un tristate buffer



Esercizi

- Esercizi 2.13 – 2.15 – 2.16 – 2.17 – 2.25 – 2.26 – 2.27

Mappe di Karnaugh

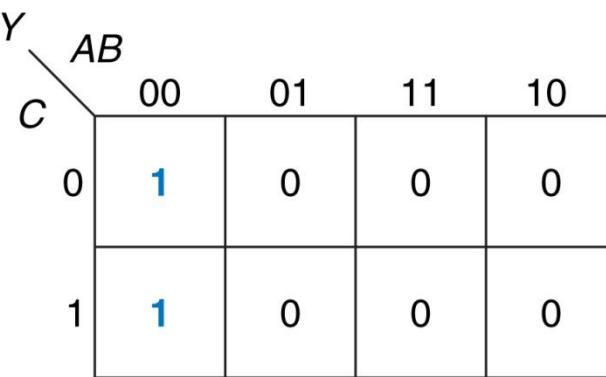
- Le mappe di Karnaugh sono un metodo per semplificare espressioni booleane in forma SOP
- In realtà non introducono tecniche di semplificazione nuove, sono semplicemente un espediente grafico che consente di rilevare più facilmente implicati che possono essere semplificati
- Quindi alla base delle mappe di Karnaugh c'è il solito principio:

$$PA + \bar{P}\bar{A} = P$$

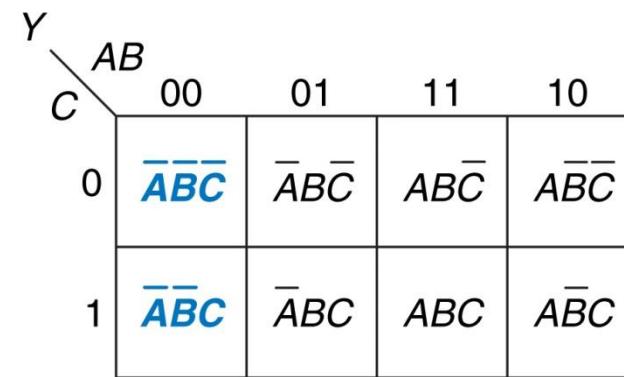
Mappe di Karnaugh

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

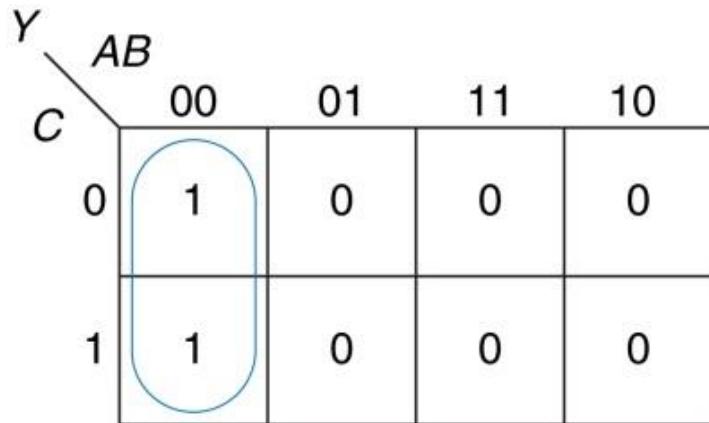
(a)



(b)



(c)

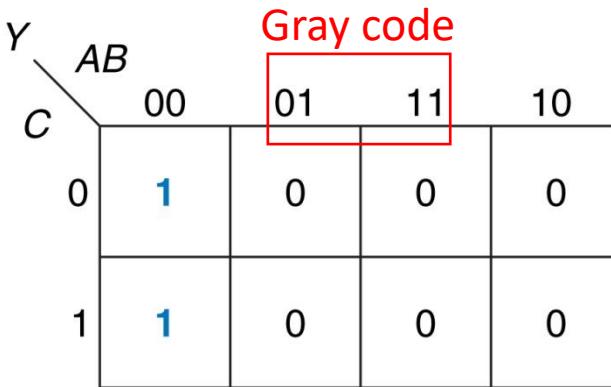


$$Y = \overline{A} \overline{B} \overline{C} + \overline{A} \overline{B} C = \overline{A} \overline{B} (\overline{C} + C) = \overline{A} \overline{B}$$

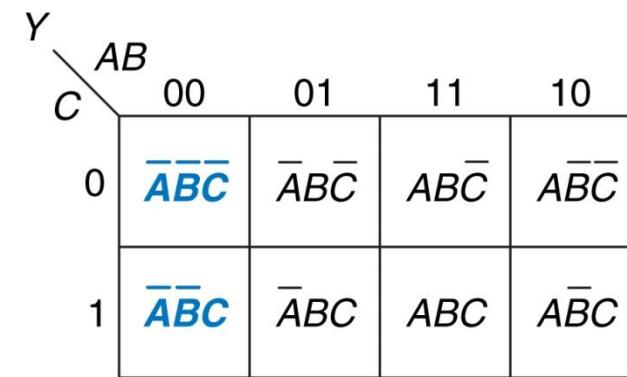
Mappe di Karnaugh

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

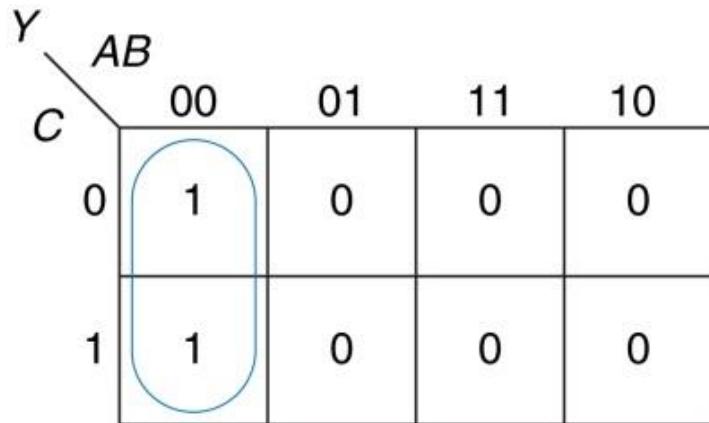
(a)



(b)

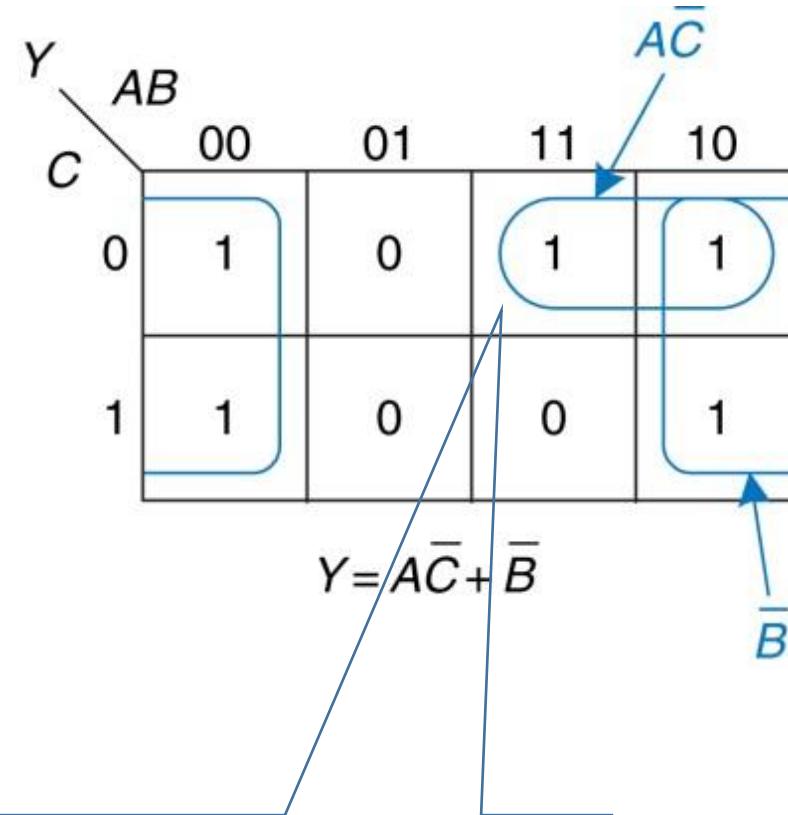


(c)



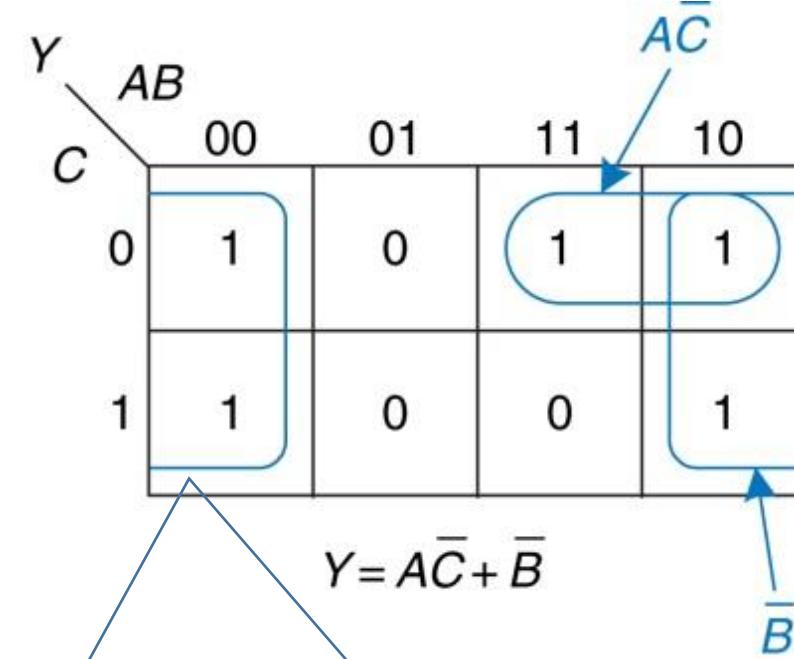
$$Y = \overline{A} \overline{B} \overline{C} + \overline{A} \overline{B} C + A \overline{B} \overline{C} + A B \overline{C} = \overline{A} \overline{B} (\overline{C} + C) = \overline{A} \overline{B}$$

		AB	C	Y	
		00	01	11	10
C	0	1	0	1	1
	1	1	0	0	1



$$AB\bar{C} + A\bar{B}\bar{C} = A\bar{C}(B + \bar{B}) = A\bar{C}$$

		A	B	C	Y
		0	1	0	1
A	B	0		00	01
		1		11	10
0	0	1	0	1	1
	1	1	0	0	1

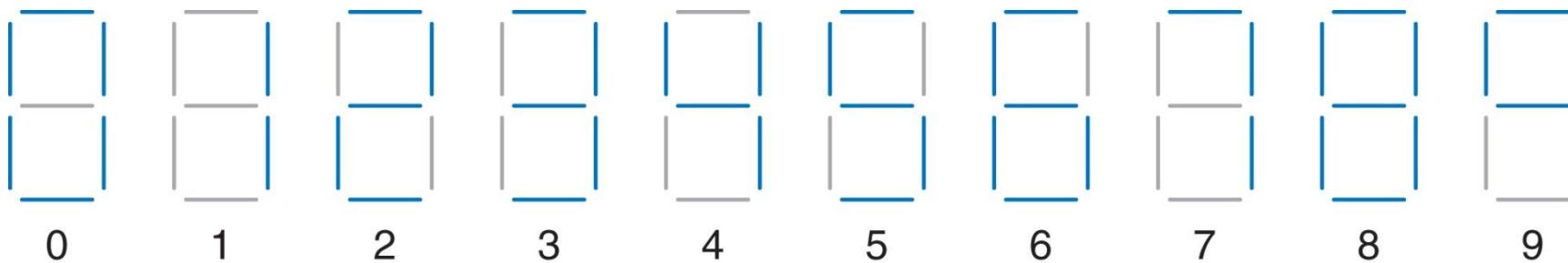
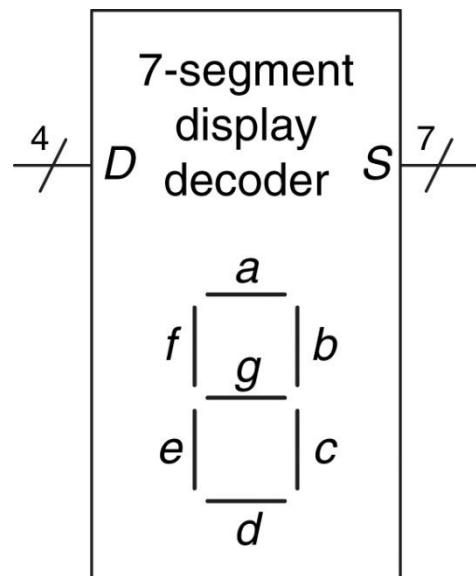


$$\bar{A}\bar{B}\bar{C} + \bar{A}\bar{B}C + A\bar{B}\bar{C} + A\bar{B}C = \bar{A}\bar{B}(\bar{C} + C) + A\bar{B}(\bar{C} + C) = \bar{A}\bar{B} + A\bar{B} = (\bar{A} + A)\bar{B} = \bar{B}$$

Regole di minimizzazione con K-maps

- Usare il minimo numero di cerchi necessari per ricoprire tutti gli 1
- Tutte le caselle sottese ad un cerchio devono contenere un 1
- Ogni cerchio ricopre un blocco di caselle che è una potenza di 2: 1,2,4,...
 - In un cerchio le variabili che non assumono valore costante devono assumere entrambi i valori 0 e 1
- Ogni cerchio deve essere il più largo possibile
- Un cerchio può «trasbordare» i bordi della K-map
 - La K-map ha una struttura *toroidale*
- Una casella può essere ricoperta da più cerchi se questo permette un numero inferiore di cerchi

Display a 7 segmenti



Display a 7 segmenti

$D_{3:0}$	S_a	S_b	S_c	S_d	S_e	S_f	S_g
0000	1	1	1	1	1	1	0
0001	0	1	1	0	0	0	0
0010	1	1	0	1	1	0	1
0011	1	1	1	1	0	0	1
0100	0	1	1	0	0	1	1
0101	1	0	1	1	0	1	1
0110	1	0	1	1	1	1	1
0111	1	1	1	0	0	0	0
1000	1	1	1	1	1	1	1
1001	1	1	1	0	0	1	1
others	0	/0	0	0	0	0	0

Display a 7 segmenti

Diagram illustrating the logic for a 7-segment display using two 4-to-16 decoders ($D_{1:0}$) and a multiplexer (S_a).

The inputs to the decoders are binary values:

	00	01	11	10
$D_{1:0}$	1	0	0	1
00	0	1	0	1
01	1	1	0	0
11	1	0	0	0
10	1	1	0	0

The outputs of the decoders are connected to the multiplexer (S_a). The multiplexer also receives a control signal $D_{3:2}$. The output of the multiplexer is the 7-segment display signal.

Diagram illustrating the logic for a 7-segment display using two 4-to-16 decoders ($D_{1:0}$) and a multiplexer (S_b).

The inputs to the decoders are binary values:

	00	01	11	10
$D_{1:0}$	1	1	0	1
00	1	0	0	1
01	1	0	0	1
11	1	1	0	0
10	1	0	0	0

The outputs of the decoders are connected to the multiplexer (S_b). The multiplexer also receives a control signal $D_{3:2}$. The output of the multiplexer is the 7-segment display signal.

Display a 7 segmenti

S_a	$D_{3:2}$	00	01	11	10
$D_{1:0}$		00	01	11	10
$\bar{D}_2 \bar{D}_1 \bar{D}_0$		1	0	0	1
$\bar{D}_3 D_2 D_0$		0	1	0	1
$D_3 \bar{D}_2 \bar{D}_1$		1	0	0	0
$\bar{D}_3 D_1$		1	1	0	0
$\bar{D}_3 \bar{D}_2$		1	1	0	0

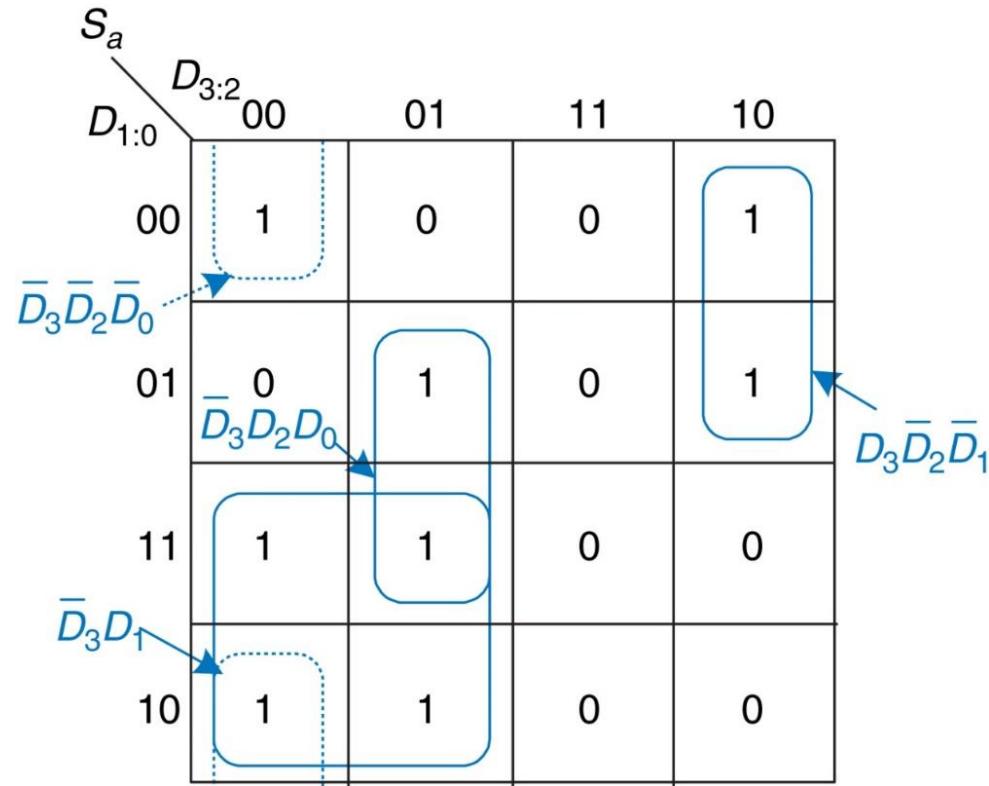
$$S_a = \bar{D}_3 D_1 + \bar{D}_3 D_2 D_0 + D_3 \bar{D}_2 \bar{D}_1 + \bar{D}_2 \bar{D}_1 \bar{D}_0$$

S_b	$D_{3:2}$	00	01	11	10
$D_{1:0}$		00	01	11	10
$\bar{D}_3 \bar{D}_1 \bar{D}_0$		1	1	0	1
$\bar{D}_3 D_1 D_0$		1	0	0	0
$\bar{D}_3 \bar{D}_2$		1	0	0	0

$$S_b = \bar{D}_3 \bar{D}_2 + \bar{D}_2 \bar{D}_1 + \bar{D}_3 D_1 D_0 + \bar{D}_3 \bar{D}_1 \bar{D}_0$$

Display a 7 segmenti

In generale, vi possono essere diverse forme minime:

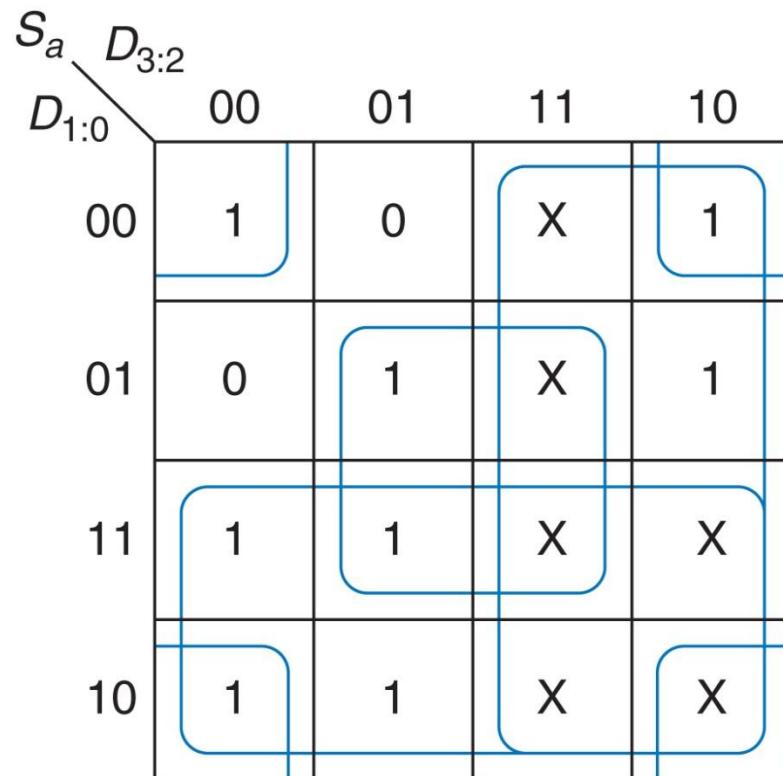


$$S_a = \bar{D}_3 D_1 + \bar{D}_3 D_2 D_0 + D_3 \bar{D}_2 \bar{D}_1 + \bar{D}_3 \bar{D}_2 \bar{D}_0$$

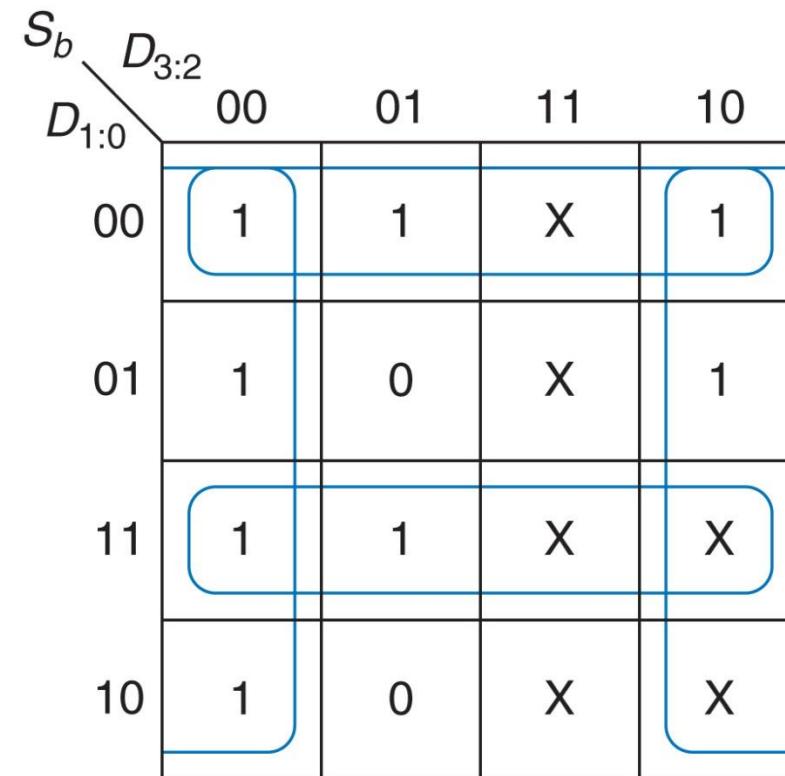
Don't care in output

- Abbiamo visto nei circuiti a priorità come si possano utilizzare degli input «don't care» per avere una rappresentazione della tabella di verità più succinta
- Valori «don't care» (X) si possono avere anche in output allorché per una data configurazione degli input il valore dell'output è irrilevante (non ci interessa se sia 0 o 1)
- Ad esempio nel display a sette segmenti per gli input «illegali» 10-15 l'output può essere di tipo X
- Poiché non ci interessa se un valore X corrisponde in realtà ad uno 0 o un 1 allora nella minimizzazione di una espressione possiamo scegliere che valore dargli come più opportuno.
- In particolare in una K-map sostituiremo i valori X con degli 1 se questo consente di avere un numero inferiore di cerchi o cerchi più larghi

K-map con valori «don't care»



$$S_a = D_3 + D_2D_0 + \bar{D}_2\bar{D}_0 + D_1$$



$$S_b = \bar{D}_2 + D_1D_0 + \bar{D}_1\bar{D}_0$$

Esercizi su K-maps

AB CD	00	01	11	10
00	1	0	0	1
01	0	1	1	1
11	0	1	1	1
10	1	0	0	1

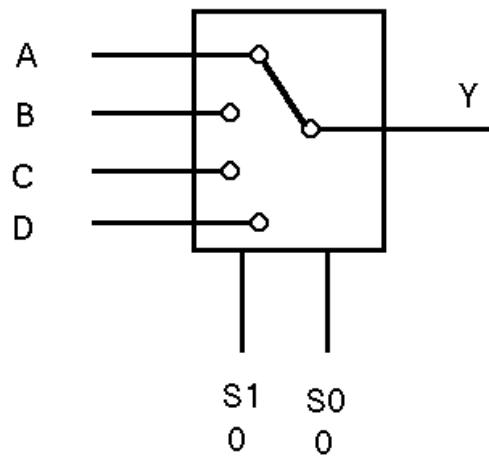
AB CD	00	01	11	10
00	1	0	0	1
01	1	1	0	1
11	1	1	1	1
10	1	0	0	1

AB CD	00	01	11	10
00	1	0	0	0
01	0	1	1	1
11	1	1	1	1
10	1	0	0	0

AB CD	00	01	11	10
00	0	1	1	0
01	1	0	0	1
11	1	0	0	1
10	0	1	1	0

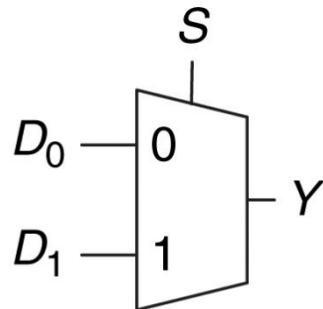
Multiplexer

- Un multiplexer è sostanzialmente un selettore di linea

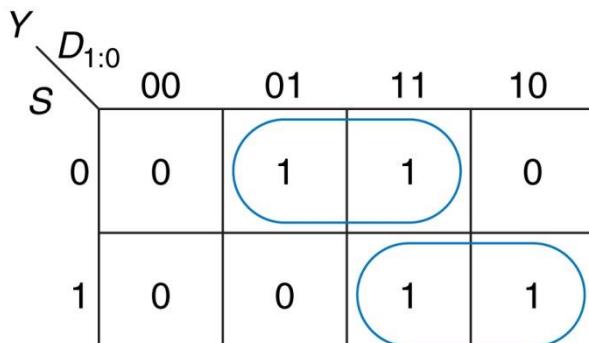


- In generale è costituito da N ingressi (dove N è una potenza di 2), 1 uscita, e $\log_2 N$ linee di selezione che indicano a quale ingresso deve corrispondere l'output

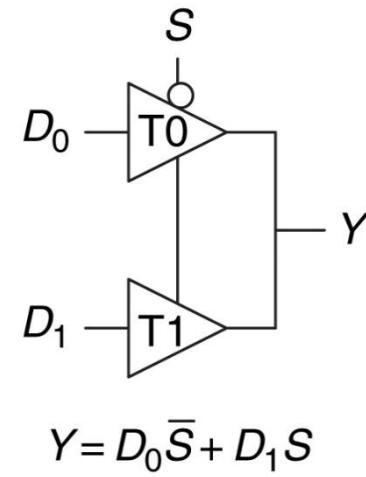
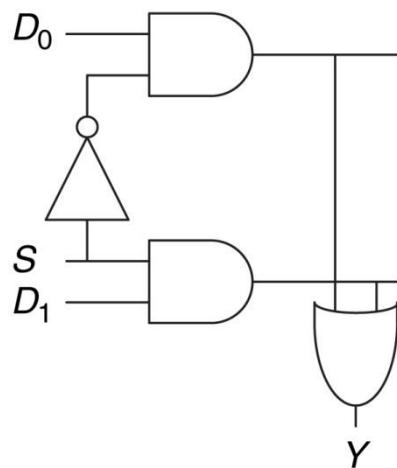
Multiplexer 2:1



S	D ₁	D ₀	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

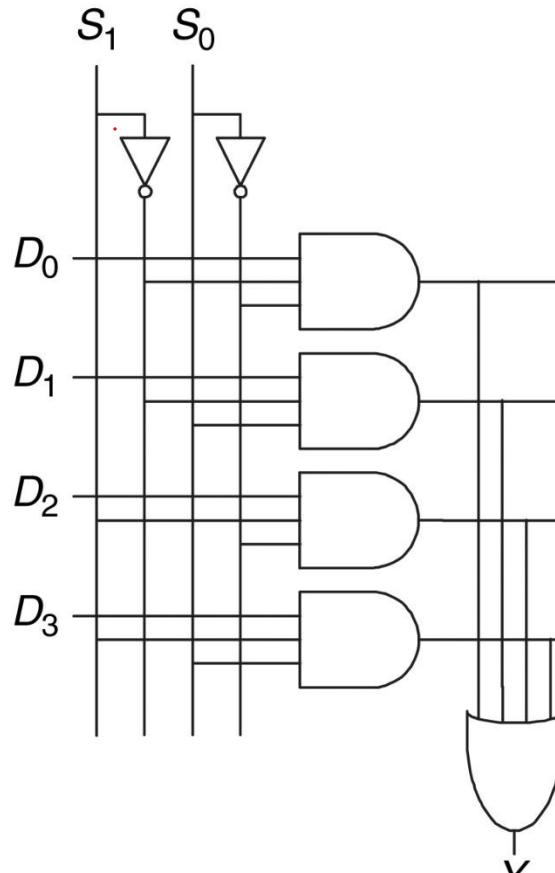
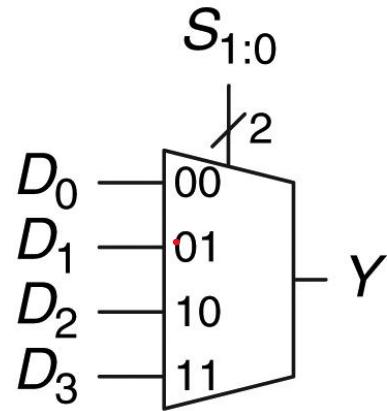


$$Y = D_0 \bar{S} + D_1 S$$

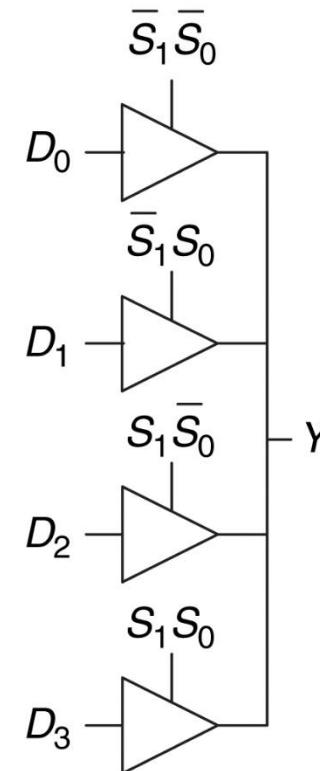


$$Y = D_0 \bar{S} + D_1 S$$

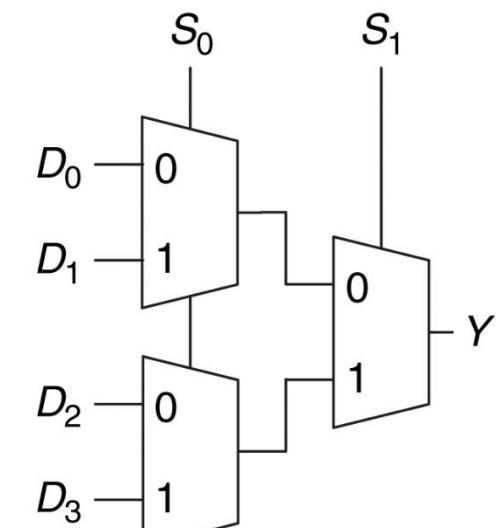
Multiplexer 4:1



(a)



(b)



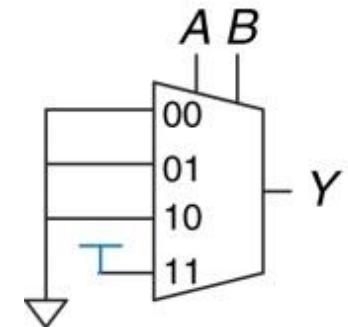
(c)

Sintetizzare funzioni booleane con mux

- I multiplexer possono essere usati anche per sintetizzare delle funzioni booleane
- Sintetizzare una funzione di m variabili con un mux a 2^m linee è molto semplice: le variabili saranno linee di selezione. Data una certa configurazione delle variabili, la linea di ingresso corrispondente sarà posta al valore della funzione in quella configurazione
- Di fatto le linee di ingresso riproducono la tabella di verità della funzione

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

$Y = AB$

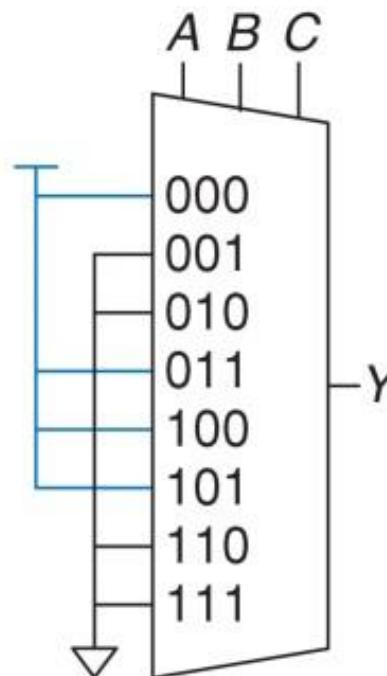


Sintetizzare funzioni booleane con mux

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

$$Y = A\bar{B} + \bar{B}\bar{C} + \bar{A}BC$$

(a)

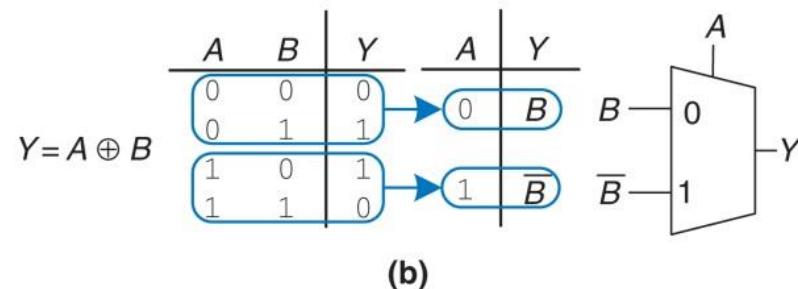
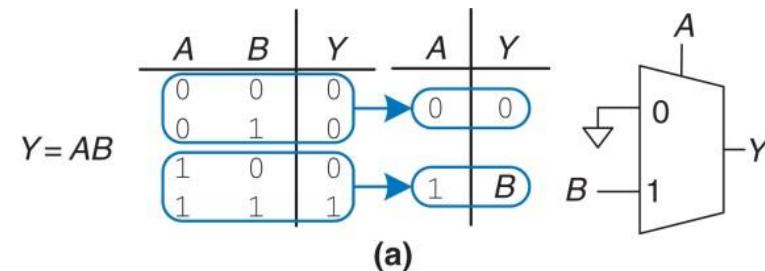


(b)

$$Y = \Sigma(0,3,4,5)$$

Sintetizzare funzioni booleane con mux

- E' possibile utilizzare un mux con 2^{m-1} ingressi per sintetizzare un funzione ad m variabili
- Le prime m-1 variabili saranno linee di selezione, mentre le linee di ingresso possono essere poste a 0,1, oppure all'ultima variabile (positiva o negata)



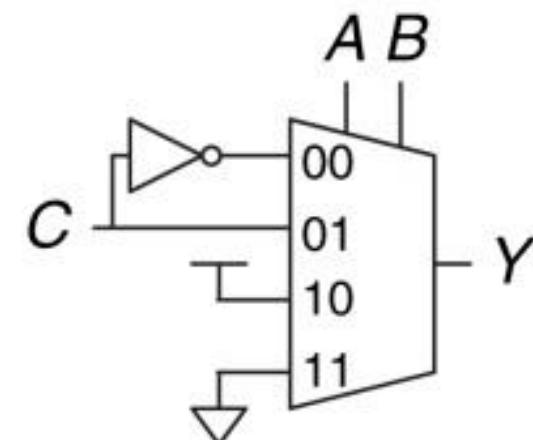
Sintetizzare funzioni booleane con mux

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

(a)

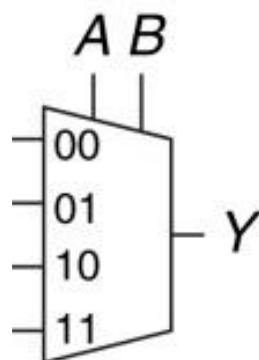
(b)

(c)



Sintetizzare funzioni booleane con mux

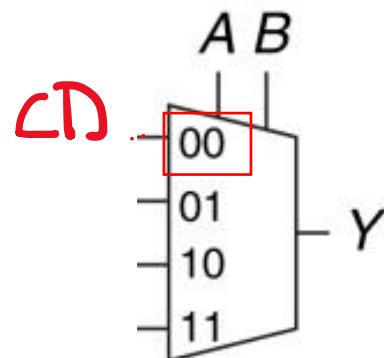
Supponiamo di avere mux 4:1 e di voler sintetizzare una funzione di 4 variabili



A B C D	Y
0000	0
0001	0
0010	0
0011	1
0100	1
0101	1
0110	0
0111	0
1000	1
1001	1
1010	0
1011	0
1100	1
1101	1
1110	1
1111	1

Sintetizzare funzioni booleane con mux

Supponiamo di avere mux 4:1 e di voler sintetizzare una funzione di 4 variabili

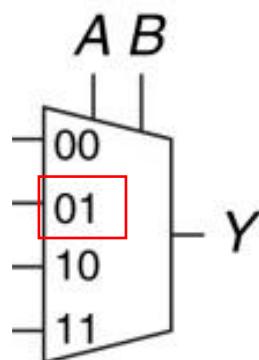


A B C D	Y
0 0 0 0	0
0 0 0 1	0
0 0 1 0	0
0 0 1 1	1
0 1 0 0	1
0 1 0 1	1
0 1 1 0	0
0 1 1 1	0
1 0 0 0	1
1 0 0 1	1
1 0 1 0	0
1 0 1 1	0
1 1 0 0	1
1 1 0 1	1
1 1 1 0	1
1 1 1 1	1

CD

Sintetizzare funzioni booleane con mux

Supponiamo di avere mux 4:1 e di voler sintetizzare una funzione di 4 variabili

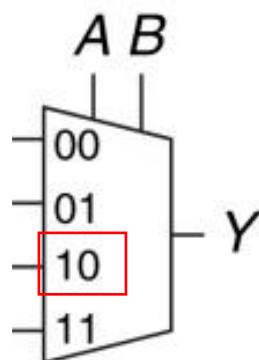


A B C D	Y
0 0 0 0	0
0 0 0 1	0
0 0 1 0	0
0 0 1 1	1
0 1 0 0	1
0 1 0 1	1
0 1 1 0	0
0 1 1 1	0
1 0 0 0	1
1 0 0 1	1
1 0 1 0	0
1 0 1 1	0
1 1 0 0	1
1 1 0 1	1
1 1 1 0	1
1 1 1 1	1

\bar{C}

Sintetizzare funzioni booleane con mux

Supponiamo di avere mux 4:1 e di voler sintetizzare una funzione di 4 variabili

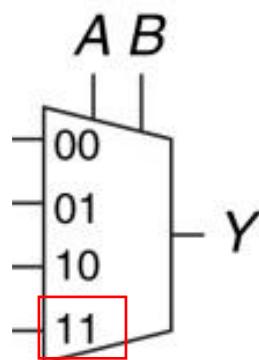


A B C D	Y
0 0 0 0	0
0 0 0 1	0
0 0 1 0	0
0 0 1 1	1
0 1 0 0	1
0 1 0 1	1
0 1 1 0	0
0 1 1 1	0
1 0 0 0	0
1 0 0 1	1
1 0 1 0	1
1 0 1 1	0
1 1 0 0	1
1 1 0 1	1
1 1 1 0	1
1 1 1 1	1

$$C \oplus D$$

Sintetizzare funzioni booleane con mux

Supponiamo di avere mux 4:1 e di voler sintetizzare una funzione di 4 variabili



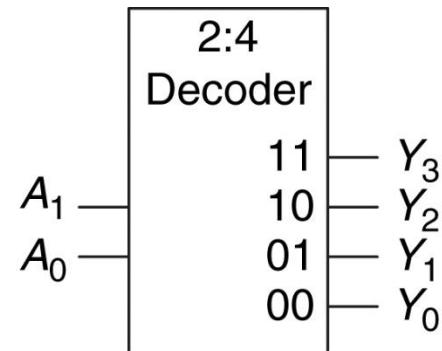
A B C D	Y
0000	0
0001	0
0010	0
0011	1
0100	1
0101	1
0110	0
0111	0
1000	1
1001	1
1010	0
1011	0
1100	1
1101	1
1110	1
1111	1

Esercizi

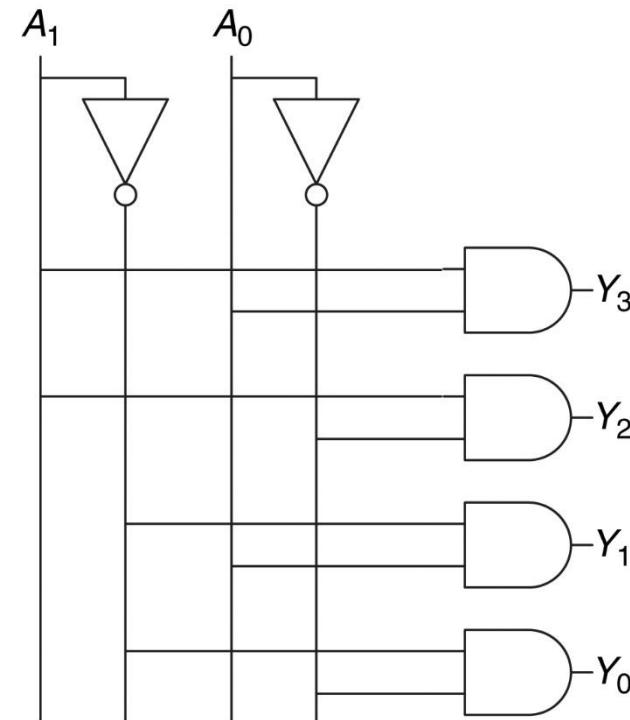
- Sintetizzare con multiplexer 4:1 la funzione di 4 variabili $\Sigma(0,5,9,10,13)$
- Sintetizzare con multiplexer 4:1 la funzione di 4 variabili $\Sigma(1,2,3,12,13)$
- Sintetizzare con multiplexer 4:1 la funzione di 4 variabili $\Sigma(0,1,2,3,5,15)$
- Sintetizzare con multiplexer 4:1 la funzione di 4 variabili $\Sigma(1,2,6,7,8,13)$

Decoder

- Un decoder ha N linee di ingresso e 2^N linee di uscita
- se m è numero rappresentato dagli input allora solo l'm-esima linea di uscita è pari a 1 mentre tutte le altre sono a 0

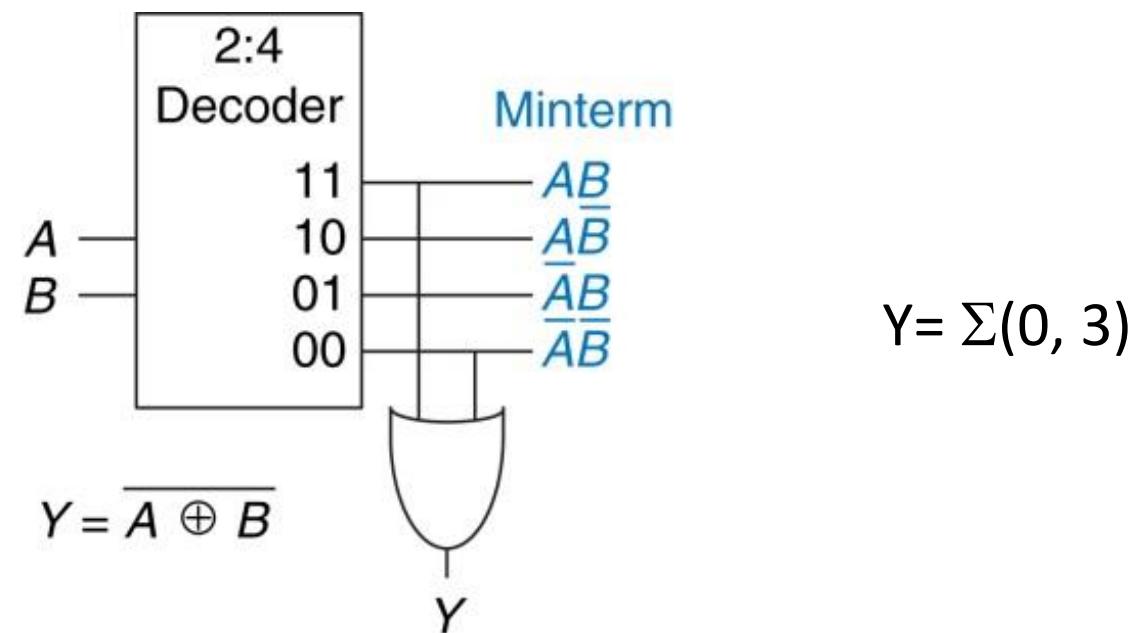


A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0



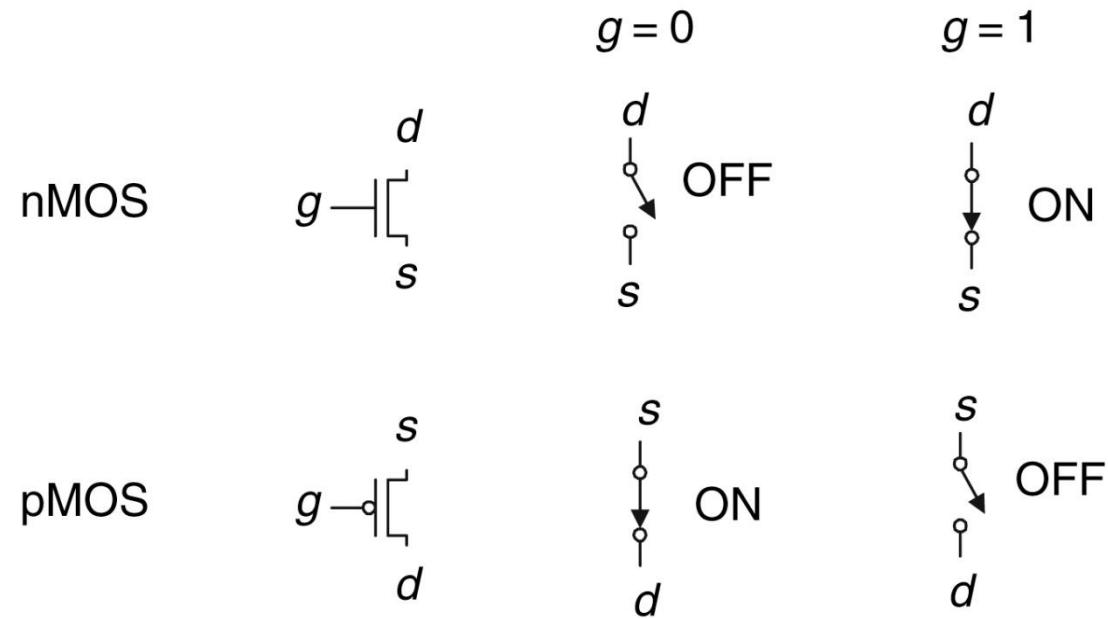
Sintetizzare funzioni booleane con decoder

- Anche i decoder possono essere usati per sintetizzare funzioni booleane
- Basta mettere in OR tutte e solo le linee di uscita che occorrono nella sigma-expresione della funzione da sintetizzare



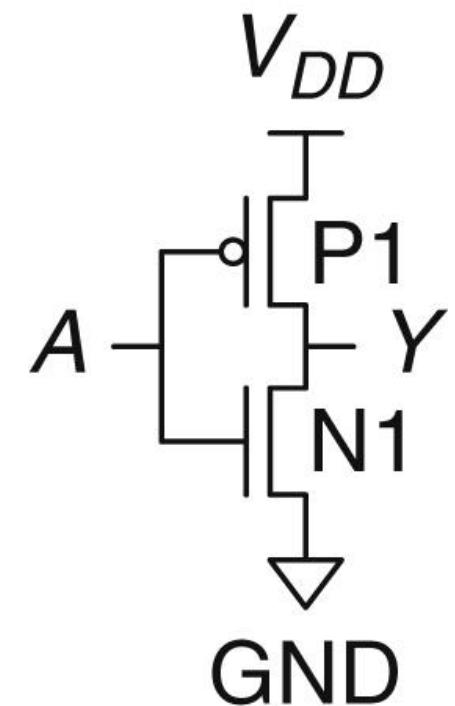
Come è fatta una porta logica?

- Oggi giorno le porte logiche sono realizzate mediante *interruttori* elettronici controllati detti MOSFET.
- Vi sono due tipi di switch MOSFET



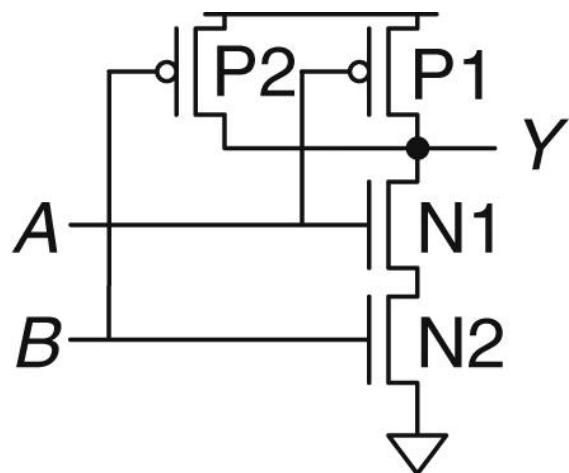
NOT al microscopio

- $A=GND \rightarrow P1$ chiuso e $N1$ aperto $\rightarrow Y=V_{DD}$
- $A=V_{DD} \rightarrow P1$ aperto e $N1$ chiuso $\rightarrow Y=GND$



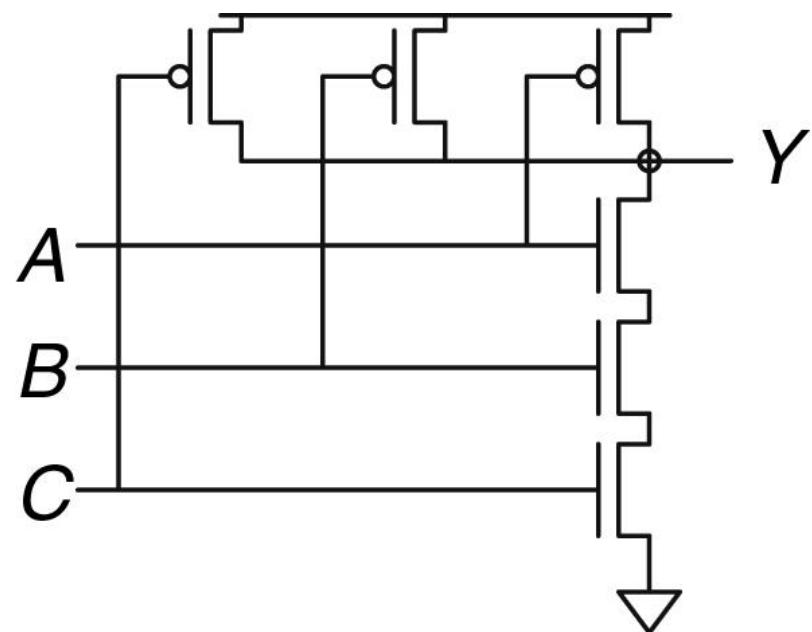
NAND al microscopio

- $A=GND$ $B= GND$ \rightarrow P1, P2 chiusi e N1,N2 aperti $\rightarrow Y= V_{DD}$
- $A=GND$ $B= V_{DD}$ \rightarrow P1, N2 chiusi e N1,P2 aperti $\rightarrow Y= V_{DD}$
- $A= V_{DD}$ $B= GND$ \rightarrow P2, N1 chiusi e N2,P1 aperti $\rightarrow Y= V_{DD}$
- $A= V_{DD}$ $B= V_{DD}$ \rightarrow N1, N2 chiusi e P1,P2 aperti $\rightarrow Y= GND$



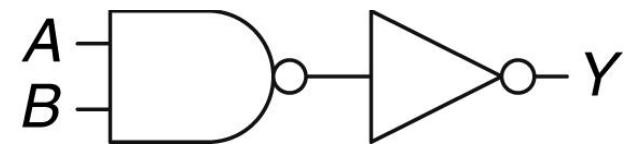
NAND al microscopio

- La logica a più ingressi non cambia.



AND al microscopio

- Nella tecnologia CMOS, la porta AND è implementata mettendo in serie una NAND con una NOT.

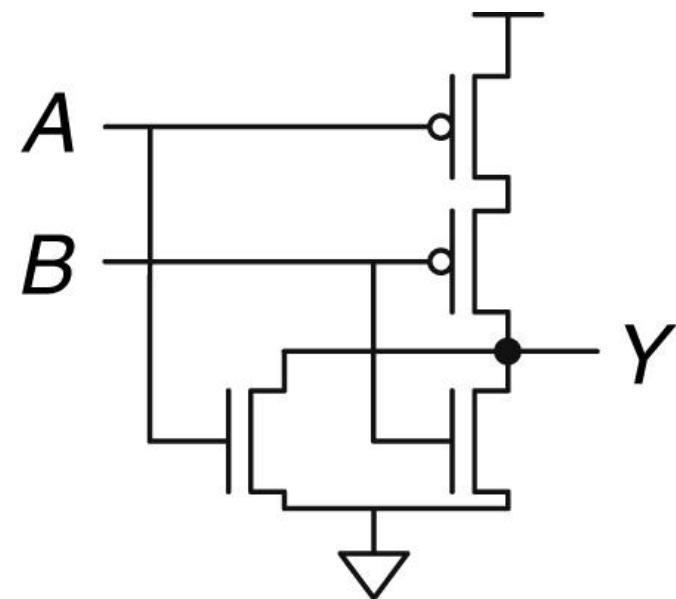


- In termini logici quindi quello che i circuiti CMOS realizzano è l'espressione

$$\overline{\overline{A} \cdot \overline{B}}$$

Esercizi

- Verificare che il seguente circuito realizza una porta NOR



Circuiti sequenziali

Logica sequenziale

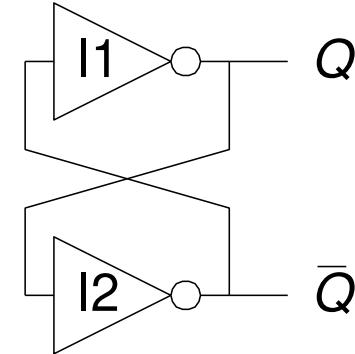
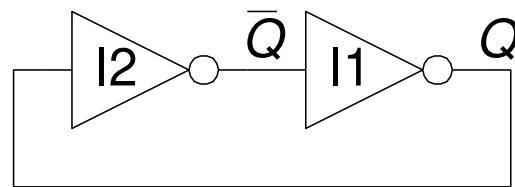
- Nei sistemi sequenziali l'output dipende sia dal valore corrente sia da valori precedenti dell'input. In tal senso si dice che il sistema ha *memoria*
- *Stato interno*: rappresenta l'informazione che mantiene la *storia* di un circuito sequenziale ed è necessaria per prevedere il suo comportamento futuro
- Come vedremo lo stato di un sistema sarà memorizzato in componenti come i latches e flip-flop
- Un circuito sequenziale (*sincrono*) avrà una topologia ben precisa:
 - logica combinatoria: definisce l'evoluzione del sistema
 - Banchi di flip-flop: servono a memorizzare gli stati del sistema
- Un aspetto peculiare dei sistemi sequenziali è quello della retroazione (*feedback*), ovvero il segnali di output vengono riportati in input

State elements

- Lo stato di un circuito influenzerà l'evoluzione del sistema
- Gli *state elements* sono tutte quelle componenti circitali che vengono adoperate per memorizzare lo stato di un circuito
 - Circuiti bistabili
 - SR Latch
 - D Latch
 - D Flip-flop

Circuito bistabile

- *building block* per altri state elements
- Two outputs: Q , \bar{Q}
- No inputs

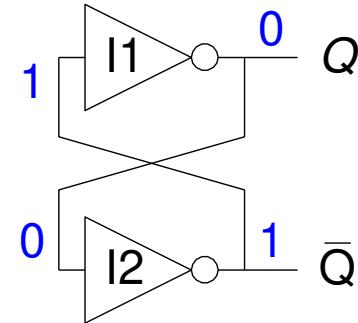


Analisi del circuito bistabile

- Considera i due possibili casi:

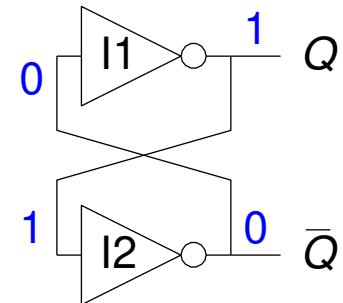
$Q = 0$:

allora $Q = 0, \bar{Q} = 1$ (consistente)



$Q = 1$:

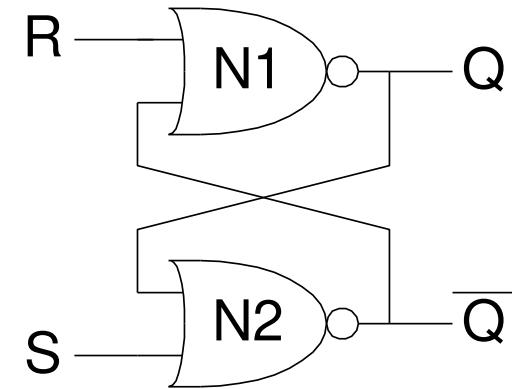
allora $Q = 1, \bar{Q} = 0$ (consistente)



- Memorizza 1 bit nella variabile di stato Q (or \bar{Q})
- Ma non ci sono input per controllare questo stato!

SR (Set/Reset) Latch

- SR Latch
- Consideriamo i 4 possibili stati:
 - $S=1, R=0$
 - $S=0, R=1$
 - $S=0, R=0$
 - $S=1, R=1$



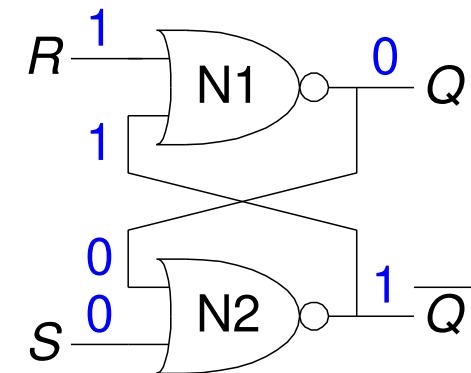
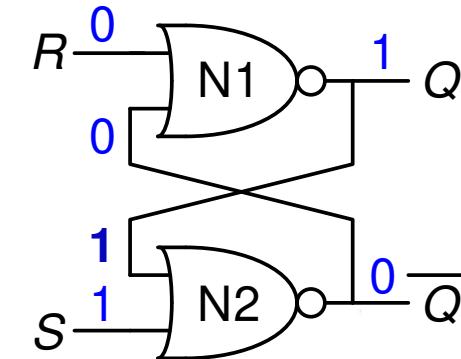
Analisi di un SR Latch

$S = 1, R = 0:$

allora $Q = 1$ e $\bar{Q} = 0$

$S = 0, R = 1:$

allora $Q = 0$ e $\bar{Q} = 1$



Analisi di un SR Latch

$S = 1, R = 0:$

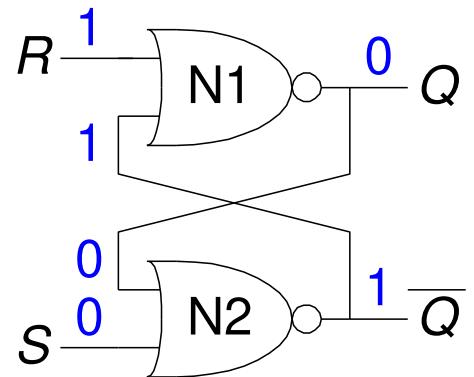
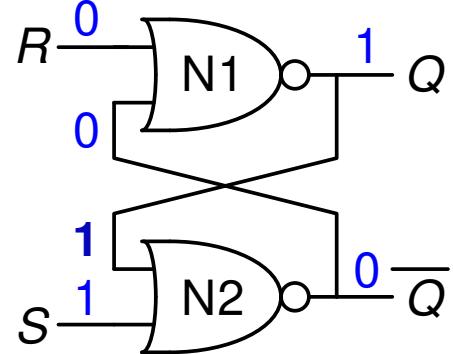
allora $Q = 1$ e $\bar{Q} = 0$

operazione Set

$S = 0, R = 1:$

allora $Q = 0$ e $\bar{Q} = 1$

operazione Reset

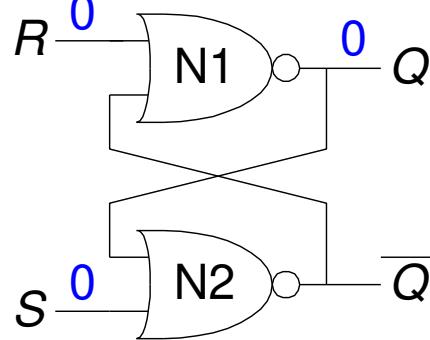


Analisi di un SR Latch

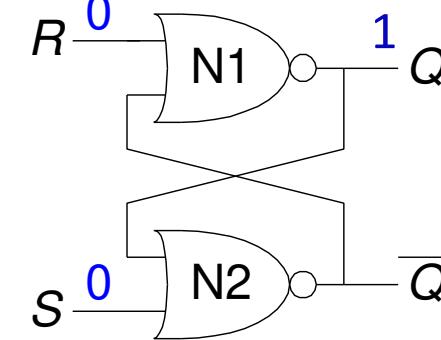
$S = 0, R = 0:$

allora $Q = Q_{prev}$

$Q_{prev} = 0$

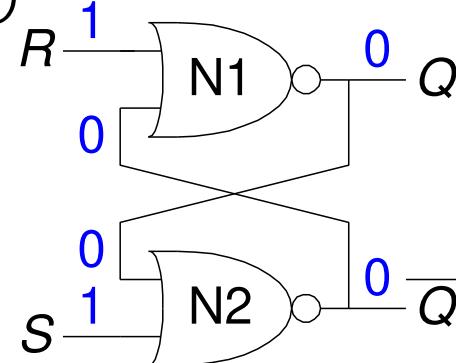


$Q_{prev} = 1$



$S = 1, R = 1:$

allora $Q = 0, \bar{Q} = 0$

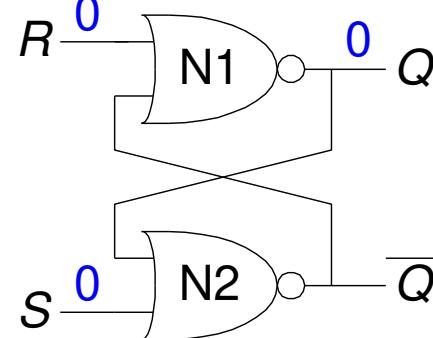


Analisi di un SR Latch

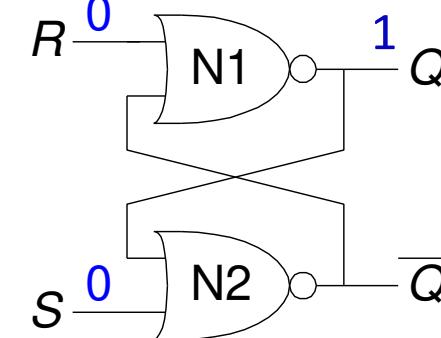
$S = 0, R = 0:$

allora $Q = Q_{prev}$
Memoria

$Q_{prev} = 0$



$Q_{prev} = 1$

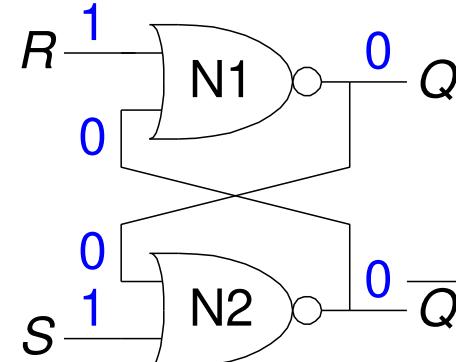


$S = 1, R = 1:$

allora $Q = 0, \bar{Q} = 0$

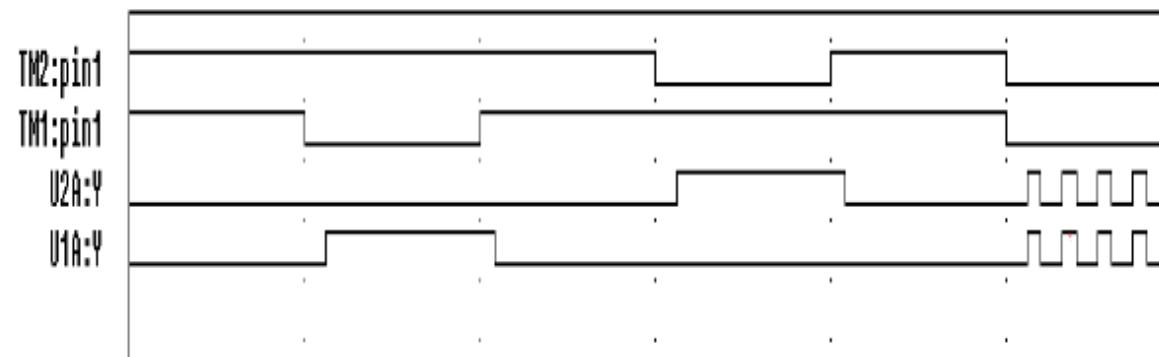
Stato non valido

$Q \neq \text{NOT } \bar{Q}$



Analisi di un SR Latch

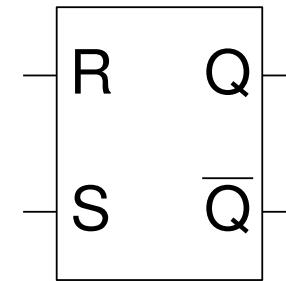
- Se dalla condizione $S=R=1$ si passa alla condizione $S=R=0$ allora
 - Se i tempi di propagazione sono uguali allora il circuito va in oscillazione
 - Nell'ipotesi, più realistica che le porte abbiano ritardi anche lievemente differenti, il circuito si mette in uno dei due stati possibili. Anche in questo caso, però, lo stato finale non è predicibile



Simbolo per un SR Latch

- SR sta per Set/Reset
 - Memorizza un bit (Q)
- **Set:** Pone l'output a 1
($S = 1, R = 0, Q = 1$)
- **Reset:** Pone l'output a 0
($S = 0, R = 1, Q = 0$)
- **Memoria:** mantiene memoria dell'output
($S = 0, R = 0, Q = Q_{\text{prev}}$)

SR Latch
Symbol

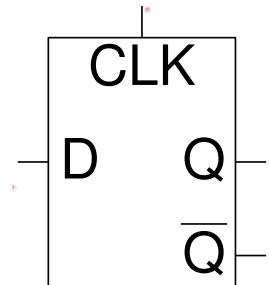


Occorre evitare lo stato non valido $S = R = 1$

D Latch

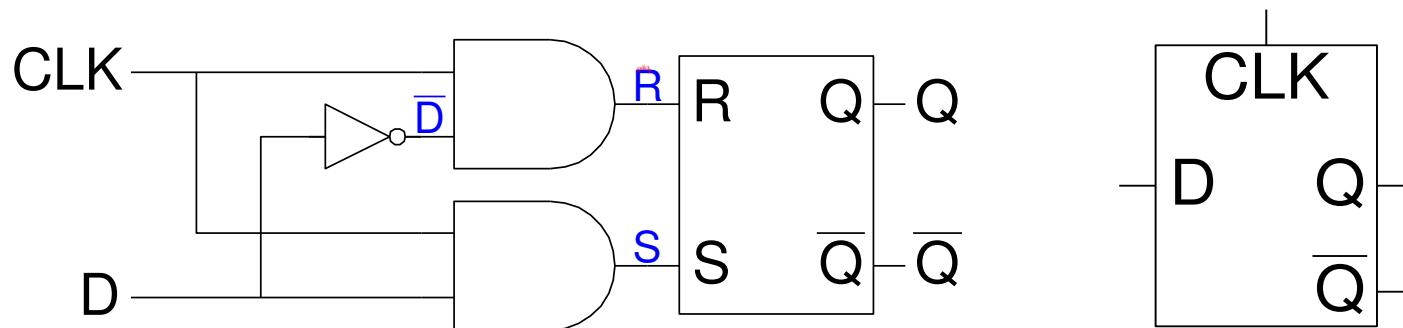
- 2 input: CLK , D
- CLK : controlla *quando* l'output cambia
- D (data input): controlla *in che cosa* l'output cambia
- Se $CLK = 1$,
 D passa fino a Q (*transparente*)
- Se $CLK = 0$,
 Q mantiene il suo valore precedente (*opaco*)

D Latch
Symbol



Evita lo stato non valido in cui $Q \neq \text{NOT } Q$

D Latch

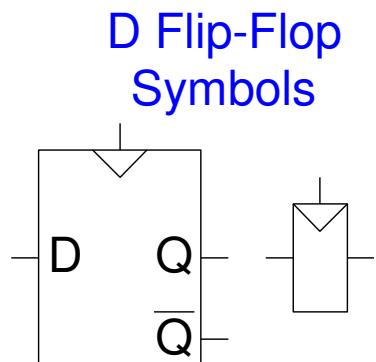


CLK	D	\overline{D}	S	R	Q	\overline{Q}
0	X	\overline{X}	0	0	Q_{prev}	\overline{Q}_{prev}
1	0	1	0	1	0	1
1	1	0	1	0	1	0

D Flip-Flop

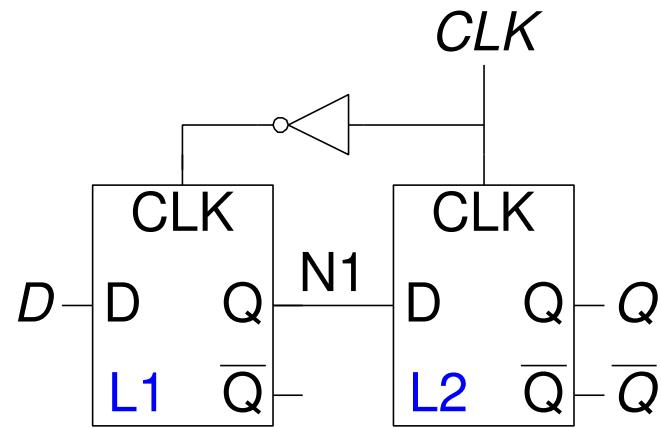
- Inputs: CLK, D
- Funzione:
 - Quando CLK passa da 0 a 1, D passa fino a Q
 - Altrimenti, Q mantiene il suo valore precedente
- Q cambia solo durante la transizione di CLK da 0 a 1

Queste tipologie di componenti sono dette edge-triggered perché sono pilotate non da un valore ma da una transizione (di CLK)

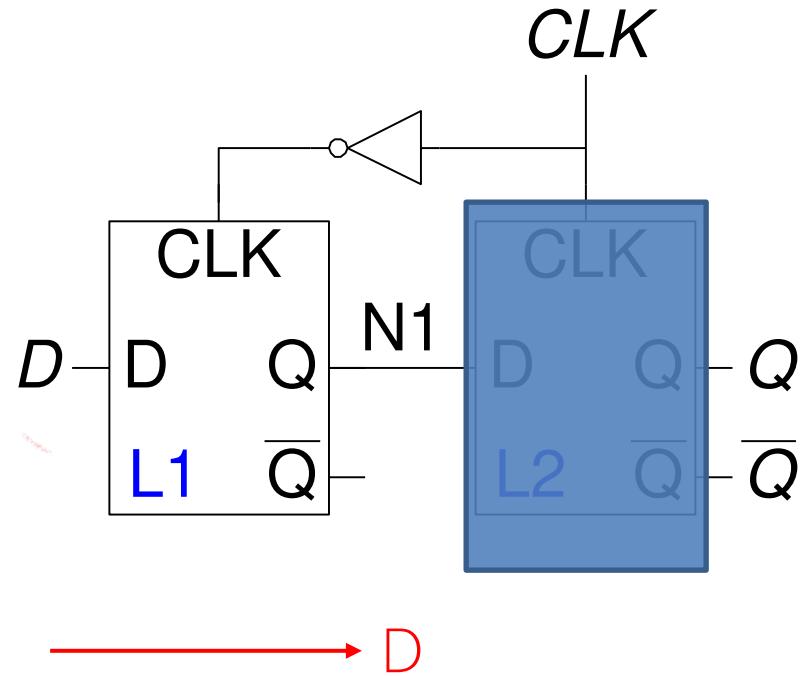


D Flip-Flop

- 2 D latch (L1 e L2) controllati da clock complementari
- Quando $CLK=0$
 - L1 è trasparente
 - L2 è opaco
 - D passa fino a N1
- Quando $CLK=1$
 - L2 è trasparente
 - L1 è opaco
 - N1 passa fino a Q
- Quindi, D passa fino a Q *sulla transizione di CLK da 0 a 1*
- Ulteriori variazioni di D quando $CLK=1$ (risp. $CLK=0$) non passano a Q perché L1 (risp. L2) è opaco



D Flip-Flop

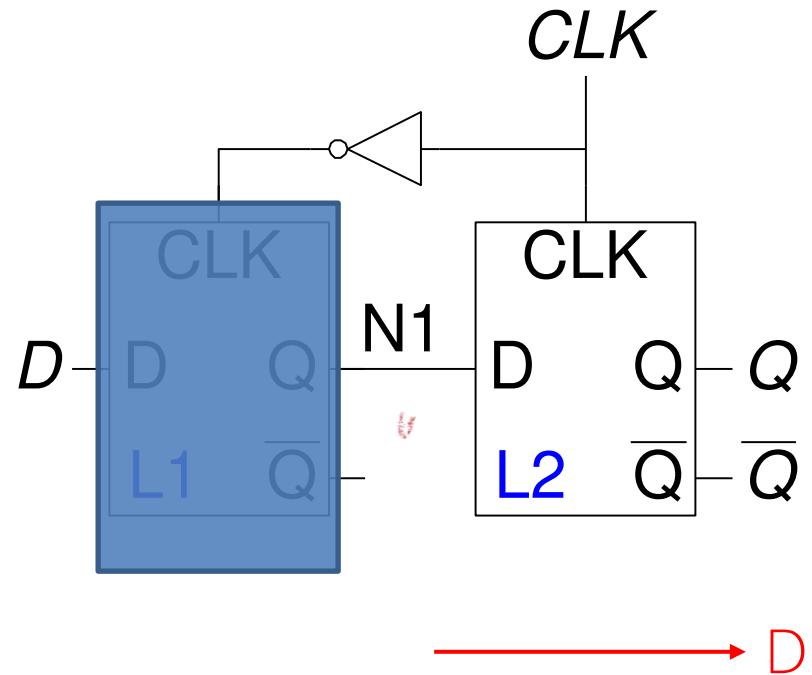


$$CLK = 0$$

$$N1 = D$$

$$Q = Q_{\text{prev}}$$

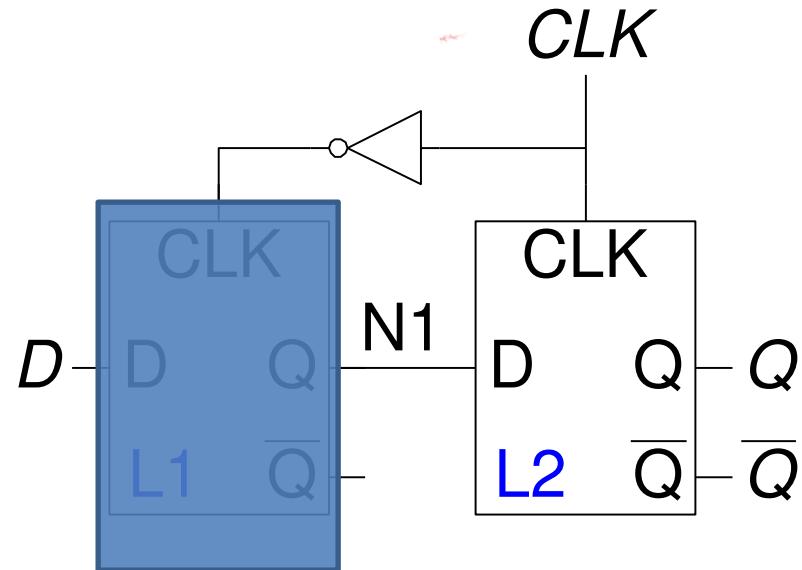
D Flip-Flop



CLK $0 \rightarrow 1$

$N1 = N1_{\text{prev}}$
 $Q = N1$ (ovvero *D*)

D Flip-Flop

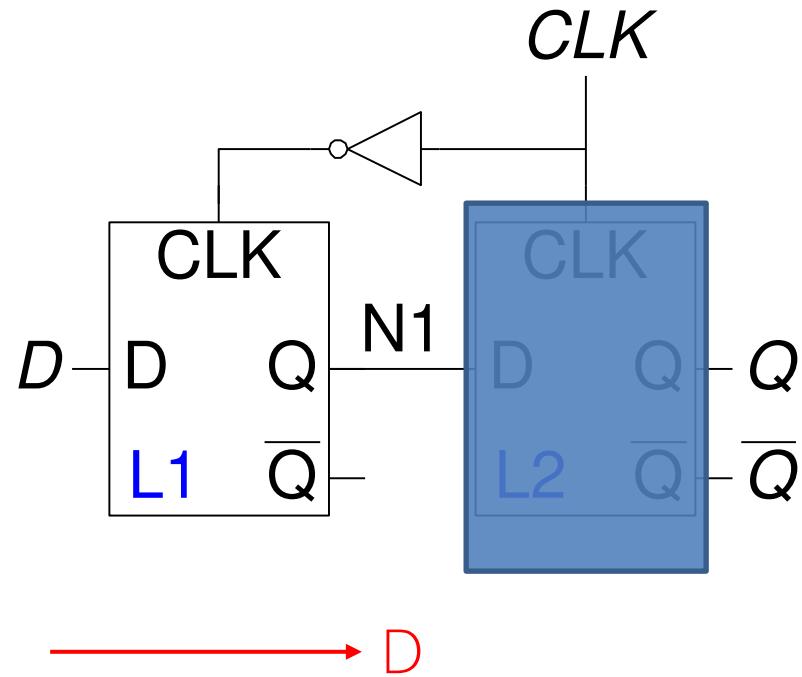


D

$CLK = 1$

$N1 = N1_{\text{prev}}$
 $Q = N1$

D Flip-Flop

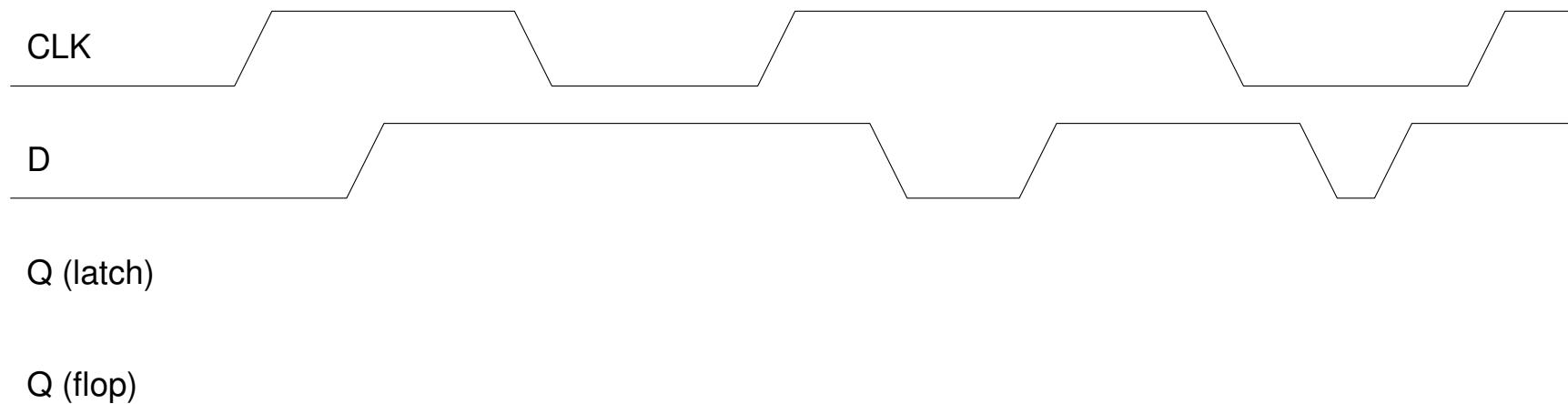
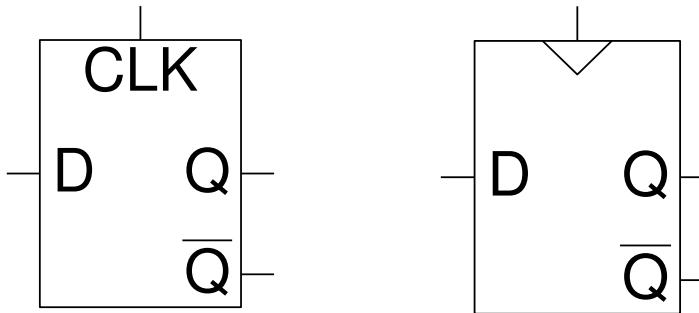


CLK 1 → 0

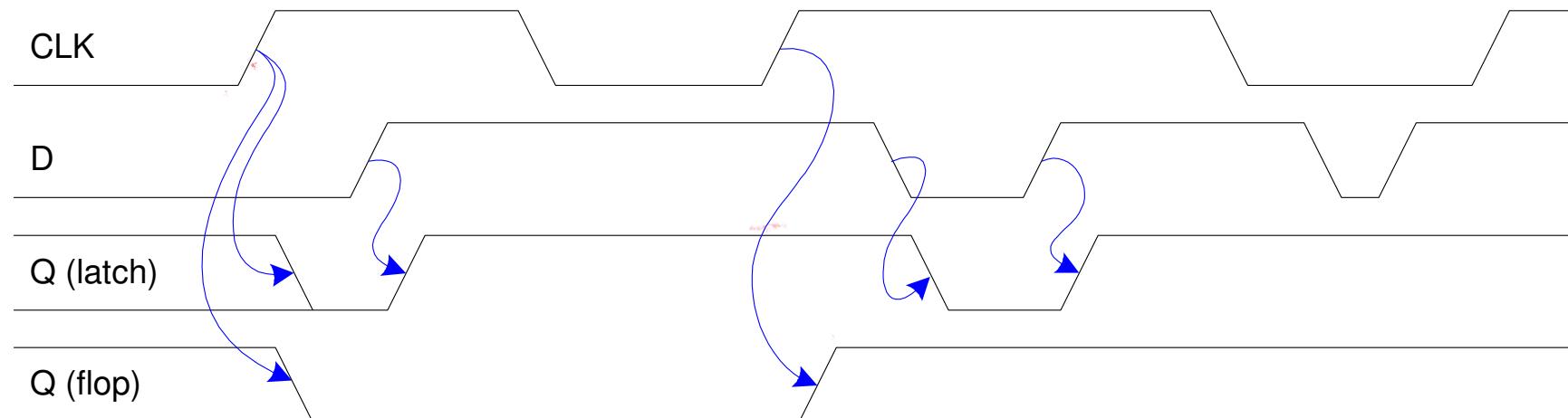
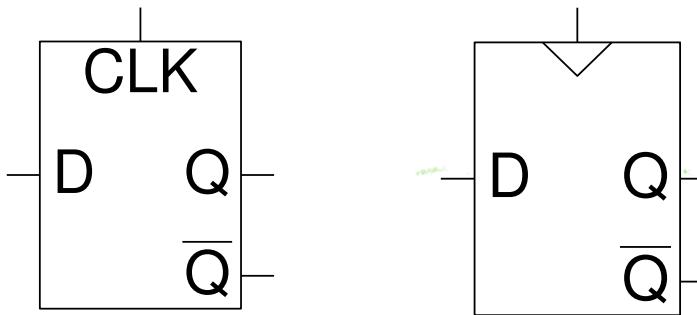
N1 = D

Q = Q_{prev}

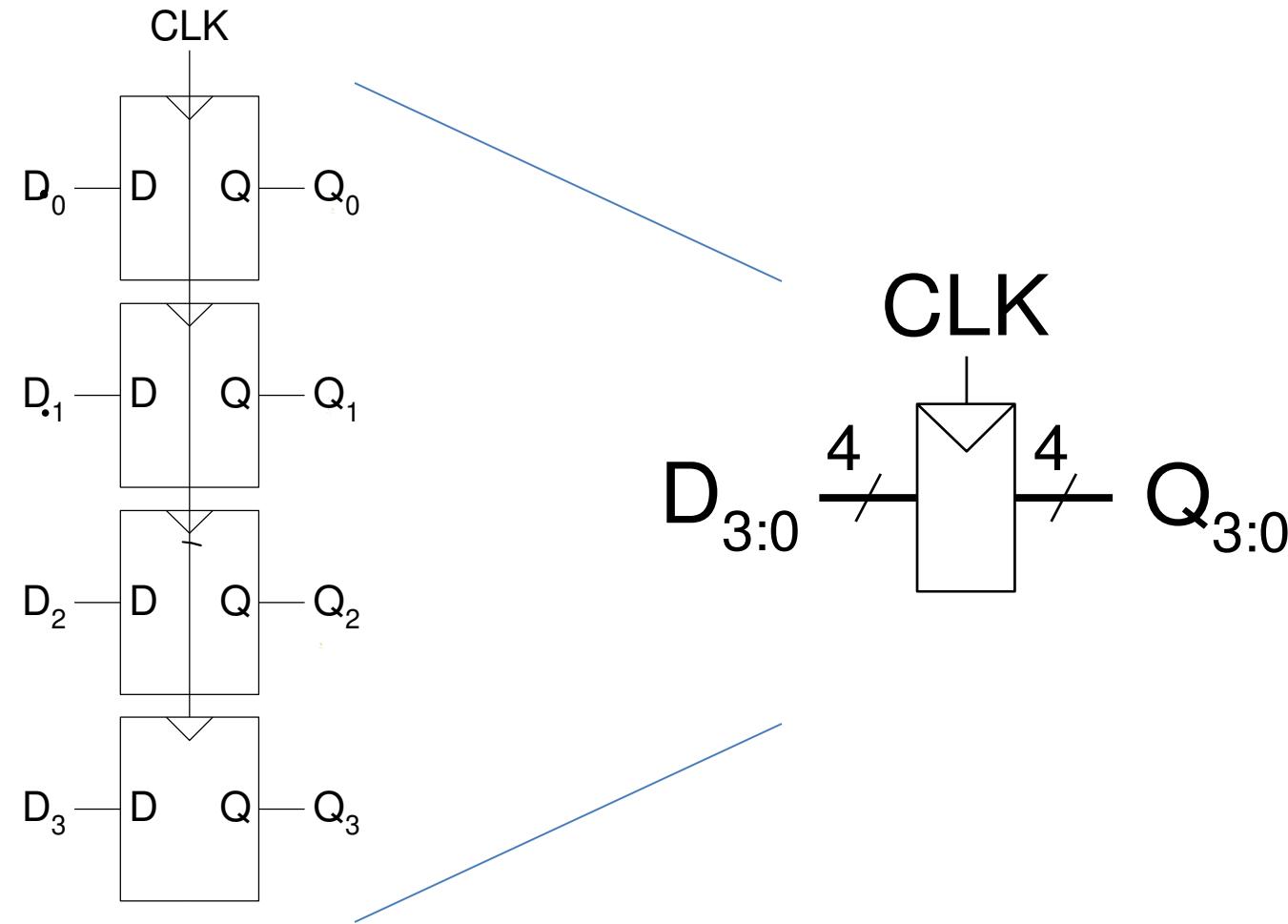
D Latch vs. D Flip-Flop



D Latch vs. D Flip-Flop

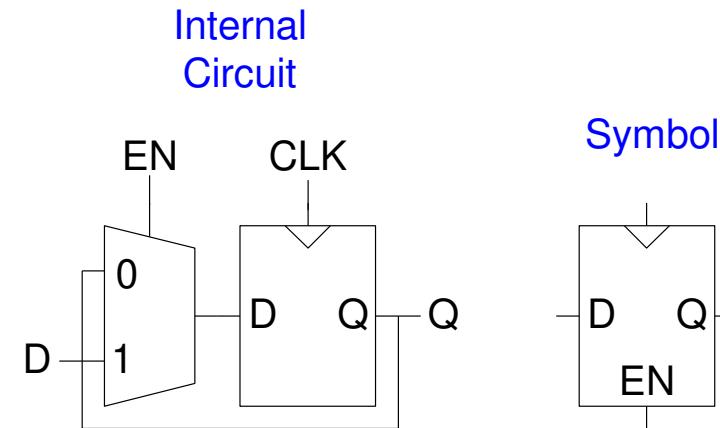


Registri: Multi-bit Flip-Flop



Flip-Flops “enabled”

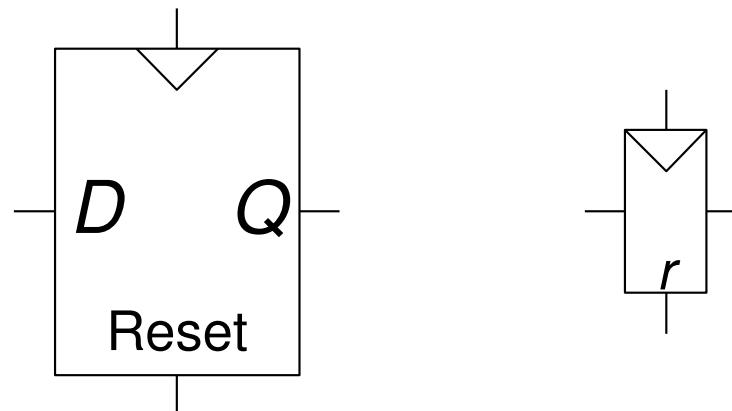
- *Inputs: CLK, D, EN*
- L'input enable (*EN*) stabilisce quando un nuovo valore di *D* è memorizzato
- *EN= 1*: *D* passa fino a *Q* (clock: 0 → 1)
- *EN= 0*: il flip-flop mantiene il suo stato precedente



Flip-Flops “resetabili”

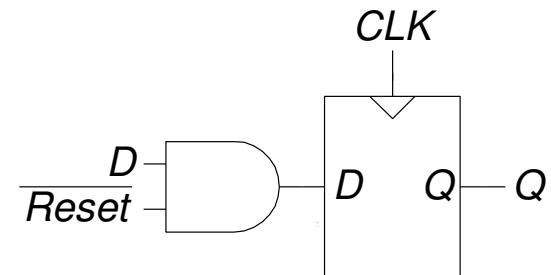
- *Inputs: CLK, D, Reset*
- *Reset = 1: Q = 0*
- *Reset = 0: il flip-flop si comporta “normalmente” come un D flip-flop*

Symbols



Flip-Flops “resettabili”

- Vi sono due tipi di flip-flop resettabili:
 - **Sincroni:** il reset è pilotato dal clock
 - **Asincroni:** il reset avviene non appena $Reset = 1$
- Flip-flop sincroni:

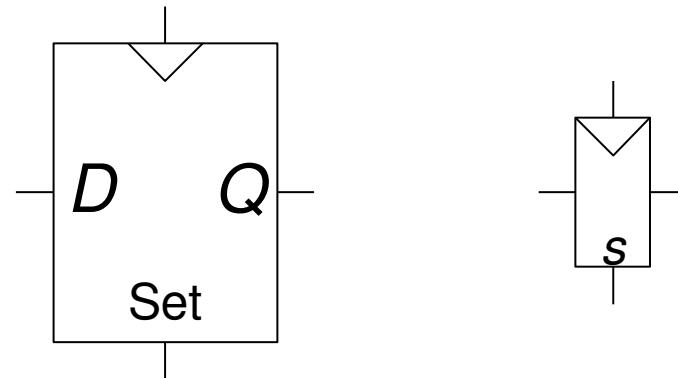


- Per i flip-flop asincroni occorre modificare il circuito interno del flip-flop

Flip-Flops “settabili”

- *Inputs: CLK, D, Set*
- *Set = 1: Q=1*
- *Set = 0: il flip-flop si comporta “normalmente” come un D flip-flop*

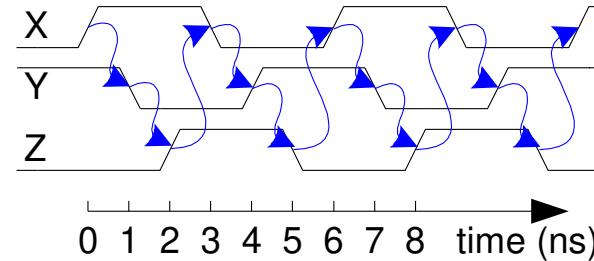
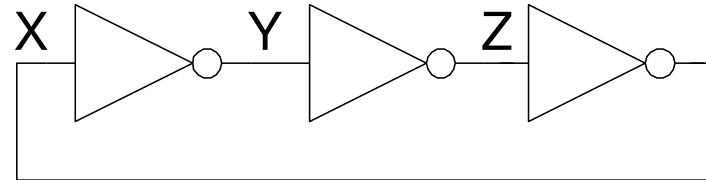
Symbols



Esercizi

- Esercizi 3.1, 3.3, 3.5, 3.7, 3.8, 3.13, 3.15

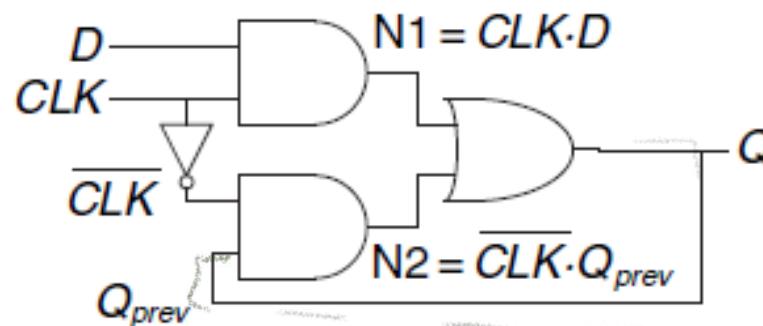
Criticità nella logica sequenziale



- Questi circuiti vengono detti “astabili” poiché hanno un comportamento oscillante
- Il periodo di oscillazione dipende dai ritardi degli inverter
- Idealmente è di 6 ns tuttavia può variare a causa di diversi fattori
 - differenze nella manifattura
 - temperatura
- Circuito *asincrono*: l'output è retroazionato in maniera diretta

Criticità nella logica sequenziale

- In casi più complessi che comprendono l'uso di più porte AND, NOT, OR il comportamento di una rete asincrona può dipendere fortemente dai ritardi accumulati sui singoli cammini

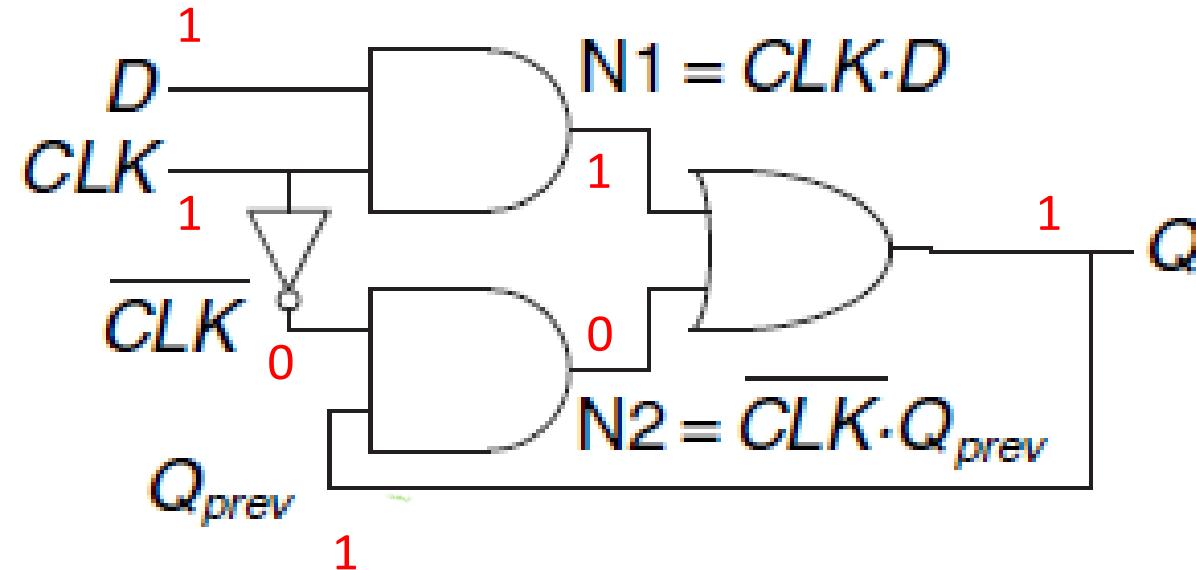


$$Q = CLK \cdot D + \overline{CLK} \cdot Q_{prev}$$

CLK	D	Q_{prev}	Q
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

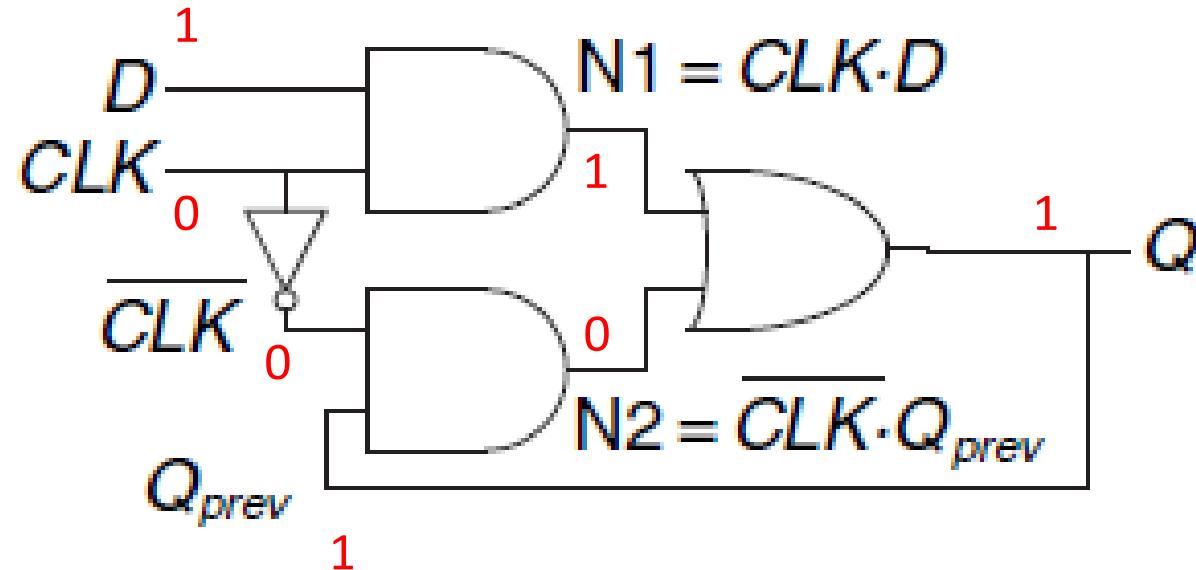
Criticità nella logica sequenziale

$$D=1, \text{CLK}=1 \rightarrow Q=1$$



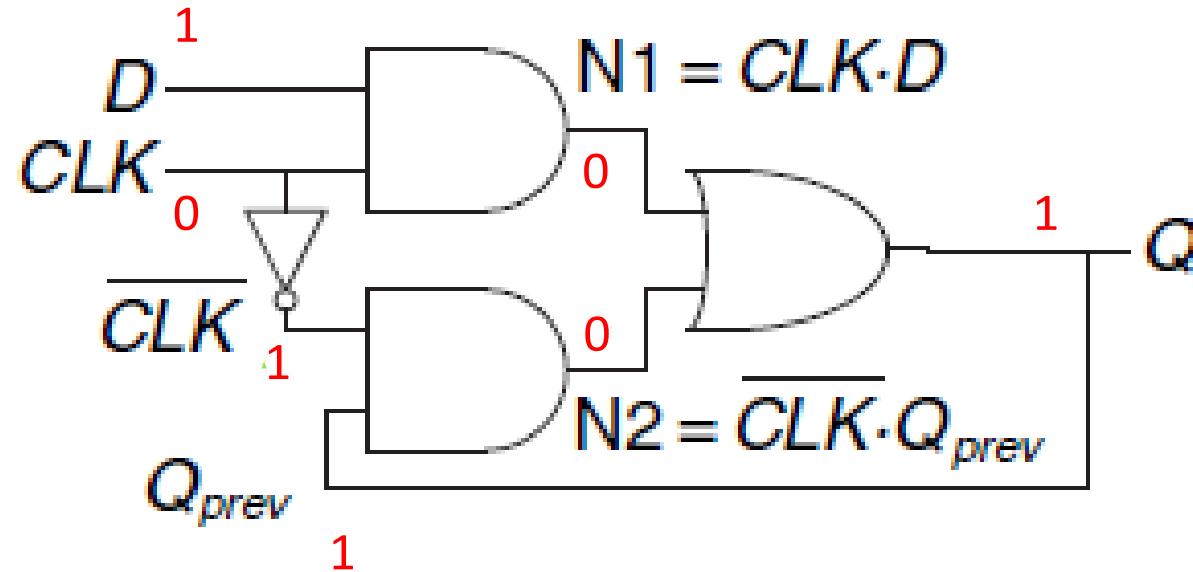
Criticità nella logica sequenziale

t_0 CLK 1 → 0



Criticità nella logica sequenziale

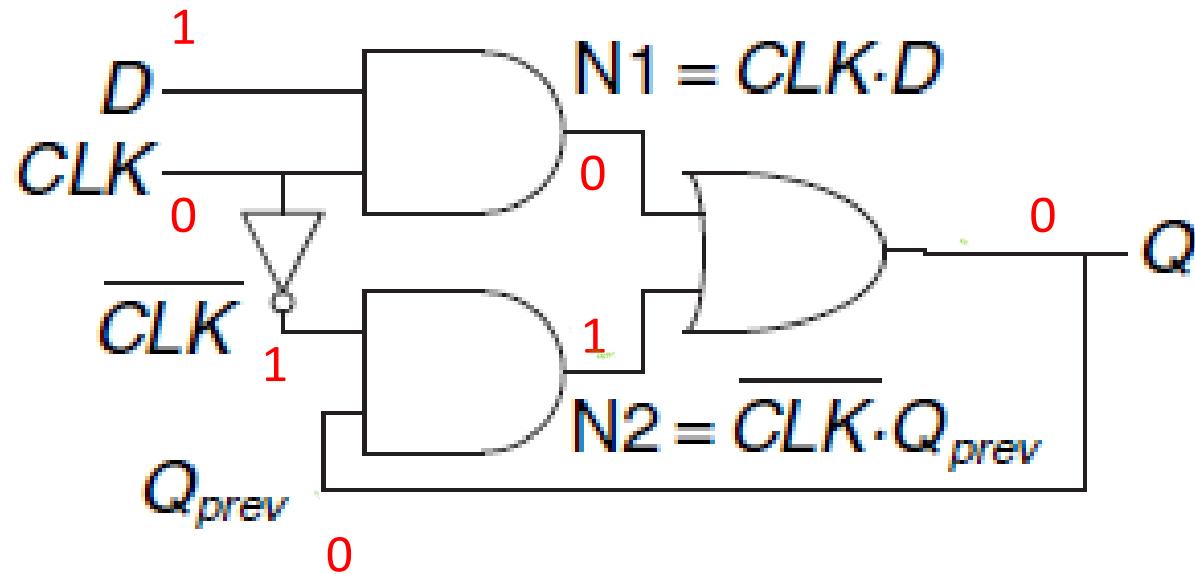
t_1



Criticità nella logica sequenziale

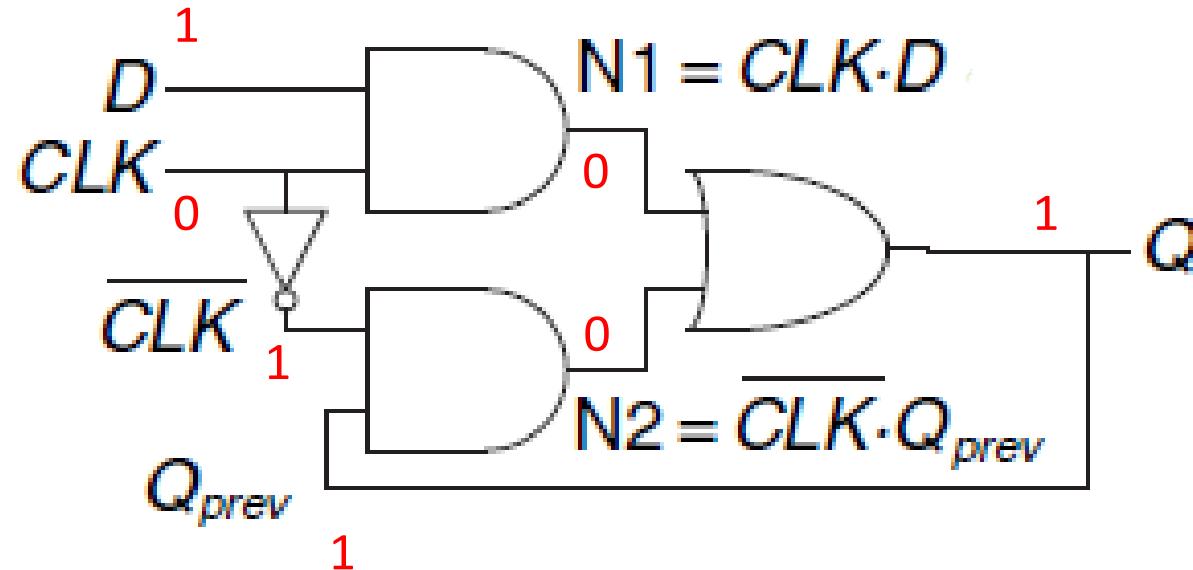
t_2

$$1 \cdot E = E$$



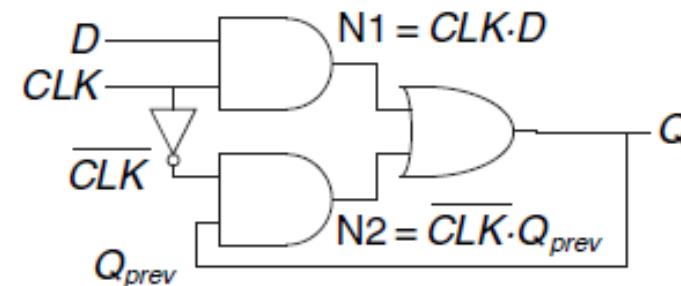
Criticità nella logica sequenziale

$t_3 (= t_1)$



Criticità nella logica sequenziale

- In casi più complessi che comprendono l'uso di più porte AND, NOT, OR il comportamento di una rete asincrona può dipendere fortemente dai ritardi accumulati sui singoli cammini



- $D=1, CLK=1 \rightarrow Q=1$
- $CLK 1 \rightarrow 0 \rightarrow Q$ oscilla $(Q=Q_{prev}=1)$

Logiche sequenziali sincrone

- I circuiti asincroni presentano delle criticità a volte difficilmente analizzabili
 - Dipendono dalla struttura fisica dei componenti
- Per questo si cerca di evitare di retroazionare l'output in maniera diretta e si interpone un registro nel ciclo di retroazione
- *Nell'ipotesi che il clock sia più lento del ritardo accumulato sul cammino, il registro consente al sistema di essere sincronizzato col clock: circuito sincrono*

Logiche sequenziali sincrone

- In generale un circuito sequenziale sincrono ha un insieme finito di stati $\{S_0, \dots, S_{k-1}\}$
- Logica combinatoria:

$$\text{out} = f(\text{in})$$

- Logica sequenziale sincrona:

$$\text{out} = f(\text{in}, s_c)$$

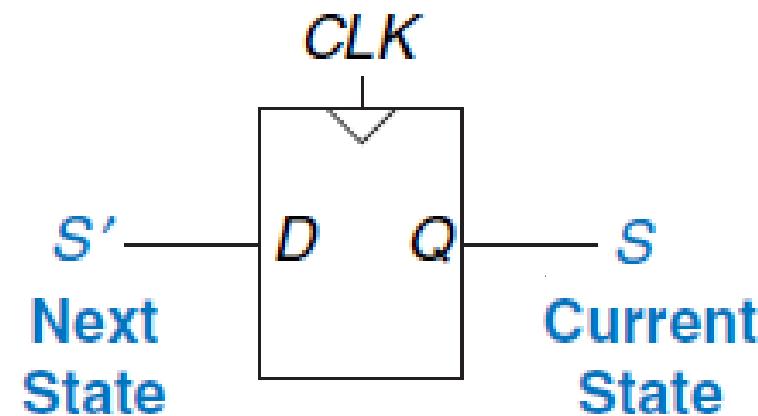
$$s_n = g(\text{in}, s_c)$$

Design di logiche sequenziali sincrone

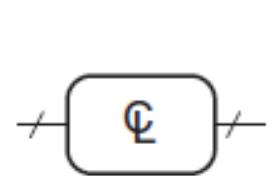
- Inserire registri nei cammini ciclici
- I registri determinano lo **stato** S_0, \dots, S_{k-1} del sistema
- I cambiamenti di stato sono determinati dalle transizioni del clock: il sistema è sincronizzato con il clock
- *Regole* di composizione:
 - Ogni componente è un registro o un circuito combinatorio
 - Almeno un componente è un registro
 - Tutti i registri sono sincronizzati con un unico clock
 - Ogni ciclo contiene almeno un registro
- Due tipici circuiti sequenziali sincroni
 - Finite State Machines (FSMs)
 - Pipelines

Current state /Next state

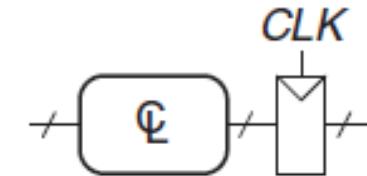
- Un flip-flop D è il più semplice circuito sequenziale sincrono
 - $Q = s_c$
 - $D = s_n$



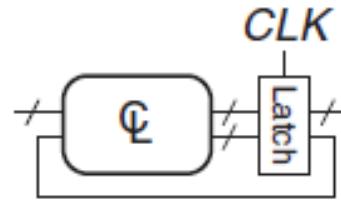
Esempi



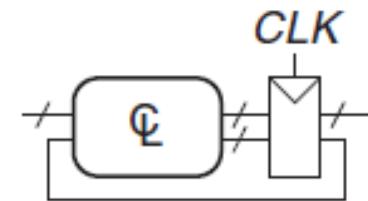
(a)



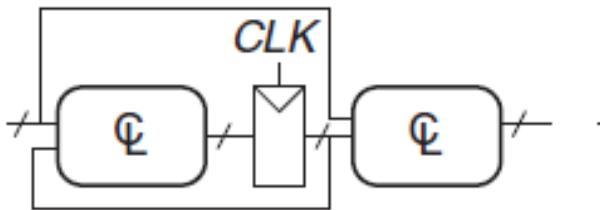
(b)



(c)



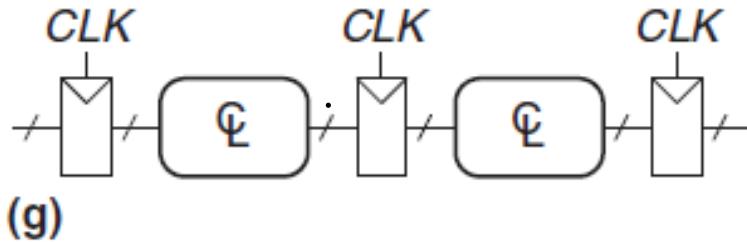
(d)



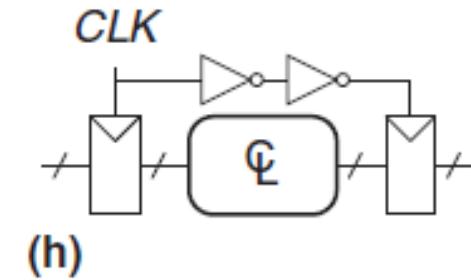
(e)



(f)



(g)



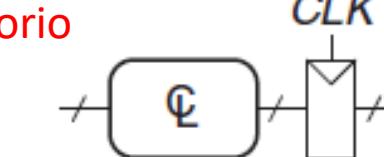
(h)

Esempi

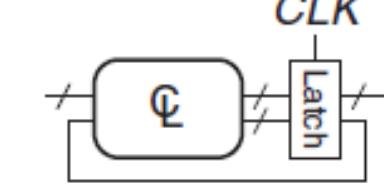
NO: circuito combinatorio



(a)

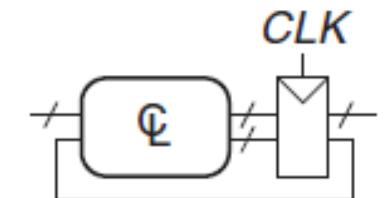


(b)

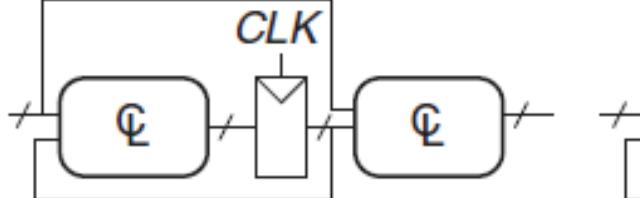


(c)

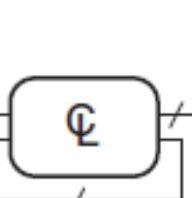
NO: latch e non flip-flop



(d)

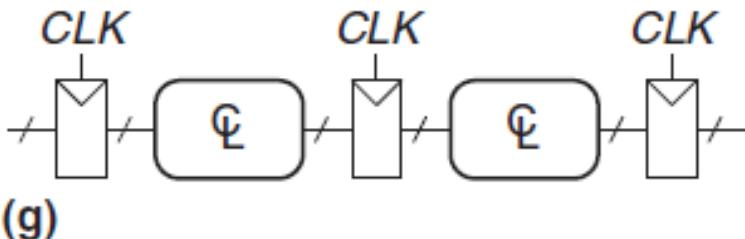


(e)

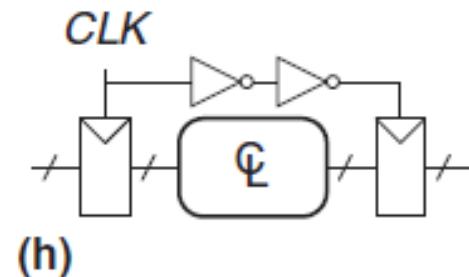


(f)

NO: circuito sequenziale asincrono



(g)

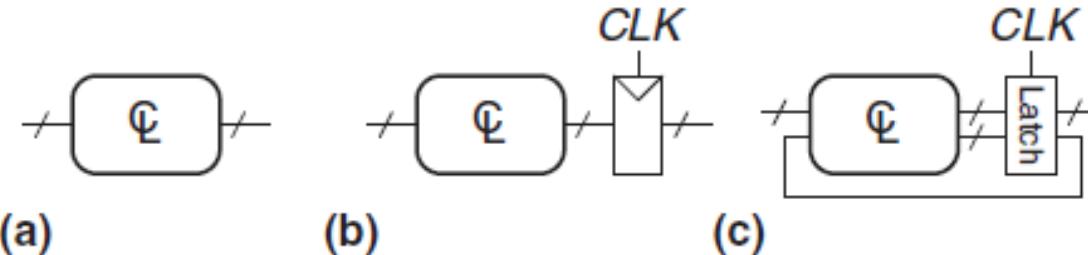


(h)

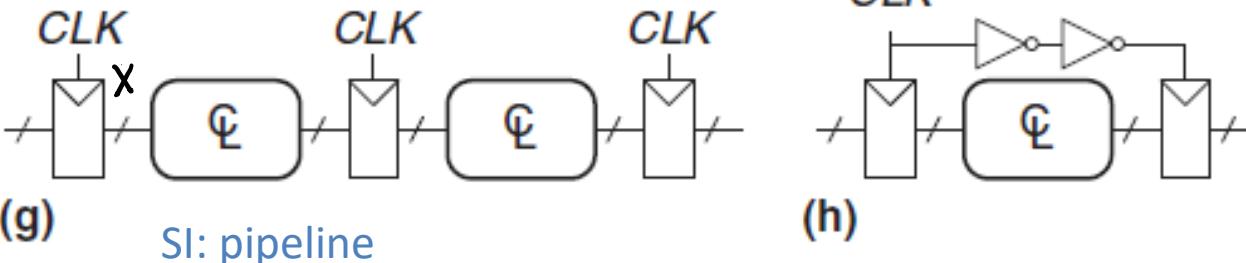
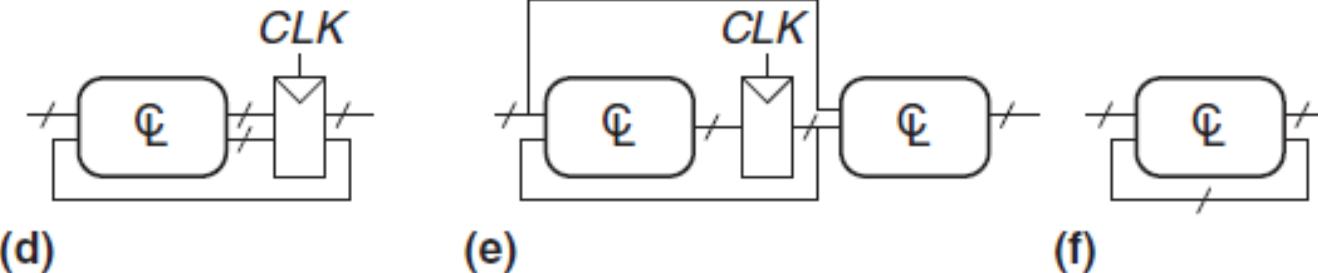
NO: i registri non hanno lo stesso clock

Esempi

SI: ma senza feedback

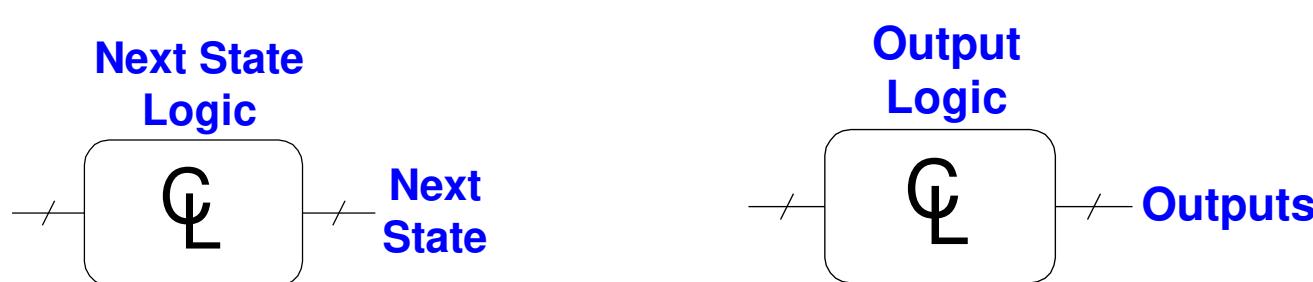
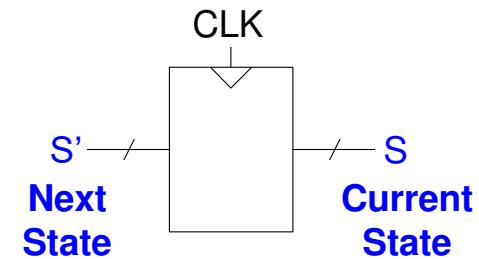


SI: FSM



Finite State Machines

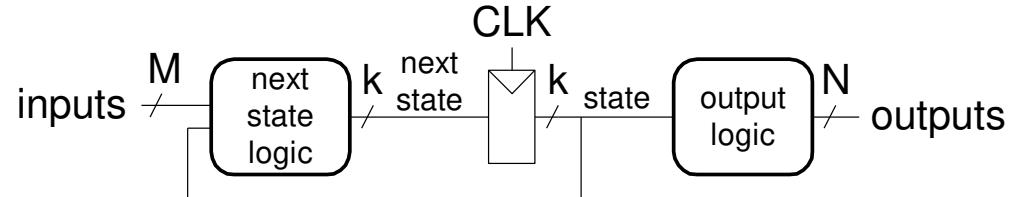
- **State register**
 - Memorizzano lo stato corrente
 - Caricano il prossimo stato (clock edge)
- **Logica combinatoria**
 - “Computa” il prossimo stato (**g**)
 - “Computa” gli output (**f**)



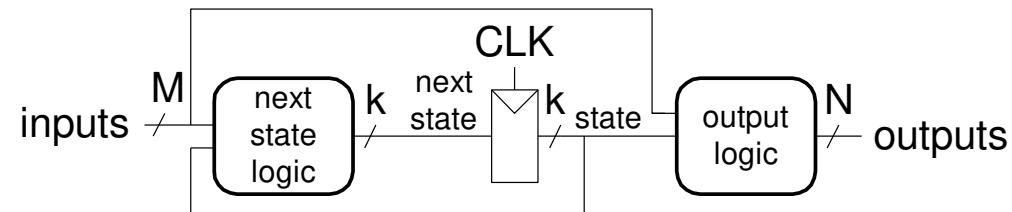
Finite State Machines

- s_n dipende sia dall'input che da s_c
$$s_n = g(in, s_c)$$
- 2 tipi di FSM a seconda della logica di output:
 - **Moore FSM:** $out=f(s_c)$
 - **Mealy FSM:** $out=f(in, s_c)$

Moore FSM

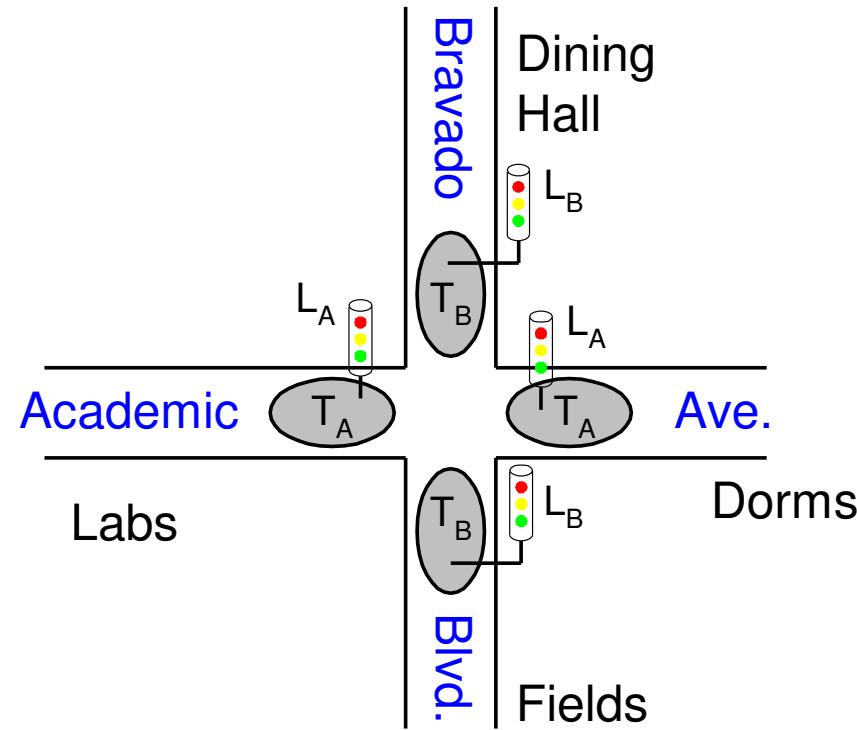


Mealy FSM



Esempio: semaforo

- Sensori: T_A, T_B (TRUE quando c'è traffico)
- Luci: L_A, L_B



Semaforo: *black box*

- Inputs: CLK , $Reset$, T_A , T_B
- Outputs: L_A , L_B

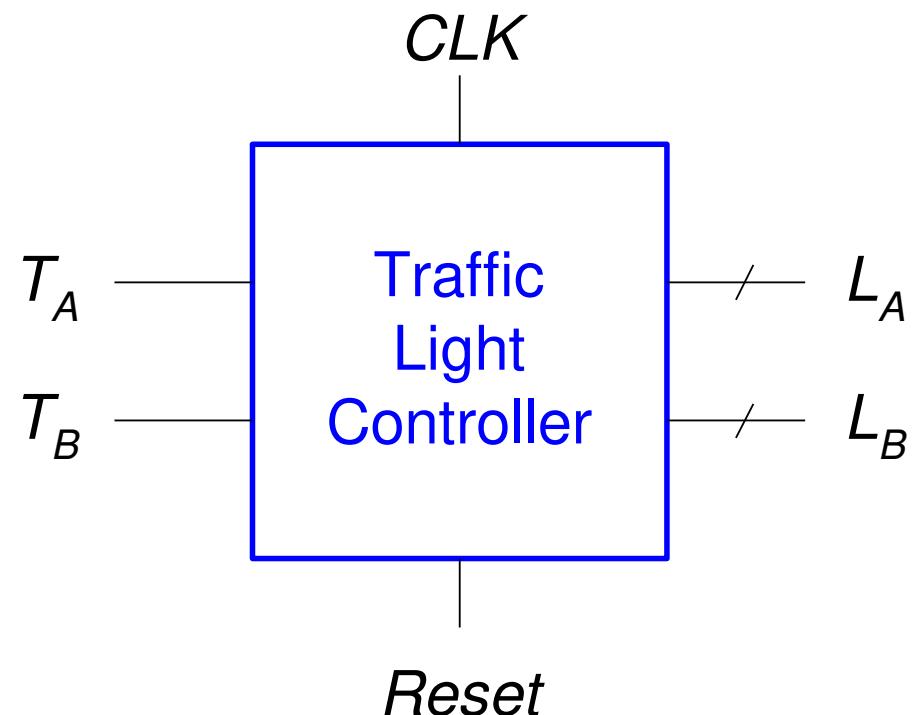
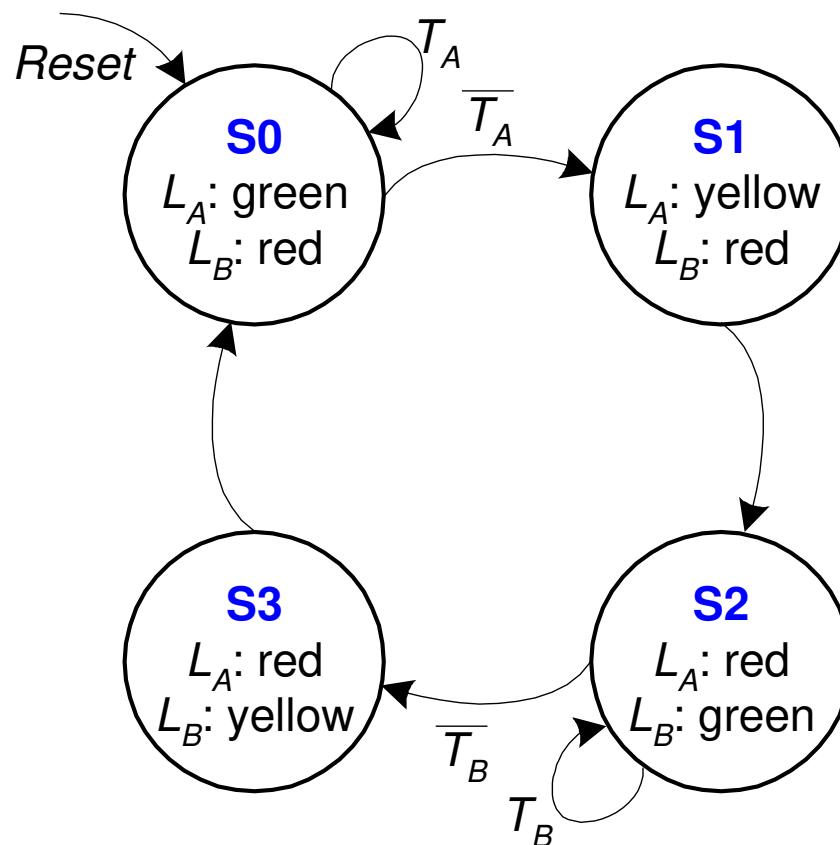


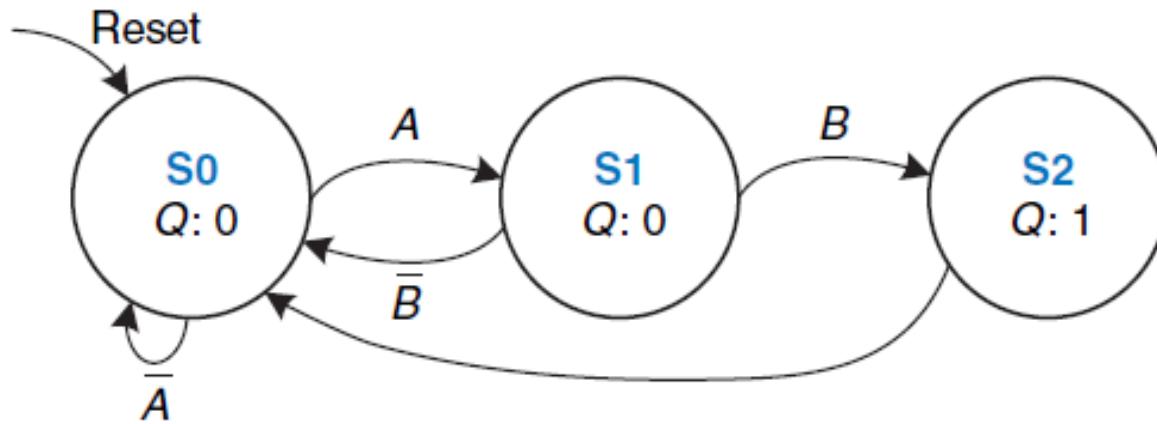
Diagramma di transizione: Moore FSM

- **Stati:** labellati con gli outputs
- **Transizioni:** labellate con gli inputs



Esempio Moore FSM

- Quale è il comportamento della FSM seguente?



Esempio Mealey FSM

- Quale è il comportamento della FSM seguente?

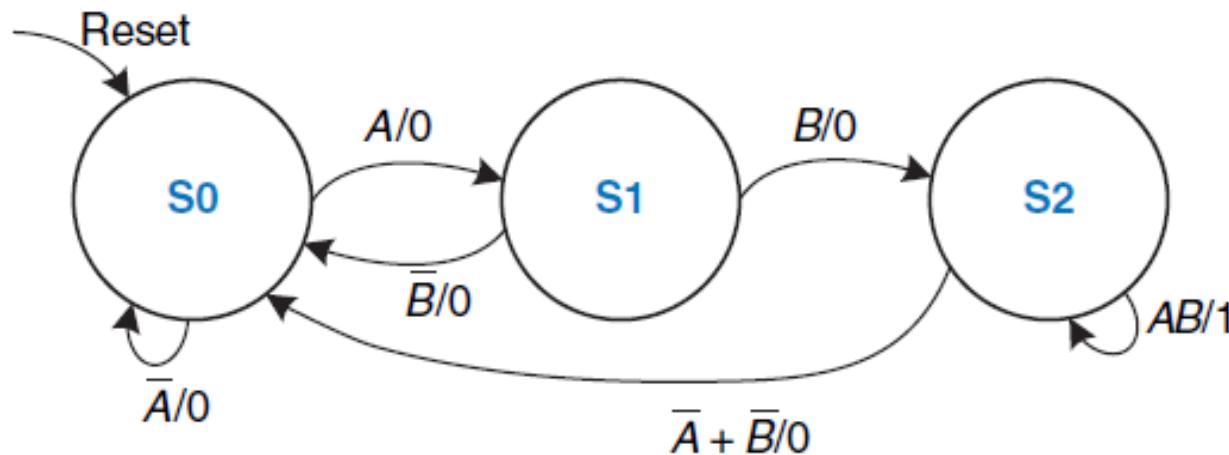


Diagramma di transizione: Moore FSM

- **Stati:** labellati con gli outputs
- **Transizioni:** labellate con gli inputs

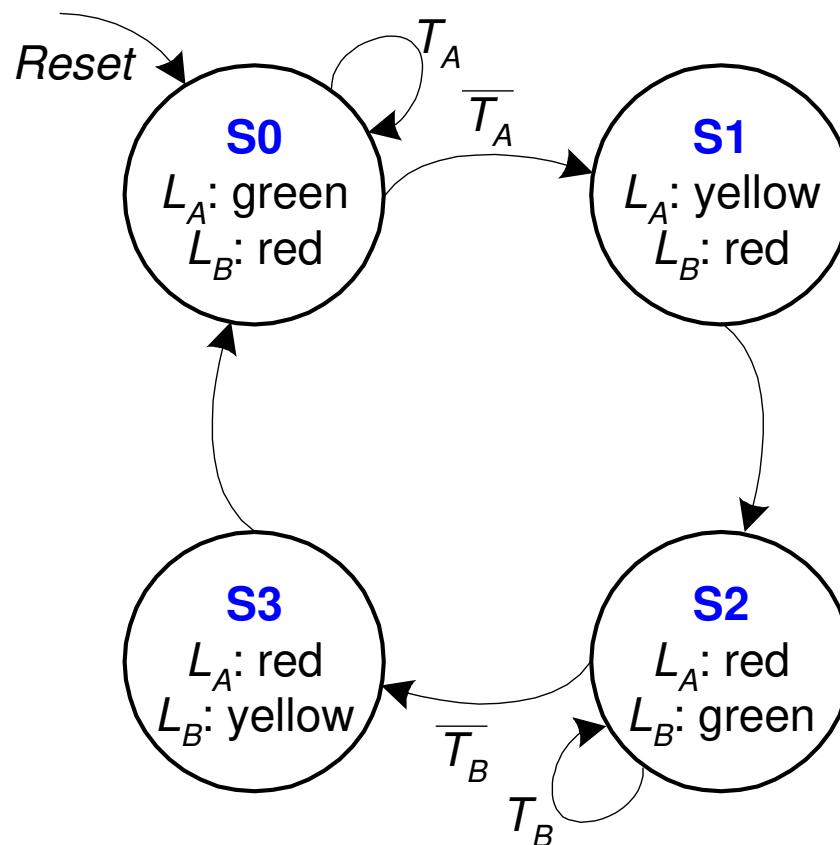


Tabella di transizione

Current State	Inputs		Next State
	T_A	T_B	
S0	0	X	S1
S0	1	X	S0
S1	X	X	S2
S2	X	0	S3
S2	X	1	S2
S3	X	X	S0

Tabella di transizione

Current State		Inputs		Next State	
S_1	S_0	T_A	T_B	S'_1	S'_0
0	0	0	X	0	1
0	0	1	X	0	0
0	1	X	X	1	0
1	0	X	0	1	1
1	0	X	1	1	0
1	1	X	X	0	0

State	Encoding
S0	00
S1	01
S2	10
S3	11

$$S'_1 = S_1 \oplus S_0$$

$$S'_0 = \overline{S_1} \overline{S_0} T_A + S_1 \overline{S_0} T_B$$

————— —————

Tabella dell'output

Current State		Outputs			
S_1	S_0	L_{A1}	L_{A0}	L_{B1}	L_{B0}
0	0	0	0	1	0
0	1	0	1	1	0
1	0	1	0	0	0
1	1	1	0	0	1

Output	Encoding
green	00
yellow	01
red	10

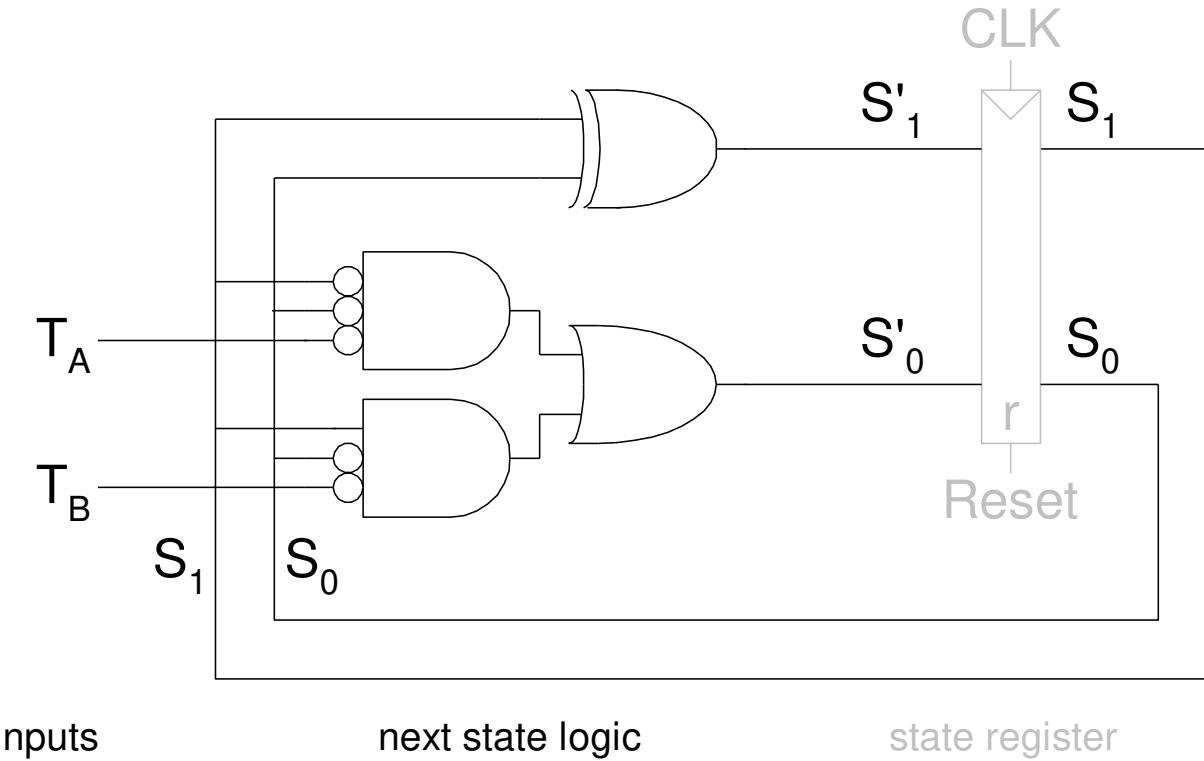
$$L_{A1} = S_1$$

$$L_{A0} = \overline{S}_1 S_0$$

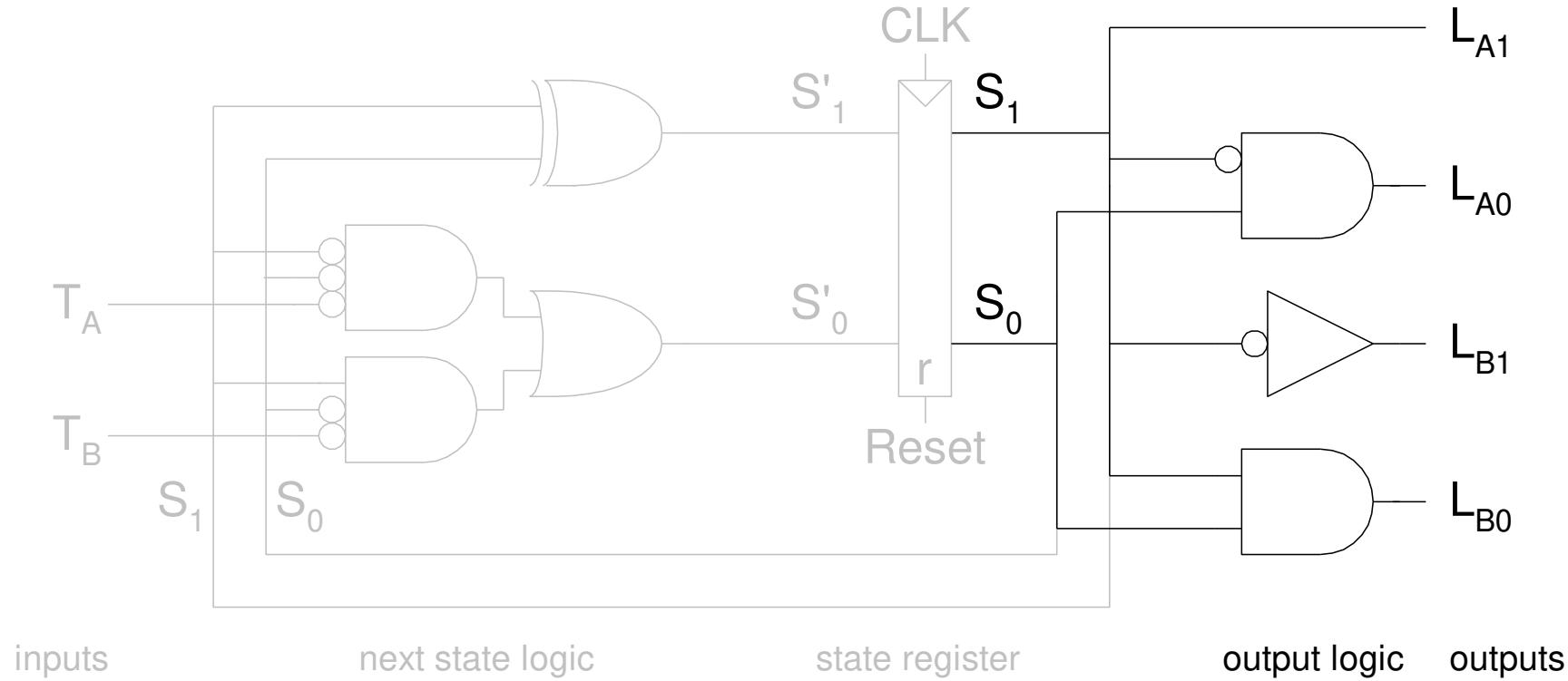
$$L_{B1} = \overline{S}_1$$

$$L_{B0} = S_1 S_0$$

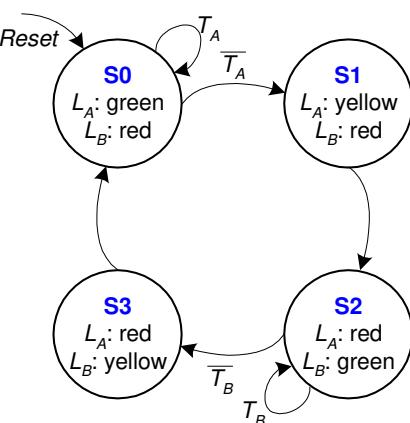
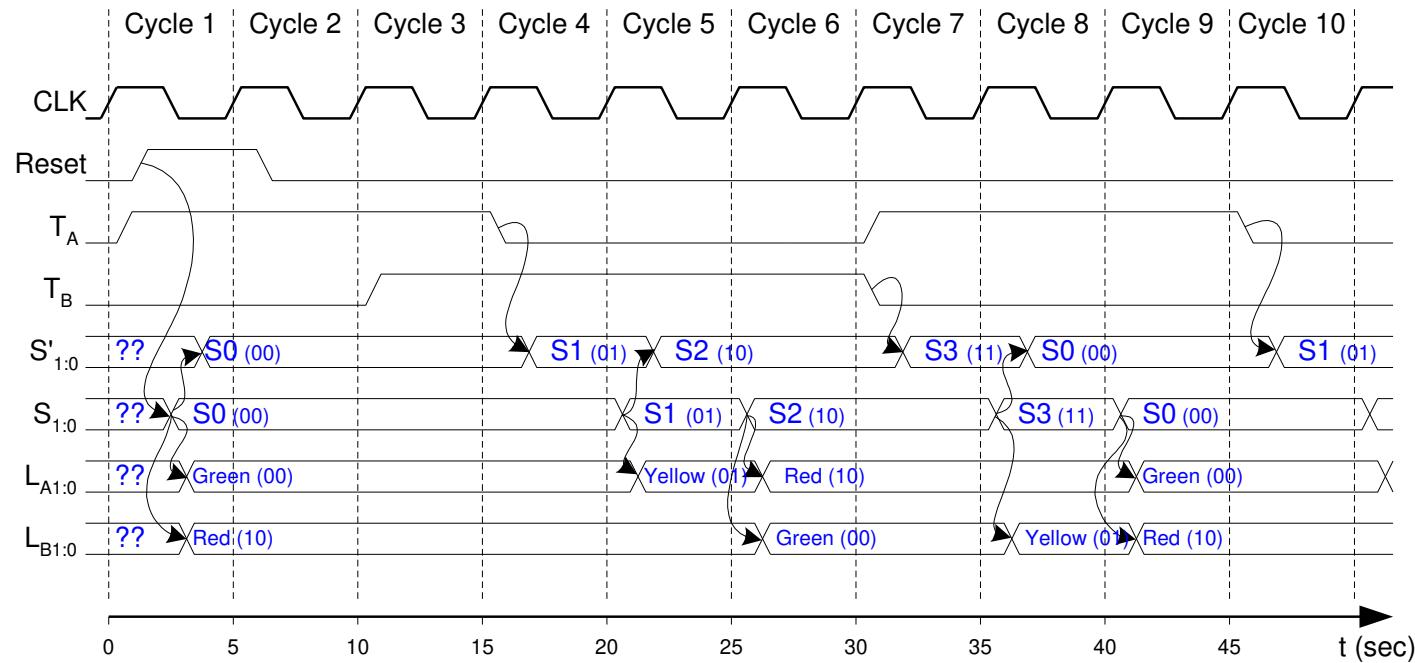
Schema della logica di transizione



Schema della logica di output



FSM Timing Diagram

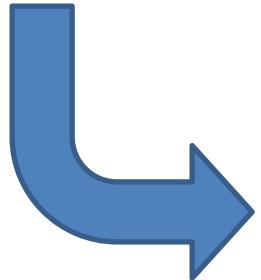


Encoding degli stati

- Encoding binario:
 - i.e., per 4 stati, 00, 01, 10, 11
- Encoding *one-hot*
 - Un bit per stato
 - Solo un bit HIGH alla volta
 - i.e., per 4 stati, 0001, 0010, 0100, 1000
 - Richiede più flip-flops
 - Spesso la logica combinatoria associata è più semplice

Moore vs Mealy FSM

Alyssa P. Hacker has a snail that crawls down a paper tape with 1's and 0's on it. The snail smiles whenever the last two digits it has crawled over are 01. Design Moore and Mealy FSMs of the snail's brain.



10011000101101

goal=1

dist_prox_goal ≥ 2

10011000101101

goal=0

dist_prox_goal ≥ 2

10011000101101

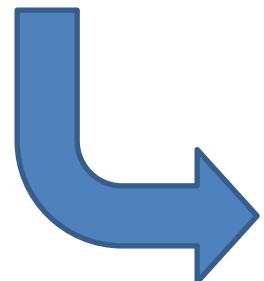
goal=0

dist_prox_goal ≥ 1

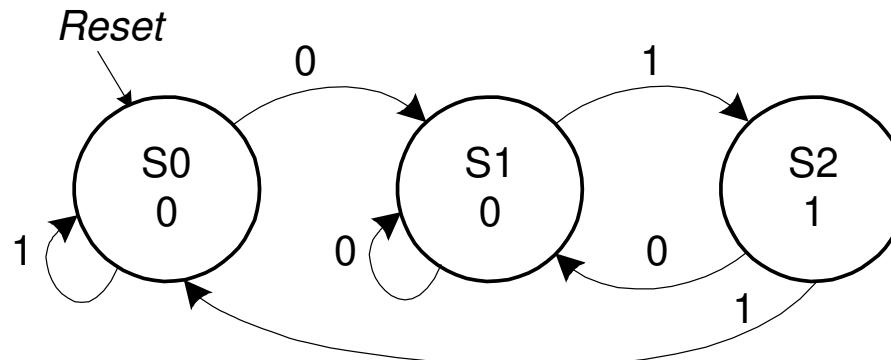
10011000101101

Moore vs Mealy FSM

Alyssa P. Hacker has a snail that crawls down a paper tape with 1's and 0's on it. The snail smiles whenever the last two digits it has crawled over are 01. Design Moore and Mealy FSMs of the snail's brain.

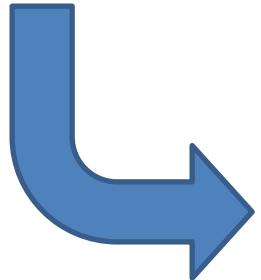


Moore FSM



Moore vs Mealy FSM

Alyssa P. Hacker has a snail that crawls down a paper tape with 1's and 0's on it. The snail smiles whenever the last two digits it has crawled over are 01. Design Moore and Mealy FSMs of the snail's brain.



10011000101101

0→

goal=0; dist_prox_goal \geq 1

1→goal=0; dist_prox_goal \geq 2

10011000101101

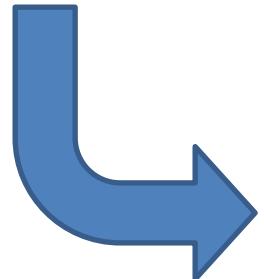
0→goal=0; dist_prox_goal \geq 1

1→ goal=1; dist_prox_goal \geq 2

10011000101101

Moore vs Mealy FSM

Alyssa P. Hacker has a snail that crawls down a paper tape with 1's and 0's on it. The snail smiles whenever the last two digits it has crawled over are 01. Design Moore and Mealy FSMs of the snail's brain.



Mealy FSM

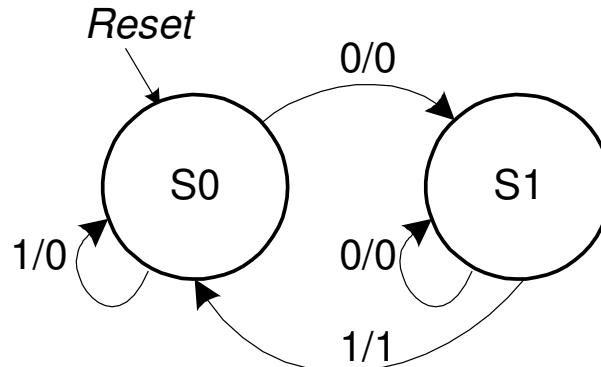


Tabella transizione Moore FSM

Current State		Inputs A	Next State	
S_1	S_0		S'_1	S'_0
0	0	0	0	1
0	0	1	0	0
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0

State	Encoding
S0	00
S1	01
S2	10

$$S_1' = S_0 A$$

$$S_0' = \overline{A}$$

Tabella transizione Moore FSM

Current State		Inputs A	Next State	
S_1	S_0		S'_1	S'_0
0	0	0	0	1
0	0	1	0	0
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0

State	Encoding
S0	00
S1	01
S2	10

$$\begin{aligned}S_1' &= S_0 A \\S_0' &= \bar{A}\end{aligned}$$

Manca S_1 negato, perché?

Tabella output Moore FSM

Current State		Output
S_1	S_0	Y
0	0	0
0	1	0
1	0	1

$$Y = S_1$$

Schema Moore FSM

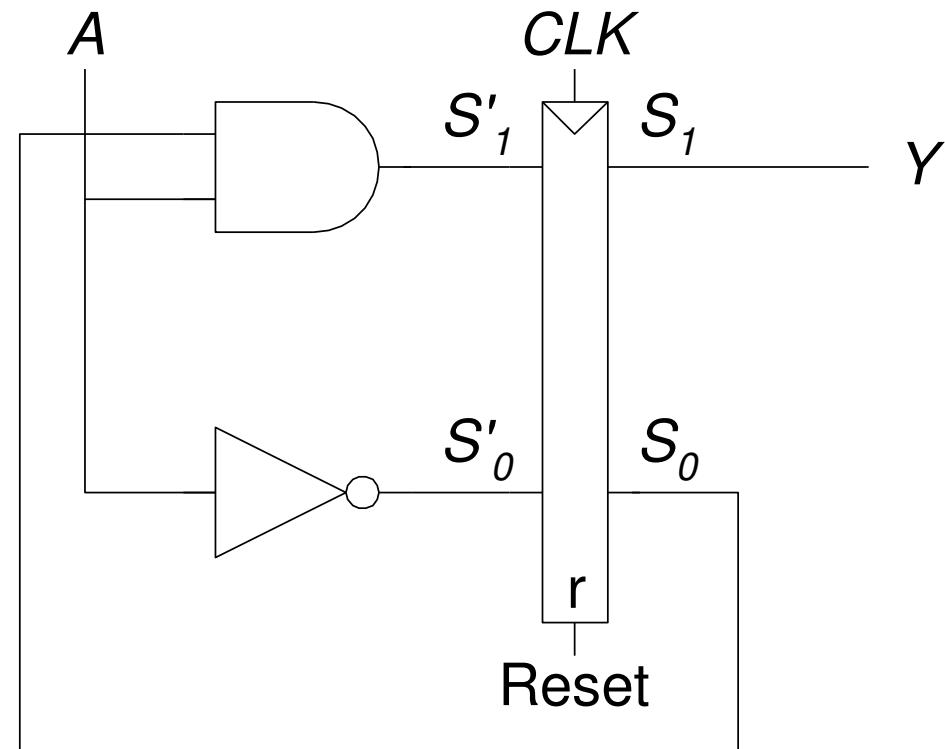


Tabella transizione/output Mealy FSM

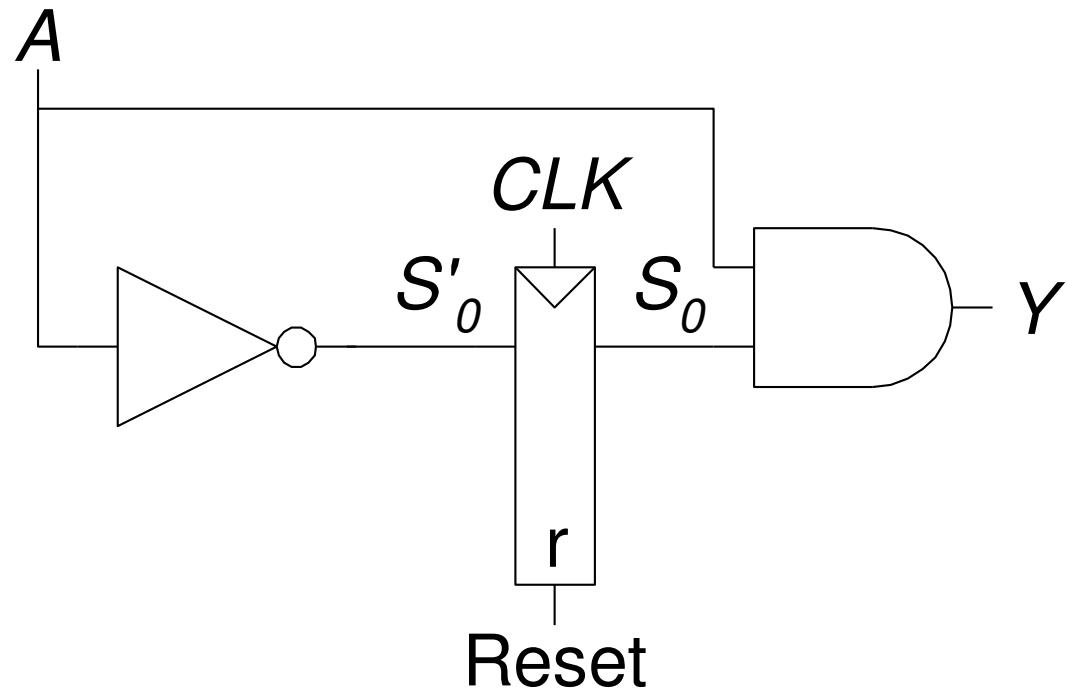
Current State	Input	Next State	Output
S	A	S'	Y
0	0	1	0
0	1	0	0
1	0	1	0
1	1	0	1

State	Encoding
S_0	0
S_1	1

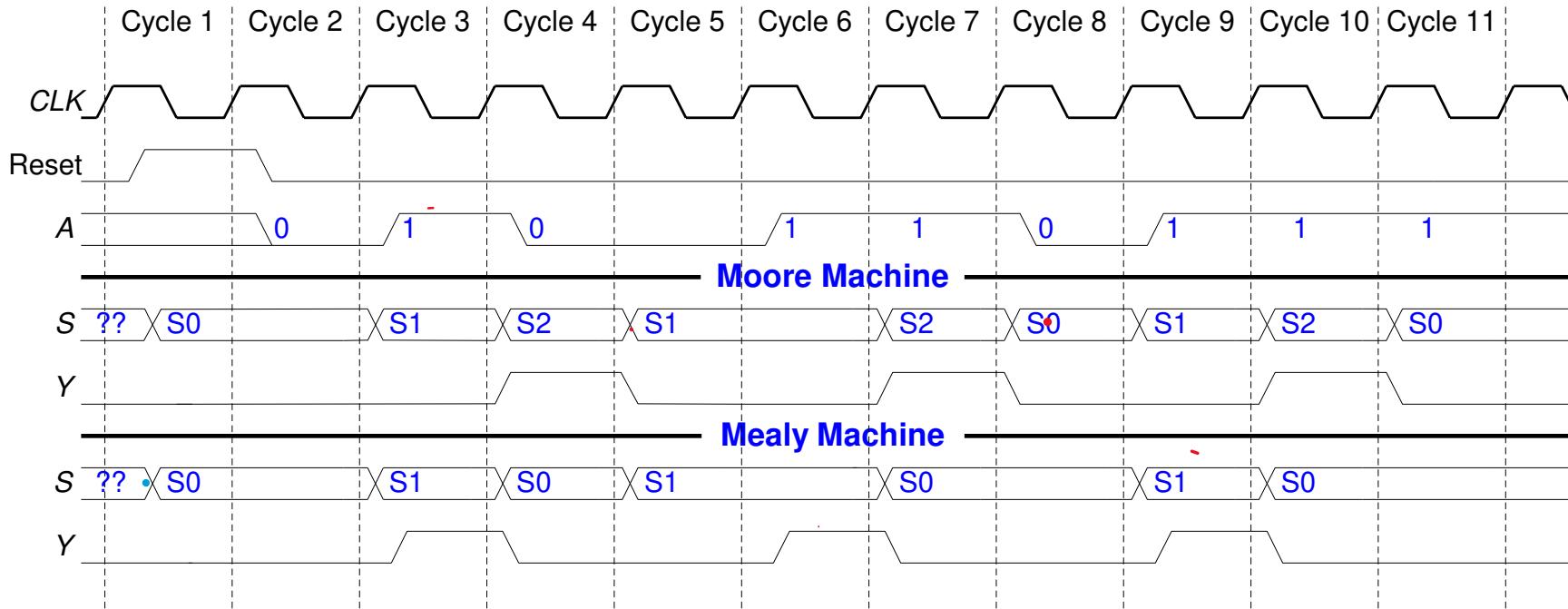
$$S' = \bar{A}$$

$$Y = SA$$

Schema Mealy FSM



Moore & Mealy Timing

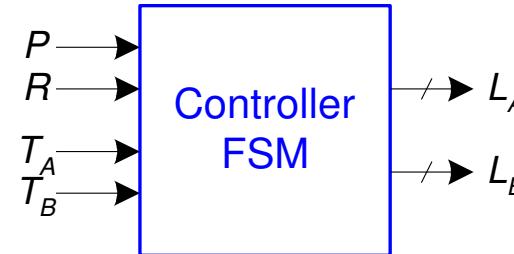


Fattorizzazione di FSM

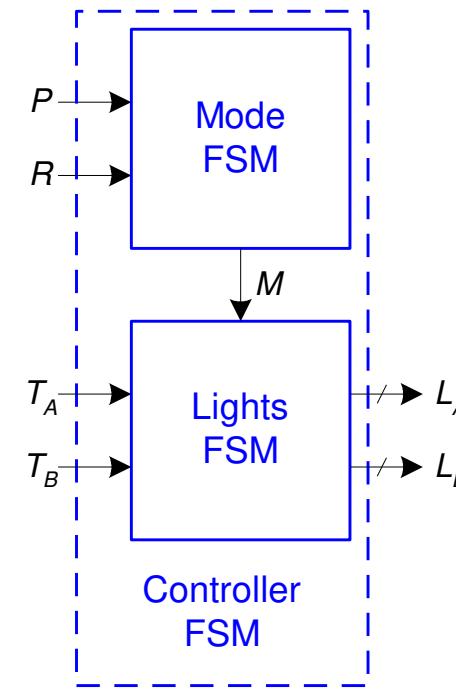
- Fattorizzare consiste nel suddividere una FSM complessa in FSM più piccole che interagiscono fra loro
- Esempio: Considerate di voler modificare il controller di semafori per tener conto di possibili parate
 - Altri due inputs: P , R
 - Se $P = 1$, entra in modalità *Parade* e il semaforo di Bravado Blvd rimane verde
 - Se $R = 1$, lascia la modalità *Parade*

Parade FSM

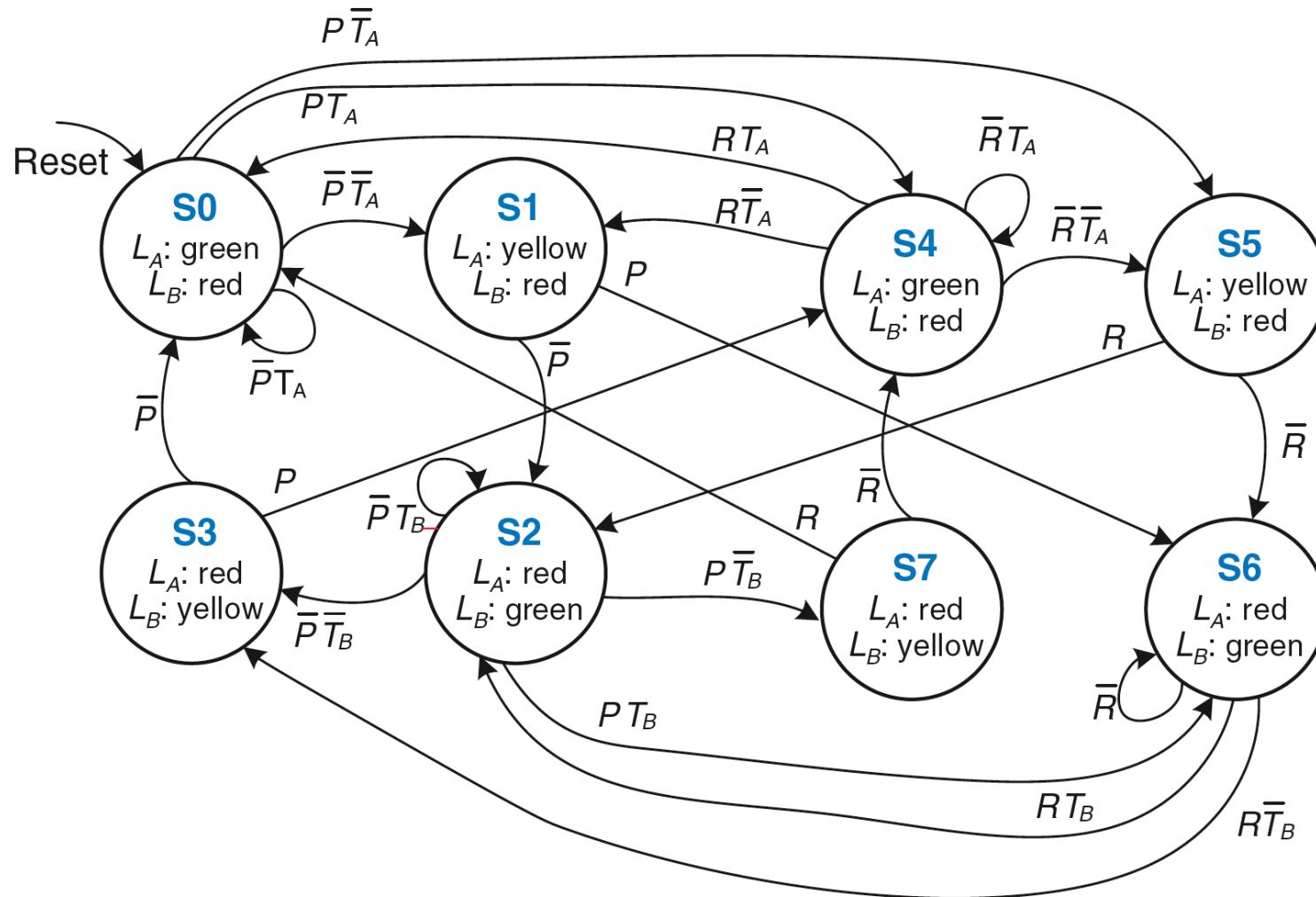
FSM non fattorizzato



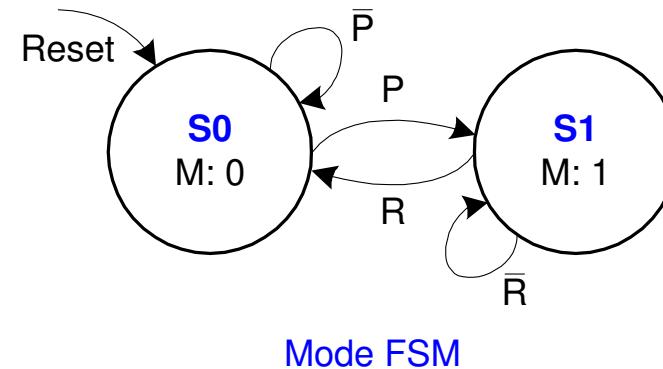
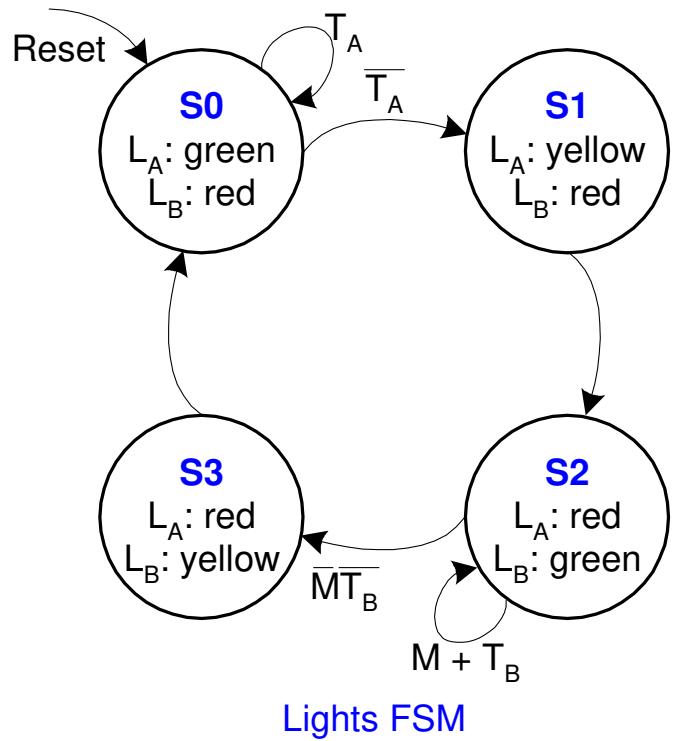
FSM fattorizzato



FSM non fattorizzato



FSM fattorizzato



Progettare una FSM

1. Identificare gli input e output
2. Abbozzare uno state transition diagram
3. Scrivere la state transition table
4. Selezionare un encoding degli stati
5. Macchina di Moore/Mealy:
 - a. Riscrivere la state transition table con l'encoding degli stati
 - b. Scrivere la output table
6. Scrivere le equazioni booleane relative alla logica di prossimo stato e alla logica di output
6. Minimizzare le equazioni
7. Fare uno schema del circuito

Esempio

- Progettare una Mealy FSM F con due input (A e B) e un output Q.
 - $Q=1$ sse A e B assumono rispettivamente il valore precedente
 - es:

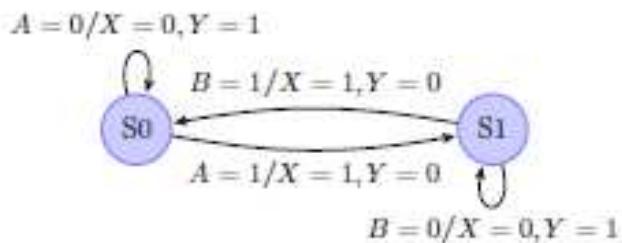
A	0	0	1	0	0	0	1	1	0
B	1	1	1	1	1	0	1	1	1
Q	0	1	0	0	1	0	0	1	0

Esercizi

■ Esercizi 3.23, 3.31

3. Il seguente diagramma di transizione per una macchina di Mealy ha due input A e B e due output X e Y . Indicare le formule SOP minime relative alla variabile di stato (S) e alle due variabili di output.

Codifica dello stato:



stato	S
S_0	0
S_1	1

Formule minime SOP:

- S^t : _____
- X : _____
- Y : _____

Parallelismo

- **2 tipi di parallelismo:**
 - **Spaziale**
 - duplicare l'hardware per eseguire più task contemporaneamente
 - **Temporale**
 - Il task è suddiviso in più fasi
 - Le diverse fasi sono eseguite in pipelining

Latenza e throughput

- **Token:** Gruppo di input da processare per ottenere un output significativo
- **Latency:** Tempo che occorre ad un token per essere processato e produrre un output
- **Throughput:** Numero di output prodotti per unità di tempo

il parallelismo incrementa il throughput

Esempio di parallelismo

- Ben vuole fare delle torte
- 5 minuti per preparare una torta
- 15 minuti per cucinarla

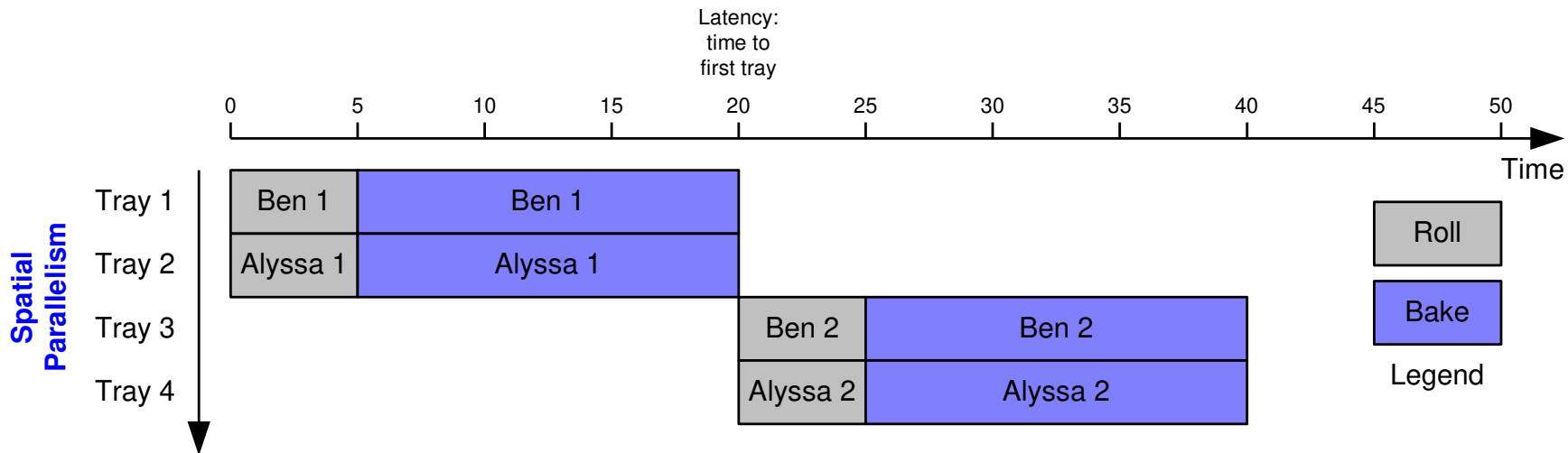
Latency = $5 + 15 = 20$ minuti = **1/3 h**

Throughput = $\frac{1}{latency}$ = **3 torte/h**

Esempio di parallelismo

- **parallelismo spaziale:** Ben chiede a Allysa di aiutarlo, usando anche il suo forno
- **parallelismo temporale:**
 - 2 fasi: preparare e cucinare
 - Mentre una torta è in forno, Ben prepara ne prepara un'altra, etc.

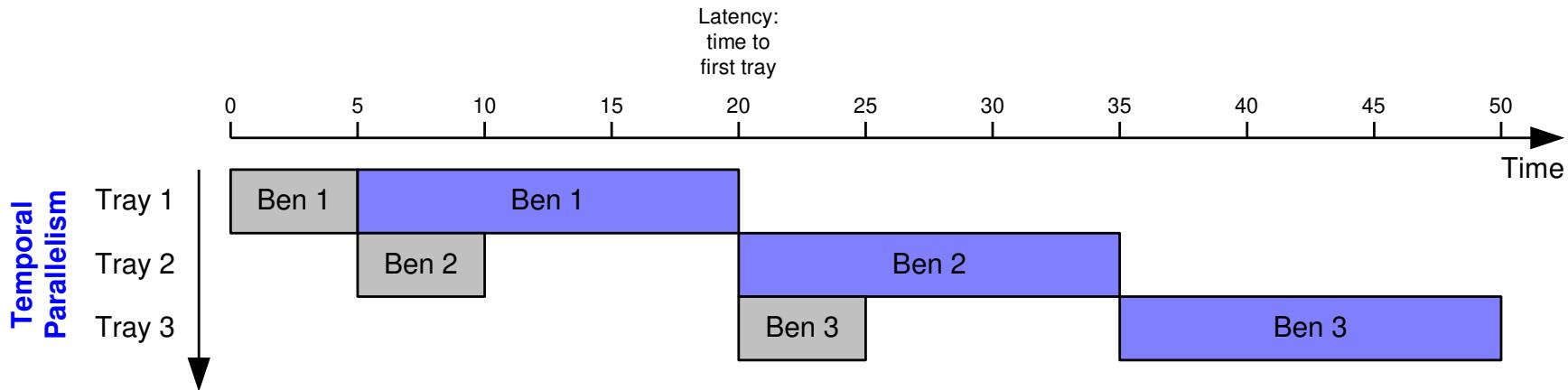
Parallelismo spaziale



$$\text{Latency} = 5 + 15 = 20 \text{ minuti} = \mathbf{1/3 \text{ h}}$$

$$\text{Throughput} = 2 \frac{1}{latency} = \mathbf{6 \text{ torte/h}}$$

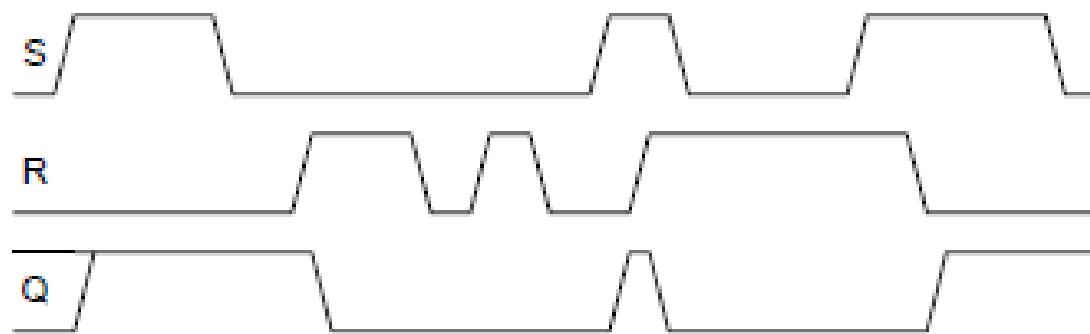
Parallelismo temporale



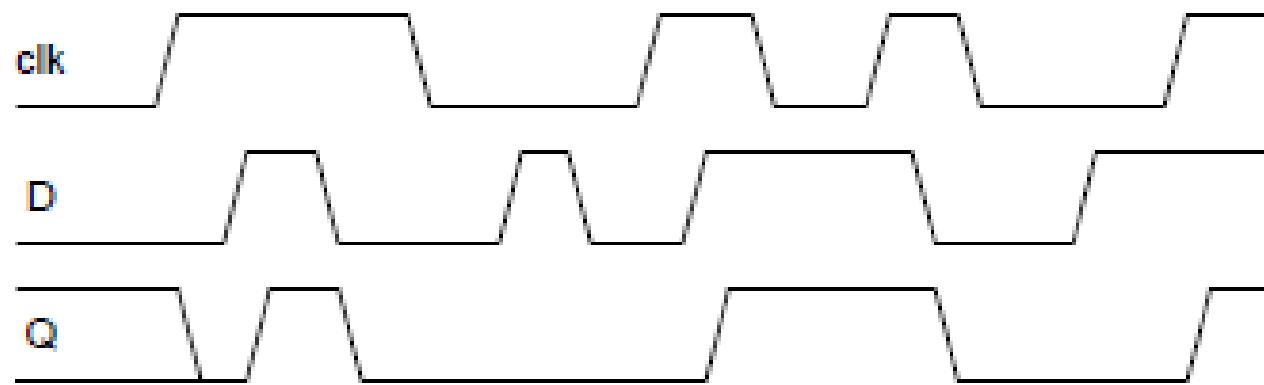
Latency = $5 + 15 = 20$ minuti = **1/3 h**

Throughput $\approx \frac{4}{65} 60 \approx$ **3,7 torte/h**

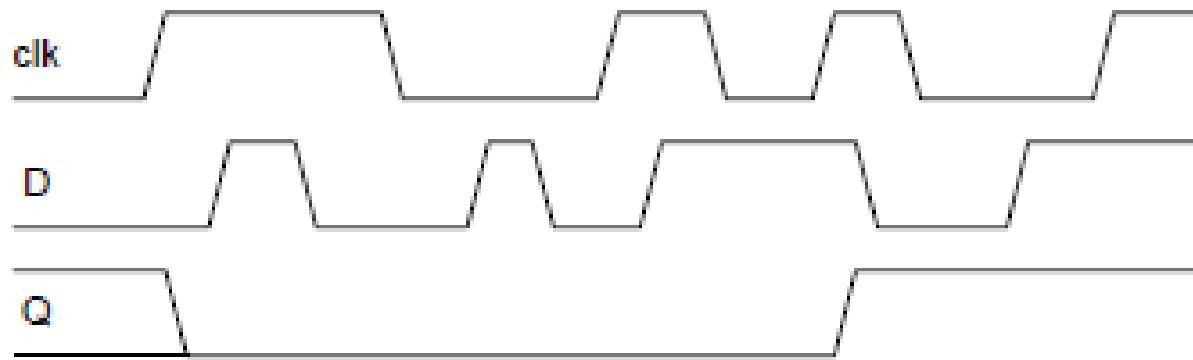
Esercizio 3.1



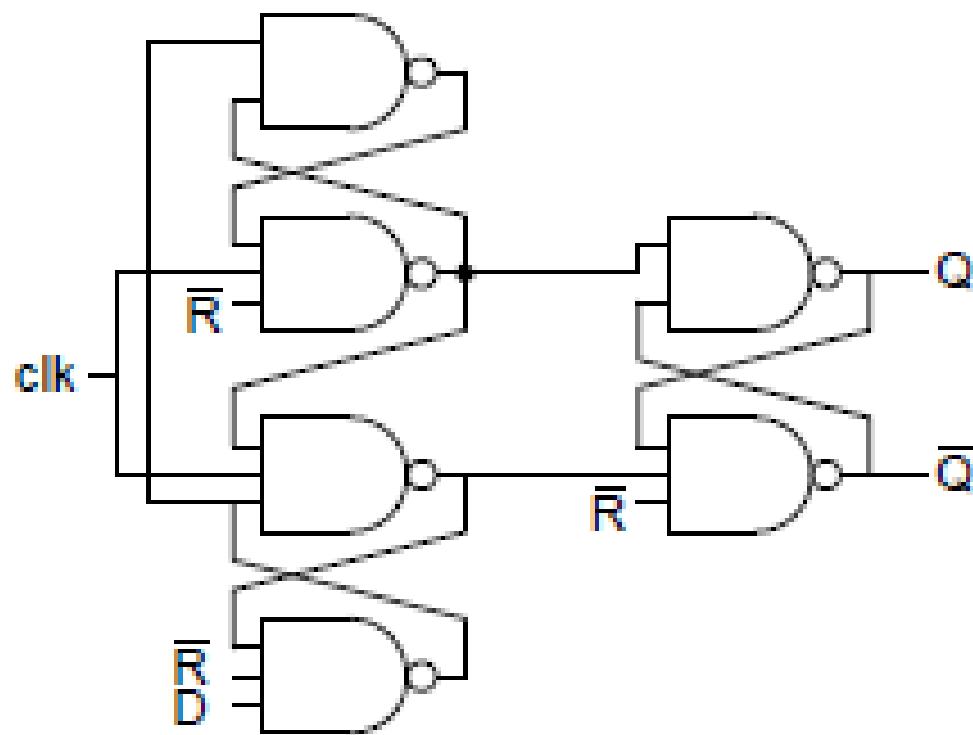
Esercizio 3.3



Esercizio 3.5

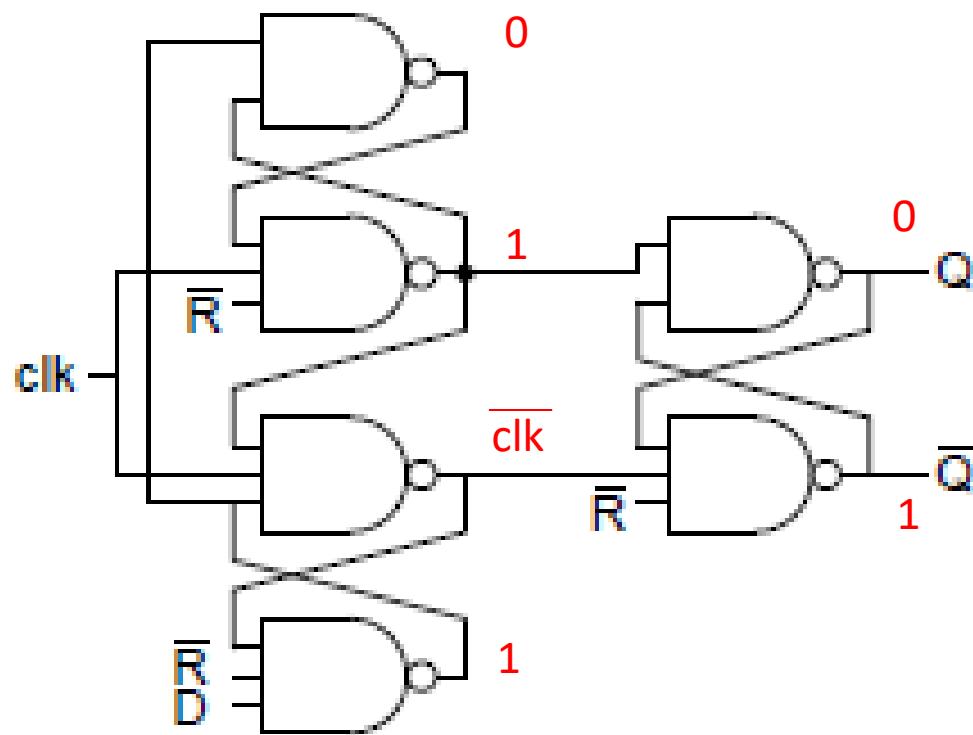


Esercizio 3.13



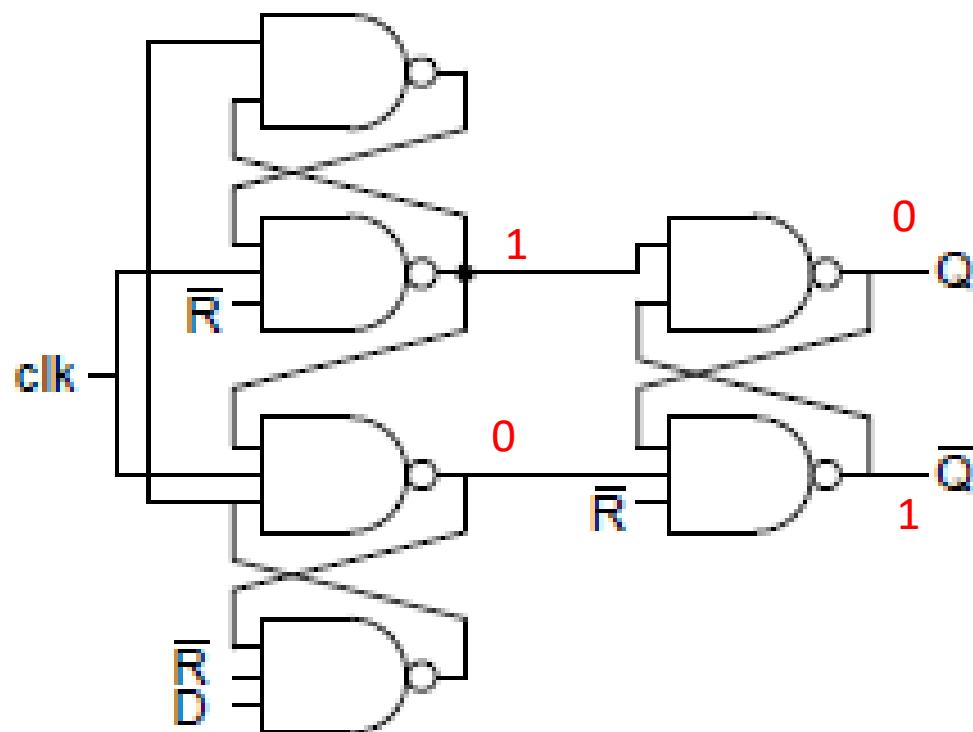
Esercizio 3.13

R=1



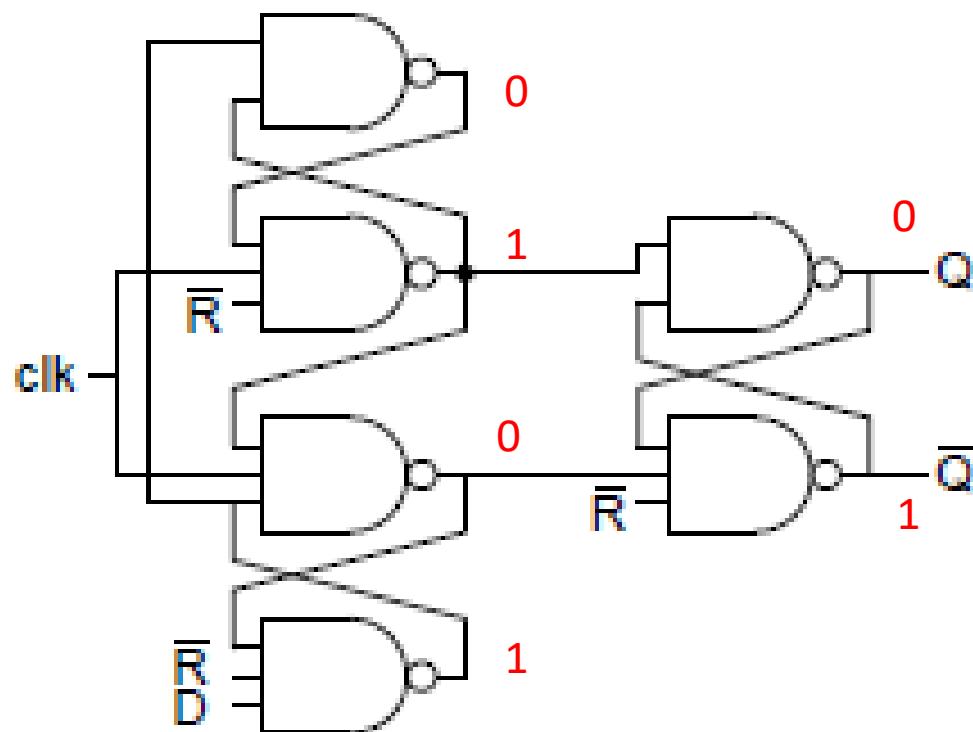
Esercizio 3.13

$R \rightarrow 0$ $\text{clk}=1$



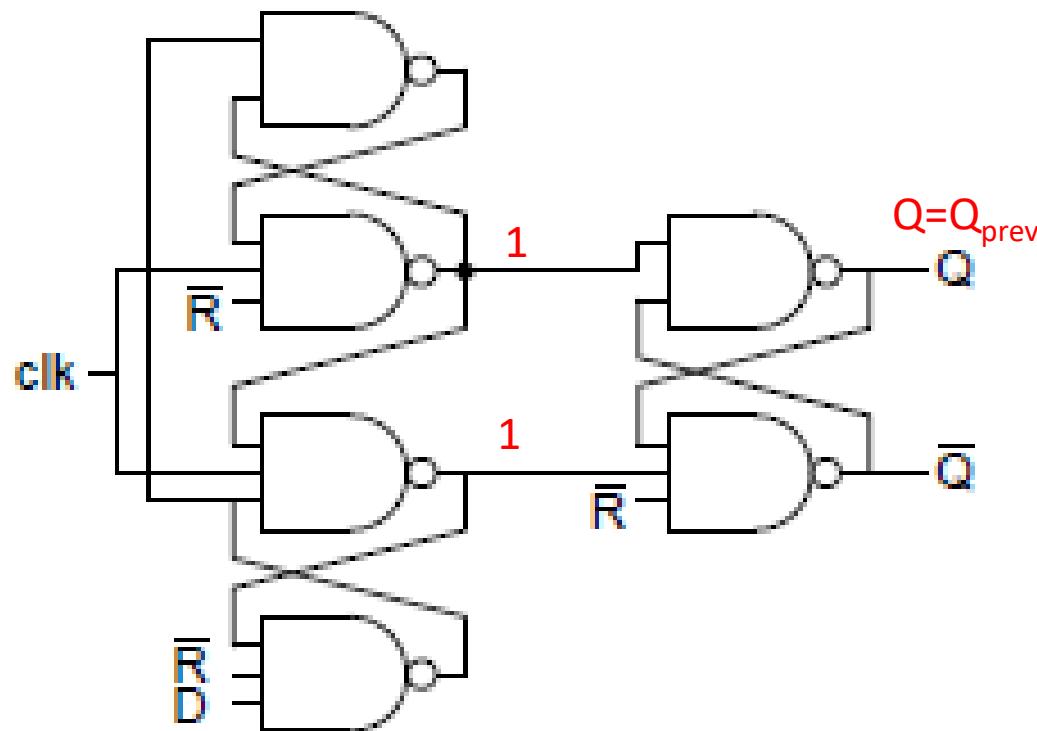
Esercizio 3.13

$R \rightarrow 0$ $\text{clk}=1$



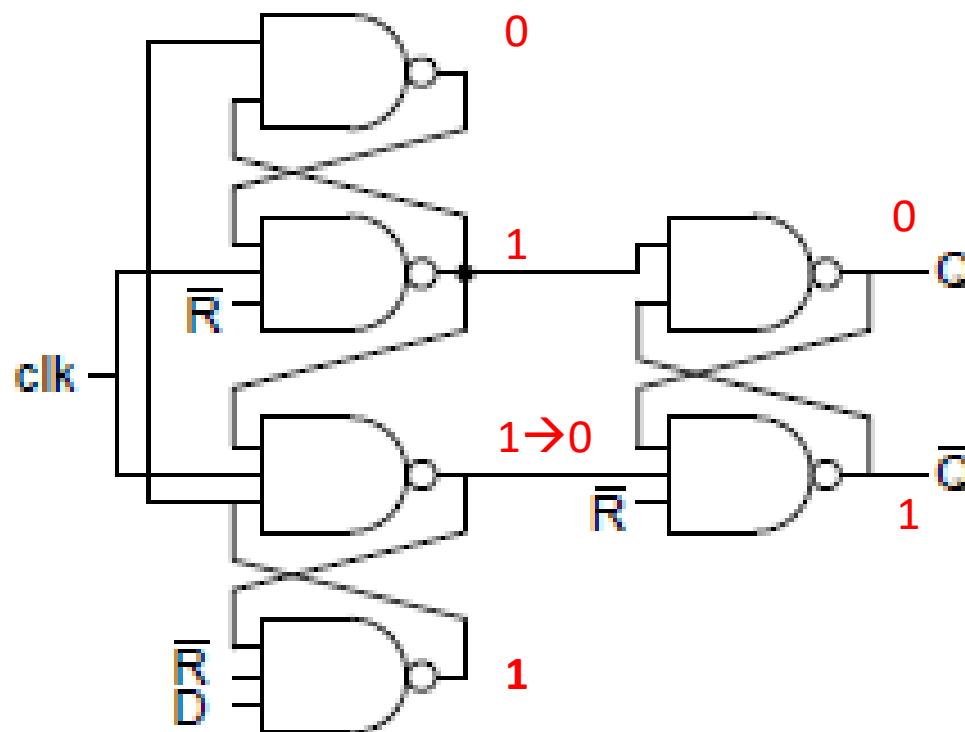
Esercizio 3.13

$R=0, \text{clk}=0$



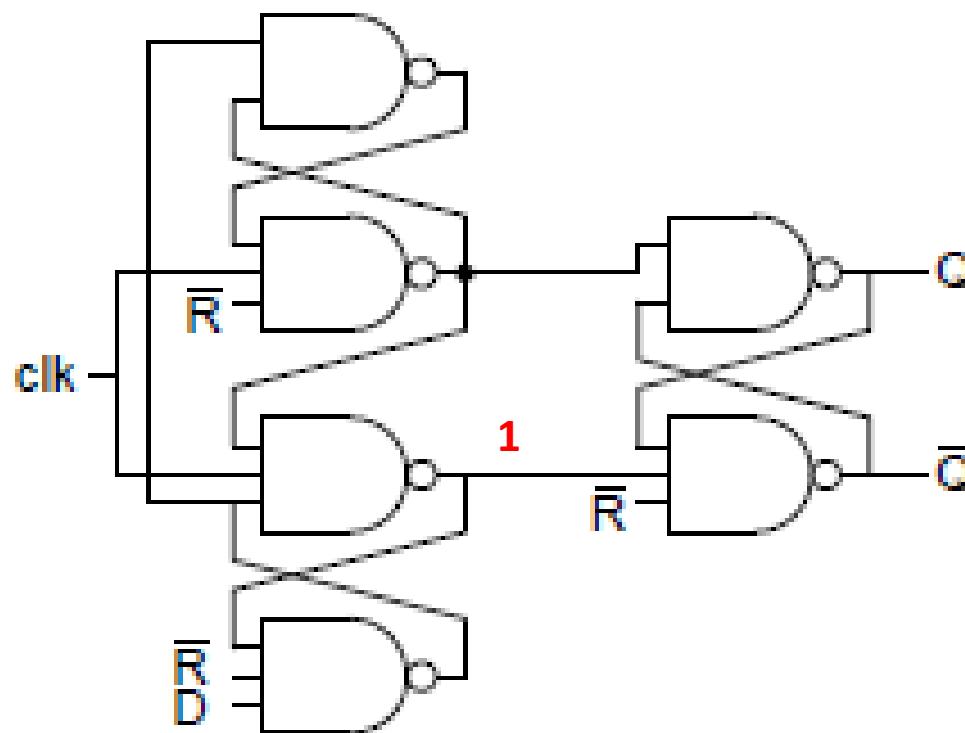
Esercizio 3.13

R=0 clk \rightarrow 1 D=0



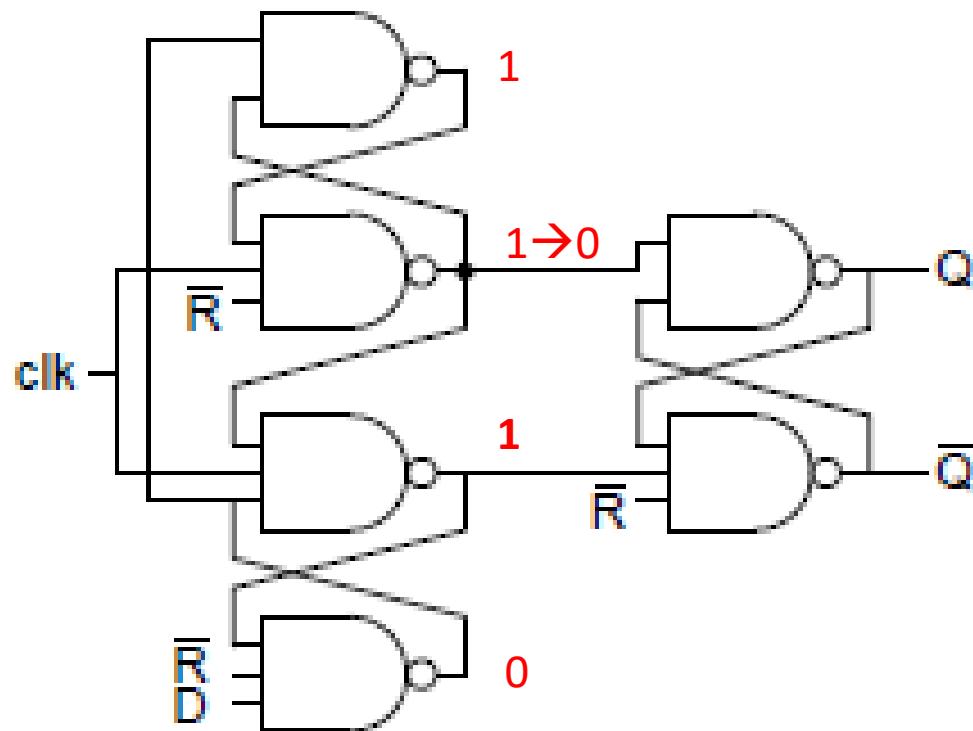
Esercizio 3.13

R=0 clk \rightarrow 1 D=1



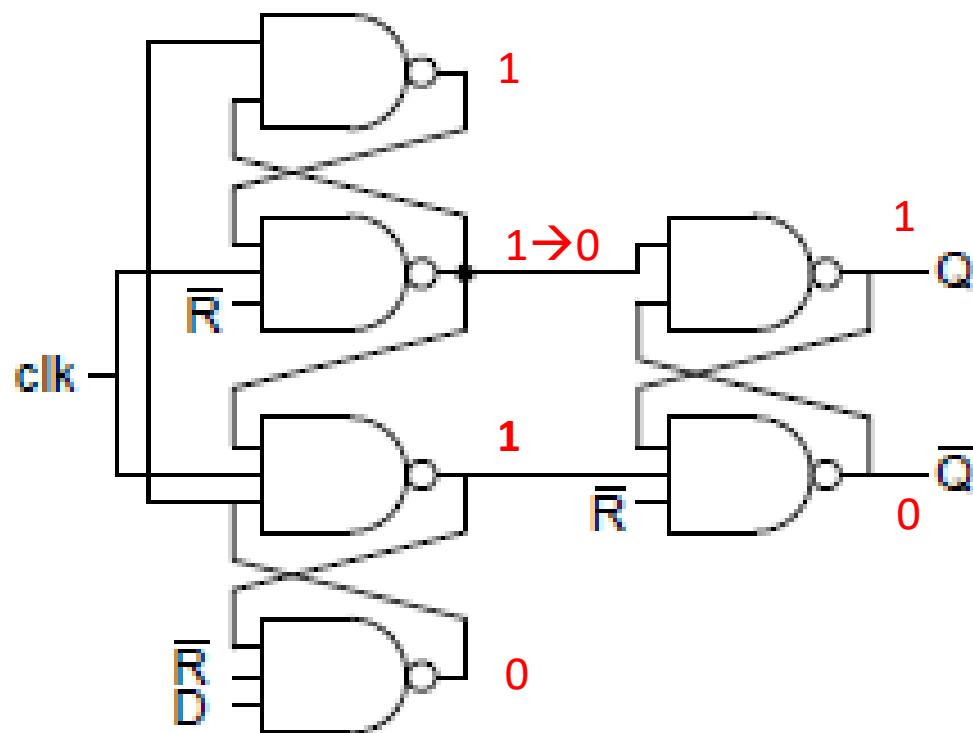
Esercizio 3.13

R=0 clk \rightarrow 1 D=1

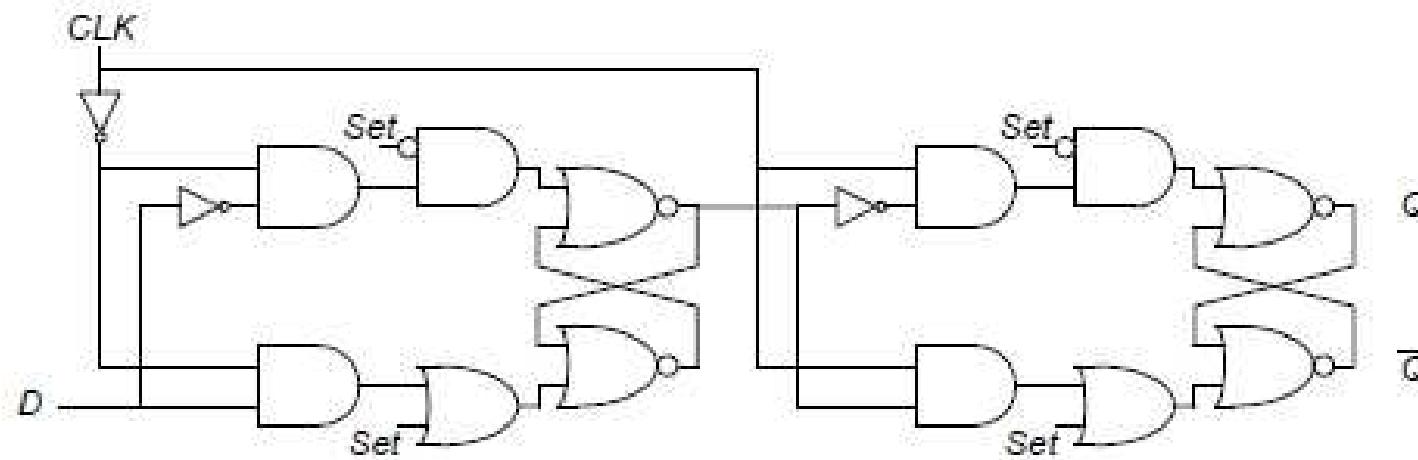


Esercizio 3.13

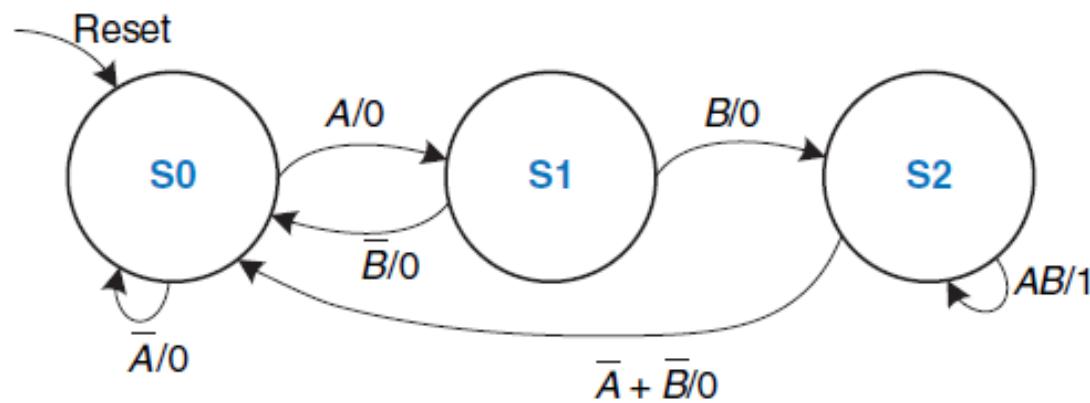
R=0 clk \rightarrow 1 D=1



Esercizio 3.15



Esercizio 3.23



Esercizio 3.23

current state		inputs		next state		output
s_1	s_0	a	b	s'_1	s'_0	q
0	0	0	X	0	0	0
0	0	1	X	0	1	0
0	1	X	0	0	0	0
0	1	X	1	1	0	0
1	0	1	1	1	0	1
1	0	0	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	0	0	0

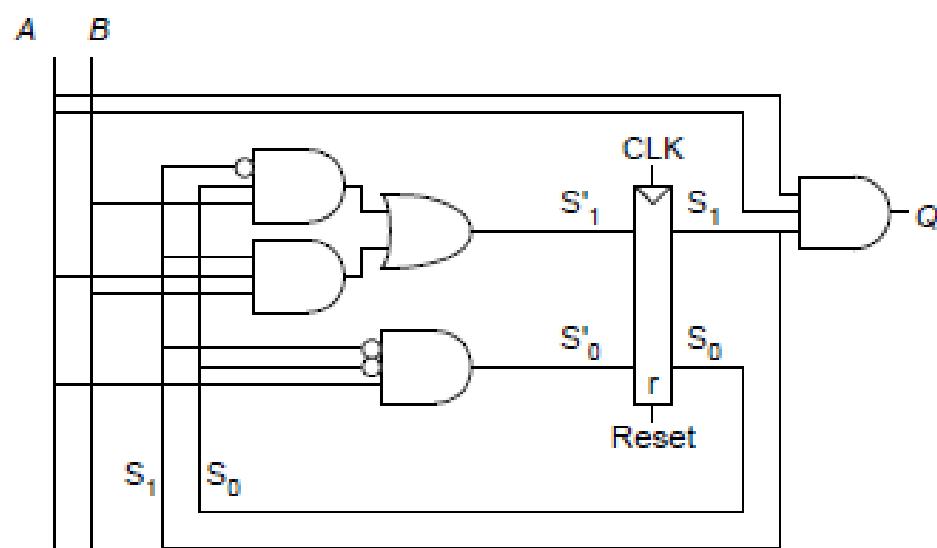
state	encoding $s_{1:0}$
S0	00
S1	01
S2	10

Esercizio 3.23

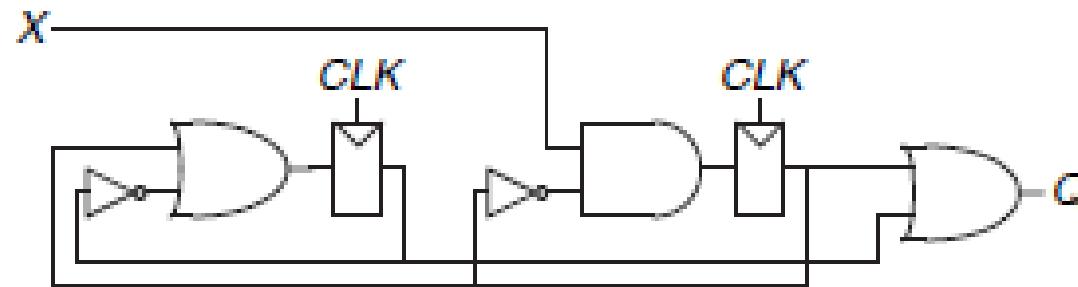
$$S_1 = \overline{S_1} S_0 B + S_1 A B$$

$$S_0 = \overline{S_1} \overline{S_0} A$$

$$Q' = S_1 A B$$



Esercizio 3.31



Esercizio 3.31

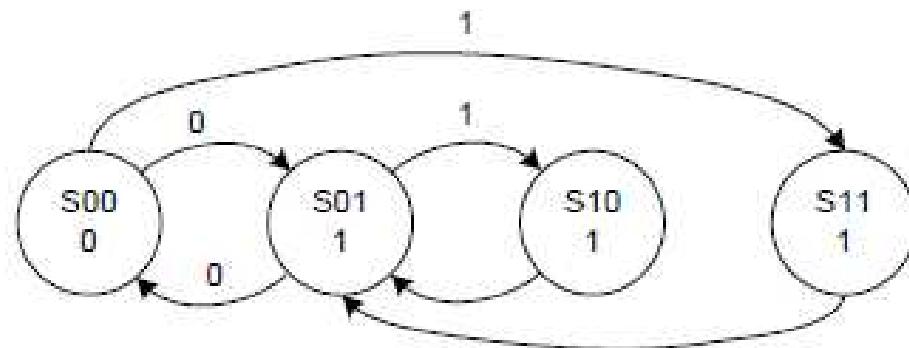
current state		input	next state	
s_1	s_0	x	s'_1	s'_0
0	0	0	0	1
0	0	1	1	1
0	1	0	0	0
0	1	1	1	0
1	X	X	0	1

TABLE 3.7 State transition table with binary encodings for Exercise 3.31

current state		output
s_1	s_0	q
0	0	0
0	1	1
1	X	1

TABLE 3.8 Output table for Exercise 3.31

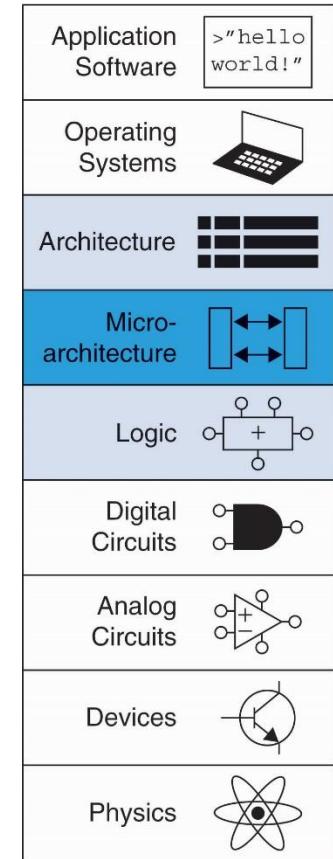
Esercizio 3.31



Microarchitettura

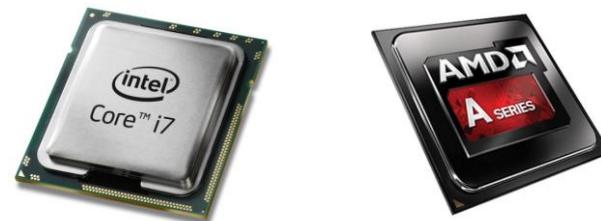
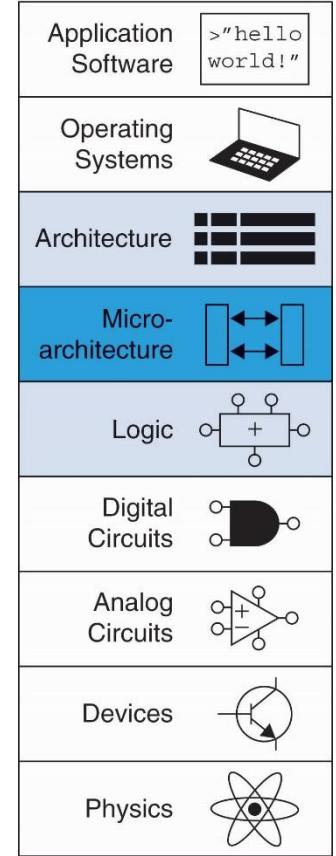
Architettura: rappresenta la struttura di un calcolatore dal punto di vista di un programmatore

- È definita dal set di istruzioni e operandi che costituisce il *linguaggio macchina*
- Al linguaggio macchina corrisponde un linguaggio *assembly*
 - **Assembly language:** formato human-readable delle istruzioni
 - **Machine language:** formato computer-readable (1 e 0)
- Esistono molte architetture differenti
 - **ARM**
 - **X86**
 - **MIPS**
 - **SPARC**
 - **PowerPC**



Microarchitettura: definisce la disposizione specifica di registri, ALU, macchine a stati finiti, le memorie e altri blocchi logici necessari per implementare una architettura

- Come due algoritmi di ordinamento (bubble e quick sort) realizzano la stessa funzione con procedure differenti, così due microarchitetture possono realizzare la medesima architettura ma soluzioni tecnologiche e prestazioni molto differenti
- Ad esempio Intel e AMD realizzano diversi modelli (microarchitetture) della medesima architettura x86.



Architettura ARM

- Sviluppata negli anni 80 dalla Advanced RISC Machines – ora chiamata ARM Holdings
- Circa 10 miliardi di processor ARM venduti ogni anno
- Impiegata soprattutto nel settore degli smartphone e tablet
- Circa il 75% della popolazione umana usa prodotti equipaggiati da un processore ARM

Design principles

L'architettura ARM segue quattro principi di progettazione fondamentali:

1. Regularity supports design simplicity
2. Make the common case fast
3. Smaller is faster
4. Good design demands good compromises

Regularity supports simplicity

C Code

a = b + c;

ARM Assembly Code

ADD a, b, c

C Code

a = b - c;

ARM Assembly Code

SUB a, b, c

ADD: mnemonico – indica l'operazione da eseguire

b, c: operandi sorgente (source operands)

a: operando destinazione (destination operand)

Regularity supports design simplicity

- Ogni istruzione è rappresentata da una parola di 32 bit (anche se alcune istruzioni richiederebbero meno di 32 bit di codifica, gestire istruzioni di lunghezza variabile aumenterebbe la complessità)
- I formati sono consistenti, nell'esempio precedente stesso numero di operandi (due sorgenti e una destinazione)
- *Questo facilita l'encoding in hardware*

Make the common case fast

- L'architettura ARM comprende solo istruzioni semplici e di uso frequente
- L'hardware necessario per decodificare ed eseguire le istruzioni è così *semplice* e veloce
- Istruzioni complesse (che sono meno comuni) corrispondono a più istruzioni ARM

C Code

```
a = b + c - d;
```

ARM assembly code

```
ADD t, b, c ; t = b + c  
SUB a, t, d ; a = t - d
```

- ARM è un **Reduced Instruction Set Computer (RISC)**: poche istruzioni e semplici
- X86 è un **Complex Instruction Set Computers (CISC)**: molte istruzioni complesse

Smaller is faster

I dati sono memorizzati:

- Direttamente all'interno di una istruzione.
 - Costanti (anche dette *immediates*)
- In registri
 - ARM ha solo 16 registri
 - I registri sono più veloci della memoria
 - Ci focalizzeremo su di una architettura in cui ogni registro è costituito da 32 bits
- In memoria
 - Più lenta dei registri ma più capiente
 - A sua volta suddivisa in diversi livelli: cache, centrale, di massa
 - le istruzioni ARM operano esclusivamente sui registri, quindi i dati residenti nella memoria devono essere spostati in un registro prima di poter essere elaborati.

Good design demands good compromises

Al fine di massimizzare la semplicità sarebbe preferibile anche avere un unico formato per le istruzioni. Tuttavia, questa scelta risulterebbe troppo restrittiva.

Il compromesso scelto da ARM (esempio del principio 4) è di avere **tre** possibili **formati di istruzione**:

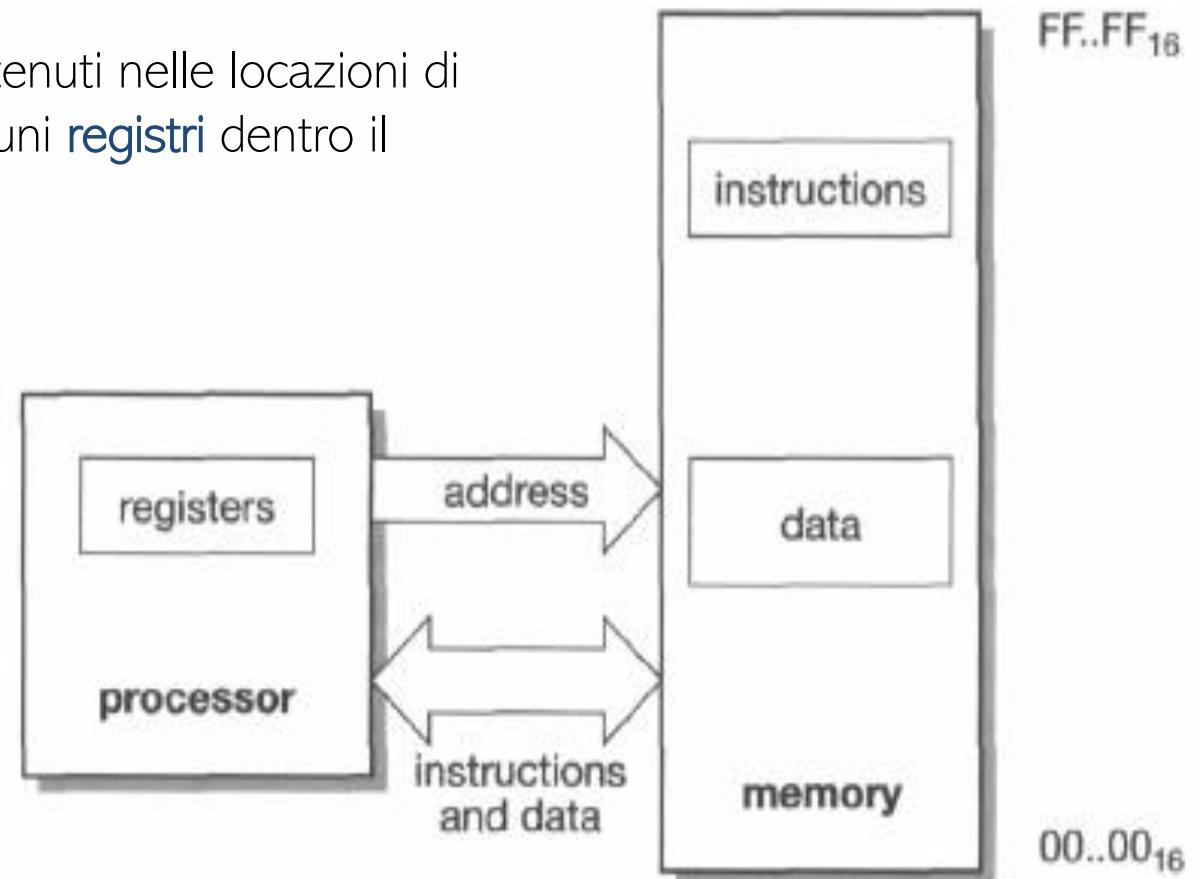
- ▶ Data-processing;
- ▶ Memory;
- ▶ Branch.

Cos'è un processore

Un processore general-purpose è un **automa** a stati finiti, che esegue **istruzioni** rilocate in una memoria.

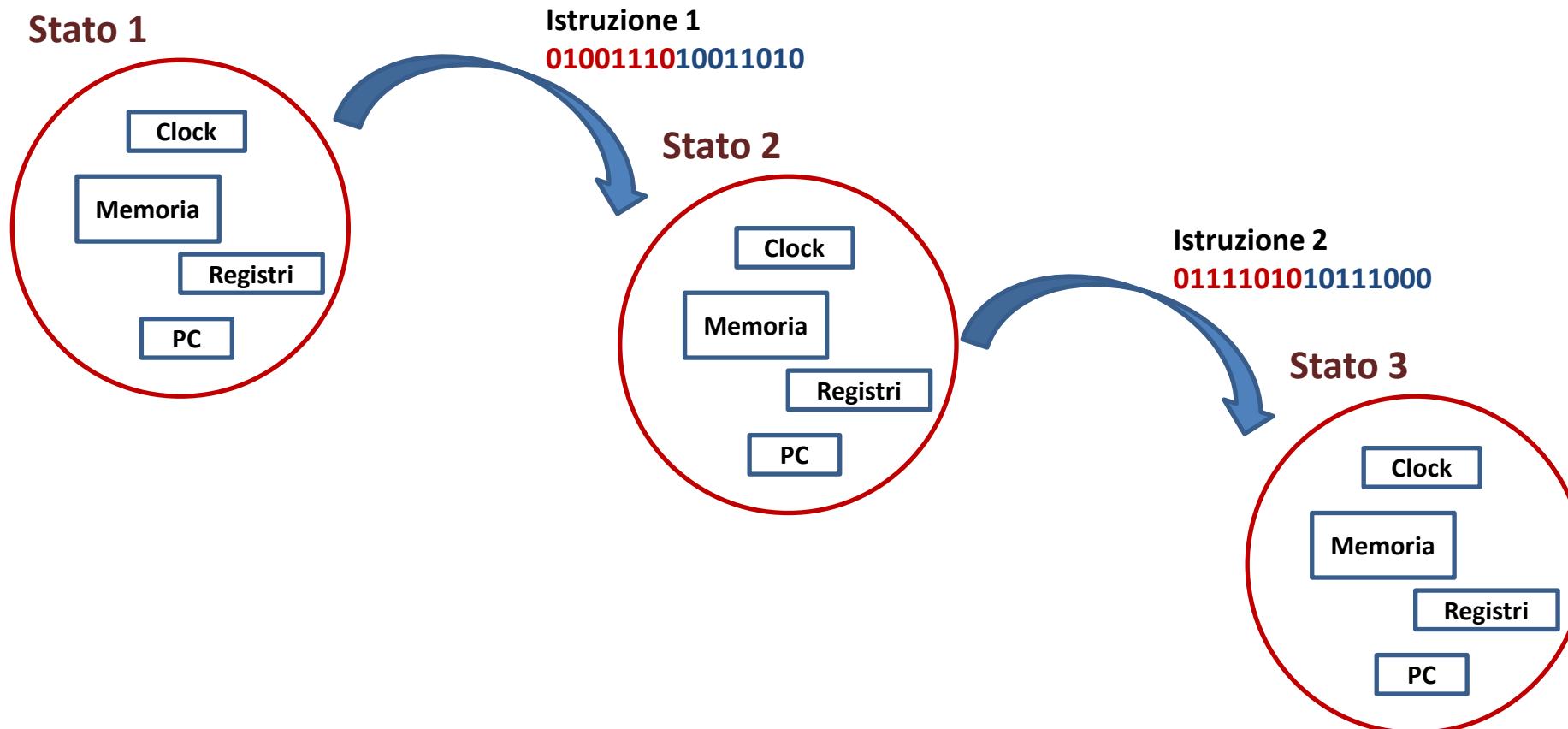
Lo **stato** del sistema è definito dai valori contenuti nelle locazioni di **memoria** insieme con i valori contenuti in alcuni **registri** dentro il processore stesso.

Ogni **istruzione** definisce in che modo lo stato deve cambiare e quale istruzione deve essere eseguita successivamente.



Microarchitettture come automi

Una microarchitettura può essere vista come un automa.



Variazioni di stato

- I registri, la memoria dati e la memoria istruzioni operano mediante logica combinatoria.
- Tutti i componenti effettuano la scrittura sul fronte alto del clock, cosicché lo stato del sistema cambia solo su un fronte del clock.
- Gli indirizzi, i dati ed il segnale di write enable devono essere impostati prima del fronte del clock e mantenuti immutati per un tempo superiore al ritardo di propagazione.
- Sia gli elementi di stato, che il microprocessore sono costituiti da logica combinatoria e da componenti sincronizzate dal clock. L'intero sistema è, quindi, sincrono e può essere visto come una complessa macchina a stati finiti o come l'insieme di macchine a stati finiti semplici, che interagiscono fra loro.

Microarchitetture a ciclo singolo

Saranno prese in considerazione tre diverse microarchitetture ARM, le quali differiscono fra loro principalmente per il modo in cui gli elementi di stato sono interconnessi.

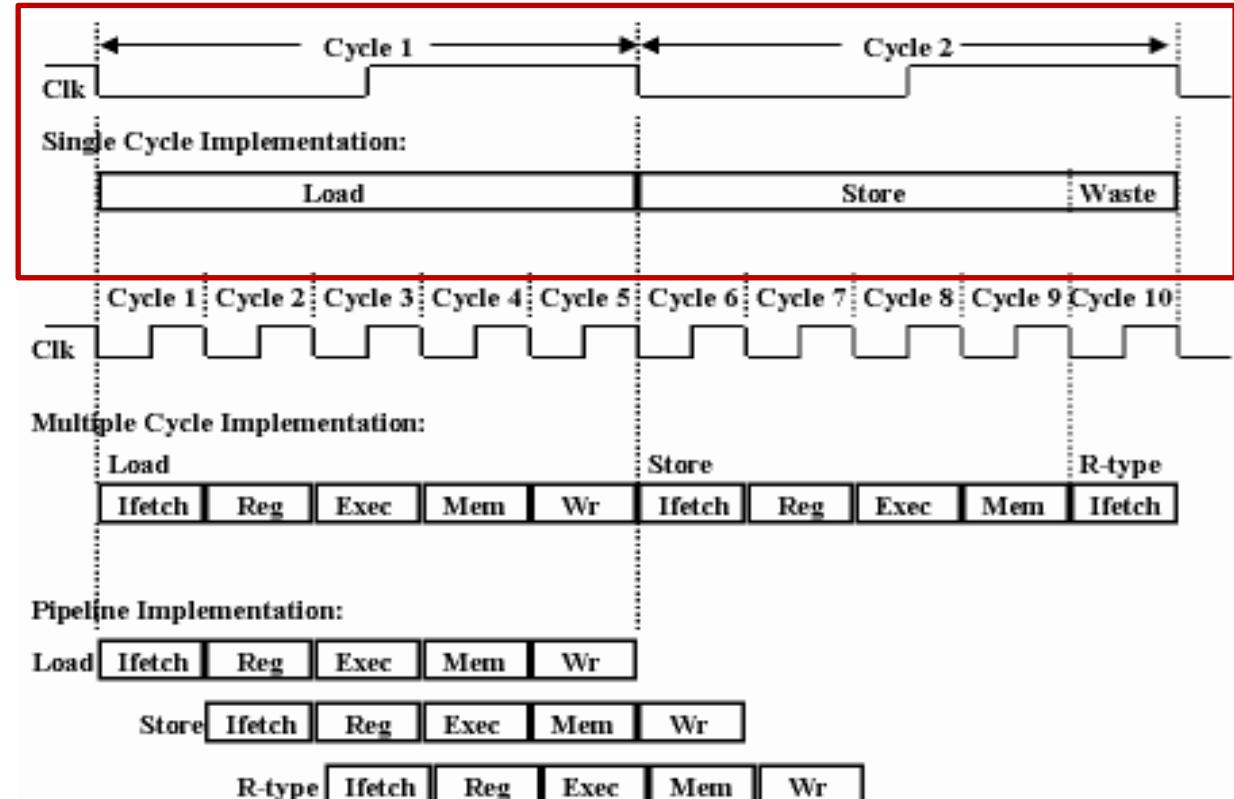
Ciclo singolo: l'intera istruzione è eseguita in un singolo ciclo.

Vantaggi:

- ▶ semplice da comprendere;
- ▶ unità di controllo molto semplice;
- ▶ non richiede stati non architetturali.

Svantaggi:

- ▶ tempo pari a quello dell'istruzione più lenta;
- ▶ memoria dati e memoria istruzioni separate;



Microarchitettture a ciclo multiplo

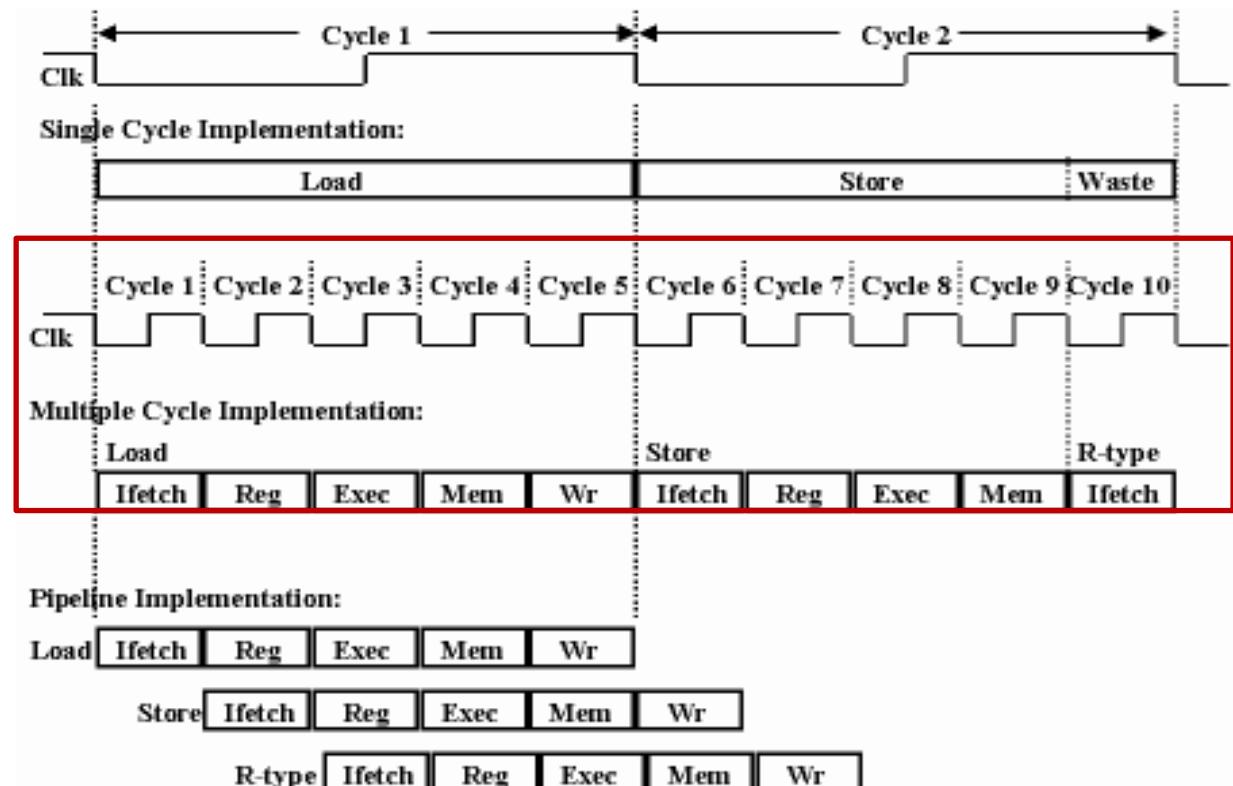
Ciclo multiplo: esegue una istruzione in più cicli brevi.

Vantaggi:

- ▶ riuso dei componenti;
- ▶ durata variabile delle istruzioni;
- ▶ non richiede la separazione delle memorie.

Svantaggi:

- ▶ richiede stati non architetturali;
- ▶ esegue una istruzione per volta.



Microarchitetture con pipeline

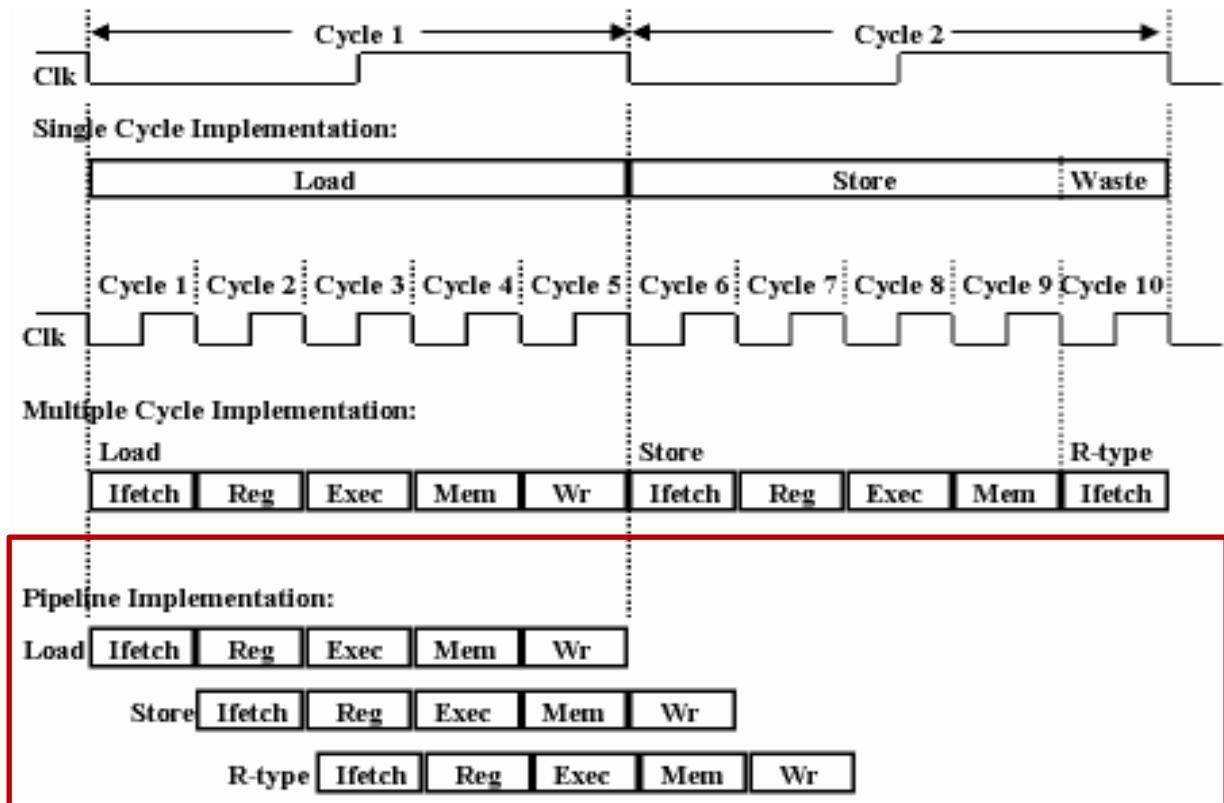
Pipelining: si applica al processore a ciclo singolo e ne migliora le performance.

Vantaggi:

- ▶ esegue più istruzioni contemporaneamente;
- ▶ si può accedere a dati e registri contemporaneamente.

Svantaggi:

- ▶ logica di controllo più complessa.
- ▶ richiede registri di pipeline.



Misura delle prestazioni

Ci sono molti modi per misurare le performance di un processore.

Una misura affidabile consiste nel valutare le prestazioni di tempo rispetto all'esecuzione di un insieme fissato di programmi, che prende il nome di benchmark.

CINT2006 (Integer Component of SPEC CPU2006):

Benchmark	Language	Application Area	Brief Description
400.perlbench	C	Programming Language	Derived from Perl V5.8.7. The workload includes SpamAssassin, MHonArc (an email indexer), and specdiff (SPEC's tool that checks benchmark outputs).
401.bzip2	C	Compression	Julian Seward's bzip2 version 1.0.3, modified to do most work in memory, rather than doing I/O.
403.gcc	C	C Compiler	Based on gcc Version 3.2, generates code for Opteron.
429.mcf	C	Combinatorial Optimization	Vehicle scheduling. Uses a network simplex algorithm (which is also used in commercial products) to schedule public transport.
445.gobmk	C	Artificial Intelligence: Go	Plays the game of Go, a simply described but deeply complex game.
456.hmmer	C	Search Gene Sequence	Protein sequence analysis using profile hidden Markov models (profile HMMs)
458.sjeng	C	Artificial Intelligence: chess	A highly-ranked chess program that also plays several chess variants.
462.libquantum	C	Physics / Quantum Computing	Simulates a quantum computer, running Shor's polynomial-time factorization algorithm.
464.h264ref	C	Video Compression	A reference implementation of H.264/AVC, encodes a videotream using 2 parameter sets. The H.264/AVC standard is expected to replace MPEG2
471.omnetpp	C++	Discrete Event Simulation	Uses the OMNet++ discrete event simulator to model a large Ethernet campus network.
473.astar	C++	Path-finding Algorithms	Pathfinding library for 2D maps, including the well known A* algorithm.
483.xalancbmk	C++	XML Processing	A modified version of Xalan-C++, which transforms XML documents to other document types.

Misura delle prestazioni

Il tempo di esecuzione è calcolato come:

$$\text{Tempo di esecuzione} = (\# \text{istruzioni}) \left(\frac{\text{cicli}}{\text{istruzione}} \right) \left(\frac{\text{secondi}}{\text{ciclo}} \right)$$

CPI: Cycles/instruction

clock period: seconds/cycle

IPC: instructions/cycle = IPC

Misura delle prestazioni

Il tempo di esecuzione è calcolato come:

$$\text{Tempo di esecuzione} = (\# \text{istruzioni}) \left(\frac{\text{cicli}}{\text{istruzione}} \right) \left(\frac{\text{secondi}}{\text{ciclo}} \right)$$

CPI: Cycles/instruction

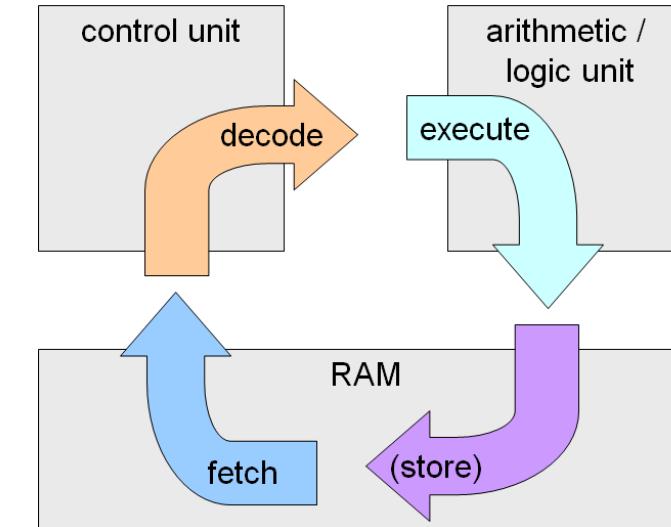
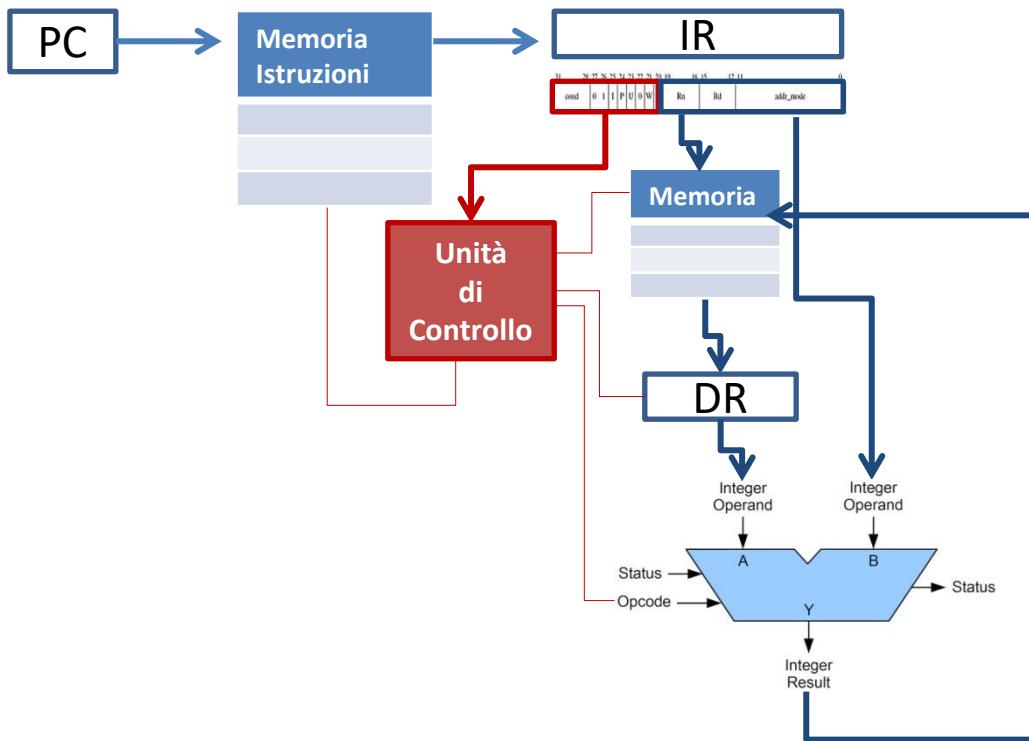
clock period: seconds/cycle

IPC: instructions/cycle = IPC

La frequenza di clock corrisponde
all'inverso del clock period

Microarchitettture: schema generale

Una istruzione ha un ciclo di vita che consta di quattro fasi principali



Progettazione

Una microarchitettura consta di due componenti che interagiscono fra loro:

► il **datapath**:

- determina il flusso dei dati durante l'esecuzione di una istruzione
- opera su parole dati e contiene strutture quali le memorie, i registri, l'ALU, e i multiplexer
- considereremo un datapath a 32 bit.

► l'**unità di controllo**:

- riceve l'istruzione corrente dal datapath ed "istruisce" il datapath su come eseguire tale istruzione.
- Produce i valori di selezione dei multiplexer, i segnali di abilitazione alla scrittura dei registri e della memoria.

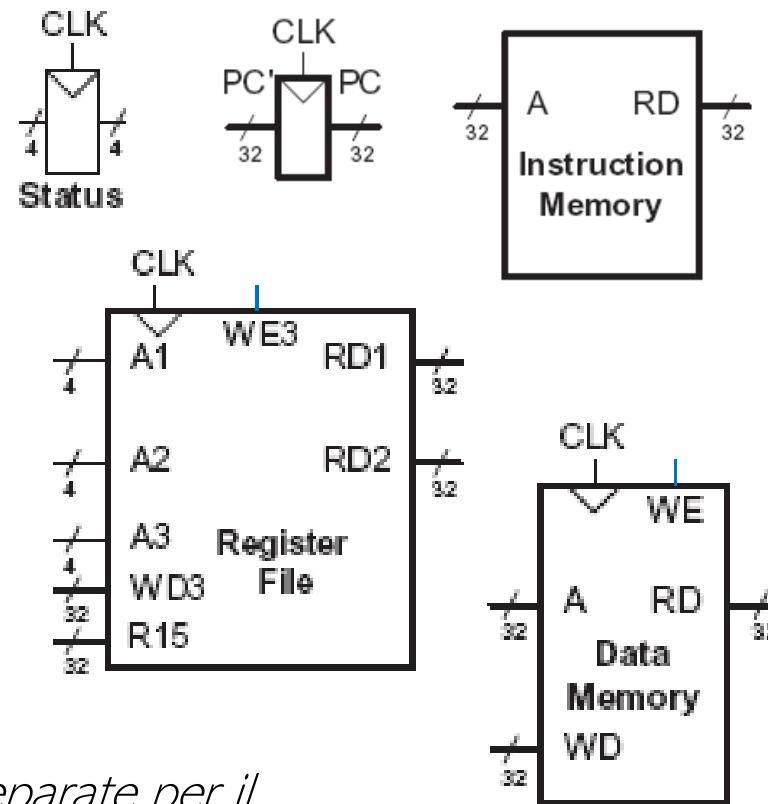
Procederemo in modo incrementale: Elementi di stato (program counter, registri, e registro di stato). Logica combinatoria fra gli elementi di stato. Gestione della memoria.

Progettazione

Il miglior modo di procedere nel processo di progettazione consiste nel considerare prima gli elementi di stato (program counter, registri, registro di stato) e aggiungere successivamente la logica combinatoria.

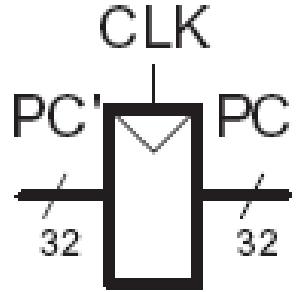
Suddivideremo la memoria in cinque elementi di stato:

- ▶ il program counter
- ▶ i registri (register file)
- ▶ il registro di stato (status register)
- ▶ la memoria istruzioni (instruction memory)
- ▶ la memoria dati (data memory).

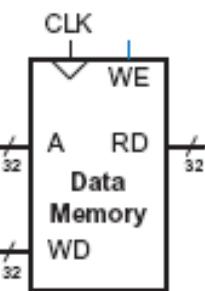
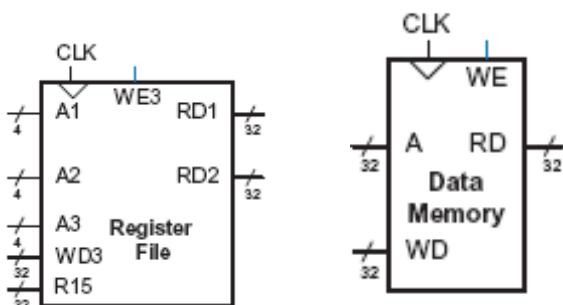
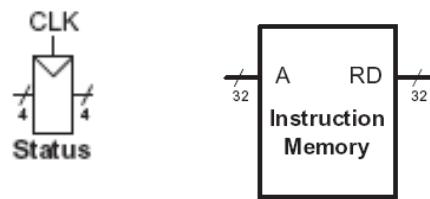


La memoria per le istruzioni e quella per i dati sono separate per il solo scopo di facilitare la comprensione.

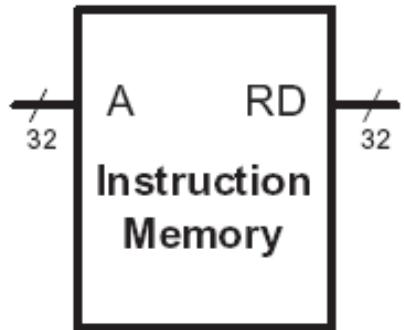
Program counter



Sebbene il PC sia logicamente parte del file registro, esso viene letto e scritto ad ogni ciclo indipendentemente dagli altri registri ed è quindi implementato come un registro autonomo 32-bit. La sua uscita, PC, indica l'indirizzo dell'istruzione corrente. Il suo ingresso, PC', indica l'indirizzo della successiva istruzione.



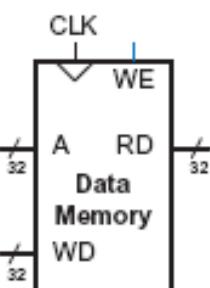
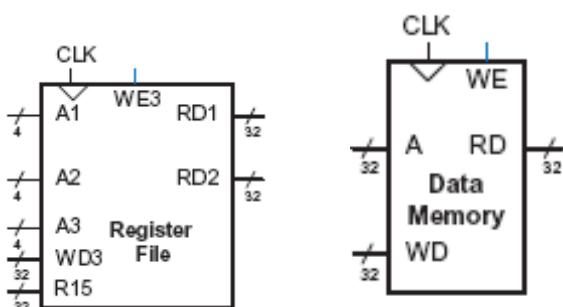
Instruction Memory



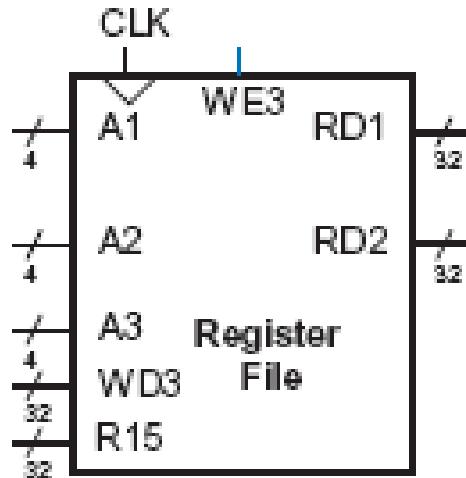
essa ha una singola porta di sola lettura (*semplificazione*).

A indica l'indirizzo di una istruzione che verrà riportata (*letta*) nel registro RD.

Considerando che in genere le memorie sono byte addressable ed essendo sia A che RD a 32 bit, qual'è il numero massimo di istruzioni che una architettura 32 bit supporta?



File register



consiste di 16 registri (R_0, \dots, R_{15}) a 32 bit

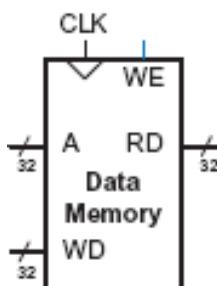
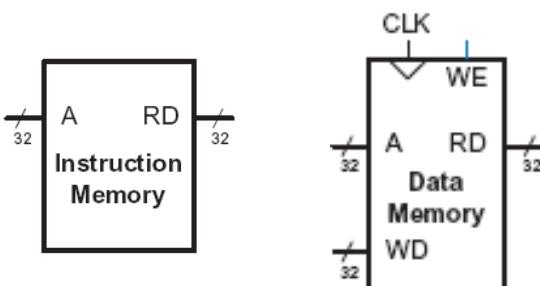
Dal punto di vista logico i registri non sono equivalenti, svolgono al livello architetturale funzioni diverse. In particolare:



R_{13} : stack pointer

R_{14} : link register

R_{15} : proveniente dal program counter



File register

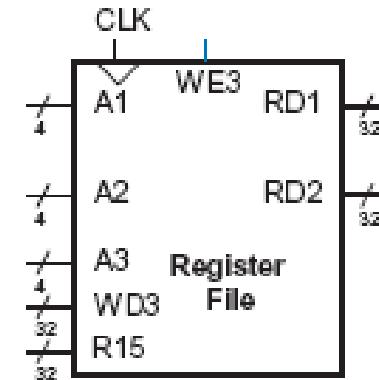
Esso ha due porte per la lettura **A1** e **A2** ed una per la scrittura **A3**.

Ciascuna porta di lettura ha un input di 4 bit ($2^4=16$), che specificano uno dei registri come operando, che viene letto nei registri **RD1** o **RD2**, rispettivamente.

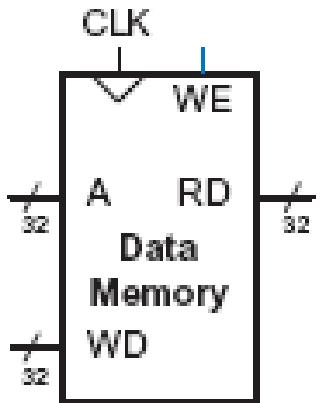
La porta di scrittura ha:

- ▶ un indirizzo di 4 bit **A3** (dove scrivere);
- ▶ un elemento di 32 bit **WD3** (cosa scrivere);
- ▶ un segnale di Write Enable (**WE3**).

Se **WE** è 1, il dato **WD3** è scritto nel registro specificato da **A3**.



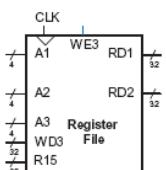
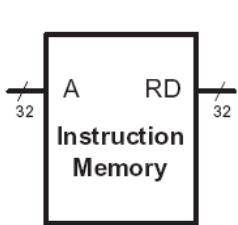
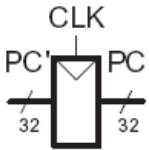
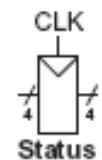
Data Memory



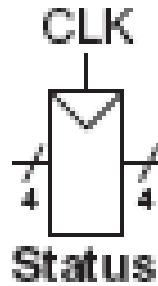
Il **Data Memory**: ha una singola porta di lettura/scrittura.

Quando il segnale **WE** (Write Enable) è attivo, essa scrive il dato **WD** nella cella puntata dall'indirizzo **A** durante il fronte alto del clock.

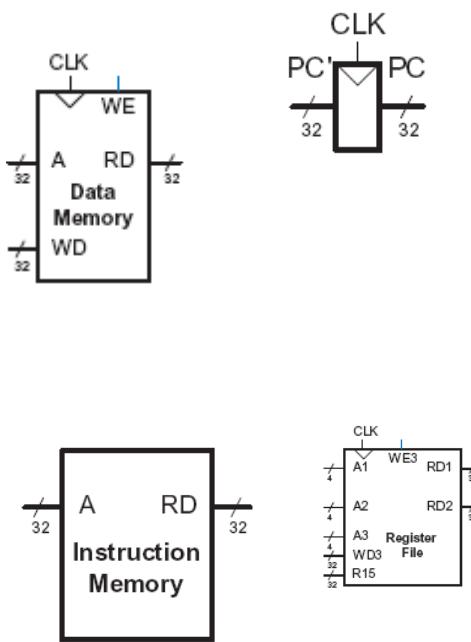
Quando il segnale **WE** (Write Enable) è 0, essa legge i dati nella cella puntata dall'indirizzo **A** durante il fronte alto del clock e li pone nel registro **RD**.



Status



Memorizza i flags **N,Z,C,V** fornite dall'ALU al fine di eseguire istruzioni condizionate



Flag	Name	Description
<i>N</i>	Negative	Instruction result is negative
<i>Z</i>	Zero	Instruction results in zero
<i>C</i>	Carry	Instruction causes an unsigned carry out
<i>V</i>	oVerflow	Instruction causes an overflow

Istruzioni di base

Al fine di facilitare la comprensione dell'architettura di un processore ARM, considereremo un limitato **set** di istruzioni:

- **Istruzioni di Data-processing:**
 - **ADD, SUB, AND, ORR**
 - Con registri e modalità di indirizzamento diretto, ma senza shifts
- **Instrukzioni di memoria:**
 - **LDR, STR**
 - with **positive immediate offset**
- **Istruzioni di salto:**
 - **B**

Indirizzi di memoria

- Ad ogni **byte** corrisponde un unico indirizzo
 - Una parola è di 32-bit = 4 bytes, quindi gli indirizzi delle parole incrementano di 4 in 4

Byte address				Word address			
• • •				• • •			
13 12 11 10				00000010			
F E D C				0000000C			
B A 9 8				00000008			
7 6 5 4				00000004			
3 2 1 0				00000000			
MSB				LSB			

- Indirizzo delle seconda parola = $2 \times 4 = 8$
 - Indirizzo della decima parola = $10 \times 4 = 40$

Istruzioni di memoria: load

Una istruzione che legge in memoria è detta **load**

- **Mnemonico:** LDR (*load register*)
- Vi sono differenti modi per eseguire un load, ad esempio:

LDR R0, [R1, #12]

,

indirizzo:

- *base address + offset*
- indirizzo = (R1 + 12)

risultato:

- Il dato memorizzato all'indirizzo (R1 + 12) viene *caricato* nel registro R0

Istruzioni di memoria: load

Esempio: Carica nel registro R3 la parola all'indirizzo 8

```
MOV R2, #0  
LDR R3, [R2, #8]
```

$$\text{indirizzo} = (R2 + 8) = 8$$

R3 = 0x01EE2842 (dopo il load)

Word address	Data				Word number
...
00000010	C D	1 9	A 6	5 B	Word 4
0000000C	4 0	F 3	0 7	8 8	Word 3
00000008	0 1	E E	2 8	4 2	Word 2
00000004	F 2	F 1	A C	0 7	Word 1
00000000	A B	C D	E F	7 8	Word 0

(b) Width = 4 bytes

Istruzioni di memoria: load

Vi sono anche altre segnature di LDR, ad esempio:

LDR R0, [R1, R2] LDR R0, [R1, R2, LSL #2]

$R0 \leftarrow \text{mem32}[R1 + R2]$ $R0 \leftarrow \text{mem32}[R1 + (R2 * 4)]$

Istruzioni di memoria: store

Una istruzione che scrive in memoria è detta **store**

- **Mnemonico:** STR (*store register*)
- Semantica speculare a LDR:

STR R0, [R1, #12]

mem32[R1 + 12] \leftarrow R0

- Segnature analoghe

STR R1, [R0, R2]

STR R0, [R1, R2, LSL #1]

mem32[R0 + R2] \leftarrow R1

mem32[R1 + (R2*2)] \leftarrow R0

Istruzioni di memoria: store

Esempio: Memorizza il valore di R7 nella parola di memoria 21.

Indirizzo di memoria = $4 \times 21 = 84 = 0x54$

MOV R5, #0

STR R7, [R5, #0x54]

L'offset può essere scritto in
decimale o esadecimale

Word address	Data	Word number
00000010	C D 1 9 A 6 5 B	Word 4
0000000C	4 0 F 3 0 7 8 8	Word 3
00000008	0 1 E E 2 8 4 2	Word 2
00000004	F 2 F 1 A C 0 7	Word 1
00000000	A B C D E F 7 8	Word 0

Width = 4 bytes

Varianti: LDRB, STRB

LDRB/STRB eseguono rispettivamente il load e lo store di un singolo byte

Come sono indirizzati i byte all'interno di una parola?

Little-endian: la numerazione dei byte inizia dal byte meno significativo

Big-endian: la numerazione dei byte inizia dal byte più significativo

Big-Endian				Little-Endian			
Byte Address	Word Address	Byte Address	Word Address	Byte Address	Word Address	Byte Address	Word Address
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
C	D	E	F	F	E	D	C
8	9	A	B	B	A	9	8
4	5	6	7	7	6	5	4
0	1	2	3	3	2	1	0
MSB		LSB		MSB		LSB	

Varianti: LDRB, STRB

LDRB/STRB eseguono rispettivamente il load e lo store di un singolo byte

Come sono indirizzati i byte all'interno di una parola?

Little-endian: la numerazione dei byte inizia dal byte meno significativo

Big-endian: la numerazione dei byte inizia dal byte più significativo

Big-Endian				Little-Endian			
Byte Address				Word Address			
	⋮			⋮	⋮		
C	D	E	F				
8	9	A	B	8	F	E	D
4	5	6	7	4	B	A	9
0	1	2	3	0	7	6	5
MSB		LSB		MSB		LSB	

IBM PowerPC Intel x86

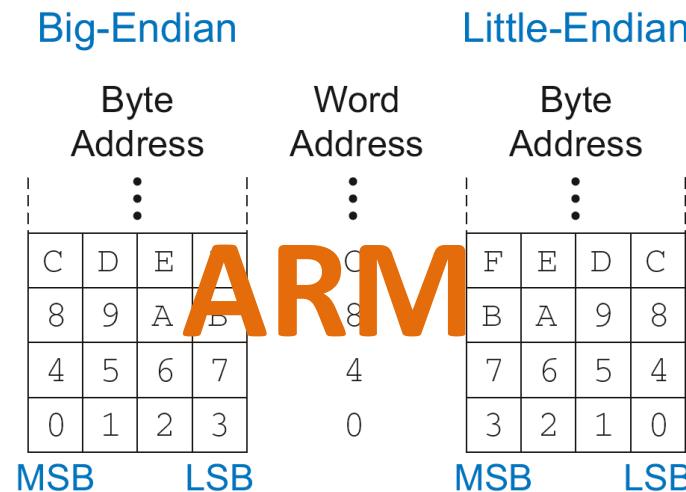
Varianti: LDRB, STRB

LDRB/STRB eseguono rispettivamente il load e lo store di un singolo byte

Come sono indirizzati i byte all'interno di una parola?

Little-endian: la numerazione dei byte inizia dal byte meno significativo

Big-endian: la numerazione dei byte inizia dal byte più significativo

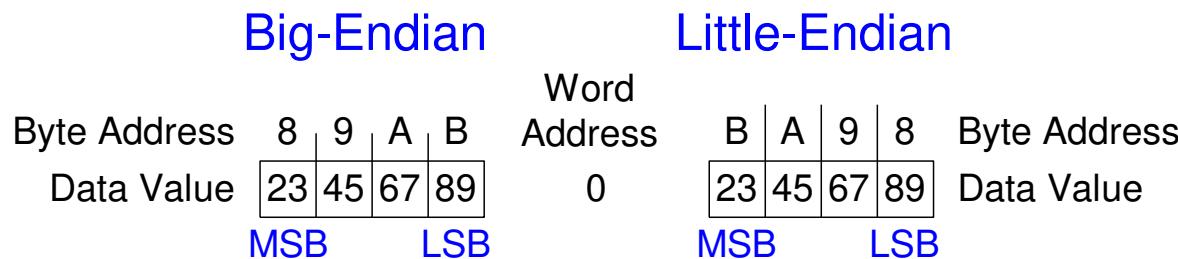


Varianti: LDRB, STRB

Esempio: si assuma che R2 *vale* 0 (0x00000000) e che si esegua l'istruzione

LDRB R7, [R2, #1]

Quale sarà il valore di R7 in un sistema little-endian/ big-endian?



Big-endian:

0x00000045

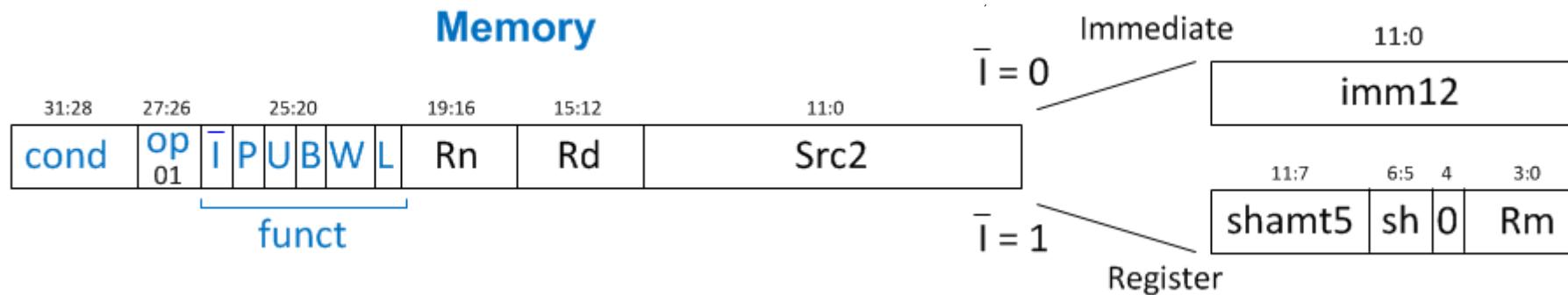
Little-endian:

0x00000067

Codifiche istruzioni di memoria

LDR, STR, LDRB, STRB

- ***op*** = 01_2
- ***Rn*** = registro *base address*
- ***Rd*** = destinazione (load), sorgente (store)
- ***Src2*** = offset: registro (*shiftato* opzionalmente) o immediate
- ***funct*** = 6 bit di controllo



Indicizzazione

ARM fornisce tre tipi di indicizzazione:

Offset:

L'indirizzo è calcolato sommando o sottraendo un offset al contenuto del registro di base.

Preindex

L'indirizzo viene calcolato come nel caso dell'offset e viene scritto nel registro di base;

Postindex

L'indirizzo corrisponde al contenuto del registro di base;

L'offset viene aggiunto o sottratto al contenuto del registro di base e riscritto nel registro di base

Table 6.4 ARM indexing modes

Mode	ARM Assembly	Address	Base Register
Offset	LDR R0, [R1, R2]	R1 + R2	Unchanged
Pre-index	LDR R0, [R1, R2]!	R1 + R2	R1 = R1 + R2
Post-index	LDR R0, [R1], R2	R1	R1 = R1 + R2

Indicizzazione

ARM fornisce tre tipi di indicizzazione:

Offset:

L'indirizzo è calcolato sommando o sottraendo un offset al contenuto del registro di base.

Preindex

L'indirizzo viene calcolato come nel caso dell'offset e viene scritto nel registro di base:

Postindex

L'indirizzo corrisponde al contenuto del registro di base;

L'offset viene aggiunto o sottratto al contenuto del registro di base e riscritto nel registro di base

Esempi

Offset: LDR R1, [R2, #4] ; R1 = mem[R2+4]

Preindex: LDR R3, [R5, #16] ! ; R3 = mem[R5+16]
; R5 = R5 + 16

Postindex: LDR R8, [R1], #8 ; R8 = mem[R1]
; R1 = R1 + 8

Campi di funct

Type of Operation

L	B	Instruction
0	0	STR
0	1	STRB
1	0	LDR
1	1	LDRB

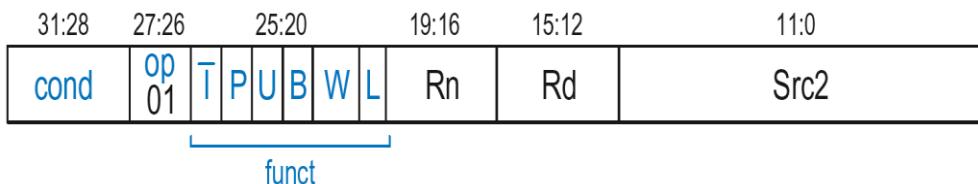
Indexing Mode

P	W	Indexing Mode
0	1	Not supported
0	0	Postindex
1	0	Offset
1	1	Preindex

Add/Subtract Immediate/Register Offset

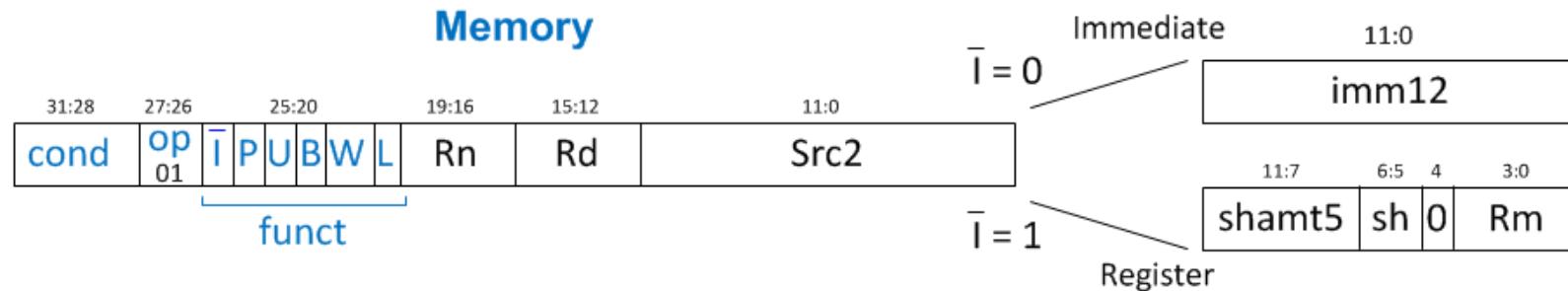
Value	T	U
0	Immediate offset in Src2	Subtract offset from base
1	Register offset in Src2	Add offset to base

Memory



Campo sh

Sh codifica il tipo di shift (i.e., >>, <<, >>>, ROR)



Shift Type	sh
LSL	00_2
LSR	01_2
ASR	10_2
ROR	11_2

Esempi

STR R11, [R5], #-26

- **Operation:** mem[R5] <= R11; R5 = R5 - 26
- **cond** = 1110_2 (14) for unconditional execution
- **op** = 01_2 (1) for memory instruction
- **funct** = 0000000_2 (0)
- **I** = 0 (immediate offset), **P** = 0 (postindex),
- **U** = 0 (subtract), **B** = 0 (store word), **W** = 0 (postindex),
- **L** = 0 (store)
- **Rd** = 11, **Rn** = 5, **imm12** = 26

Field Values

31:28	27:26	25:20	19:16	15:12	11:0
1110_2	01_2	0000000_2	5	11	26
cond	op	I	PUBWL	Rn	Rd
E	4	0	5	B	0 1 A

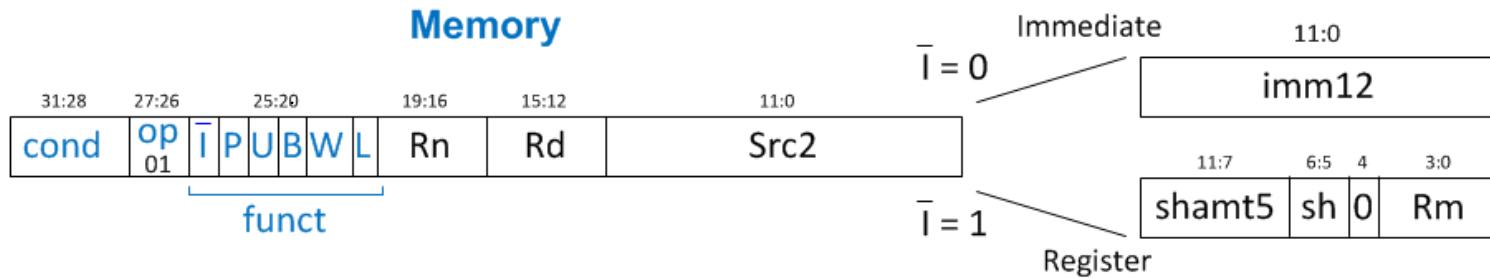
1110	01	000000	0101	1011	0000	0001	1010
E	4	0	5	B	0	1	A

Esempi

LDR R3, [R4, R5]

- **Operation:** $R3 \leftarrow \text{mem}[R4 + R5]$
- **cond** = 1110_2 (14) for unconditional execution
- **op** = 01_2 (1) for memory instruction
- **funct** = $1T1001_2$ (57)
- $I = 1$ (register offset), $P = 1$ (offset indexing),
- $U = 1$ (add), $B = 0$ (load word), $W = 0$ (offset indexing),
- $L = 1$ (load)
- $Rd = 3$, $Rn = 4$, $Rm = 5$ ($shamt5 = 0$, $sh = 00$)

1110 01 111001 0100 0011 00000 00 0 0101 = **0xE7943005**

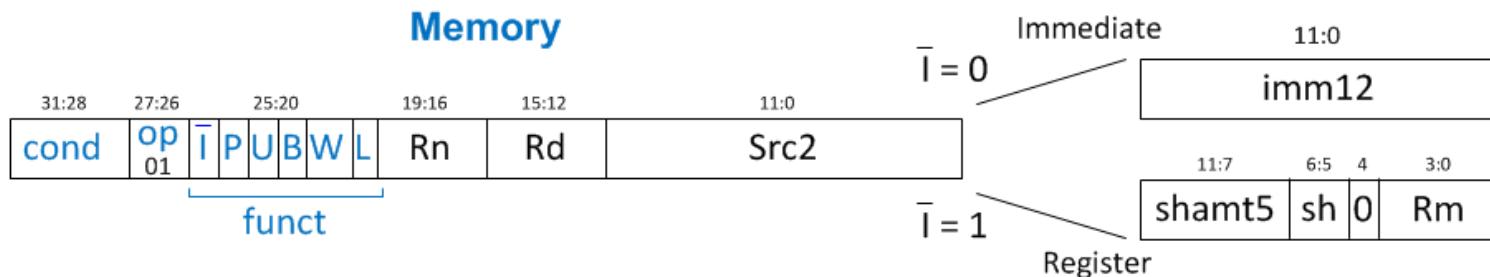


Esempi

STR R9, [R1, R3, LSL #2]

- **Operation:** $\text{mem}[\text{R1} + (\text{R3} \ll 2)] \leq \text{R9}$
- **cond** = 1110_2 (14) for unconditional execution
- **op** = 01_2 (1) for memory instruction
- **funct** = 111000_2 (0)
- **I** = 1 (register offset), **P** = 1 (offset indexing),
- **U** = 1 (add), **B** = 0 (store word), **W** = 0 (offset indexing),
- **L** = 0 (store)
- **Rd** = 9, **Rn** = 1, **Rm** = 3, **shamt** = 2, **sh** = 00_2 (LSL)

1110 01 111000 0001 1001 00010 00 0 0011 = **0xE7819103**



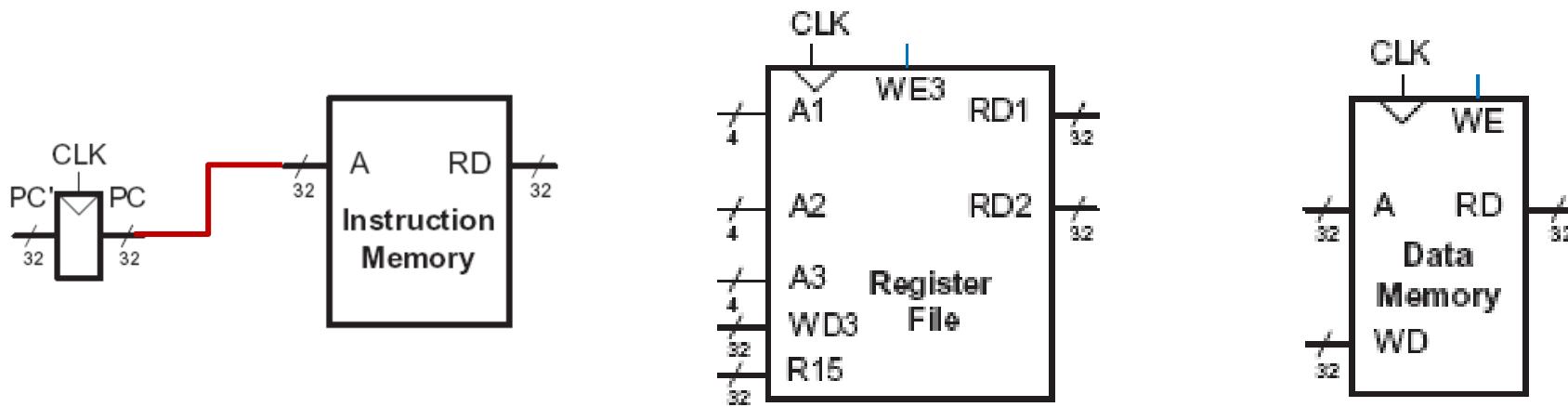
Datapath LDR

LDR Rd, [Rn, imm12]

Il **PC** contiene l'indirizzo dell'istruzione da eseguire.

Il primo passo è quello di leggere questa istruzione dalla memoria istruzioni, per cui il PC viene collegato all'indirizzo di ingresso della memoria di istruzioni.

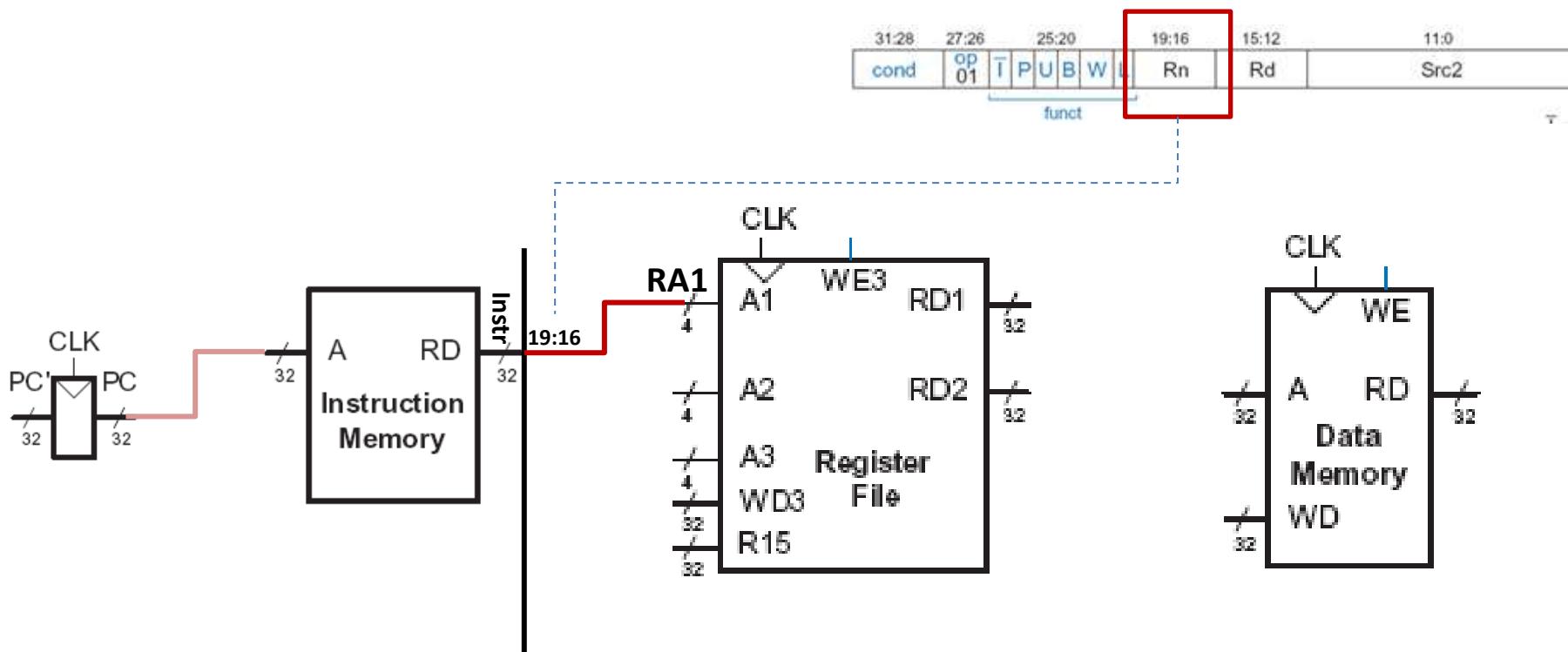
L'istruzione a 32 bit viene letta ed è rappresentata dall'etichetta **Instr**.



Datapath LDR

Il passo successivo è quello di leggere il registro contenente l'indirizzo di base.
(L'indirizzo di) questo registro è specificato nel campo **Rn** dell'istruzione,
Instr_{19:16}

Questi bit vengono collegati all'ingresso indirizzo di una delle porte del file register (**A1**). Il register file *legge* il valore di registro in **RD1**.

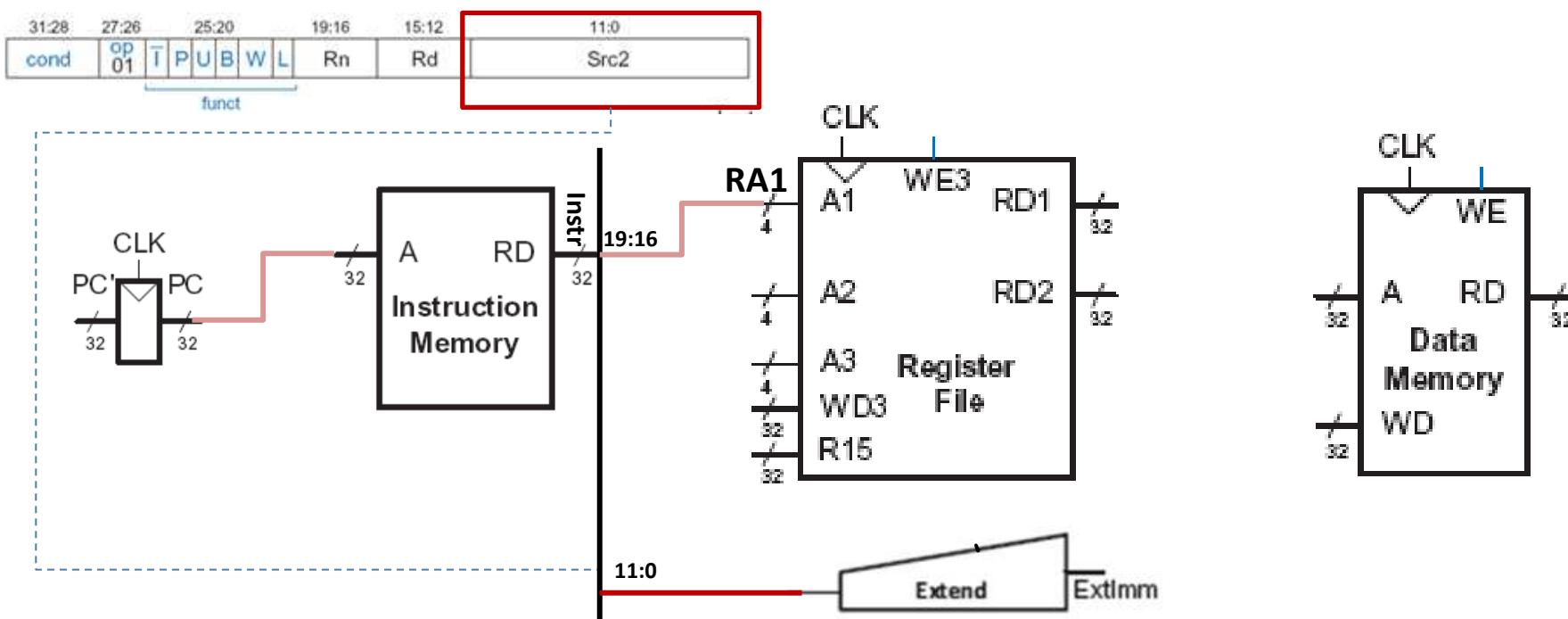


Datapath LDR

L'istruzione **LDR** richiede anche un **offset**, il quale è memorizzato nell'istruzione stessa e corrisponde ai bit $\text{Instr}_{11:0}$.

L'offset è un valore senza segno, quindi deve essere esteso a 32 bit.

Il valore a 32 bit (**ExtImm**) è tale che $\text{ExtImm}_{31:12} = 0$ e $\text{ExtImm}_{11:0} = \text{Instr}_{11:0}$.

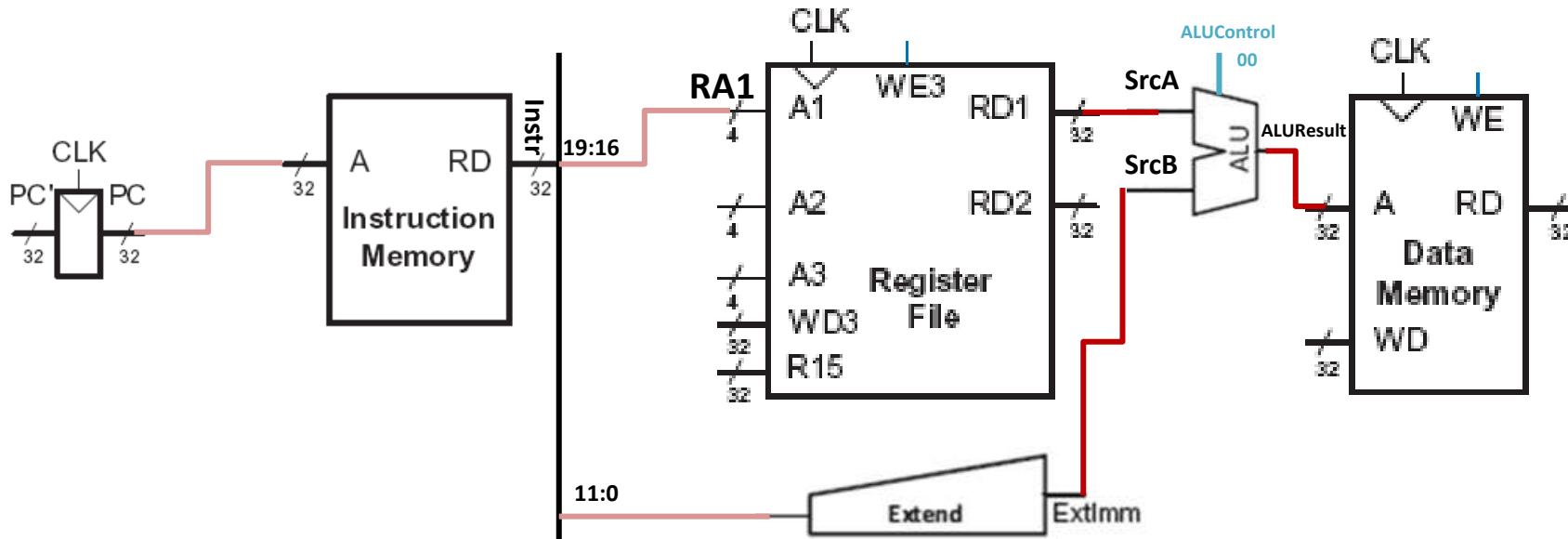


Datapath LDR

Il processore deve aggiungere l'**indirizzo di base** all'offset per trovare l'indirizzo di memoria a cui leggere. La somma è effettuata per mezzo di una **ALU**.

La **ALU** riceve due operandi (**srcA** e **srcB**). **srcA** proviene dal register file, mentre **srcB** in questo esempio proviene da **ExtImm**. Inoltre, il segnale a 2-bit **ALUControl** specifica l'operazione: una somma (indicata con 00), se il valore del bit U è uguale a 1, una sottrazione (indicata 01), viceversa.

La **ALU** genera un valore a 32 bit **ALUResult**, che viene inviato alla memoria dati come indirizzo di lettura.

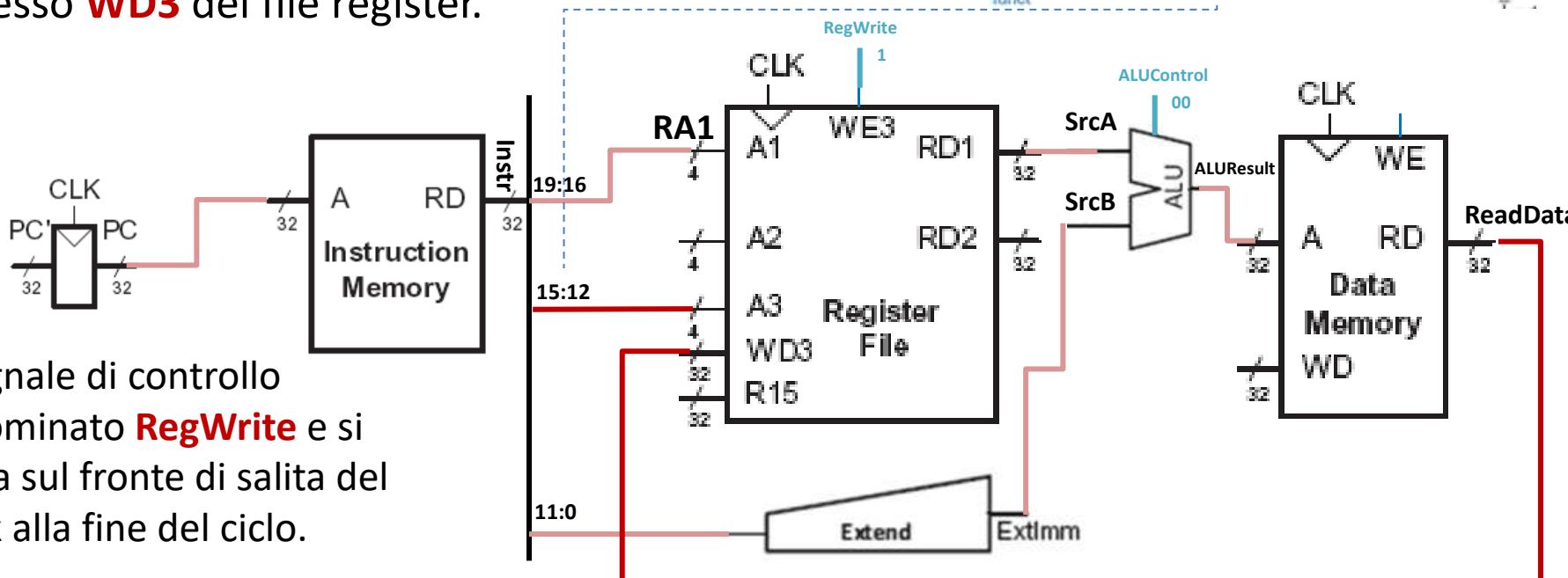
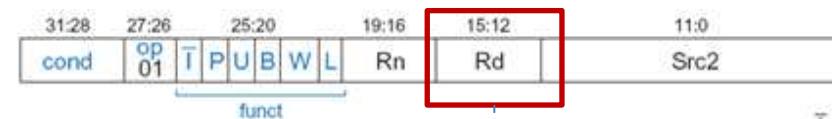


Datapath LDR

I dati vengono letti dalla memoria dati sul bus **ReadData** e poi sono scritti nel registro destinazione alla fine del ciclo.

il registro di destinazione per l'istruzione **LDR** è specificato nel campo **Rd**, **Instr_{15:12}**, che è collegato all'indirizzo di ingresso **A3**.

Il bus **ReadData** è collegato alla porta di ingresso **WD3** del file register.

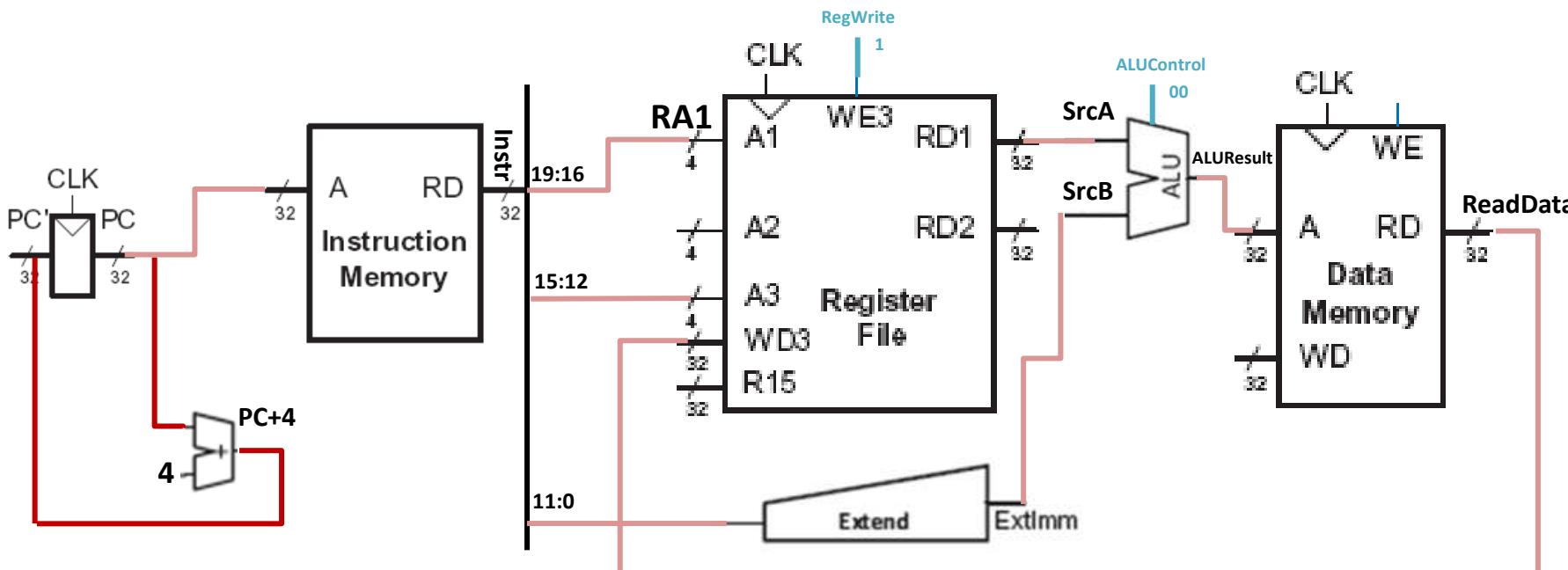


Datapath LDR

Contemporaneamente il processore deve calcolare l'indirizzo della successiva istruzione **PC'**.

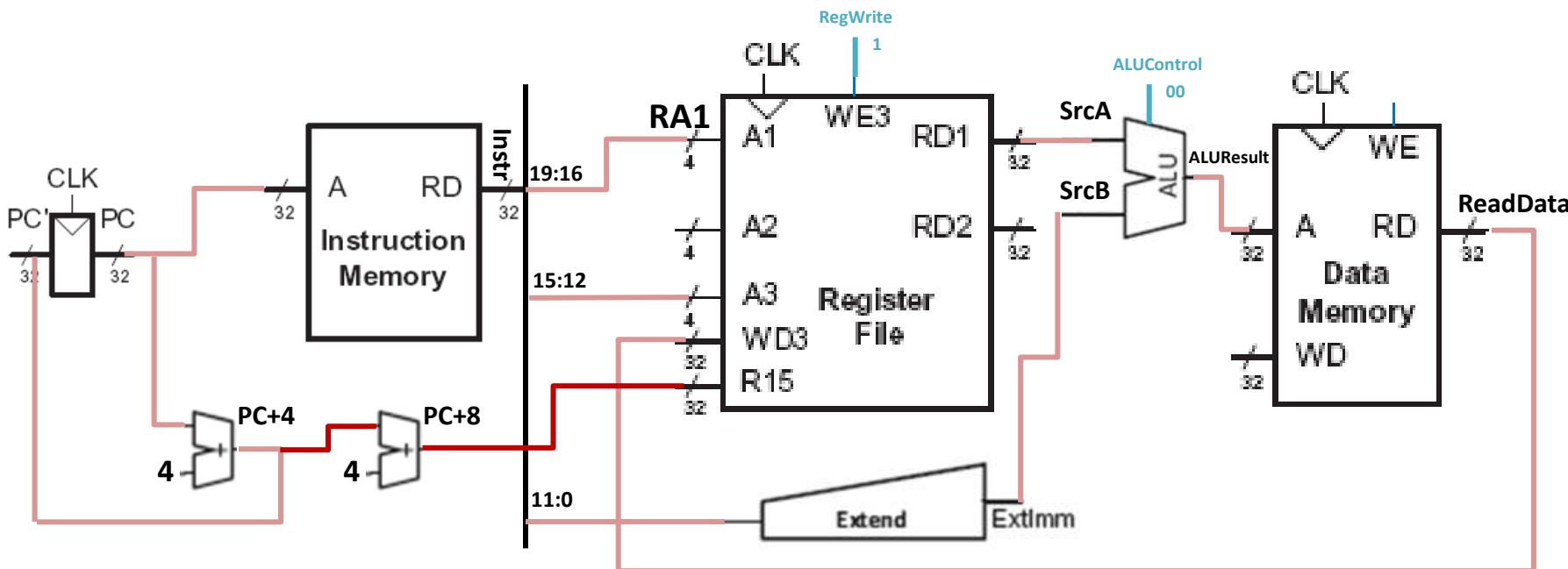
Le istruzioni sono a 32 bit (4 byte), quindi l'istruzione successiva è a **PC + 4**.

Si utilizza un **sommatore** per incrementare il **PC** di 4. Il nuovo indirizzo viene scritto nel contatore di programma sul successivo fronte di salita del clock.



Datapath LDR

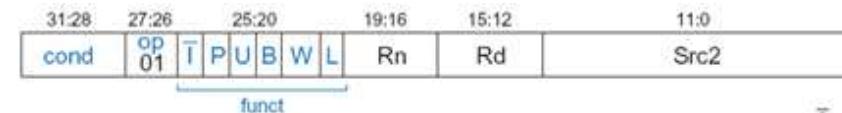
Nelle architetture ARM il registro **R15** contiene il valore **PC+8**, per cui è necessario un ulteriore **sommatore** (+4), la cui uscita sia collegata all'ingresso di **R15**.



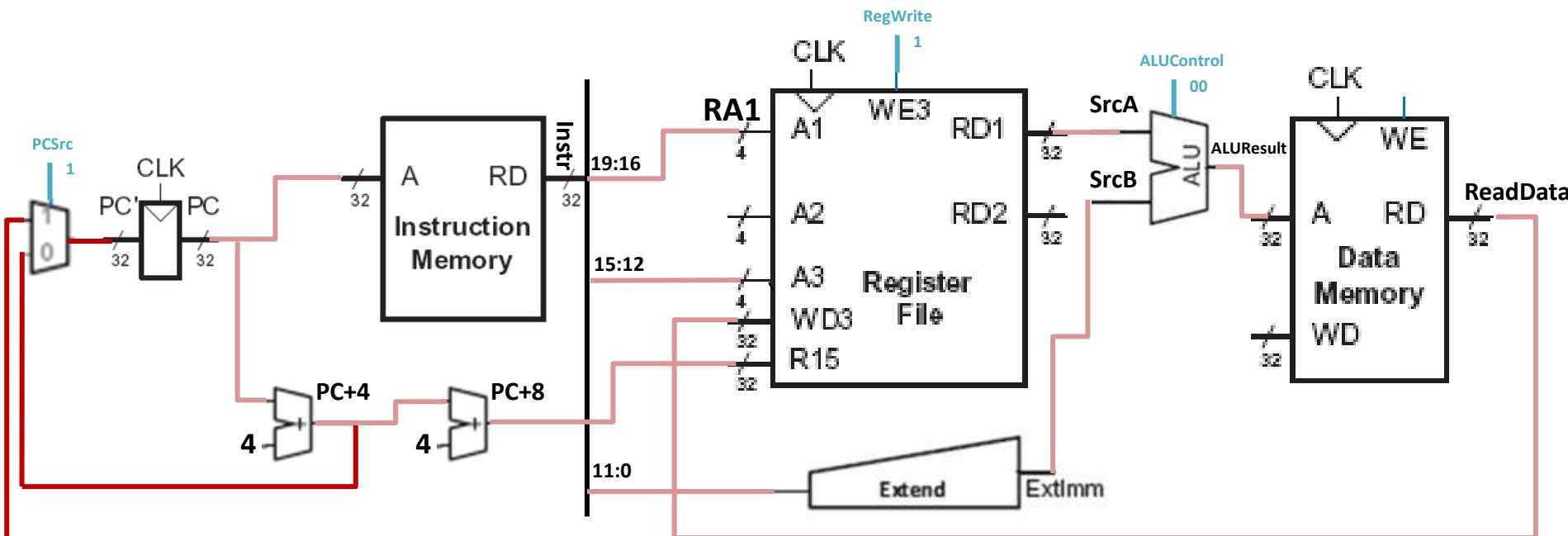
Datapath LDR

Infine, trattandosi di uno stored program paradigm, l'istruzione successiva potrebbe essere letta anche dalla memoria e quindi corrispondere al contenuto di **ReadData**. Un **multiplexer**, permette di selezionare fra:

- ▶ 0 – **PCPlus4**
- ▶ 1 – **ReadData**.

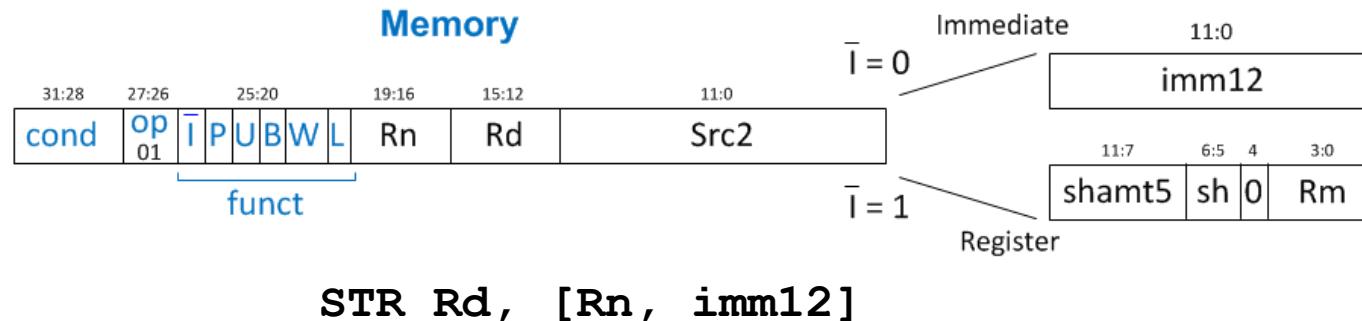


Il segnale di controllo associato al multiplexer è **PCSrc**.



Datapath STR

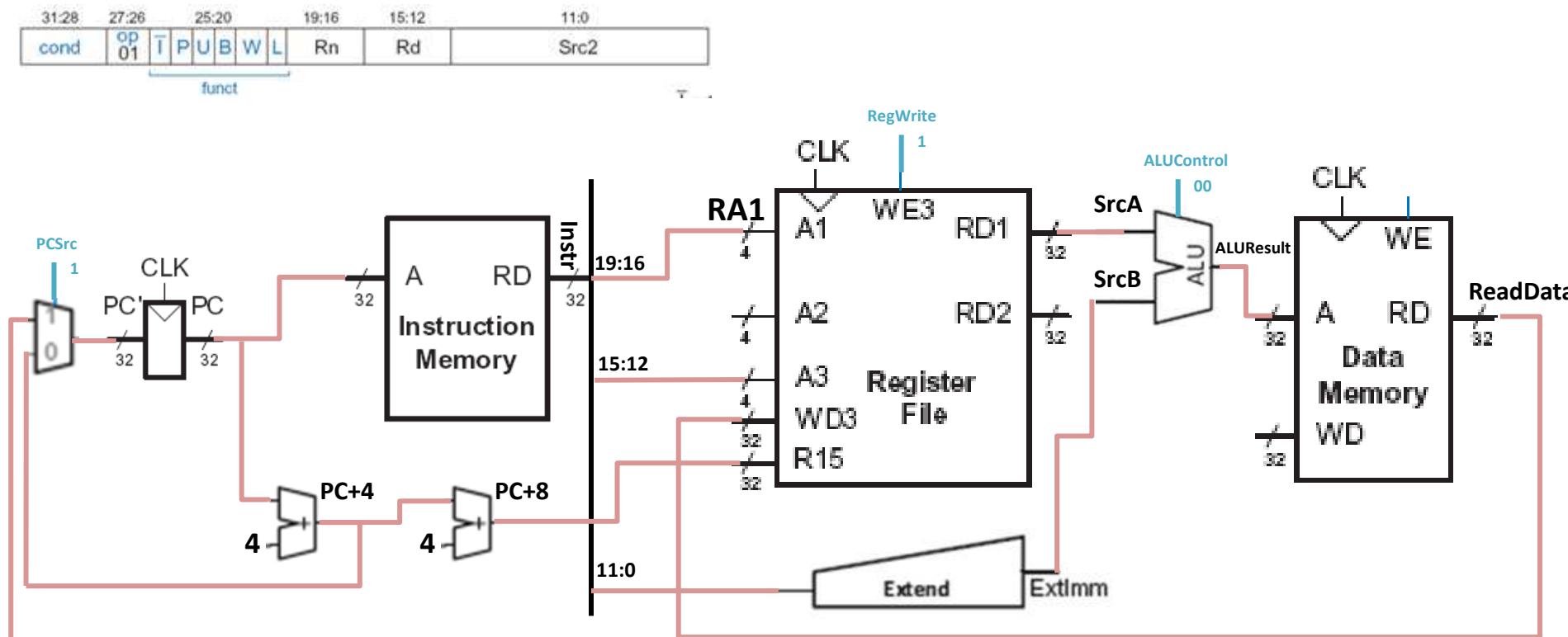
L'istruzione **STR** scrive una parola di 32 bit contenuta in un registro nella memoria centrale. Il modo in cui questa operazione viene effettuata dipende dalla politica di indirizzamento specificata.



Datapath STR

Estendiamo il **datapath** in modo da poter gestire anche l'istruzioni **STR**.

Come LDR, **STR** legge un indirizzo di base dalla porta **A1** del register file e completa l'immediate. L'**ALU** aggiunge l'indirizzo di base alla costante per trovare l'indirizzo di memoria. Tutte queste funzioni sono già supportate nel datapath.

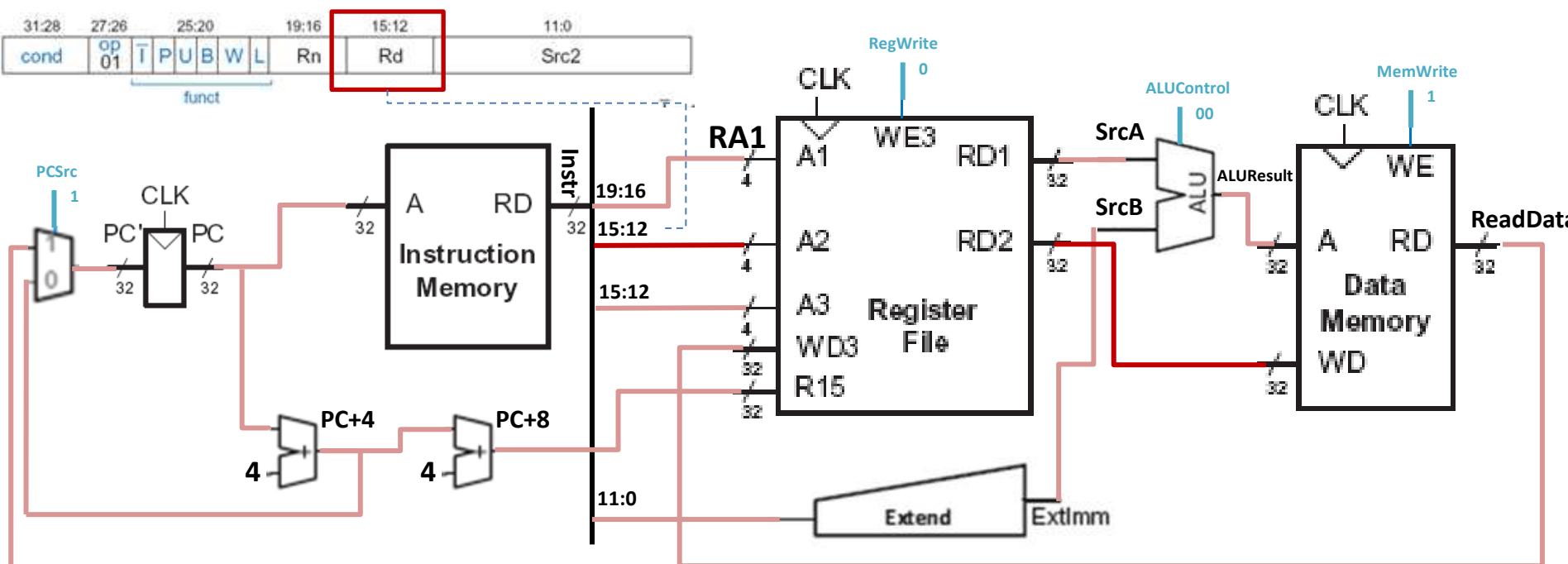


Datapath STR

Il registro è specificato nel campo **Rd**, **Instr_{15:12}**, che è collegato alla porta **A2** del register file.

Il valore del registro viene letto sulla porta **RD2**, che è collegata alla porta dati di scrittura (**WD**) della memoria dati.

L'abilitazione del segnale di scrittura **WE** è controllato da **MemWrite**, il quale è 1 se i dati devono essere scritti in memoria.

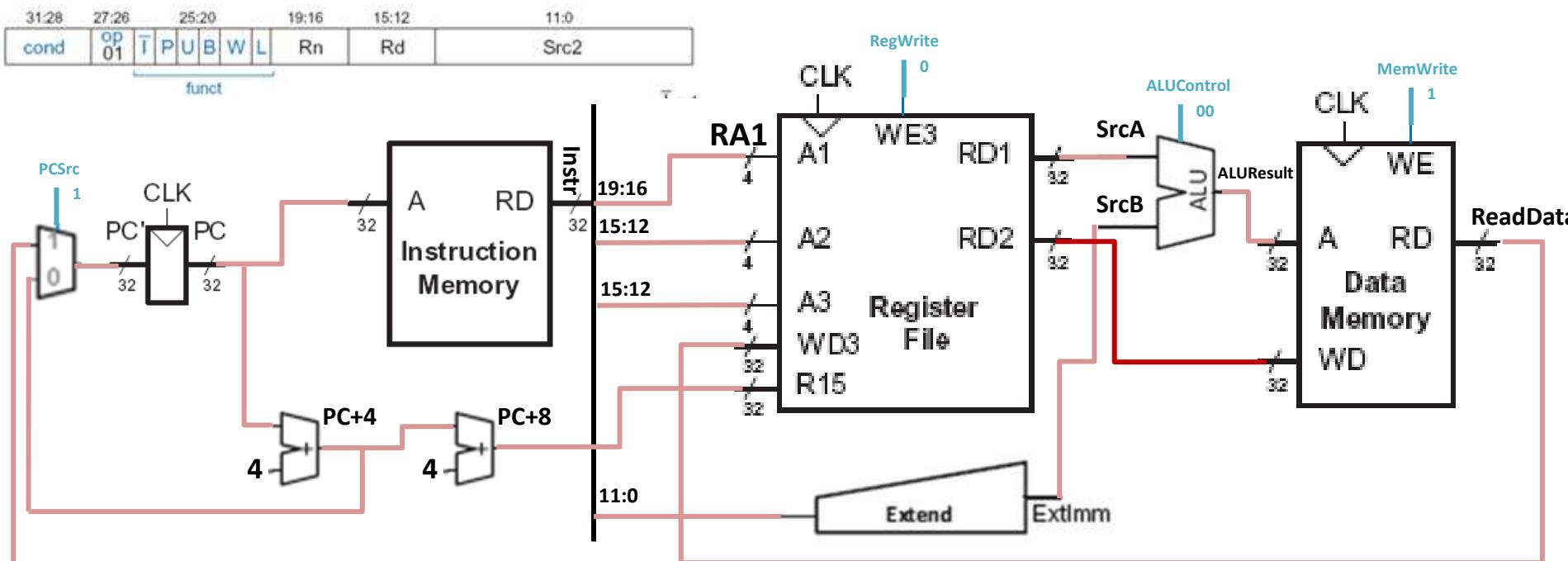


Datapath STR

Il segnale **ALUControl** deve essere impostato a **00** per sommare l'indirizzo di base e l'offset.

Il segnale **RegWrite** è impostato a **0**, perché nulla deve essere scritto nel register file.

Si noti che, pur essendo **Rd** associato anche a A3 e **ReadData** a WD3, questo non produce effetti sul File register, essendo **RegWrite** impostato a **0**.



Istruzioni di data processing

MOV	Move a 32-bit value	MOV Rd,n	Rd = n
MVN	Move negated (logical NOT) 32-bit value	MVN Rd,n	Rd = ~n
ADD	Add two 32-bit values	ADD Rd,Rn,n	Rd = Rn+n
ADC	Add two 32-bit values and carry	ADC Rd,Rn,n	Rd = Rn+n+C
SUB	Subtract two 32-bit values	SUB Rd,Rn,n	Rd = Rn-n
SBC	Subtract with carry of two 32-bit values	SBC Rd,Rn,n	Rd = Rn-n+C-1
RSB	Reverse subtract of two 32-bit values	RSB Rd,Rn,n	Rd = n-Rn
RSC	Reverse subtract with carry of two 32-bit values	RSC Rd,Rn,n	Rd = n-Rn+C-1
AND	Bitwise AND of two 32-bit values	AND Rd,Rn,n	Rd = Rn AND n
ORR	Bitwise OR of two 32-bit values	ORR Rd,Rn,n	Rd = Rn OR n
EOR	Exclusive OR of two 32-bit values	EOR Rd,Rn,n	Rd = Rn XOR n
BIC	Bit clear. Every '1' in second operand clears corresponding bit of first operand	BIC Rd,Rn,n	Rd = Rn AND (NOT n)
CMP	Compare	CMP Rd,n	Rd=n & change flags only
CMN	Compare Negative	CMN Rd,n	Rd+n & change flags only
TST	Test for a bit in a 32-bit value	TST Rd,n	Rd AND n, change flags
TEQ	Test for equality	TEQ Rd,n	Rd XOR n, change flags

MUL	Multiply two 32-bit values	MUL Rd,Rm,Rs	Rd = Rm*Rs
MLA	Multiple and accumulate	MLA Rd,Rm,Rs,Rn	Rd = (Rm*Rs)+Rn

Istruzioni logiche

- AND
- ORR
- EOR (**XOR**)
- BIC (**Bit Clear**)
- MVN (**MoVe and NOT**)

Source registers				
R1	0100 0110	1010 0001	1111 0001	1011 0111
R2	1111 1111	1111 1111	0000 0000	0000 0000

Assembly code	Result			
AND R3, R1, R2	R3	0100 0110	1010 0001	0000 0000
ORR R4, R1, R2	R4	1111 1111	1111 1111	1111 0001
EOR R5, R1, R2	R5	1011 1001	0101 1110	1111 0001
BIC R6, R1, R2	R6	0000 0000	0000 0000	1011 0111
MVN R7, R2	R7	0000 0000	0000 0000	1111 1111

Istruzioni logiche

- Le istruzioni AND or BIC sono utili per **mascherare** bit
 - **Esempio:** Maschera tutti i bit eccetto il byte meno significativo
 - $0xF234012F \text{ AND } 0x000000FF = 0x0000002F$
 - $0xF234012F \text{ BIC } 0xFFFFF00 = 0x0000002F$
- L'istruzione ORR: è utile per **combinare** bit fields
 - **Esempio:** combina 0xF2340000 con 0x000012BC:
 - $0xF2340000 \text{ ORR } 0x000012BC = 0xF23412BC$

Istruzioni di shifting

- LSL: logical shift left
 - **Esempio:** LSL R0, R7, #5 ; R0=R7 << 5
- LSR: logical shift right
 - **Esempio:** LSR R3, R2, #31 ; R3=R2 >> 31
- ASR: arithmetic shift right
 - **Esempio:** ASR R9, R11, #4 ; R9=R11 >>> 4
- ROR: rotate right
 - **Esempio:** ROR R8, R1, #3 ; R8=R1 ROR 3
- Lo shift aritmetico a sinistra si comporta come lo shift logico
- Non vi è un'istruzione per la rotazione a sinistra: la rotazione a sinistra di **N** posizioni corrisponde a una rotazione a destra di 32- **N** posizioni
- Lo shift di un valore a sinistra di **N** posizioni è equivalente a moltiplicare tale valore per 2^N . Analogamente, uno shift aritmetico aritmetico a destra di un valore **N** è equivalente a dividere per 2^N .

Esempi istruzioni di shifting

Source register				
R5	1111 1111	0001 1100	0001 0000	1110 0111

Assembly Code		Result			
LSL R0, R5, #7	R0	1000 1110	0000 1000	0111 0011	1000 0000
LSR R1, R5, #17	R1	0000 0000	0000 0000	0111 1111	1000 1110
ASR R2, R5, #3	R2	1111 1111	1110 0011	1000 0010	0001 1100
ROR R3, R5, #21	R3	1110 0000	1000 0111	0011 1111	1111 1000

Source registers				
R8	0000 1000	0001 1100	0001 0110	1110 0111
R6	0000 0000	0000 0000	0000 0000	0001 0100

Assembly code		Result			
LSL R4, R8, R6	R4	0110 1110	0111 0000	0000 0000	0000 0000
ROR R5, R8, R6	R5	1100 0001	0110 1110	0111 0000	1000 0001

Istruzioni di moltiplicazione

- **MUL:** moltiplicazione 32×32 -bit, risultato 32-bit
 - MUL R1, R2, R3
 - **Risultato:** $R1 = (R2 \times R3)_{31:0}$
- **UMULL:** moltiplicazione unsigned 32×32 -bit, risultato 64-bit
 - UMULL R1, R2, R3, R4
 - **Risultato:** $\{R1, R4\} = R2 \times R3$ ($R2, R3$ unsigned)
- **SMULL:** moltiplicazione signed 32×32 -bit, risultato 64-bit
 - SMULL R1, R2, R3, R4
 - **Risultato:** $\{R1, R4\} = R2 \times R3$ ($R2, R3$ signed)

Formato istruzioni di data processing

Il formato delle istruzioni di **data-processing** è il più comune.

Il primo operando sorgente è un registro. Il secondo operando sorgente può essere una costante o un registro. La destinazione è un registro.

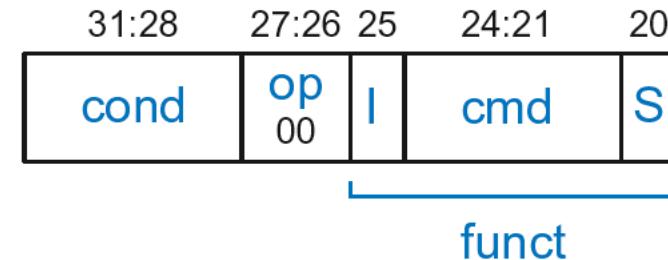
- **Operandi:**
 - **Rn:** primo registro sorgente
 - **Src2:** secondo registro sorgente o immediate
 - **Rd:** registro destinazione
- **Campi di controllo:**
 - **cond:** specifica l'esecuzione condizionale in base ai flag
 - **op:** 00 è l'opcode per istruzioni di data processing
 - **funct:** specifica il tipo di funzione da eseguire

Data-processing

31:28	27:26	25:20	19:16	15:12	11:0
cond	op	funct	Rn	Rd	Src2
4 bits	2 bits	6 bits	4 bits	4 bits	12 bits

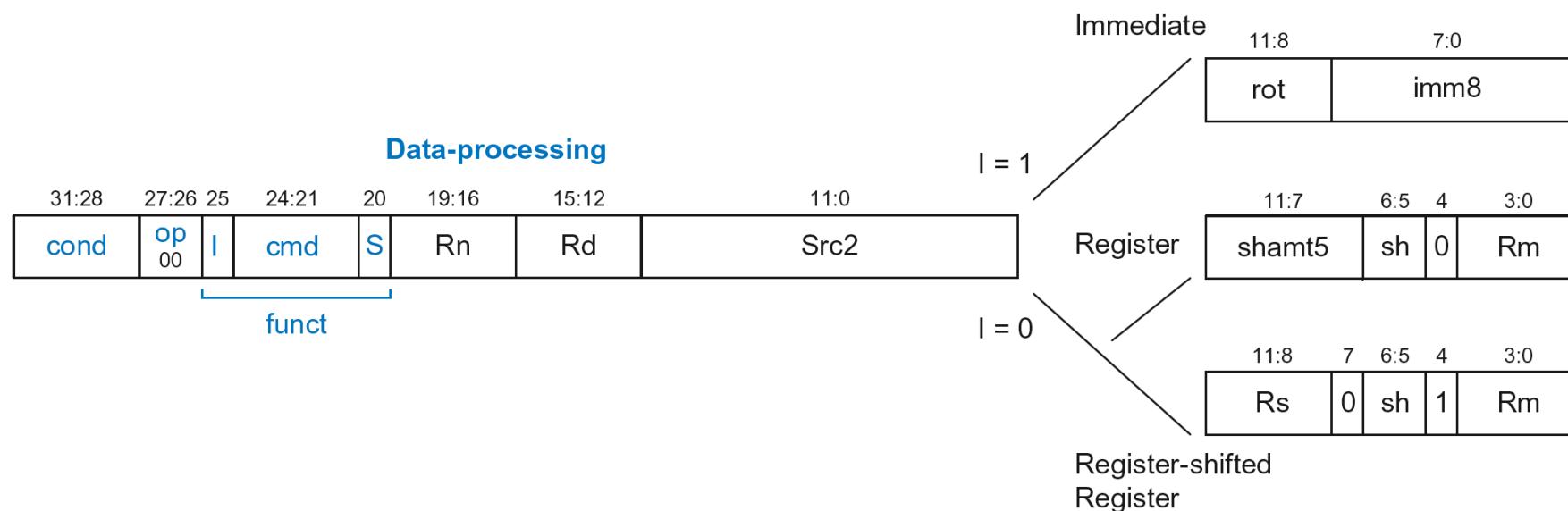
Formato istruzioni di data processing

- **cmd:** indica l'istruzione specifica da eseguire
 - **cmd** = 0100_2 sta per ADD
 - **cmd** = 0010_2 sta per SUB
- **I-bit**
 - **I** = 0: *Src2* è un registro
 - **I** = 1: *Src2* è un immediate
- **S-bit:** quando è 1 allora vengono aggiornate le condition flags
 - **S** = 0: SUB R0, R5, R7
 - **S** = 1: ADDS R8, R2, R4 or CMPS R3, #10



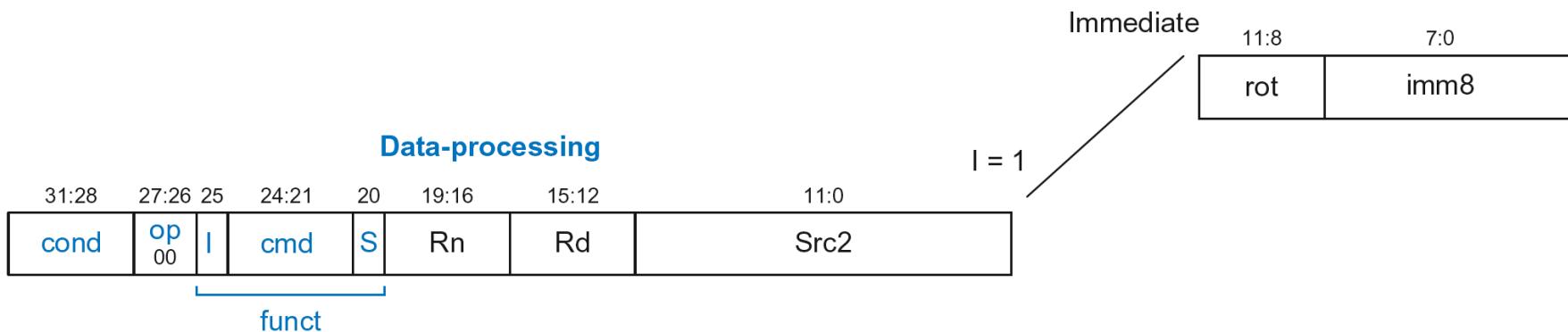
Formato istruzioni di data processing

- *Src2* può essere:
 - Immediate
 - Registro
 - Registro “shiftato” da un altro registro



Formato immediate

- Un immediate è codificato come segue:
 - $imm8$: 8-bit unsigned immediate
 - rot : 4-bit rotation value
- Da cui si ricava una costante a 32-bit :
 $imm8 \text{ ROR } (rot \times 2)$



Formato immediate

- Un immediate è codificato come segue:
 - $imm8$: 8-bit unsigned immediate
 - rot : 4-bit rotation value
- Da cui si ricava una constant a 32-bit :
 $imm8 \text{ ROR } (rot \times 2)$

Esempio: $imm8 = 10001111$

rot	32-bit constant
0000	0000 0000 0000 0000 0000 0000 1000 1111
0001	11 00 0000 0000 0000 0000 00 10 0011
0010	1111 0000 0000 0000 0000 0000 1000
...	...
1111	0000 0000 0000 0000 00 10 0011 1100

Esempio: Add

ADD R0, R1, #42

cond = 1110_2 (14) indica un'esecuzione non condizionata

op = 00_2 (0) indica un'istruzione di data-processing

cmd = 0100_2 (4) è il codice di ADD

I = 1 indica che **Src2** è un immediate, **S** = 0,

Rd = 0, **Rn** = 1

imm8 = 42, **rot** = 0

Field Values

31:28	27:26	25	24:21	20	19:16	15:12	11:8	7:0
1110_2	00_2	1	0100_2	0	1	0	0	42
cond	op	I	cmd	S	Rn	Rd	shamt5	sh
1110	00	1	0100	0	0001	0000	0000	00101010

0xE281002A

Esempio: SUB

SUB R2, R3, #0xFF0

cond = 1110_2 (14)

op = 00_2 (0)

cmd = 0010_2 (2) è il codice di SUB

I=1, **S**=0

Rd = 2, **Rn** = 3

imm8 = 0xFF

per produrre 0xFF0 **imm8** deve essere ruotato di 4 bit a sinistra,
ovvero 28 bit a destra. Quindi, **rot** = 14

Field Values

31:28	27:26	25	24:21	20	19:16	15:12	11:8	7:0
1110_2	00_2	1	0010_2	0	3	2	14	255
cond	op	I	cmd	S	Rn	Rd	rot	imm8
1110	00	1	0010	0	0011	0010	1110	11111111

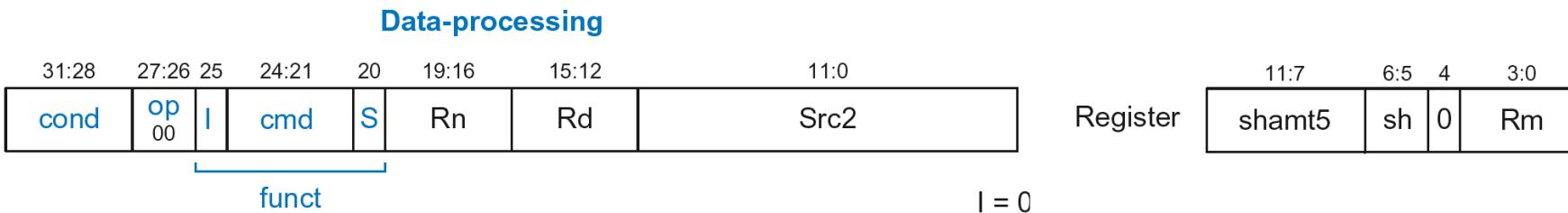
0xE2432EFF

Formato register

Rm: indica il registro che fa da secondo operando

shamt5: indica di quanto il valore in Rm è shiftato

sh: indica il tipo di shift (i.e., >>, <<, >>>, ROR)



Shift Type	sh
LSL	00_2
LSR	01_2
ASR	10_2
ROR	11_2

Esempio: Add

ADD R5, R6, R7

Operation: R5 = R6 + R7

cond = 1110_2 (14)

op = 00_2 (0)

cmd = 0100_2 (4)

I=0 indica che **Src2** è un registro

Rd = 5, **Rn** = 6, **Rm** = 7

shamt = 0, **sh** = 0

Field Values

31:28	27:26	25	24:21	20	19:16	15:12	11:7	6:5	4	3:0
1110_2	00_2	0	0100_2	0	6	5	0	0	0	7
cond	op	I	cmd	S	Rn	Rd	shamt5	sh	Rm	

1110	00	0	0100	0	0110	0101	00000	00	0	0111
------	----	---	------	---	------	------	-------	----	---	------

0xE0865007

Esempio: OR

ORR R9, R5, R3, LSR #2

Operation: R9 = R5 OR (R3 >> 2)

cond = 1110₂ (14)

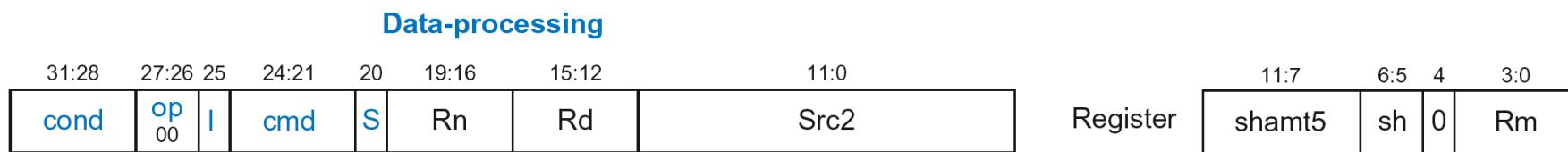
op = 00₂ (0)

cmd = 1100₂ (12) indica l'istruzione ORR

I=0

Rd = 9, **Rn** = 5, **Rm** = 3

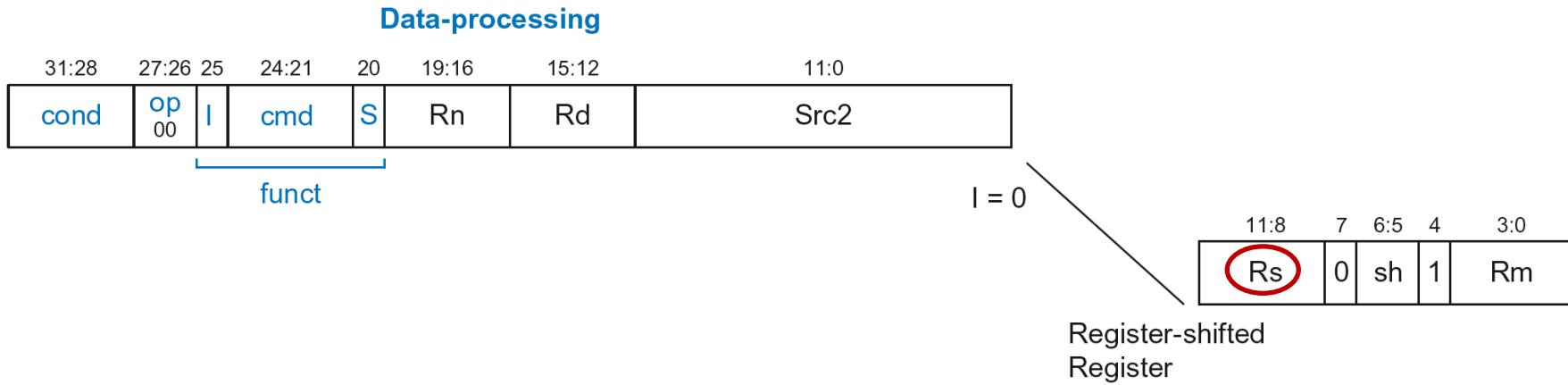
shamt5 = 2, **sh** = 01₂ (LSR)



1110 00 0 1100 0 0101 1001 00010 01 0 0011

0xE1859123

Formato register-shifted register



Simile al formato register, con la differenza che il numero di posizioni di cui Rm deve shiftare non è una costante ma è indicato dal registro Rs.

Esempio: XOR

EOR R8, R9, R10, ROR R12

Operation: $R8 = R9 \text{ XOR } (R10 \text{ ROR } R12)$

cond = 1110_2 (14)

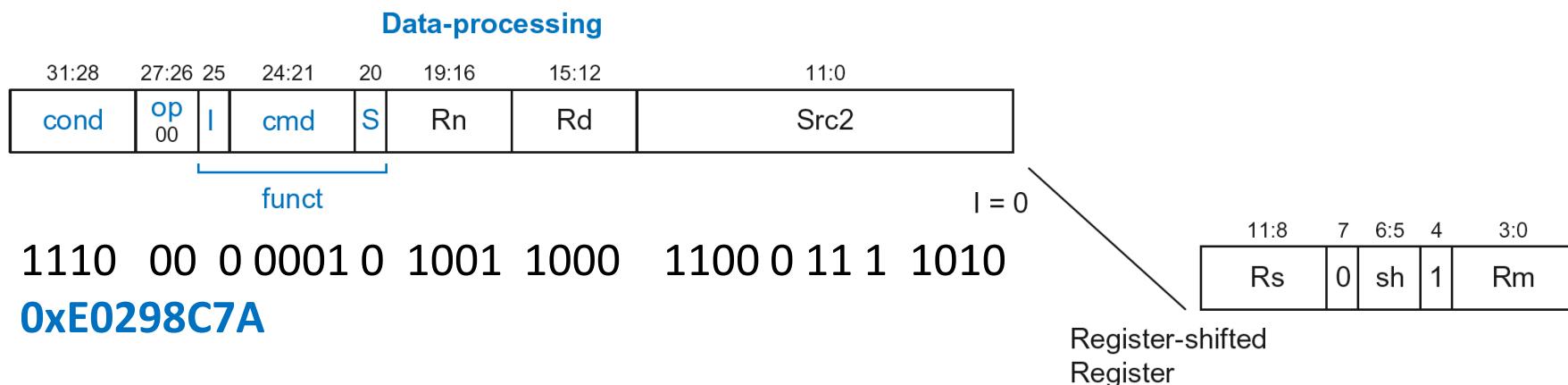
op = 00_2 (0)

cmd = 0001_2 (1) indica l'instruzione EOR

I=0

Rd = 8, **Rn** = 9, **Rm** = 10, **Rs** = 12

sh = 11_2 (ROR)



Esempio: ROR

ROR R1, R2, #23

Operation: $R1 = R2 \text{ ROR } 23$

cond = 1110_2 (14)

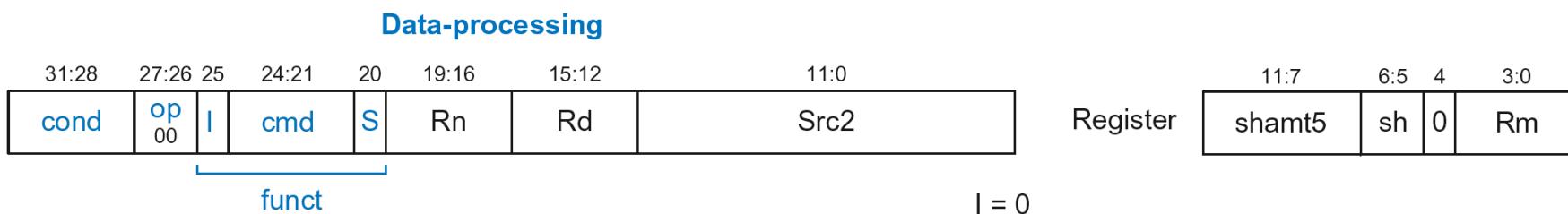
op = 00_2 (0)

cmd = 1101_2 (13) codice usato per tutti i tipi di shift (LSL, LSR, ASR, ROR)

Src2 è un immediate-shifted register quindi **I**=0

Rd = 1, **Rn** = 0, **Rm** = 2

shamt5 = 23, **sh** = 11_2 (ROR)



1110 00 011010 0000 0001 10111110 0010

0xE1A01BE2

Datapath ADD, SUB, AND, ORR

Estendiamo il **datapath** per gestire le istruzioni di data processing **ADD**, **SUB**, **AND** e **ORR**, utilizzando la modalità di indirizzamento immediato.

In tal caso, le istruzioni hanno come operandi un registro ed una costante contenuta nei bit dell'istruzione stessa. L'**ALU** esegue l'operazione e il risultato viene scritto in un terzo registro.

Esse differiscono solo nella specifica operazione eseguita dall'**ALU**. Quindi, possono essere implementate tutte con lo stesso hardware utilizzando diversi segnali **ALUControl**.

I valori per **ALUControl** sono:

- ▶ **ADD** – 00;
- ▶ **SUB** – 01;
- ▶ **AND** – 10;
- ▶ **ORR** – 11.

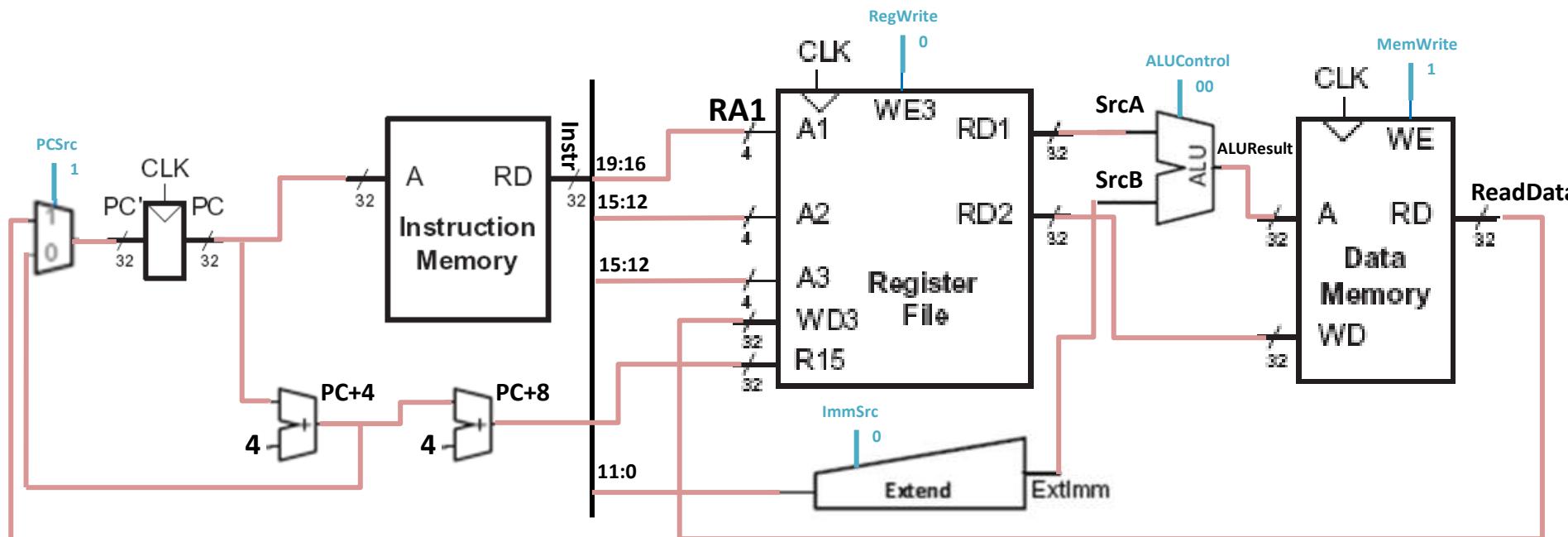
L'**ALU** imposta anche dei bit in **ALUFlags_{3:0}**:

- ▶ **Zero**;
- ▶ **Negativo**;
- ▶ **Carry**;
- ▶ **oVerflow**.

Datapath ADD, SUB, AND, ORR

Le istruzioni di data processing utilizzano costanti di **8 bit** (non **12 bit**), per cui il blocco **Extend** riceve in input un segnale di controllo **ImmSrc**:

- ▶ **ImmSrc = 0** → **ExtImm** è esteso da **Instr_{7:0}**;
- ▶ **ImmSrc = 1** → **ExtImm** è esteso da **Instr_{11:0}** (per **LDR** o **STR**);



Datapath ADD, SUB, AND, ORR

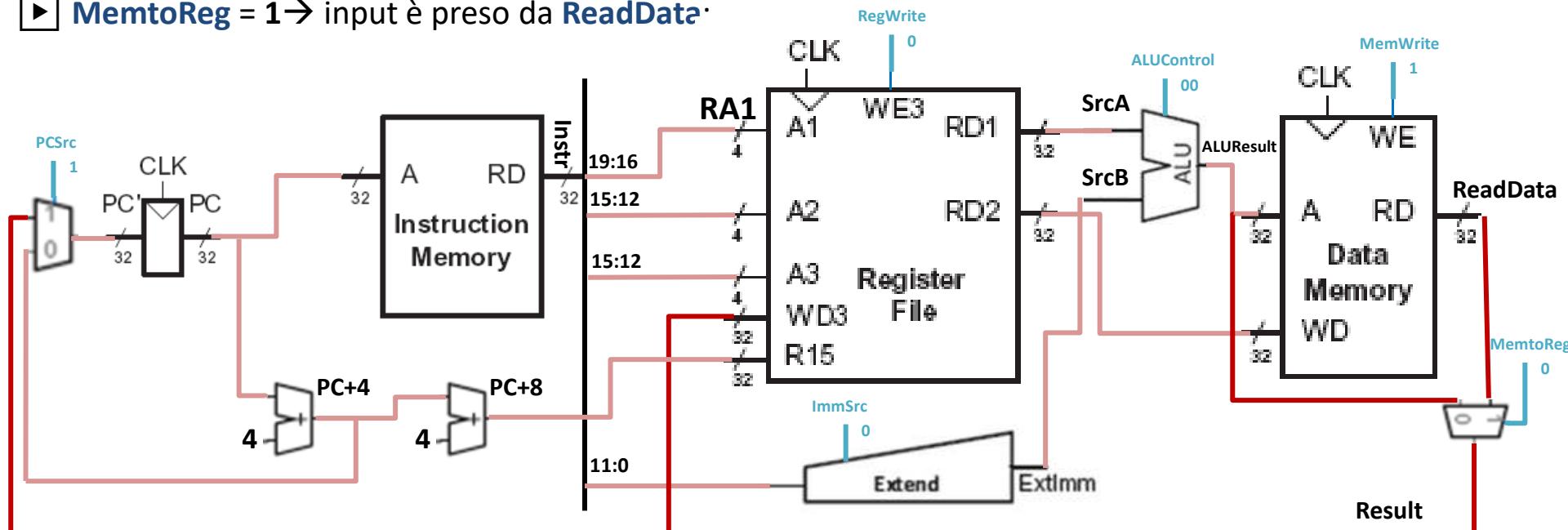
Un altro aspetto da disambiguare riguarda la scrittura nel register file.

Esso può ricevere l'input sia dalla **memoria dati** (**LDR**), che dall'**ALU** (operazioni aritmetiche).

Aggiungiamo un altro **multiplexer** che permette di selezionare la sorgente di input tra **ReadData** e **ALUResult**. L'uscita del multiplexer è indicata con **Result**.

Il multiplexer richiede un segnale di controllo, ovvero **MemtoReg**.

- ▶ **MemtoReg = 0** → input è preso da **ALUResult**;
- ▶ **MemtoReg = 1** → input è preso da **ReadData**;

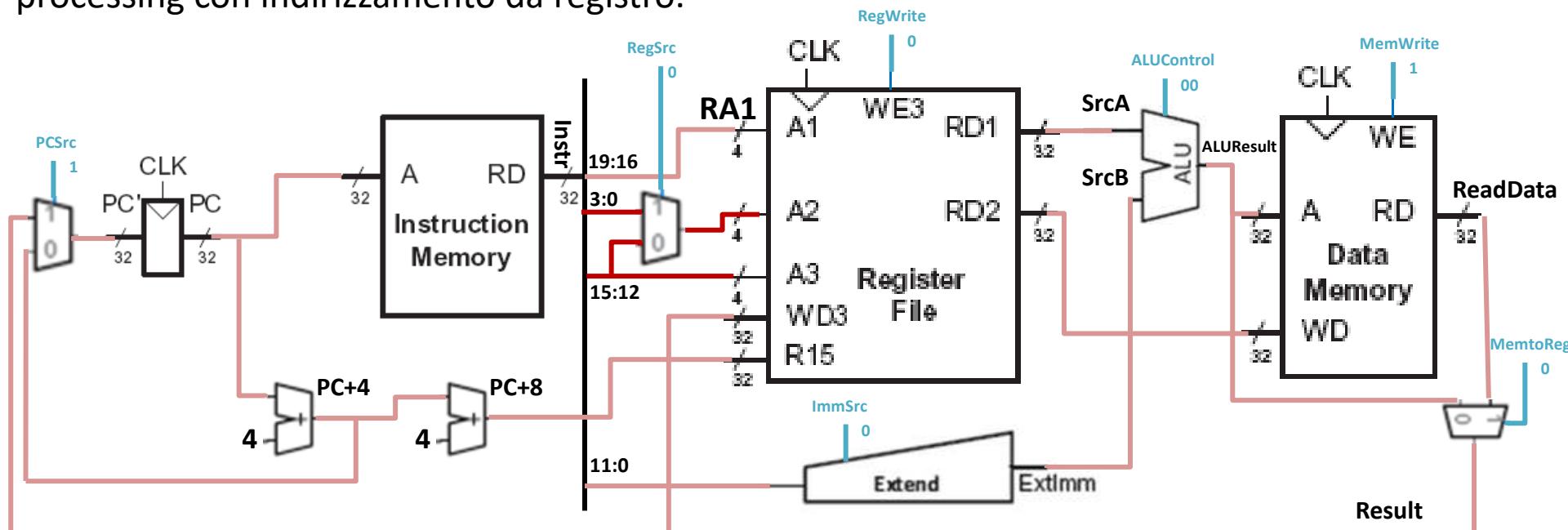


Datapath ADD, SUB, AND, ORR

Le istruzioni di data processing con indirizzamento da registro ricevono la loro seconda fonte da **Rm**, specificato da **Instr_{3:0}**, piuttosto che da una costante.

Aggiungiamo un ulteriore **multiplexer** sugli ingressi del file registro. In base al valore del segnale di controllo **RegSrc** , **RA2** può essere selezionato fra:

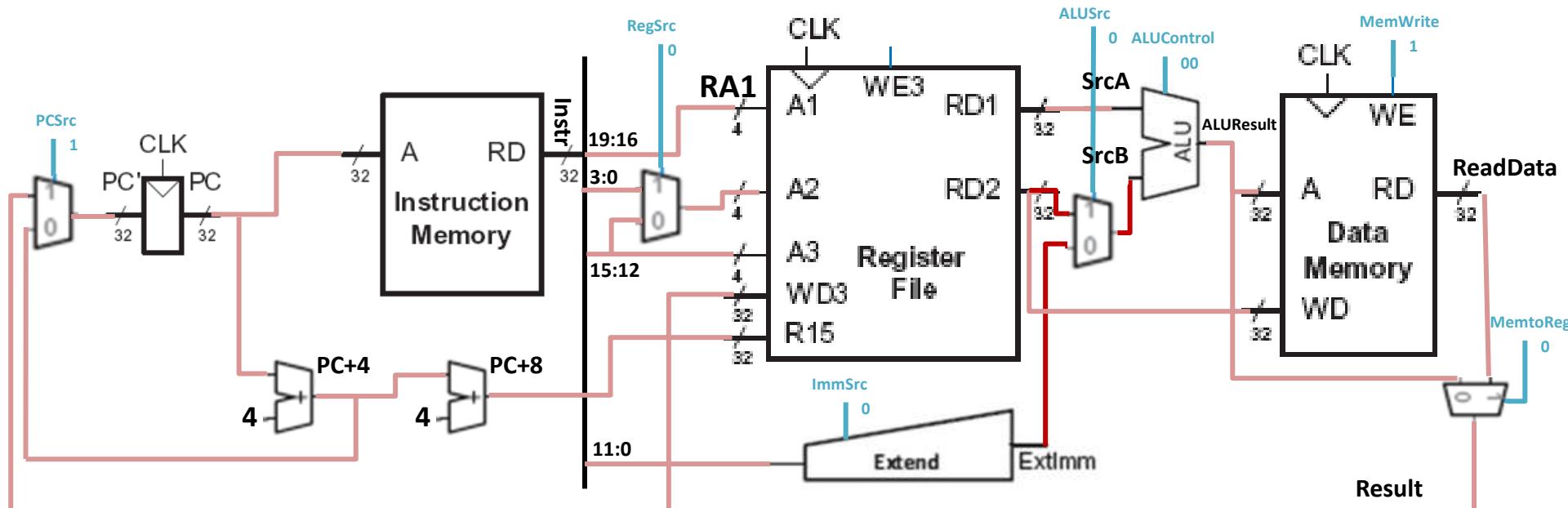
- ▶ **Rd** (**Instr_{15:12}**) per **STR**;
- ▶ **Rm** (**Instr_{3:0}**) per istruzioni di data processing con indirizzamento da registro.



Datapath ADD, SUB, AND, ORR

Aggiungiamo un ulteriore **multiplexer** sugli ingressi dell'**ALU** per selezionare questo secondo registro sorgente. In base al valore del segnale di controllo **ALUSrc**, la seconda sorgente della ALU viene selezionata tra:

- ▶ **ExtImm** per istruzioni, che utilizzano costanti;
- ▶ dal **register file** per istruzioni di data processing con indirizzamento da registro.



Esecuzione condizionata

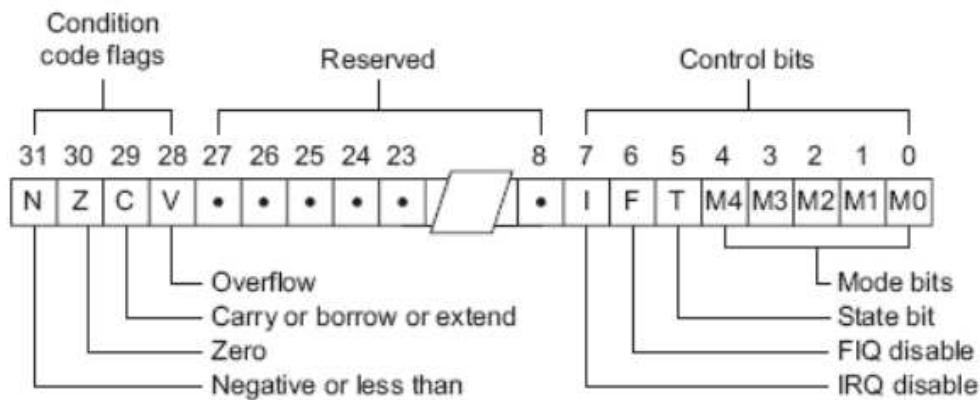
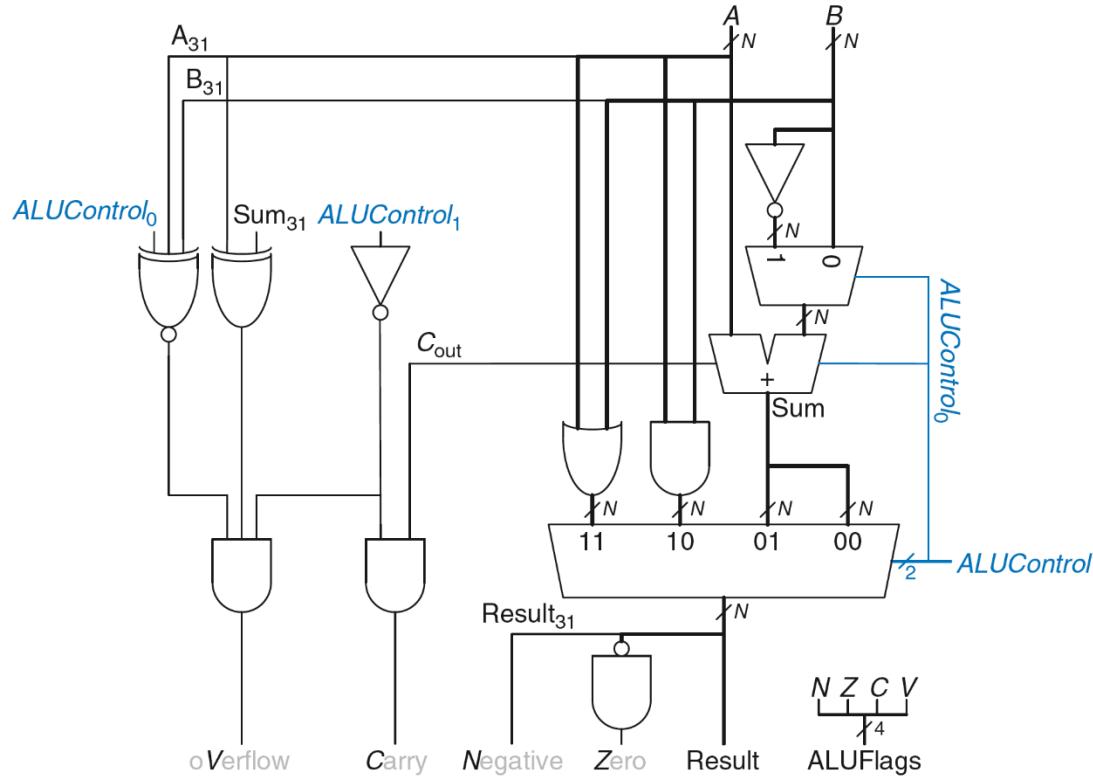
A volte si vuole che l'esecuzione di una istruzione dipenda dal verificarsi o meno di una certa condizione, per esempio:

- Istruzioni if/then/else, while, etc.: un blocco di istruzioni è eseguito solo se una data condizione valuta a true

ARM usa delle **condition flag** che sono

- memorizzate nel *Current Program Status Register (CPSR)*
- settate da una istruzione
- usate per eseguire in modo condizionato una istruzione

Flag	Name	Description
N	Negative	Instruction result is negative
Z	Zero	Instruction results in zero
C	Carry	Instruction causes an unsigned carry out
V	oVerflow	Instruction causes an overflow



Istruzione di comparazione: CMP

CMP R5, R6

- Esegue: $R5 - R6$
- Non salva il risultato
- Aggiorna i flag nel Current Program Status Register. Se il risultato di $R5 - R6$:
 - è uguale a 0, allora $Z=1$
 - è negativo, allora $N=1$
 - causa un carry out, $C=1$
 - causa un signed overflow, $V=1$

Suffisso S al nome di una istruzione

ADD**S** R1, R2, R3

- Esegue: $R2 + R3$
- Salva il risultato in R1
- Aggiorna i flag nel Current Program Status Register. Se il risultato di $R2 + R3$:
 - è uguale a 0, allora $Z=1$
 - è negativo, allora $N=1$
 - causa un carry out, $C=1$
 - causa un signed overflow, $V=1$

Istruzioni condizionate

- Un istruzione può essere *eseguita condizionalmente* sulla base dei valori delle condition flags
- In assembly, un'esecuzione condizionata è indicata da un suffisso detto *condition mnemonic*.

```
CMP    R1, R2  
SUBNE R3, R5, R8
```

- **NE**: SUB verrà eseguito solo se $R1 \neq R2$, ovvero $Z = 0$

cond	Mnemonic	Name	CondEx
0000	EQ	Equal	Z
0001	NE	Not equal	\overline{Z}
0010	CS/HS	Carry set / unsigned higher or same	C
0011	CC/LO	Carry clear / unsigned lower	\overline{C}
0100	MI	Minus / negative	N
0101	PL	Plus / positive or zero	\overline{N}
0110	VS	Overflow / overflow set	V
0111	VC	No overflow / overflow clear	\overline{V}
1000	HI	Unsigned higher	$\overline{Z}C$
1001	LS	Unsigned lower or same	Z OR \overline{C}
1010	GE	Signed greater than or equal	$\overline{N} \oplus V$
1011	LT	Signed less than	$N \oplus V$
1100	GT	Signed greater than	$\overline{Z}(\overline{N} \oplus V)$
1101	LE	Signed less than or equal	Z OR ($N \oplus V$)
1110	AL (or none)	Always / unconditional	Ignored

	Unsigned	Signed
A = 1001₂	A = 9	A = -7
B = 0010₂	B = 2	B = 2
A - B:	1001	<i>NZCV = 0011₂</i>
	+ 1110	HS: TRUE
(a)	<hr/> 10111	GE: FALSE

	Unsigned	Signed
A = 0101₂	A = 5	A = 5
B = 1101₂	B = 13	B = -3
A - B:	0101	<i>NZCV = 1001₂</i>
	+ 0011	HS: FALSE
(b)	<hr/> 1000	GE: TRUE

Esempi di istruzioni condizionate

```
CMP    R5, R9           ; performs R5-R9  
                  ; sets condition flags  
  
SUBEQ  R1, R2, R3       ; executes if R5==R9 (Z=1)  
ORRMI  R4, R0, R9       ; executes if R5-R9 is  
                  ; negative (N=1)
```

Si assume per esempio che $R5 = 17$, $R9 = 23$:

CMP esegue: $17 - 23 = -6$ (flags: $N=1, Z=0, C=0, V=0$)

SUBEQ **non è eseguita** (non sono uguali: $Z=0$)

ORRMI **è eseguita** poiché il risultato è negativo ($N=1$)

Istruzioni di Branching

Un programma di solito esegue in sequenza, incrementando il Program Counter (PC) di 4 (32 bit) dopo ciascuna istruzione, in modo da puntare alla successiva istruzione.

Le istruzioni branching permettono di cambiare il valore del PC. ARM include due tipi di branch: **simple branch** (B) e **branch and link** (BL).

Come altre istruzioni ARM, i branch possono essere condizionati o incondizionati.

Il codice assembly utilizza le etichette per indicare i blocchi di istruzioni nel programma.

Quando il codice assembly è tradotto in codice macchina, queste etichette vengono tradotte in indirizzi di istruzione.

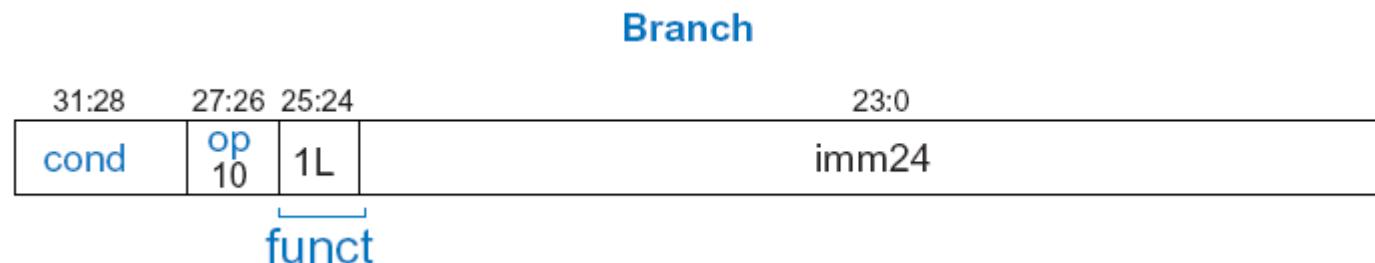
Istruzioni di Branching

Le istruzioni di branching utilizzano un unico operando costante di 24 bit, **imm24**.

Esse hanno un campo **cond** di 4 bit e un campo **op** di 2 bit, il cui valore è 10_2 .

Il campo **funct** ha solo 2 bit. Il bit più significativo è sempre 1 per i branch. Il bit meno significativo, **L**, indica il tipo di operazione di branch: 1 per **BL** e 0 per **B**.

I restanti 24 bit, **imm24**, rappresentano un valore in complemento a due, che specifica la posizione dell'istruzione relativamente all'indirizzo **PC** + 8.



Istruzioni di Branching

La costante a **24-bit** viene *moltiplicata per 4* ed estesa con segno.

Pertanto, la logica **Extend** necessita di una ulteriore modalità.

ImmSrc è, quindi, esteso a 2 bit.

L'istruzione di salto somma poi **ImmSrc** a **PC+8** e scrive il risultato di nuovo nel **PC**.

ImmSrc	ExtImm	Description
00	{24 0s} $Instr_{7:0}$	8-bit unsigned immediate for data-processing
01	{20 0s} $Instr_{11:0}$	12-bit unsigned immediate for LDR/STR
10	{6 $Instr_{23}$ } $Instr_{23:0}$ 00	24-bit signed immediate multiplied by 4 for B

Istruzioni di Branching

L'istruzione **BL** (Branch and Link) è usata per la chiamata di una subroutine

- Salva l'indirizzo di ritorno (**R15**) in **R14**
- Il ritorno dalla routine si effettua copiando **R14** in **R15**:

MOV R15, R14

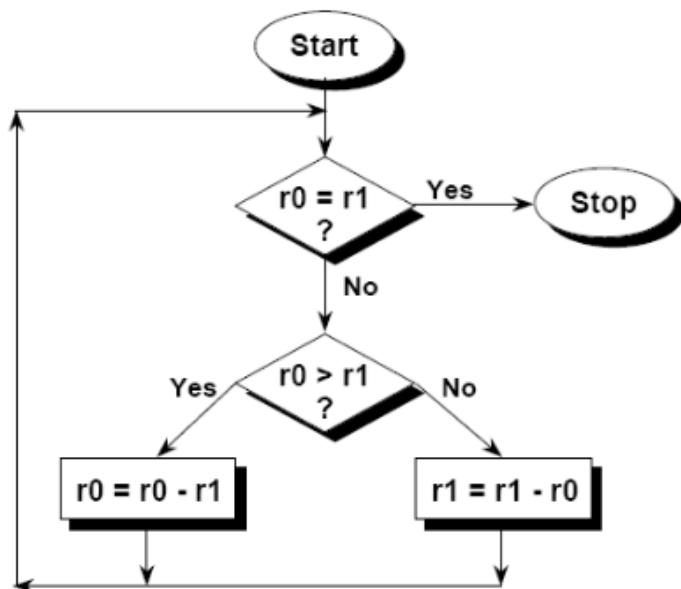
Il registro **R14** ha la funzione (architetturale) di subroutine Link Register (**LR**).

In esso viene salvato l'indirizzo di ritorno (ovvero il contenuto del registro **R15**) quando viene eseguita l'istruzione **BL** (Branch and Link).

```
....  
BL    function      ; call 'function'  
....  
; procedure returns to here  
....  
function          ; function body  
....  
....  
....  
MOV   PC, LR       ; Put R14 into PC to return
```

Esempi di istruzioni di branching

Il processore ARM consente un codice molto compatto.



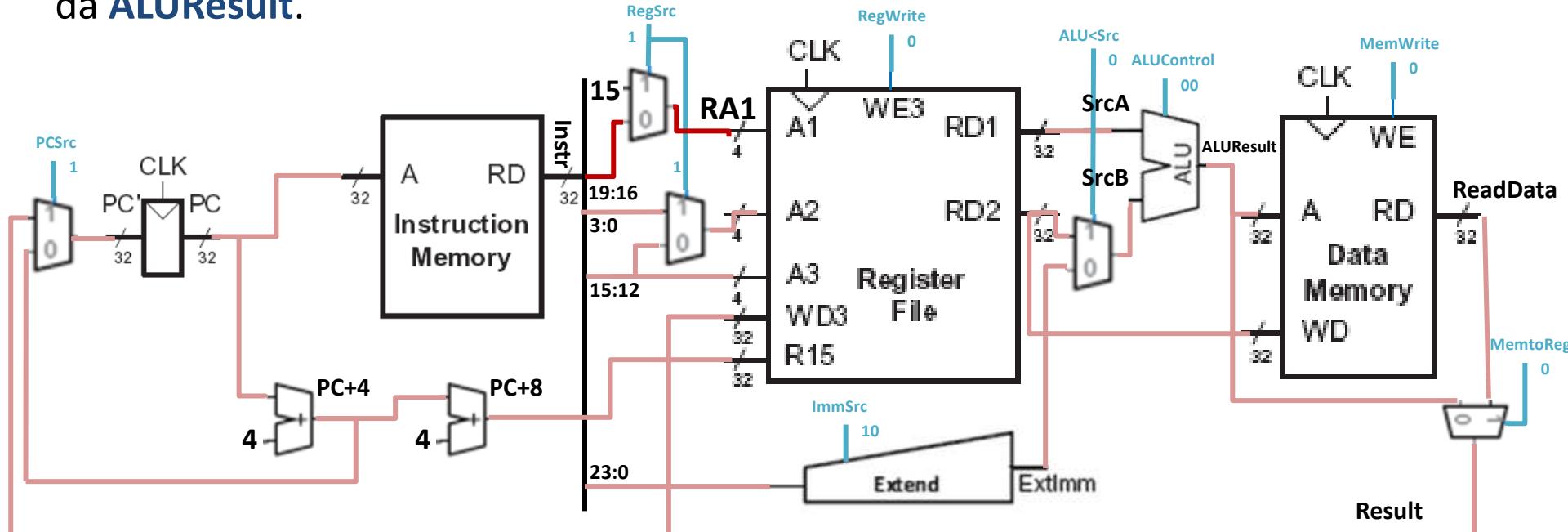
MCD	cmp r0,r1	;if r0 > r1
	subgt r0,r0,r1	;then r0 <- r0-r1
	sublt r1,r1,r0	;else r1 <- r1-r0
	bne MCD	;raggiunta la fine?

1011	LT	Signed less than	$N \oplus V$
1100	GT	Signed greater than	$\overline{Z}(\overline{N} \oplus \overline{V})$

Datapath istruzioni di branching

Dato che **PC+8** è letto dalla prima porta del register file, è necessario un **multiplexer** per selezionare **R15** come ingresso di **RA1**. Il multiplexer è controllato dal segnale **RegSrc**, il cui valore è preso dai bit **Instr_{19:16}**, per la maggior parte delle istruzioni ed è impostato a **15** per le istruzioni di branch (B).

MemtoReg è impostato a **0** e **PCSrc** è impostato a **1** per selezionare il nuovo PC da **ALUResult**.

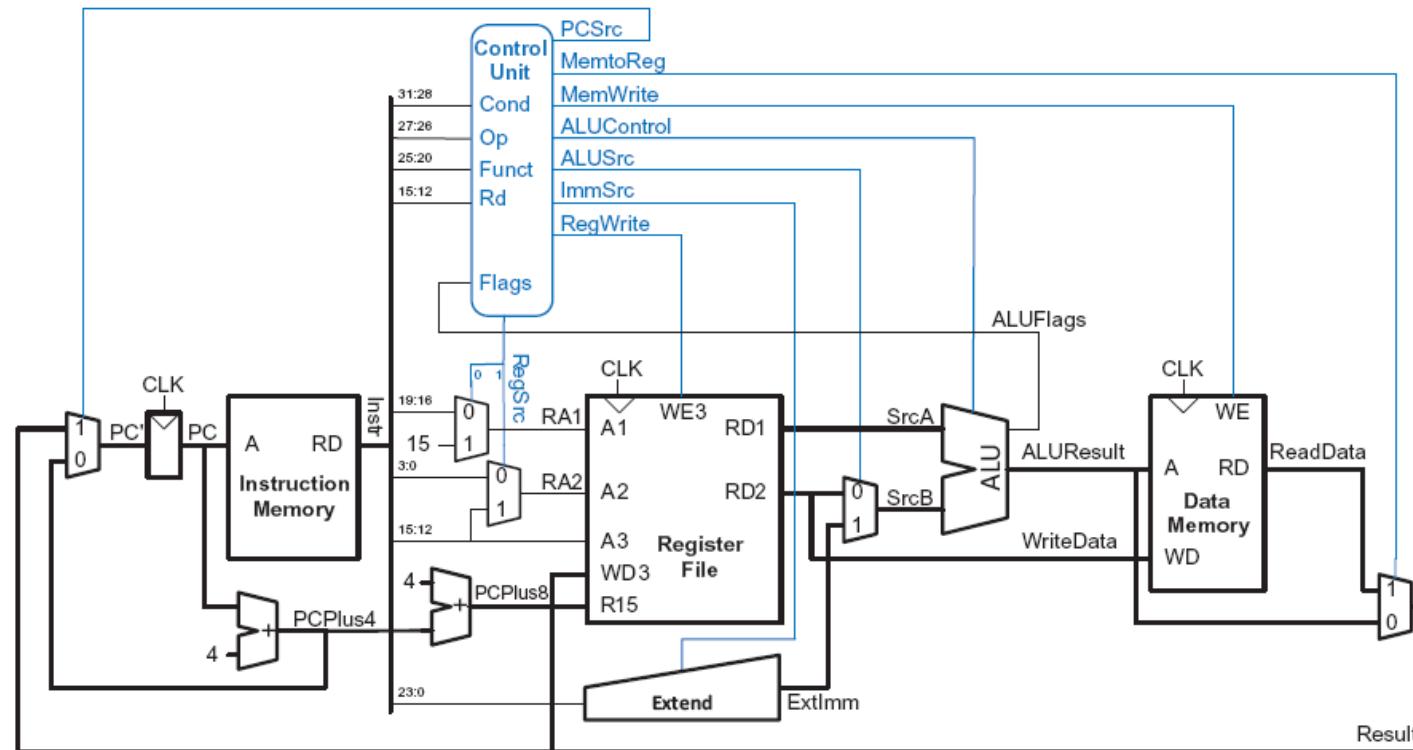


Unità di controllo

L'unità di controllo calcola i segnali di controllo in base a:

- i campi **cond**, **op**, e **funct** dell'istruzione ($\text{Instr}_{31:28}$, $\text{Instr}_{27:26}$, e $\text{Instr}_{25:20}$);
- i **flag**;
- se il registro di destinazione è il **PC**.

Il controller memorizza anche i **flag di stato** attuali nel *Current Program Status Register* e li aggiorna in modo appropriato.



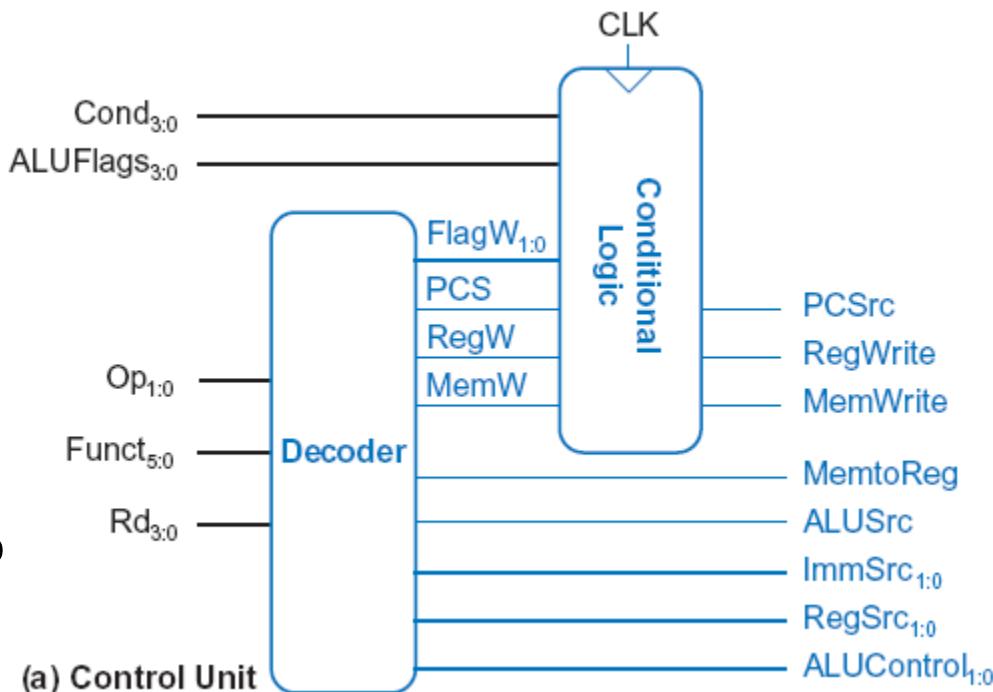
Unità di controllo

Dividiamo l'unità di controllo in due parti principali:

- ▶ il **decoder** – genera i segnali di controllo sulla base dei campi di Instr;
- ▶ la **logica condizionale** – gestisce i flag di stato e li aggiorna quando l'istruzione deve essere eseguita su condizione.

Il **Decoder** è composto da:

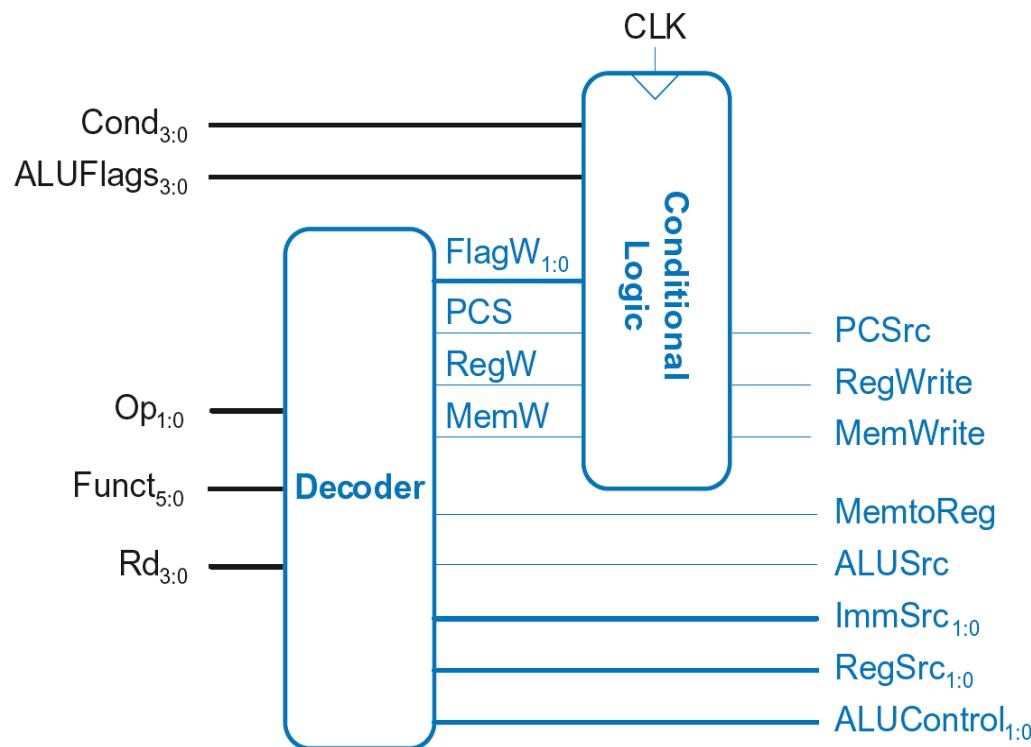
- ▶ un **decodificatore principale**, che produce la maggior parte dei segnali di controllo;
- ▶ un **decoder ALU**, che utilizza il campo Funct per determinare il tipo di istruzione data-processing;
- ▶ la **logica di controllo del PC**, che determina se il PC deve essere aggiornato a causa di una istruzione di branch o di una scrittura in R15.



Unità di controllo

Dividiamo l'unità di controllo in due parti principali:

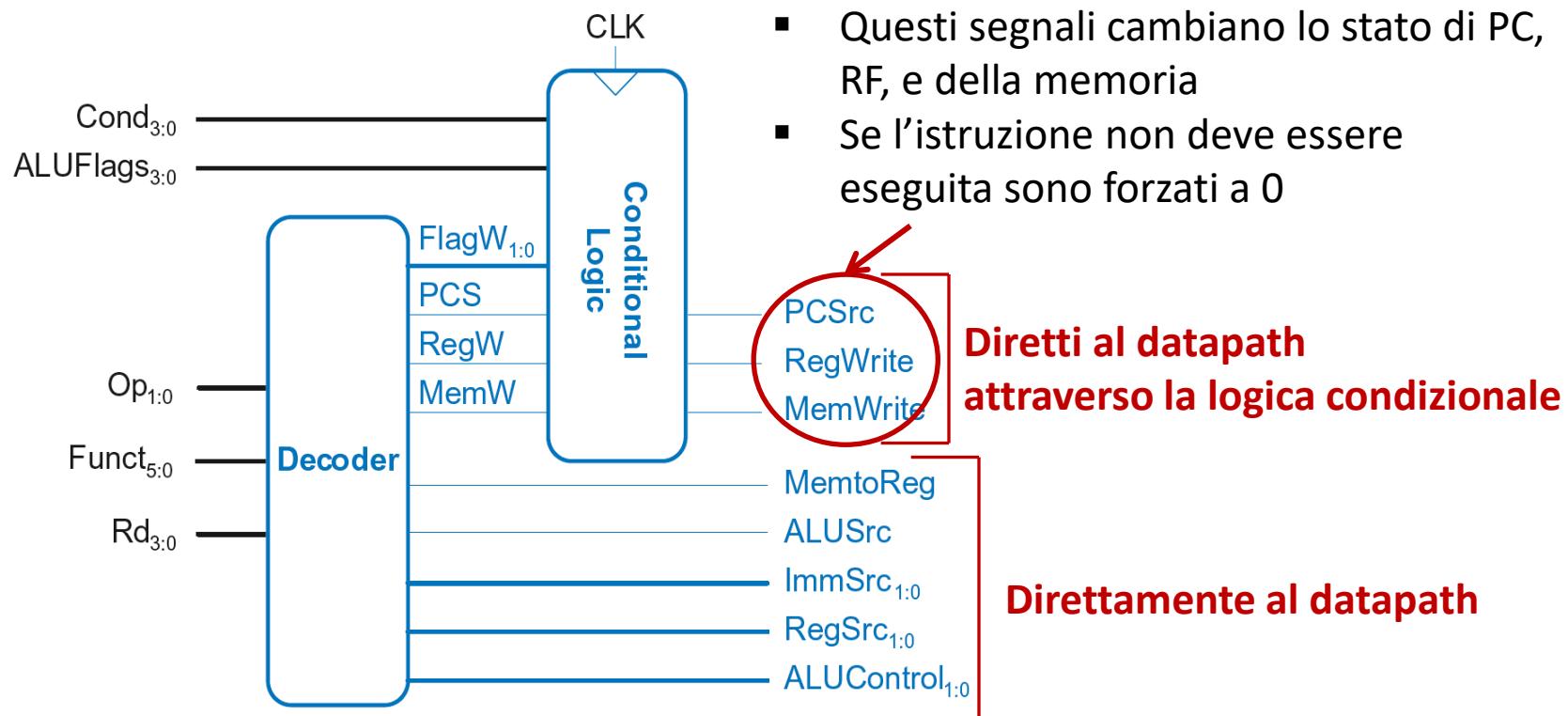
- ▶ il **decoder** – genera i segnali di controllo sulla base dei campi di Instr;
- ▶ la **logica condizionale** – gestisce i flag di stato e li aggiorna quando l'istruzione deve essere eseguita su condizione.



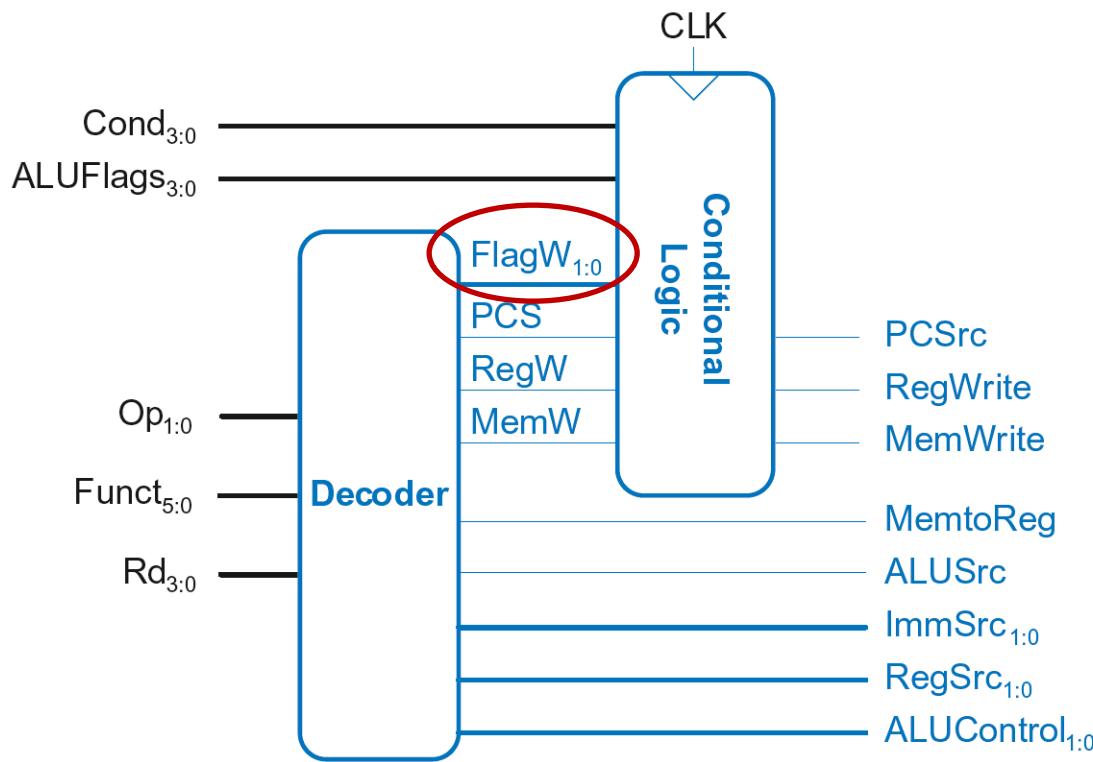
Unità di controllo

Dividiamo l'unità di controllo in due parti principali:

- ▶ il **decoder** – genera i segnali di controllo sulla base dei campi di Instr;
- ▶ la **logica condizionale** – gestisce i flag di stato e li aggiorna quando l'istruzione deve essere eseguita su condizione.



Unità di controllo



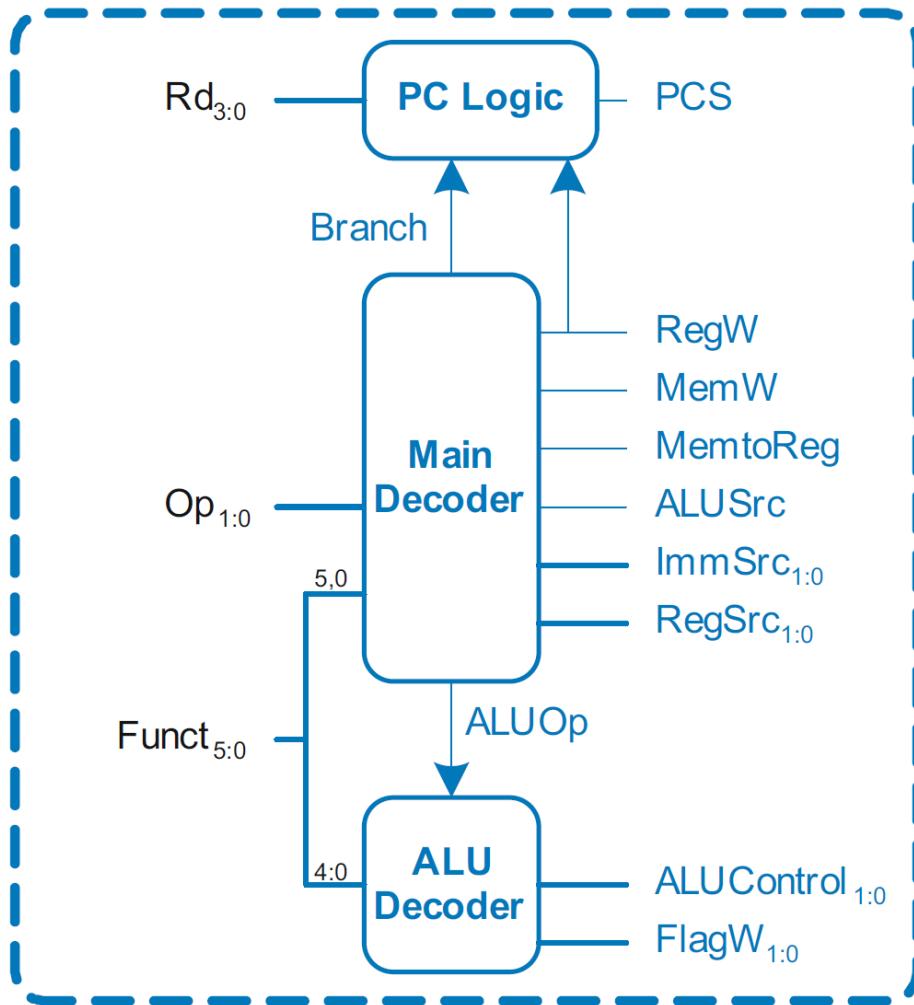
- **FlagW_{1:0}:** Flag Write signal, indica quando le *ALUFlags* devono essere aggiornate, ovvero quando in una istruzione S=1
- ADD, SUB aggiornano tutti i flag (**NZCV**)
- AND, ORR aggiornano solo **N e Z**
- Quindi sono necessari due bit:
 - **FlagW₁** = 1: *NZ*
(*ALUFlags_{3:2}* saved)
 - **FlagW₀** = 1: *CV*
(*ALUFlags_{1:0}* saved)

Decoder

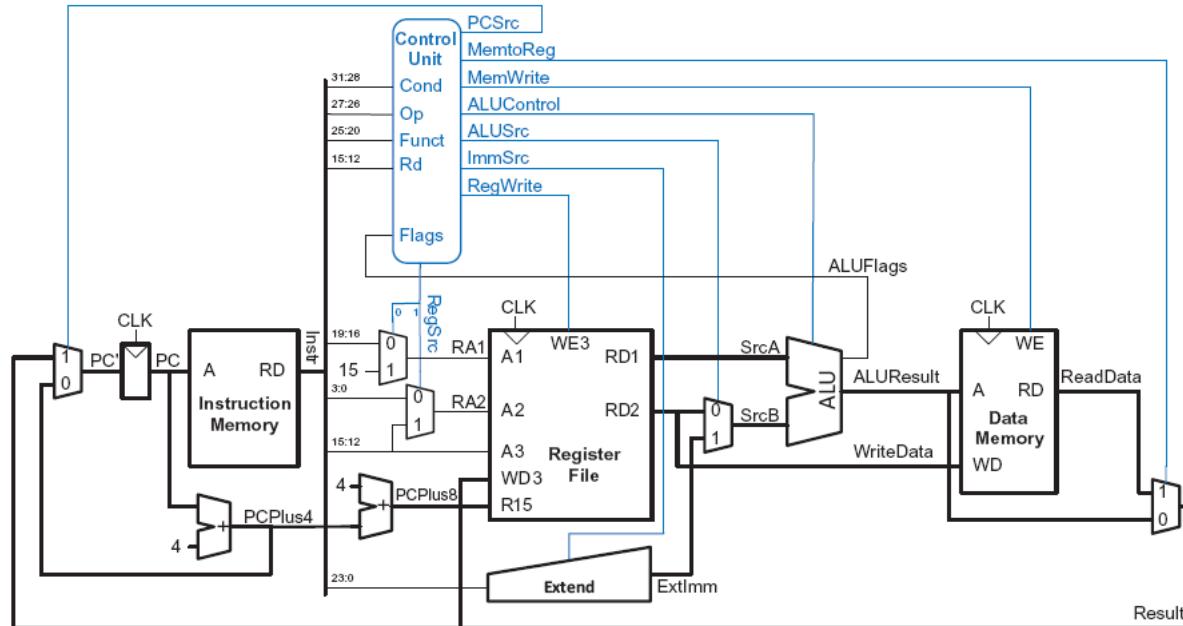
- ▶ determinare il tipo di istruzione: data processing con registro o costante, STR, LDR, o B.
- ▶ produrre i segnali di controllo adeguati per il datapath. Alcuni segnali sono inviati direttamente al datapath: **MemtoReg**, **ALUSrc**, **ImmSrc1:0**, e **RegSrc1:0**.
- ▶ generare i segnali che abilitano la scrittura (**MemW** e **RegW**), i quali devono passare attraverso la logica condizionale prima di diventare segnali datapath (**MemWrite** e **RegWrite**). Tali segnali possono essere azzerati dalla logica condizionale, se la condizione non è soddisfatta.
- ▶ generare i segnali **Branch** e **ALUOp**, utilizzati rispettivamente per indicare l'istruzione B o il tipo di istruzione data processing.

La logica per il decoder principale può essere sviluppata dalla tabella di verità utilizzando le tecniche standard per la progettazione della logica combinatoria.

- **Main Decoder**
- ALU Decoder
- PC Logic

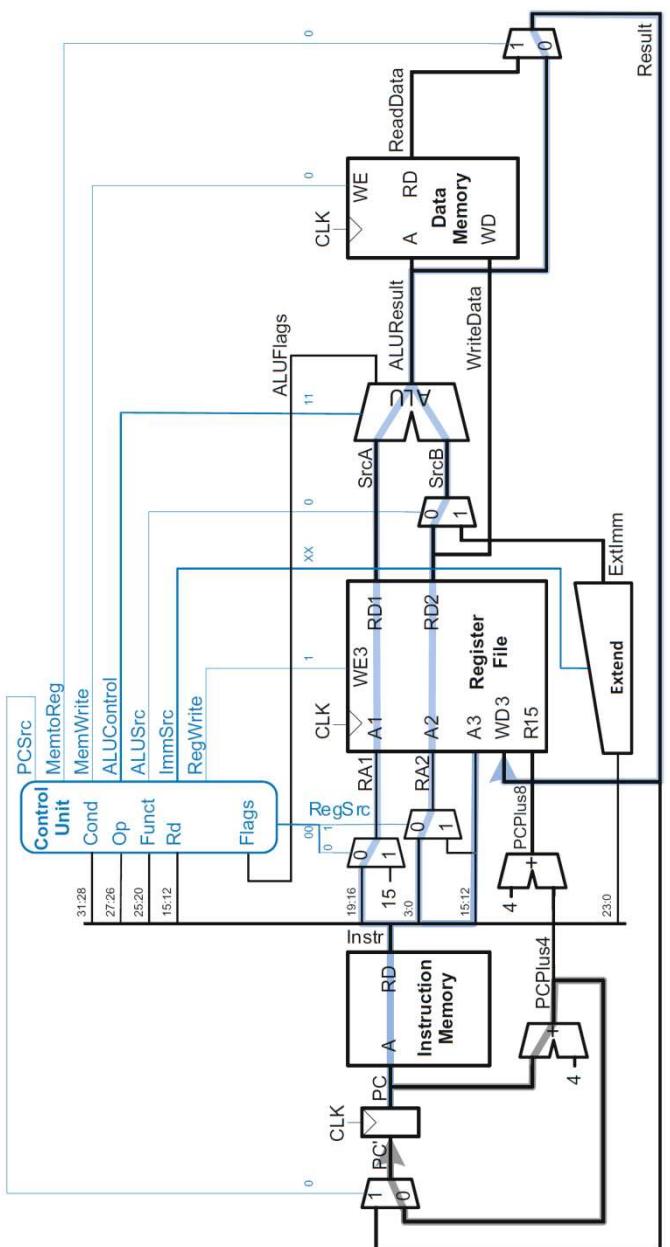


Op	Funct ₅	Type	ALUOp	RegSrc	RegW	ImmSrc	ALUSrc	MemW	MemtoReg	Branch	
00	0	X	DP Reg	0	0	0	0	XX	1	00	1
00	1	X	DP Imm	0	0	0	1	00	1	X0	1
01	X	0	STR	0	X	1	1	01	0	10	0
01	X	1	LDR	0	1	0	1	01	1	X0	0
11	X	X	B	1	0	0	1	10	0	X1	0

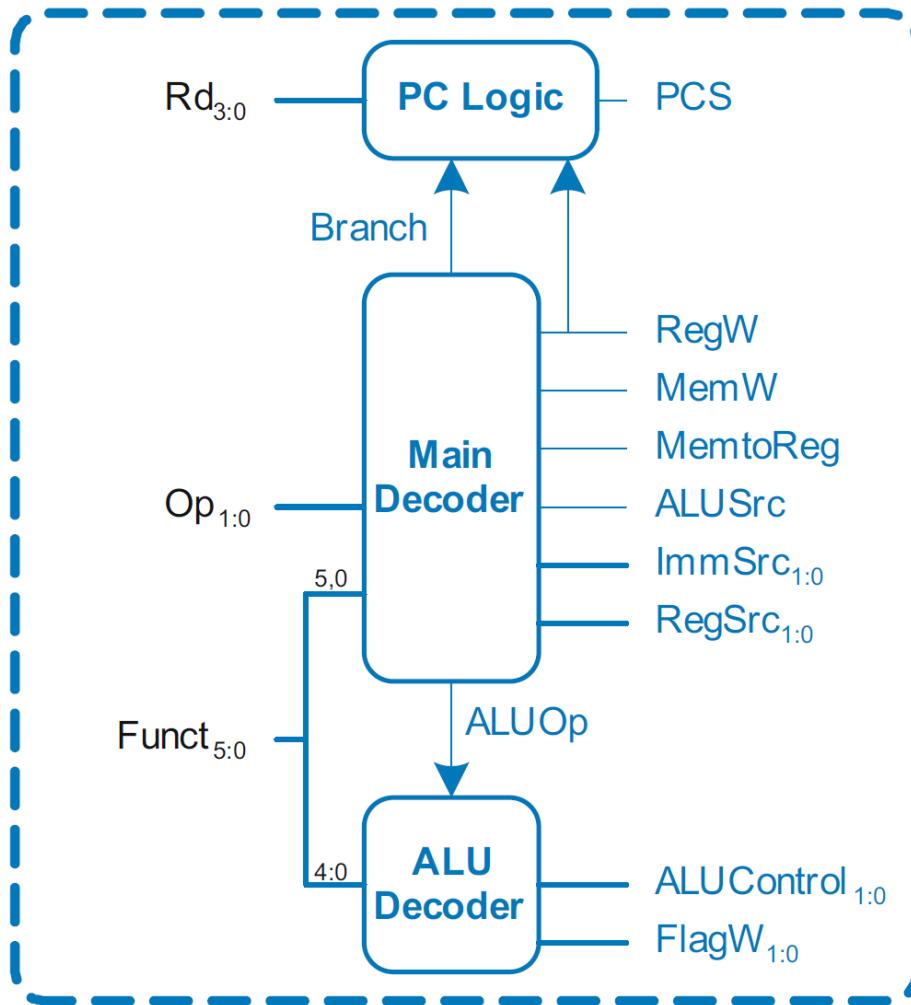


Esempio: DPReg

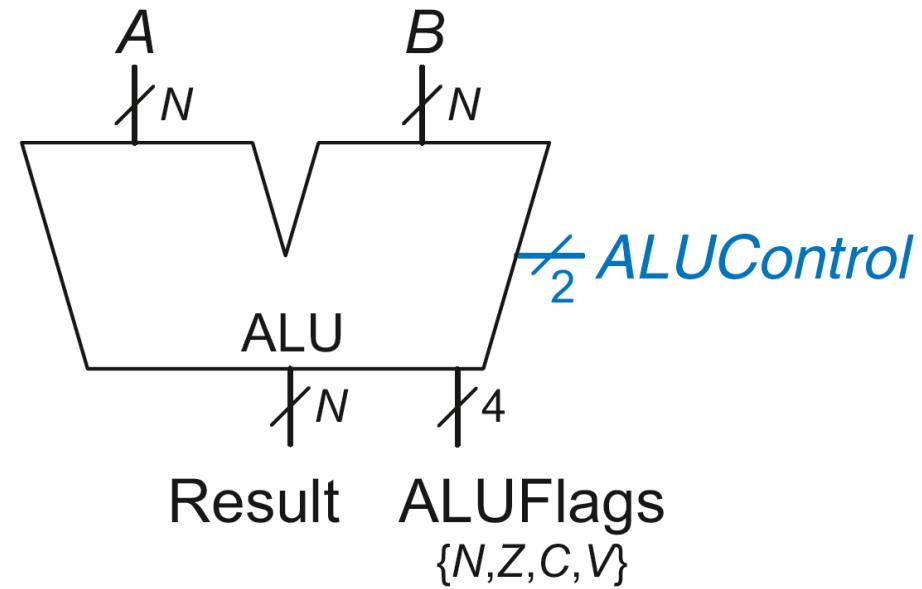
ALUOp								
RegSrc								
RegW								
ImmSrc								
ALUSrc								
MemW								
MemtoReg								
Branch								
Type								
Funct ₀	00	0	X	DP Reg	0	0	0	1
Funct ₅								
Op	00	0	X	DP Reg	00	XX	1	00



- Main Decoder
- **ALU Decoder**
- PC Logic



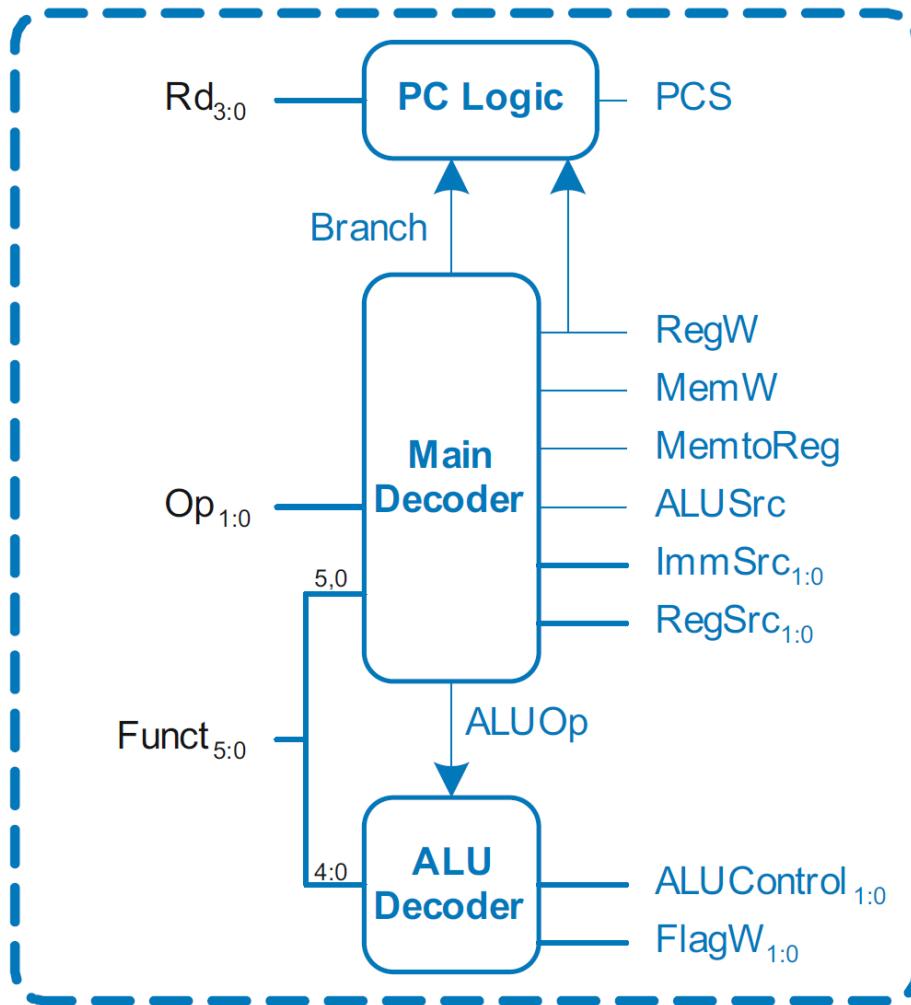
ALUControl_{1:0}	Function
00	Add
01	Subtract
10	AND
11	OR



ALUOp	Funct_{4:1} (cmd)	Funct₀ (S)	Type	ALUControl_{1:0}	FlagW_{1:0}
0	X	X	Not DP	00	00
1	0100	0	ADD	00	00
		1			11
	0010	0	SUB	01	00
		1			11
	0000	0	AND	10	00
		1			10
	1100	0	ORR	11	00
		1			10

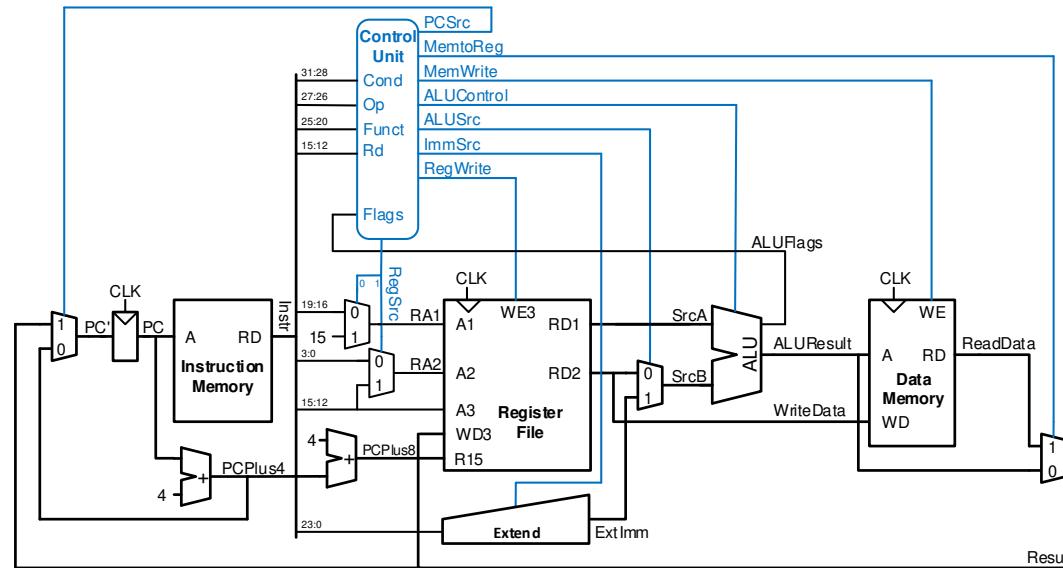
- **FlagW₁** = 1: NZ (Flags_{3:2}) devono essere salvate
- **FlagW₀** = 1: CV (Flags_{1:0}) devono essere salvate

- Main Decoder
- ALU Decoder
- **PC Logic**



La **logica del PC** controlla se l'istruzione è una scrittura in **R15** o un branch secondo la condizione:

$$PCS = ((Rd == 15) \text{ AND } RegW) \text{ OR Branch}$$

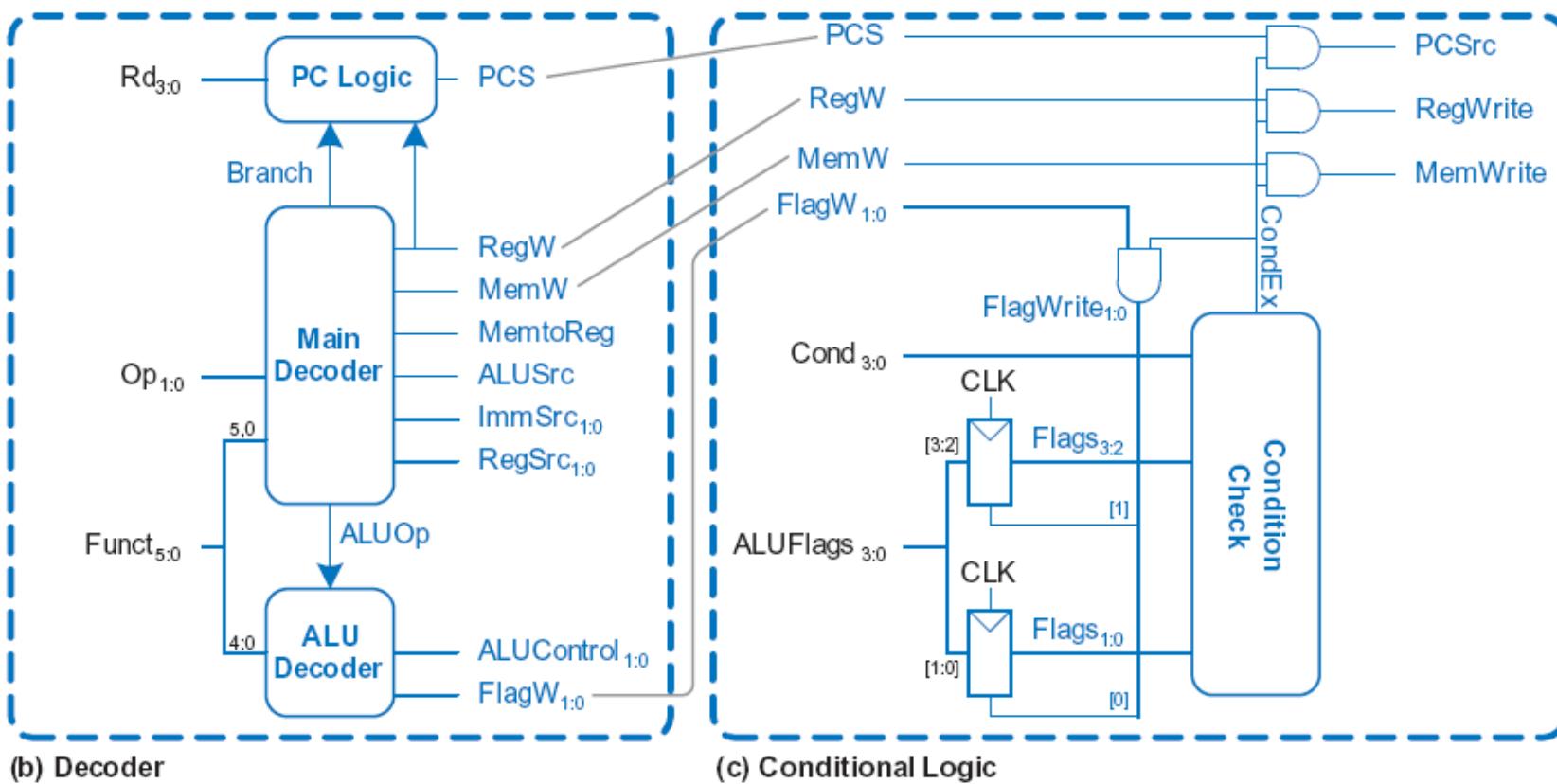


Se l'istruzione è eseguita: $PCSrc = PCS$

Altrimenti: $PCSrc = 0$ (e quindi, $PC = PC + 4$)

Logica condizionale

I segnali che abilitano la scrittura (**MemW** and **RegW**) e l'aggiornamento dei flag (**FlagWrite**) e del PC (**PCS**) devono *passare* attraverso la logica condizionale prima di diventare operativi (e.g. segnali datapath **MemWrite**, **RegWrite** e **PCSrc**). Tali segnali possono essere azzerati dalla logica condizionale, se la condizione non è soddisfatta.



Condizioni su flags di stato

Mnemonic	Name	CondEx
EQ	Equal	Z
NE	Not equal	\bar{Z}
CS / HS	Carry set / Unsigned higher or same	C
CC / LO	Carry clear / Unsigned lower	\bar{C}
MI	Minus / Negative	N
PL	Plus / Positive or zero	\bar{N}
VS	Overflow / Overflow set	V
VC	No overflow / Overflow clear	\bar{V}
HI	Unsigned higher	$\bar{Z}C$
LS	Unsigned lower or same	$Z \text{ OR } \bar{C}$
GE	Signed greater than or equal	$\overline{N \oplus V}$
LT	Signed less than	$N \oplus V$
GT	Signed greater than	$\bar{Z}(N \oplus V)$
LE	Signed less than or equal	$Z \text{ OR } (N \oplus V)$
AL	Always / unconditional	ignored

Condizioni su flags di stato

Condizioni sul risultato di una operazione aritmetica in complemento a 2

Mnemonic	Name	CondEx
MI	Minus / Negative	N
PL	Plus / Positive or zero	\bar{N}
VS	Overflow / Overflow set	V
VC	No overflow / Overflow clear	\bar{V}
AL	Always / unconditional	ignored

Condizioni su flags di stato

Per confrontare due interi (signed o unsigned) A e B si esegue la differenza A-B e si verificano le seguenti condizioni sui flag

Mnemonic	Name	CondEx
EQ	Equal	Z
NE	Not equal	\bar{Z}
CS / HS	Carry set / Unsigned higher or same	C
CC / LO	Carry clear / Unsigned lower	\bar{C}
HI	Unsigned higher	$\bar{Z}C$
LS	Unsigned lower or same	$Z \text{ OR } \bar{C}$
GE	Signed greater than or equal	$\overline{N \oplus V}$
LT	Signed less than	$N \oplus V$
GT	Signed greater than	$\bar{Z}(\overline{N \oplus V})$
LE	Signed less than or equal	$Z \text{ OR } (N \oplus V)$

Condizioni su flags di stato

Mnemonic	Name	CondEx
EQ	Equal	Z
NE	Not equal	\bar{Z}

$A==B$ sse $A-B==0$ sse $Z=1$

$A!=B$ sse $A-B!=0$ sse $Z=0$

Vale sia per la rappresentazione in complemento a 2 (interi con segno) che nella rappresentazione senza segno

Condizioni su flags di stato

Mnemonic	Name	CondEx
GE	Signed greater than or equal	$\bar{N} \oplus V$
LT	Signed less than	$N \oplus V$
GT	Signed greater than	$\bar{Z}(\bar{N} \oplus V)$
LE	Signed less than or equal	$Z \text{ OR } (N \oplus V)$

$A < B \text{ sse } A - B < 0 \text{ sse } (N \oplus V)$

- Quando $A - B$ non genera overflow allora deve essere negativo
 - Questo accade sempre quando A e B hanno lo stesso segno
- Quando $A - B$ genera overflow?
 - Allora A e B hanno segni diversi
 - $A < B$ è vera sse A negativo e B positivo sse A negativo e $(-B)$ negativo, e quindi, essendoci overflow il risultato deve essere positive (ovvero $N=0$)

$A \leq B \text{ sse } A - B < 0 \text{ or } A - B == 0 \text{ sse } Z + (N \oplus V)$

Condizioni su flags di stato

Mnemonic	Name	CondEx
CS / HS	Carry set / Unsigned higher or same	C
CC / LO	Carry clear / Unsigned lower	\bar{C}
HI	Unsigned higher	$\bar{Z}C$
LS	Unsigned lower or same	$Z \text{ OR } \bar{C}$

Problema A-B che semantica ha nella rappresentazione senza segno?

- Sappiamo che $A-B$ è in complemento a due $A+(\sim B+1)$, dove $\sim B$ è la negazione bit a bit di B
- Nella rappresentazione senza segno invece
 - $\sim(X_{N-1}\dots X_0) = (1-X_{N-1})\dots(1-X_0) = 1\dots 1 - (X_{N-1}\dots X_0) = 2^N - 1 - B$

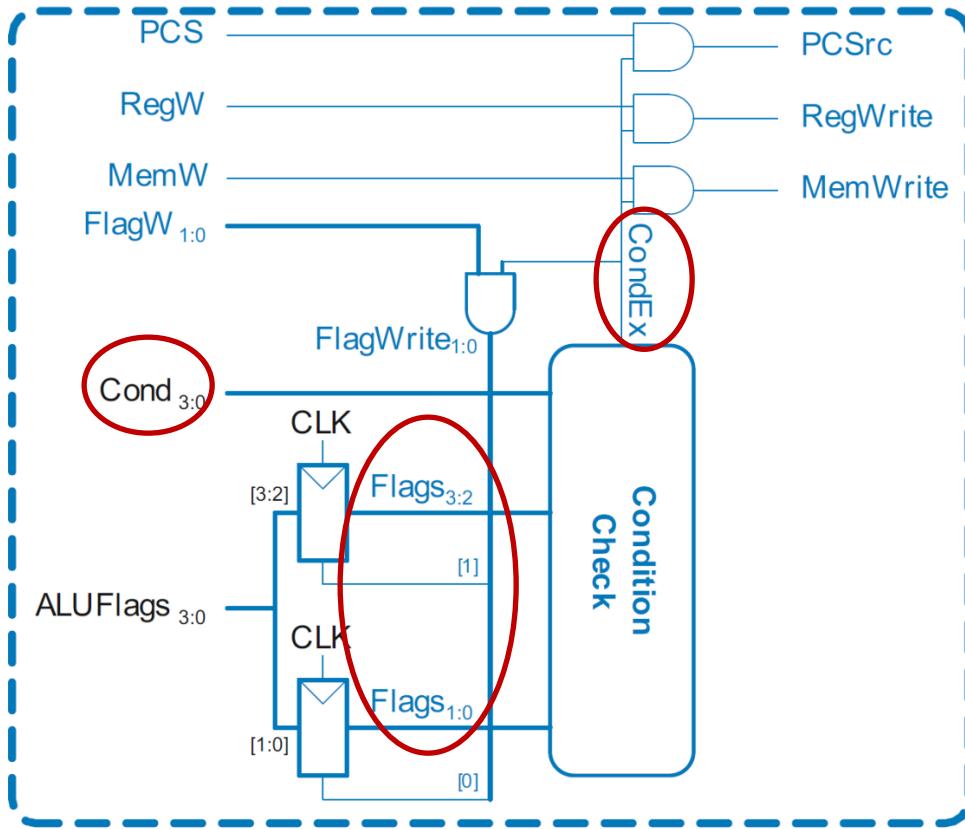
1	1	1	1	1	1	1	1
1	0	0	0	1	1	1	0
0	1	1	1	0	0	0	1

Condizioni su flags di stato

- Quindi $A + (\sim B + 1)$ corrisponde nella rappresentazione senza segno a $2^N + A - B$
 - Ora $2^N + A - B$ non genera un carry out sse $2^N + A - B \leq 2^N - 1$ sse $A + 1 \leq B$ sse $A < B$
 - Quindi $A < B$ è verificato dalla condizione \bar{C}
-
- $A \leq B$ sse $A < B$ o $A == B$ sse $Z + \bar{C}$
 - $A > B$ sse $!(A \leq B)$ sse $\bar{Z}C$

Esempi

$\text{Flags}_{3:0} = \text{NZCV}$

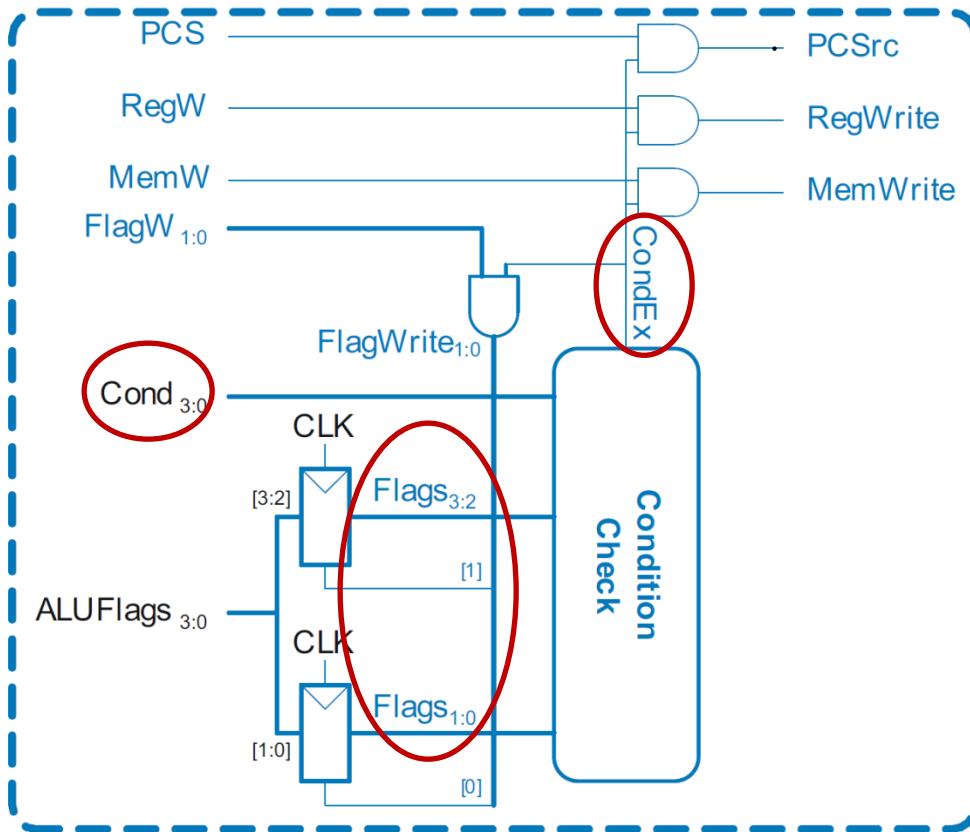


AND R1, R2, R3

$\text{Cond}_{3:0} = 1110$ (istruzione non condizionata) =>
 $\text{CondEx} = 1$

Esempi

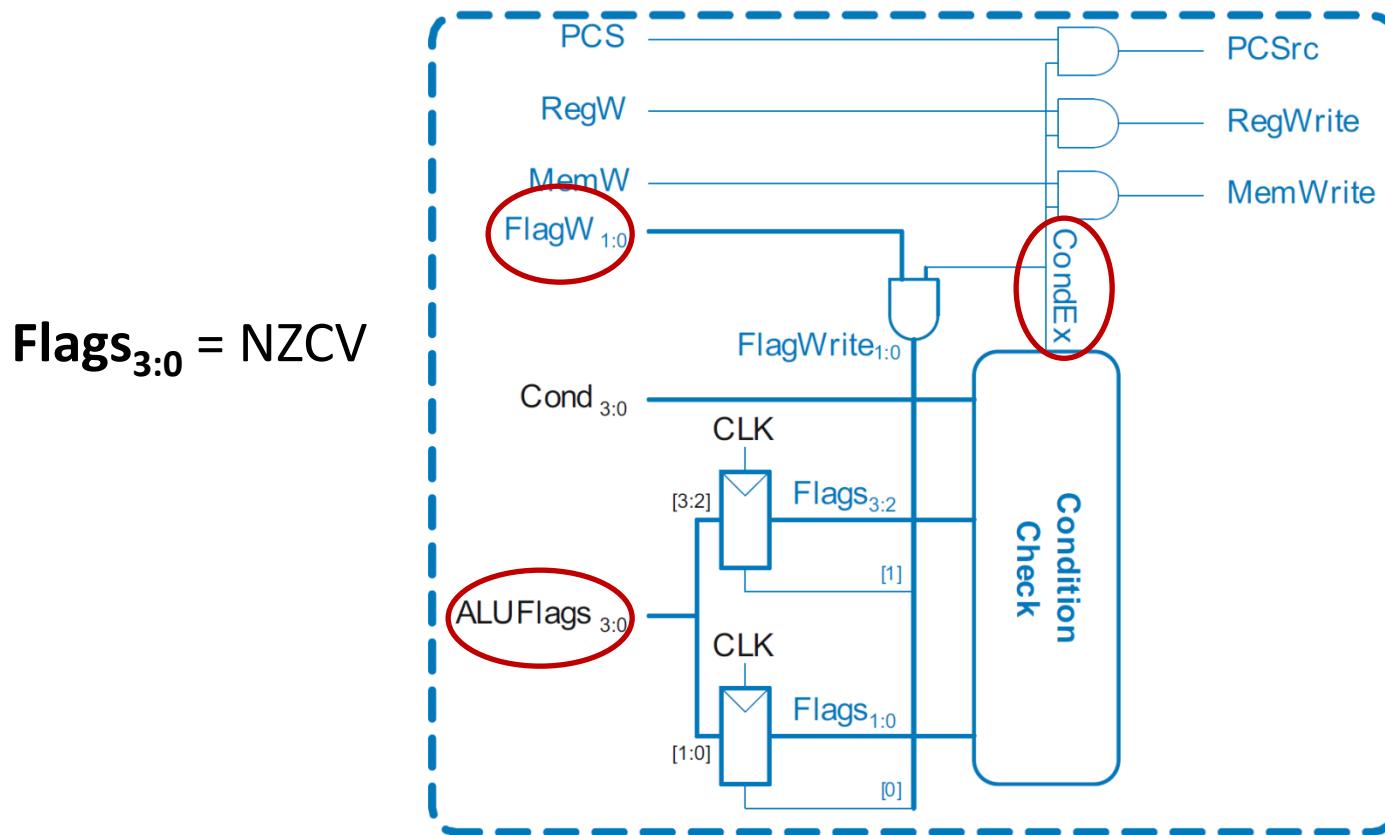
$\text{Flags}_{3:0} = \text{NZCV}$



EOREQ R5, R6, R7

Cond_{3:0}=0000 (EQ): if $\text{Flags}_{3:2}=\text{X}1\text{XX} \Rightarrow \text{CondEx} = 1$

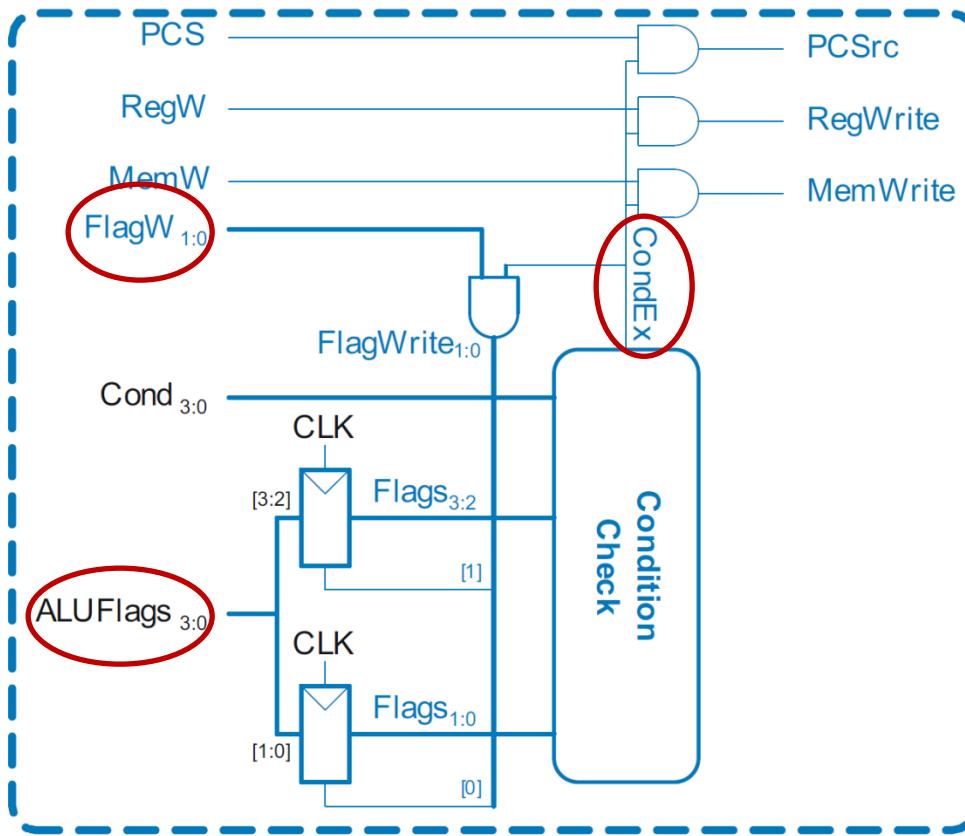
Update dei flags di stato



Flags_{3:0} vengono aggiornati con i valori di ALUFlags_{3:0} se si verificano le seguenti condizioni:

- **FlagW** è 1 (es. S-bit dell'istruzione corrente è 1)
- **CondEx** è 1 (l'istruzione deve essere eseguita)

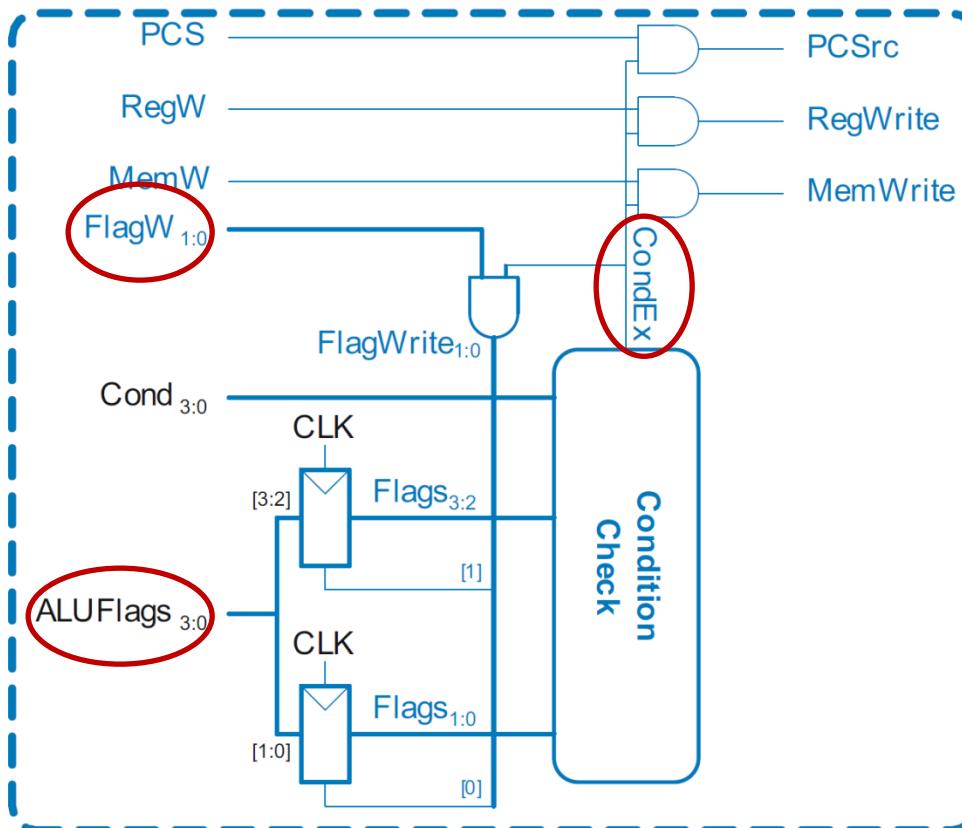
Esempi



SUBS R5, R6, R7

FlagW_{1:0} = 11 e **CondEx** = 1 (istruzione incondizionata) =>
FlagWrite_{1:0} = 11 Tutti i flag vengono aggiornati

Esempi



ANDS R7, R1, R3

FlagW_{1:0} = 10 E **CondEx** = 1 (istruzione incondizionata) =>

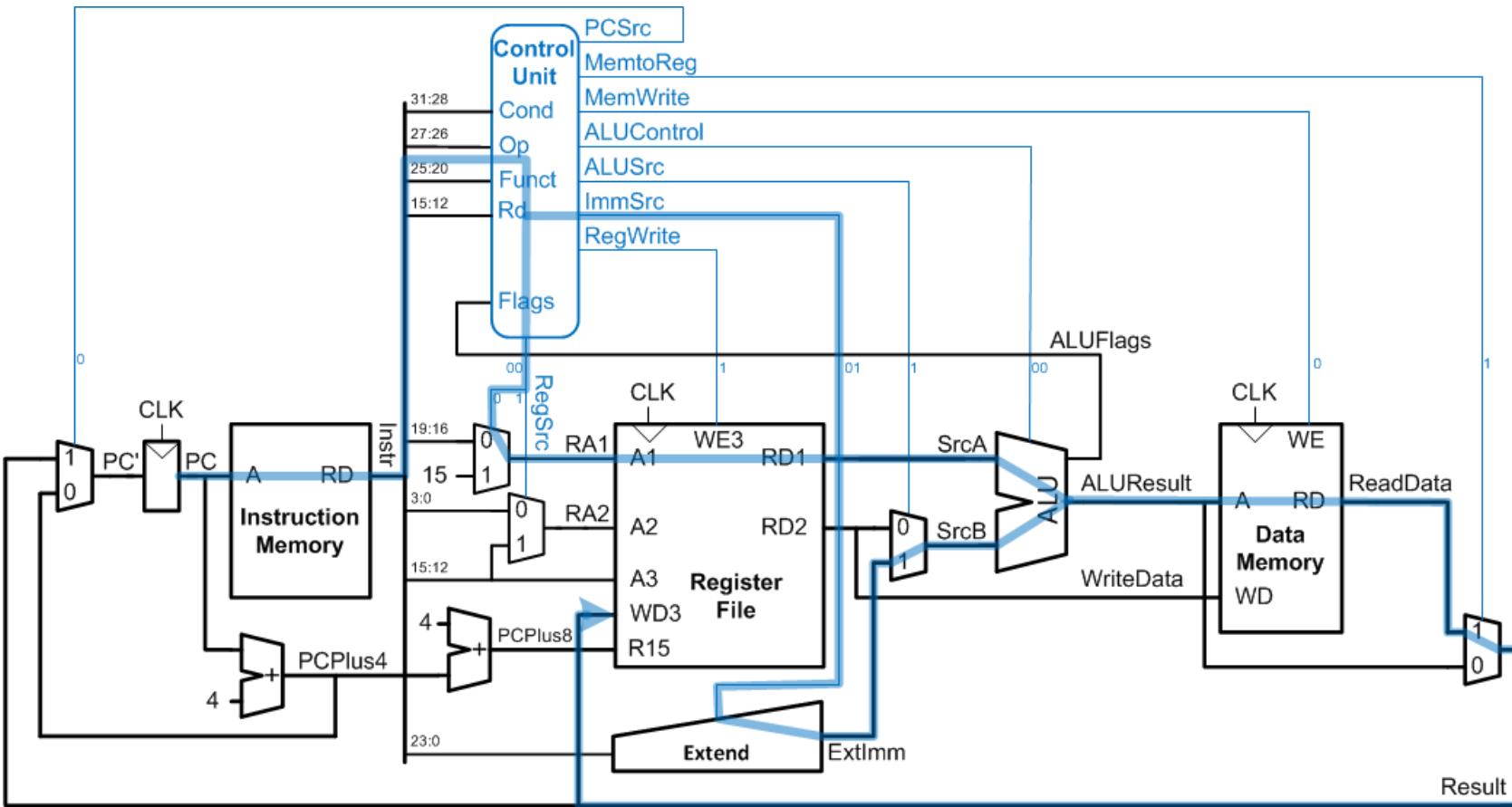
FlagWrite_{1:0} = 10 Only **Flags_{3:2}** sono aggiornati

Analisi delle prestazioni

Ogni istruzione nel processore a ciclo singolo impiega un ciclo di clock, quindi il CPI è 1.

I critical path per l'istruzione LDR sono:

- ▶ (t_{pcq_PC}) – caricamento di un nuovo indirizzo (PC) sul fronte di salita del clock;
- ▶ (t_{mem}) – lettura dell'istruzione in memoria;
- ▶ (t_{dec}) – il Decoder principale calcola RegSrc0, che induce il multiplexer a scegliere Instr_{19:16} come RA1, e il register file legge questo registro come srcA;
- ▶ ($\max[t_{mux} + t_{RFread}, t_{ext} + t_{mux}]$) – mentre il register file viene letto, il campo costante viene esteso e viene selezionata dal multiplexer ALUSrc per determinare srcB.
- ▶ (t_{ALU}) – l'ALU somma srcA e srcB per trovare l'indirizzo effettivo.
- ▶ (t_{mem}) – La memoria di dati legge da questo indirizzo.
- ▶ (t_{mux}) – il multiplexer MemtoReg seleziona ReadData.
- ▶ ($t_{RFsetup}$) – viene impostato il segnale Result ed il risultato viene scritto nel register file.



T_c limited by critical path (LDR)

Analisi delle prestazioni

Il tempo totale è dato dalla somma dei parziali:

$$T_{c1} = t_{pcq_PC} + t_{mem} + t_{dec} + \max[t_{mux} + t_{RFread}, t_{ext} + t_{mux}] + t_{ALU} + t_{mem} + t_{mux} + t_{RFsetup};$$

Nella maggior parte delle implementazioni, l'ALU, la memoria ed il register file sono sostanzialmente più lenti di altri blocchi combinatori. Pertanto, il tempo di ciclo può essere semplificato come:

$$\begin{aligned} T_{c1} = & \\ & t_{pcq_PC} + 2t_{mem} + t_{dec} + t_{RFread} \\ & + t_{ALU} + 2t_{mux} + t_{RFsetup}; \end{aligned}$$

Table 7.5 Delay of circuit elements

Element	Parameter	Delay (ps)
Register clk-to-Q	t_{pcq}	40
Register setup	t_{setup}	50
Multiplexer	t_{mux}	25
ALU	t_{ALU}	120
Decoder	t_{dec}	70
Memory read	t_{mem}	200
Register file read	t_{RFread}	100
Register file setup	$t_{RFsetup}$	60

Analisi delle prestazioni

Domanda: qual è il tempo di esecuzione per un programma con 100 miliardi di istruzioni?

Risposta:

secondo l'equazione

$$T_{c1} = t_{pcq_PC} + 2t_{mem} + t_{dec} + t_{RFread} + t_{ALU} + 2t_{mux} + t_{RFsetup}$$

il tempo di ciclo del processore singolo ciclo è

$$\begin{aligned} T_{c1} &= 40 + 2(200) + 70 + 100 + 120 + 2(25) + 60 \\ &= 840 \text{ ps.} \end{aligned}$$

Secondo l'equazione

$$\text{Tempo di esecuzione} = (\# \text{istruzioni}) \left(\frac{\text{cicli}}{\text{istruzione}} \right) \left(\frac{\text{secondi}}{\text{ciclo}} \right)$$

il tempo di esecuzione totale è

$$\begin{aligned} T_1 &= (100 \times 10^9 \text{ istruzioni}) (1 \text{ ciclo / di istruzione}) \\ &\quad (840 \times 10^{-12} \text{ s / ciclo}) = 84 \text{ secondi.} \end{aligned}$$

Table 7.5 Delay of circuit elements

Element	Parameter	Delay (ps)
Register clk-to-Q	t_{pcq}	40
Register setup	t_{setup}	50
Multiplexer	t_{mux}	25
ALU	t_{ALU}	120
Decoder	t_{dec}	70
Memory read	t_{mem}	200
Register file read	t_{RFread}	100
Register file setup	$t_{RFsetup}$	60

Limiti del ciclo singolo

Le architetture a ciclo singolo hanno tre principali limiti:

- ▶ **separazione delle memorie**: la memoria istruzioni e la memoria dati devono essere necessariamente separate, poiché dati e istruzioni devono essere gestiti all'interno dello stesso ciclo.
- ▶ **inefficienza temporale**: il ciclo di clock deve avere una durata pari al tempo impiegato dall'istruzione più lenta, sprecando tempo per tutte quelle istruzioni molto più veloci.
- ▶ **duplicazione delle componenti**: lo stesso componente non può essere riutilizzato per scopi distinti; ad esempio sono necessarie tre ALU, due per la gestione del PC e una per l'esecuzione delle istruzioni.

Vantaggi del ciclo multiplo

Le architetture a ciclo multiplo risolvono tali problemi, partizionando una intera istruzione in più passi, ciascuno dei quali viene eseguito in un ciclo di clock differente.

- ▶ È possibile utilizzare una sola memoria comune sia per le istruzioni, che per i dati. Infatti, l'istruzione viene letta in un ciclo, mentre i dati vengono letti o scritti in memoria in un ciclo differente.
- ▶ Istruzioni meno complesse richiedono un minor numero di cicli di clock, evitando sprechi di tempo.
- ▶ È possibile utilizzare un'unica ALU sia per gestire il PC, che per eseguire le istruzioni, purché tali operazioni siano effettuate in cicli di clock differenti.

Datapath

Il **datapath** è sviluppato in modo incrementale, i nuovi collegamenti sono evidenziati in **nero** (o **blu**), mentre quanto già introdotto è rappresentato in **grigio**.

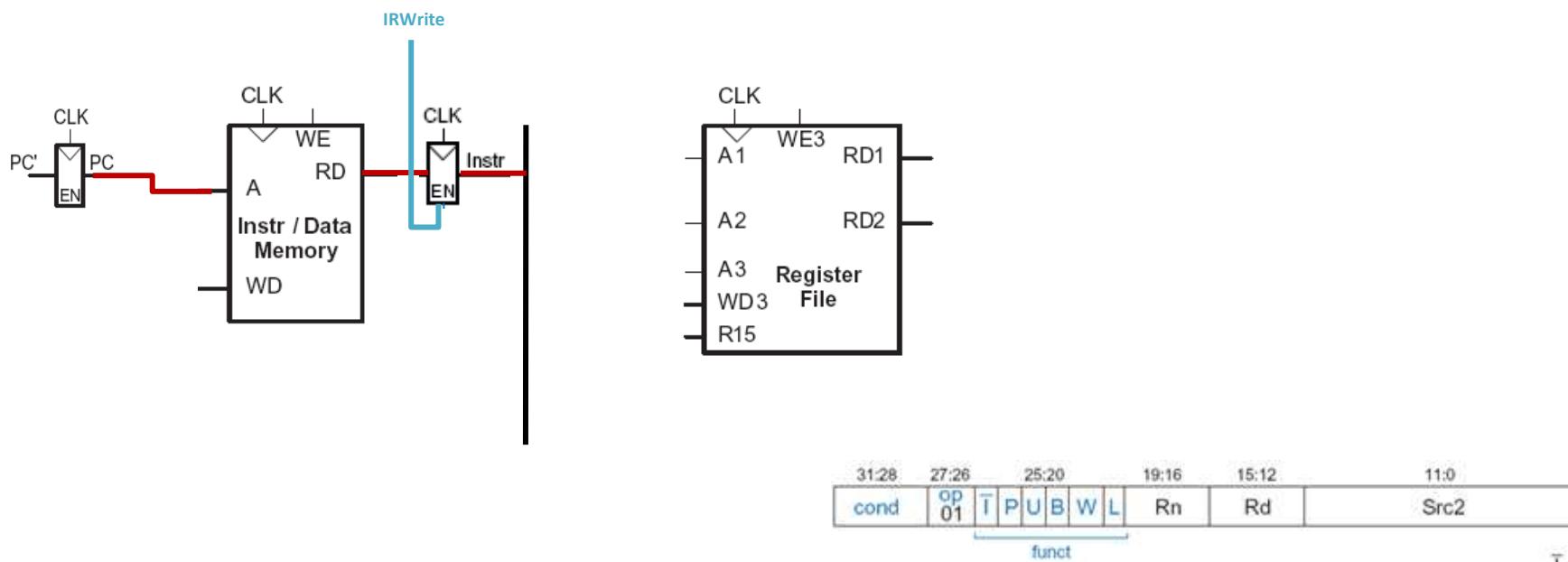
Al fine di facilitare la comprensione dell'architettura di un processore ARM, considereremo un limitato set di istruzioni:

- ▶ le istruzioni di elaborazione dati: **ADD**, **SUB**, **AND**, **ORR** (con registro e modalità di indirizzamento diretto e senza shift);
- ▶ le istruzioni di Memoria: **LDR**, **STR** (diretto e con offset positivo);
- ▶ le istruzioni di salto (branch): **B**.

Datapath LDR

Il **PC** contiene l'indirizzo dell'istruzione da eseguire. Il primo passo è quello di leggere questa istruzione dalla memoria istruzioni, per cui il PC viene collegato all'indirizzo di ingresso della memoria.

L'istruzione a 32 bit viene letta e memorizzata in un registro **IR**. Il registro **IR** riceve un segnale **IRWrite** che indica quando caricare una istruzione.

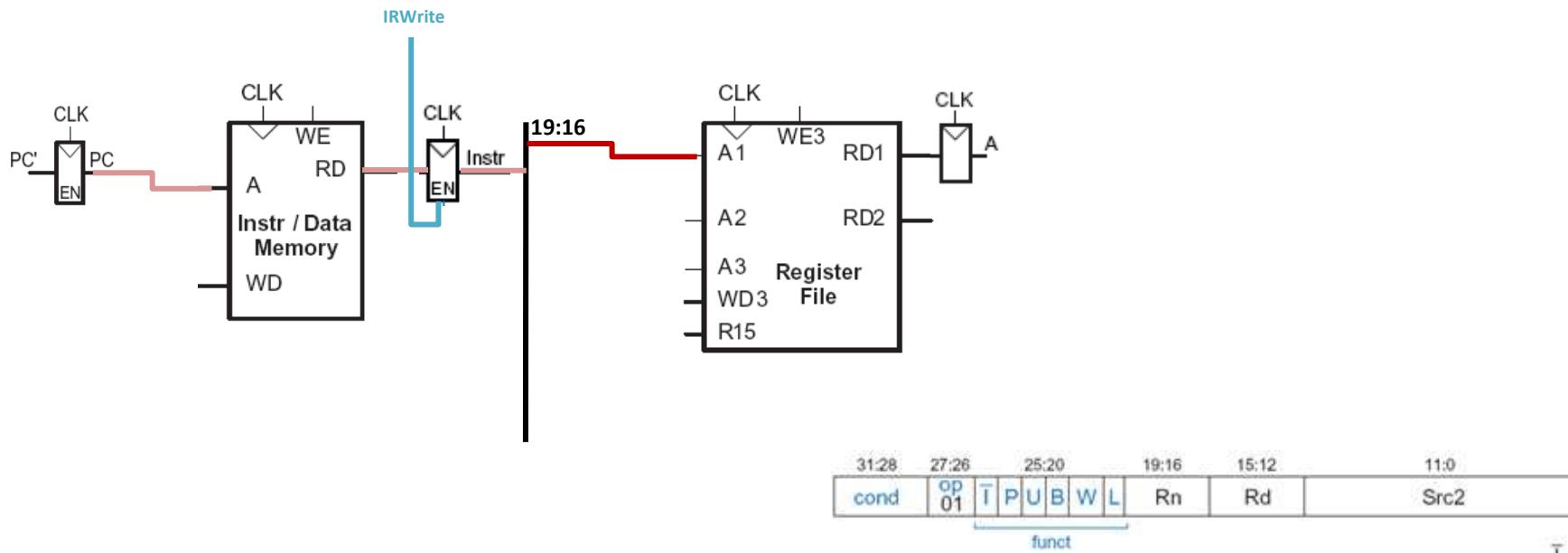


Datapath LDR

Il passo successivo è quello di leggere il registro sorgente contenente l'indirizzo di base. Questo registro è specificato nel campo **Rn** dell'istruzione, **Instr_{19:16}**

Questi bit vengono collegati all'ingresso indirizzo di una delle porte del file register (**A1**).

Il register file legge il valore di registro in **RD1** e lo memorizza in un registro **A**.



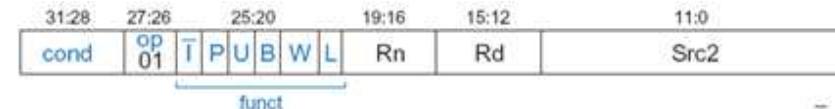
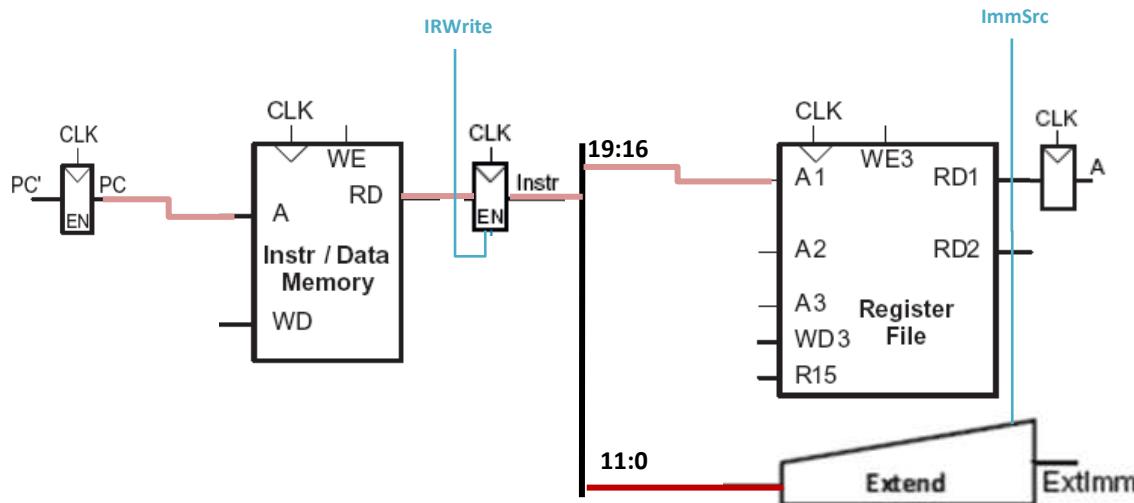
Datapath LDR

L'istruzione **LDR** richiede anche un **offset**, il quale è memorizzato nell'istruzione stessa e corrisponde ai bit $\text{Instr}_{11:0}$.

L'offset è un valore senza segno, quindi deve essere esteso a 32 bit.

Il valore a 32 bit (**ExtImm**) è tale che $\text{ExtImm}_{31:12} = 0$ e $\text{ExtImm}_{11:0} = \text{Instr}_{11:0}$.

ExtImm estende a 32 bit costanti a 8, 12 e 24 bit. Non viene memorizzato in un registro, poiché dipende solo da **Instr**, che non cambia durante l'esecuzione dell'istruzione.

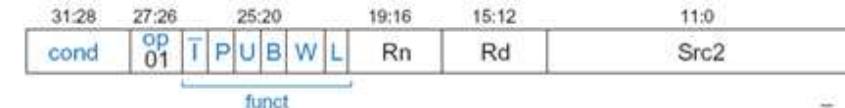
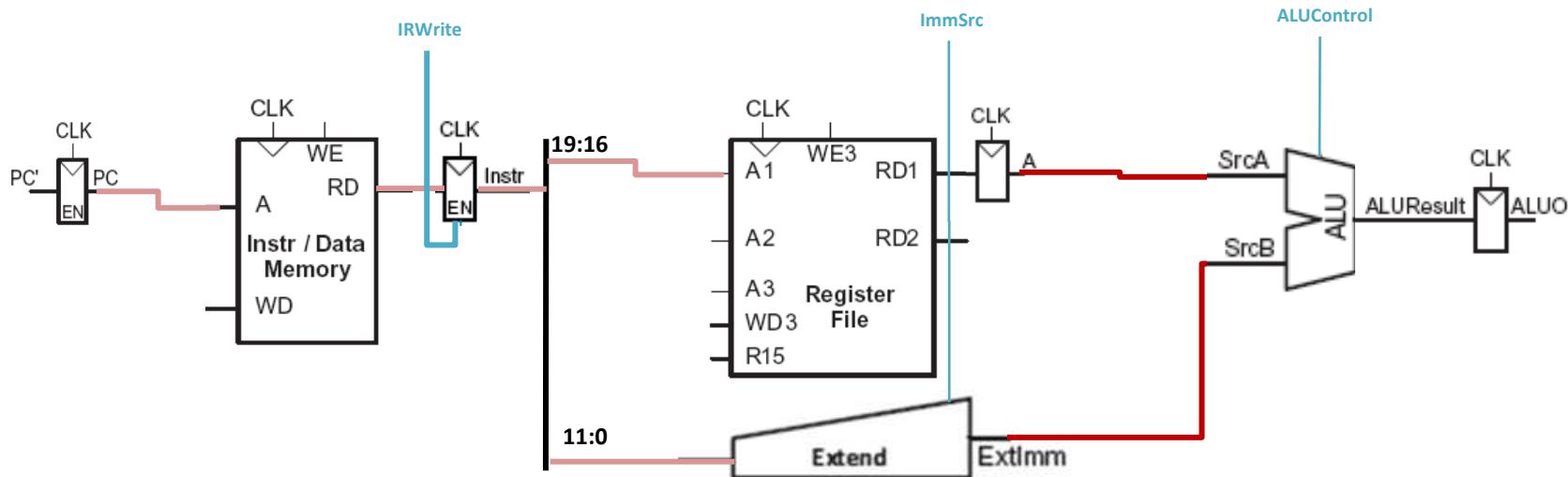


Datapath LDR

Il processore deve aggiungere l'**indirizzo di base** all'offset per trovare l'indirizzo di memoria a cui leggere. La somma è effettuata per mezzo di una **ALU**.

La **ALU** riceve due operandi (**srcA** e **srcB**). **srcA** proviene dal register file, mentre **srcB** da ExtImm. Inoltre, il segnale a 2-bit **ALUControl** specifica l'operazione (00 per somma, 01 sottrazione).

La **ALU** genera un valore a 32 bit **ALUResult**, che viene memorizzato in un registro **ALUOut**.

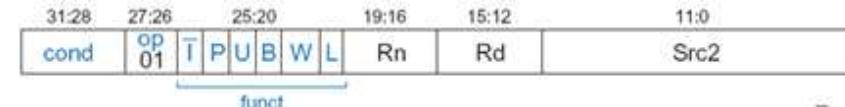
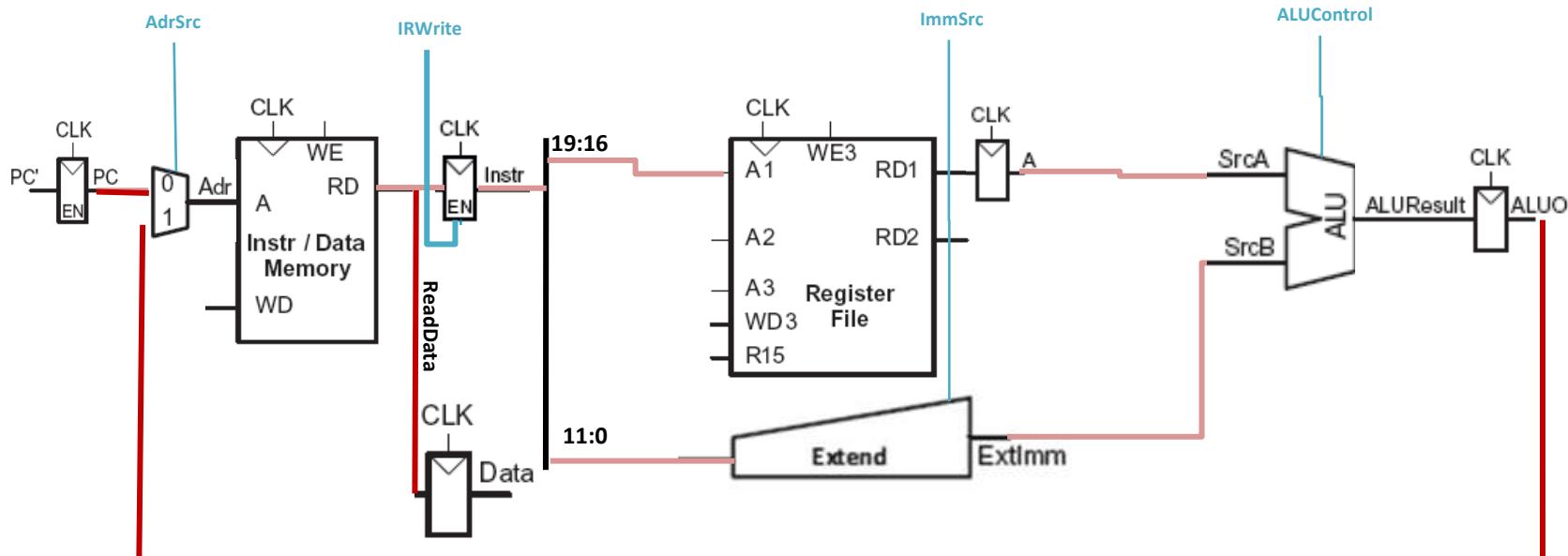


Datapath LDR

L'indirizzo calcolato dall'**ALU** deve essere inviato alla porta **A** della memoria.

Serve un multiplexer per disambiguare l'accesso con **ALUOut** o **PC**. Il multiplexer è controllato dal segnale **AdrSrc**.

I dati vengono letti dalla memoria dati sul bus **ReadData**, e poi vengono memorizzati in un registro chiamato **Data**.

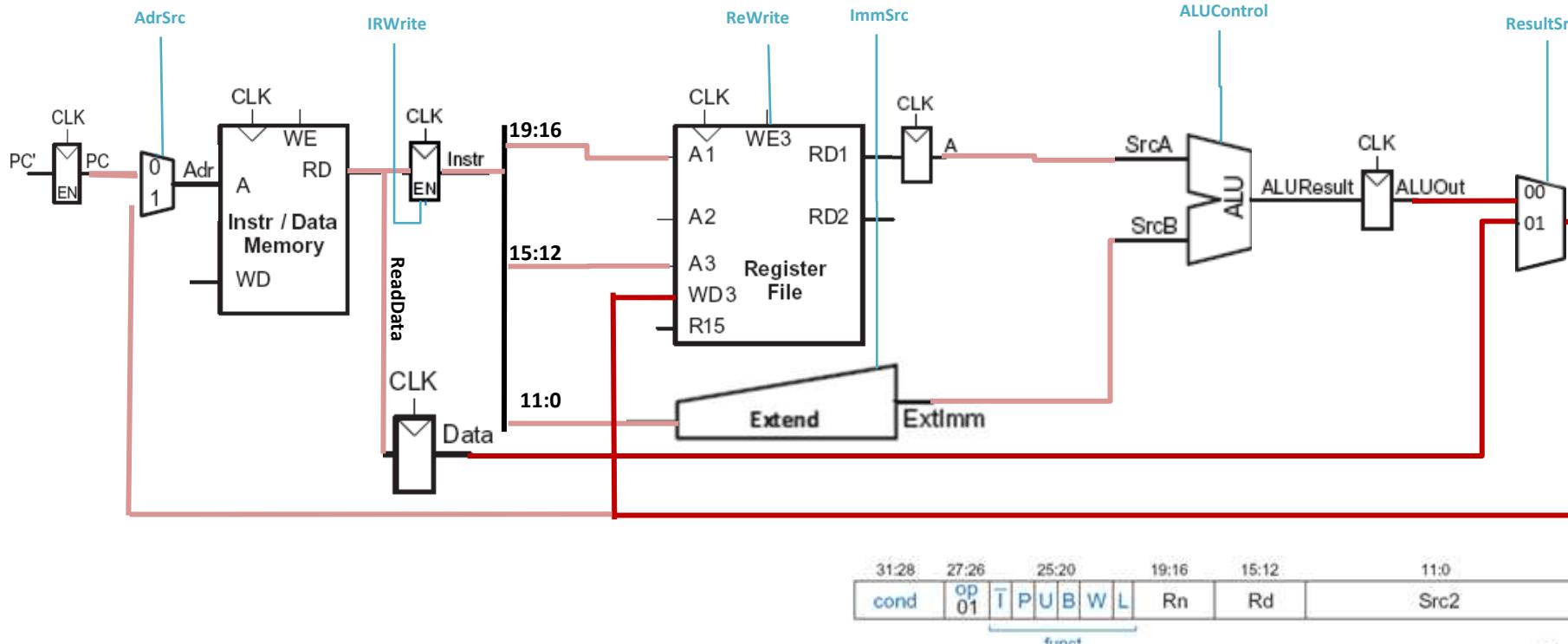


Datapath LDR

Inoltre, i dati appena letti devono essere scritti nel registro **Rd** specificato dai bit **15:12** dell'istruzione **Instr**.

Piuttosto che collegare direttamente **Data** alla porta **WD3**, si consideri che anche il risultato dell'**ALU** potrebbe dover essere scritto in **Rd**. Quindi aggiungiamo un multiplexer che seleziona fra **ALUOut** e **Data**.

Il segnale **RegWrite** deve essere impostato a **1**, per permettere la scrittura nel registro.

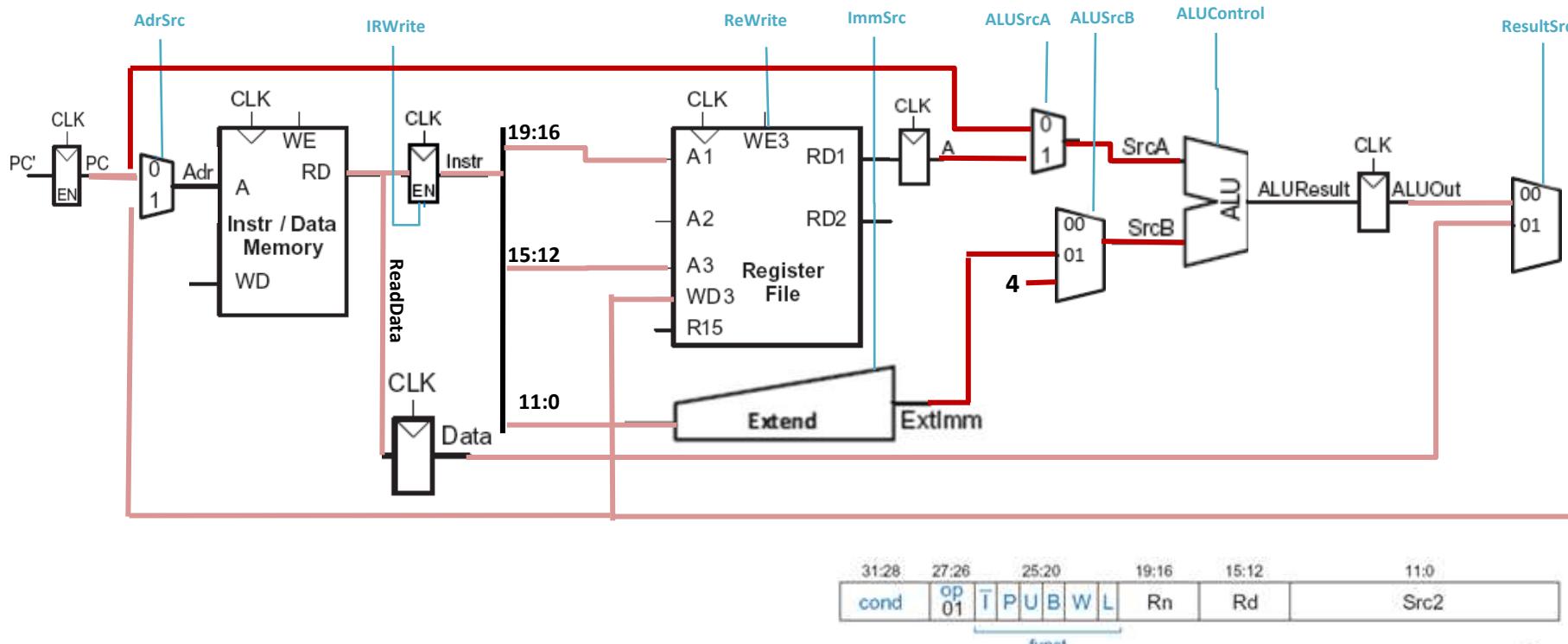


Datapath LDR

Un ulteriore compito a carico dell'**ALU** è l'incremento del **PC**, operazione che prima era svolta da una **ALU** diversa.

Aggiungiamo un multiplexer sul primo ingresso dell'**ALU**, che permette di scegliere fra il contenuto del registro **A** e il **PC**.

Sul secondo ingresso dell'ALU aggiungiamo un ulteriore multiplexer che permetta di selezionare fra ExtImm e la costante 4.

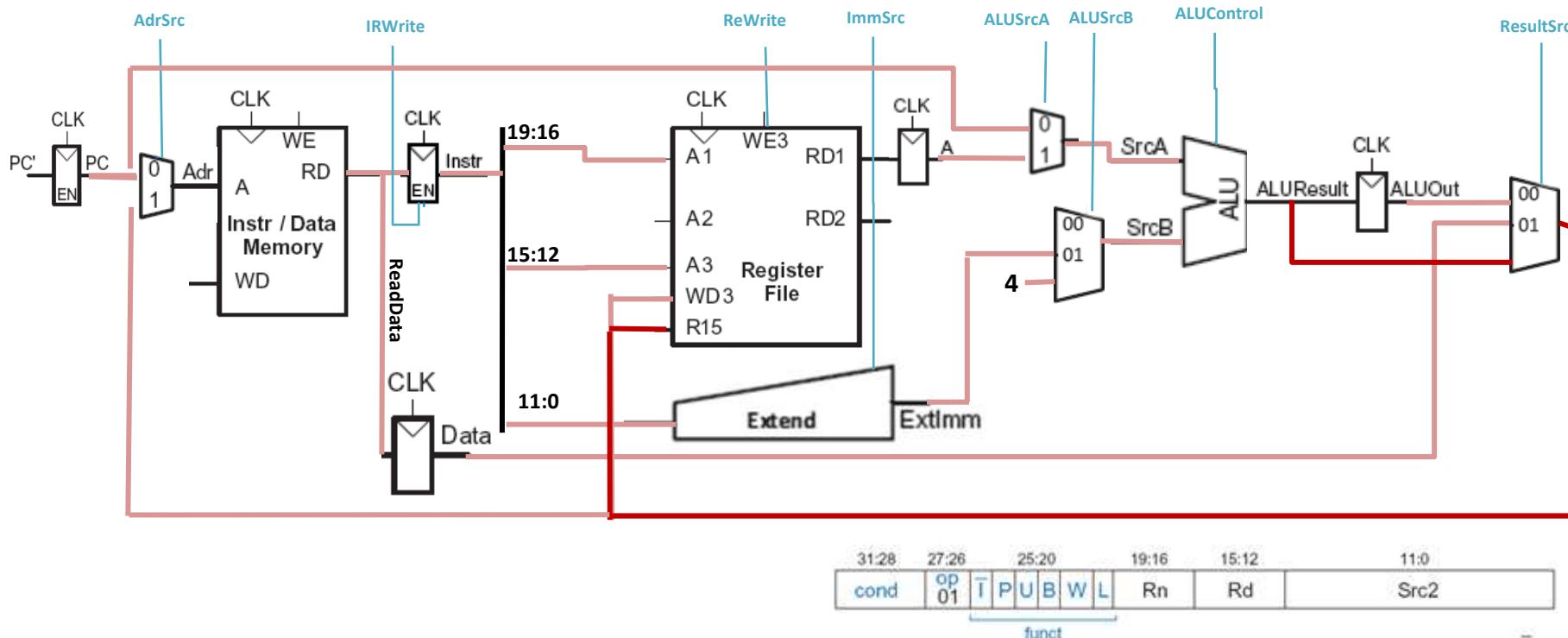


Datapath LDR

Si consideri infine, che il contenuto del registro **R15** nelle architetture ARM corrisponde a **PC+8**.

Durante il passo di fetch, il **PC** è stato aggiornato a **PC+4**, per cui sommare 4 al nuovo contenuto di **PC** produce **PC+8**, che viene memorizzato in **R15**.

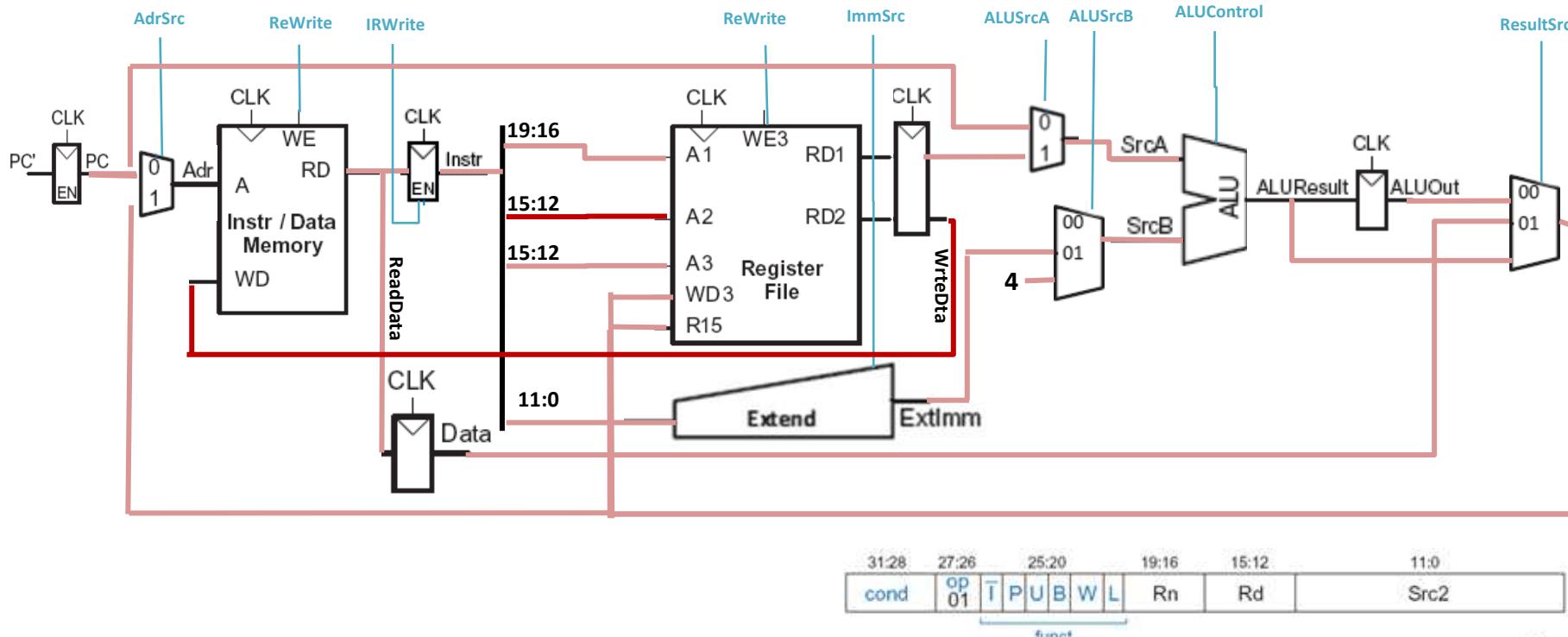
Scrivere **PC+8** in **R15**, richiede che il risultato dell'ALU possa essere collegato a tale registro. A tal fine, colleghiamo **ALUResult** con uno dei tre ingressi del multiplexer.



Datapath STR

Analogamente all'istruzione di caricamento, **STR** legge l'indirizzo di base dalla porta **RD1** del register file, estende la costante e l'**ALU** somma i due valori per calcolare l'indirizzo di memoria. Tutte queste operazioni sono già supportate.

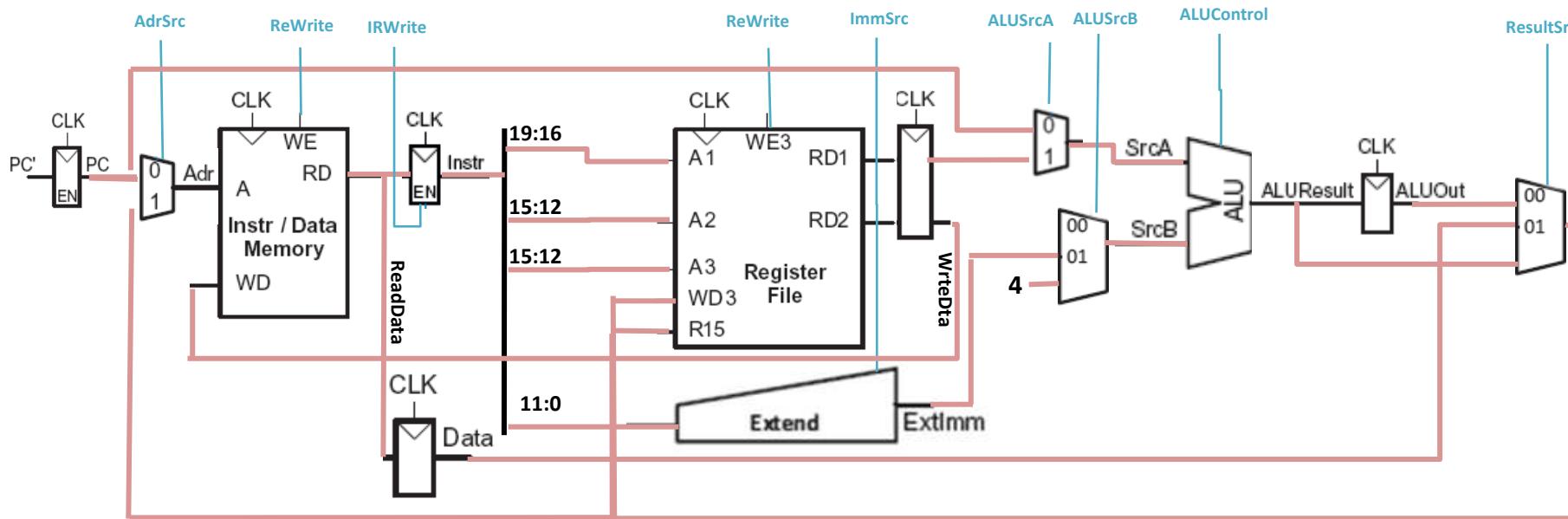
In aggiunta, **STR** legge il registro **Rd** da cui scrivere, che è specificato nei bit **Instr_{15:12}**. Il contenuto di **RD2** è inserito in un registro temporaneo **WriteData** e al passo successivo è inviato alla memoria, con segnale **MemWrite** attivo.



Datapath DataProcessing

Per le istruzioni di data processing con costante (**ADD**, **SUB**, **AND**, **OR**), il datapath legge il primo operando specificato da **Rn**, estende la costante da **8** a **32** bit, esegue l'operazione mediante l'**ALU** e scrive il risultato in un registro del register file. Tutte queste operazioni sono già supportate dal datapath.

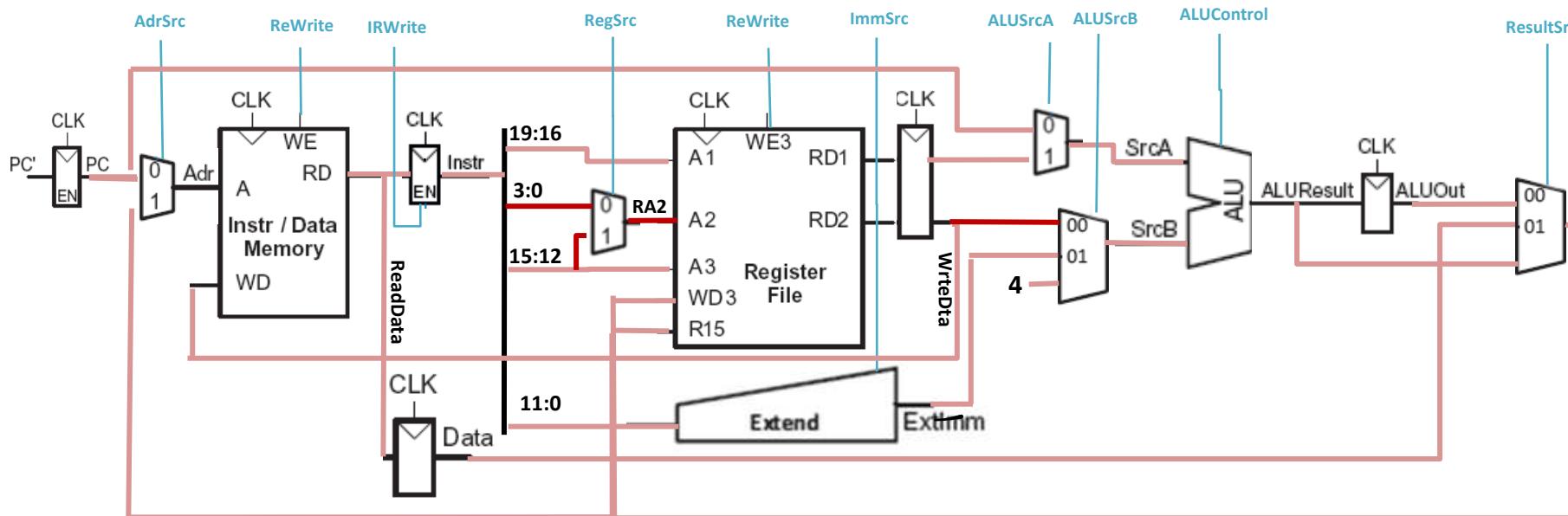
L'operazione da effettuare è specificata dal segnale **ALUControl**, mentre gli **ALUFlags** permettono di aggiornare il registro di stato.



Datapath DataProcessing

Per le istruzioni di data processing con registro (**ADD, SUB, AND, OR**), il datapath legge il secondo operando specificato da **Rm**, indicato nei bit **Instr_{3:0}**.

Inseriamo un multiplexer per selezionare tale campo sulla porta **A2** del register file. Il mutiplexer è controllato dal segnale **RegSrc**. Inoltre, estendiamo il multiplexer in **SrcB** in modo da considerare questo caso.

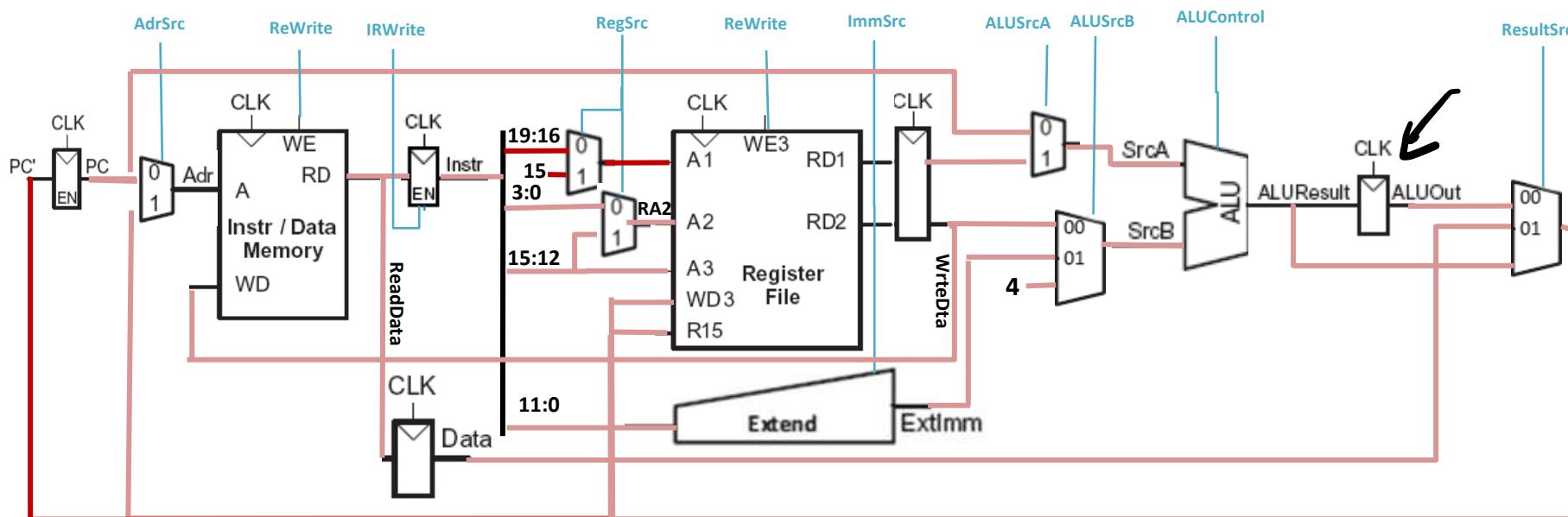


Datapath Branch

Per le istruzioni di branch, il datapath legge **PC+8** e una costante a **24** bit, che viene estesa a **32** bit. La somma di questi due valori è addizionata al **PC**.

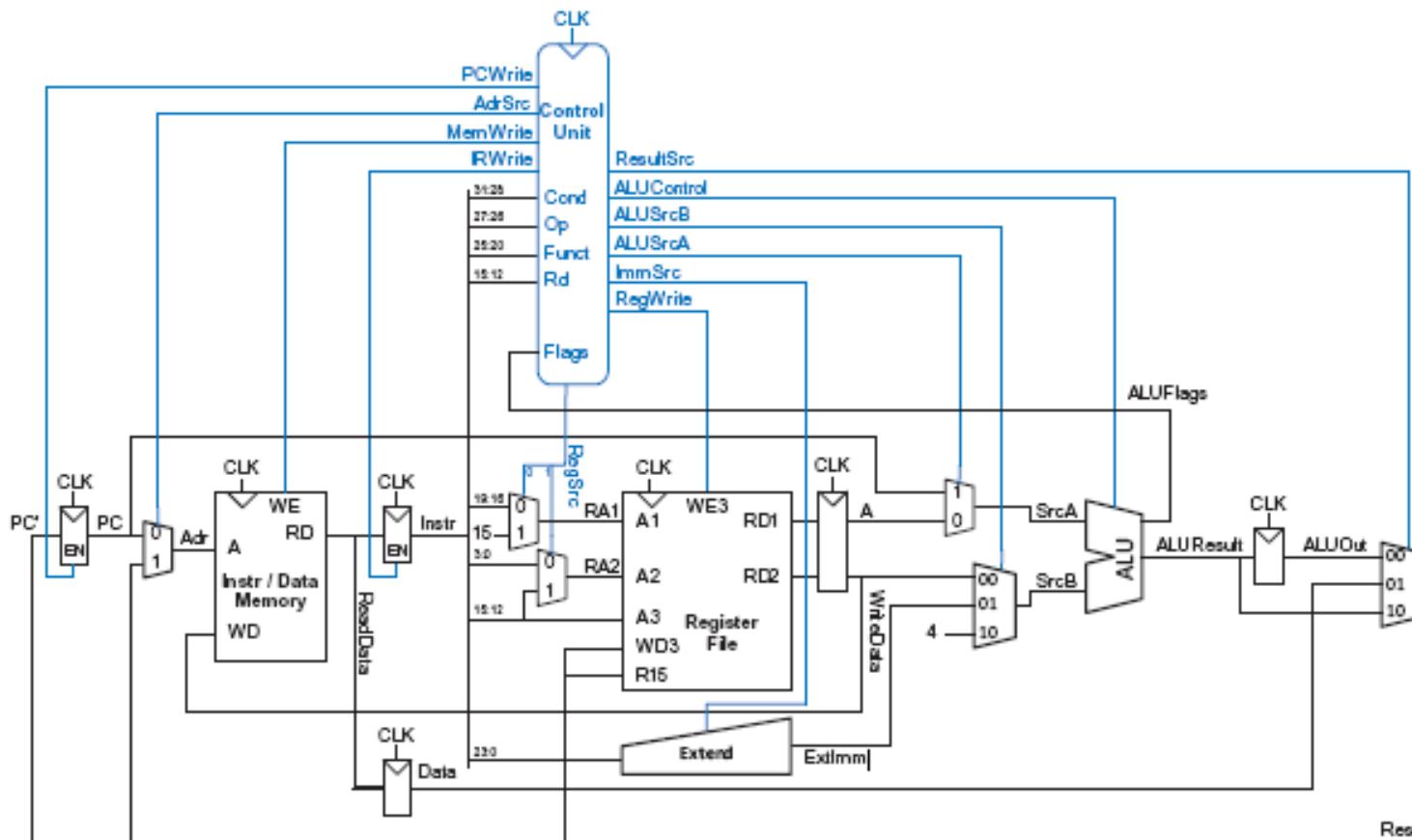
Si ricorda, inoltre, che il registro **R15** contiene il valore **PC+8** e deve essere letto per tornare da un salto. È sufficiente aggiungere un multiplexer per selezionare **R15** come input sulla porta **A1**.

Il multiplexer è controllato dal segnale **RegSrc**.



Unità di controllo

Come nel processore a ciclo singolo, l'unità di controllo genera i segnali di controllo in base ai campi **cond**, **op** e **funct** dell'istruzione ($\text{Instr}_{31:28}$, $\text{Instr}_{27:26}$, e $\text{Instr}_{25:20}$), ai flag e al fatto che il registro destinazione sia o meno il PC.

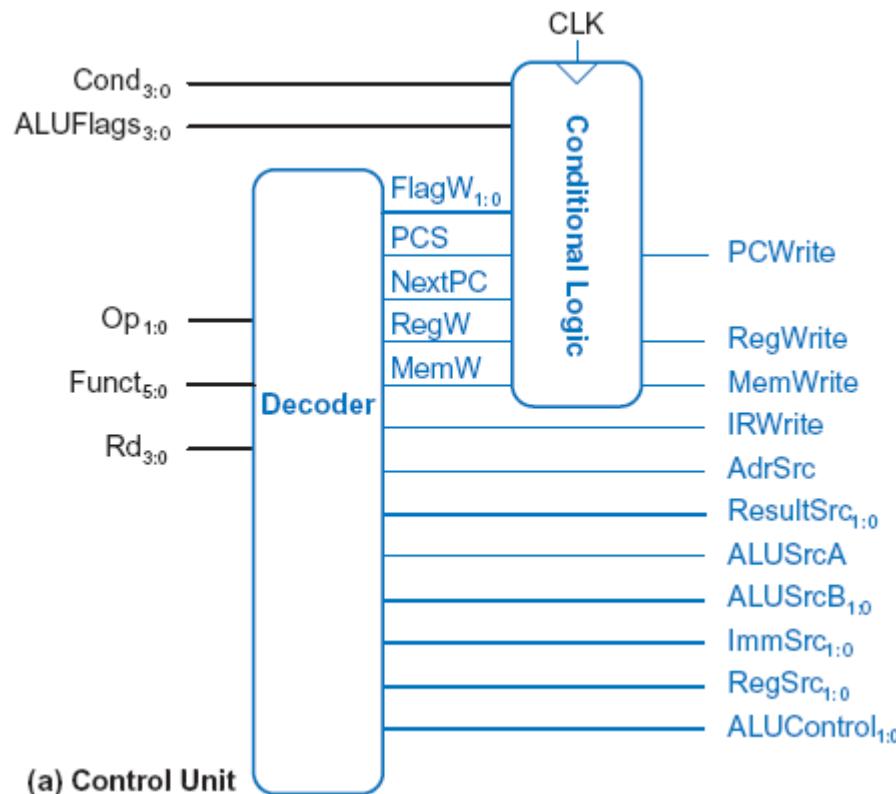


L'unità di controllo memorizza e aggiorna i flag di stato.

Unità di controllo

Come nel processore a ciclo singolo, l'unità di controllo è suddivisa in decodificatore e logica condizionale. Il decodificatore è progettato come una FSM, che produce i segnali appropriati per i diversi cicli, sulla base del proprio stato.

Il decodificatore è realizzato con una macchina di Moore, in tal modo le uscite dipendono solo dello stato attuale.



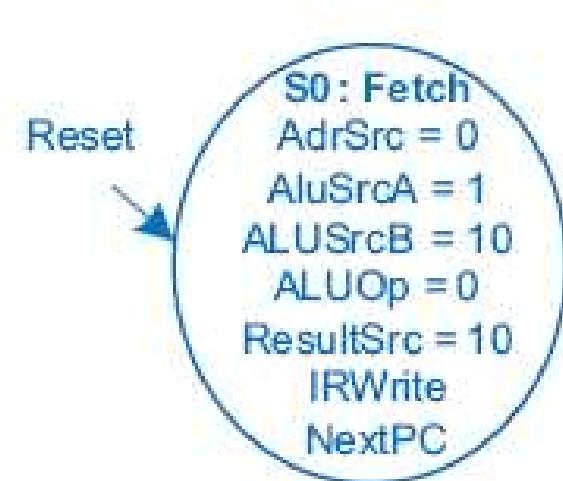
Dataflow di una istruzione

L'unità di controllo produce i segnali di attivazione per tutto il datapath (selezione nei multiplexer, abilitazione dei registri e scrittura in memoria).

Uno stato dell'automa che implementa il **main decoder** non è altro che lo stato dei segnali in un determinato momento.

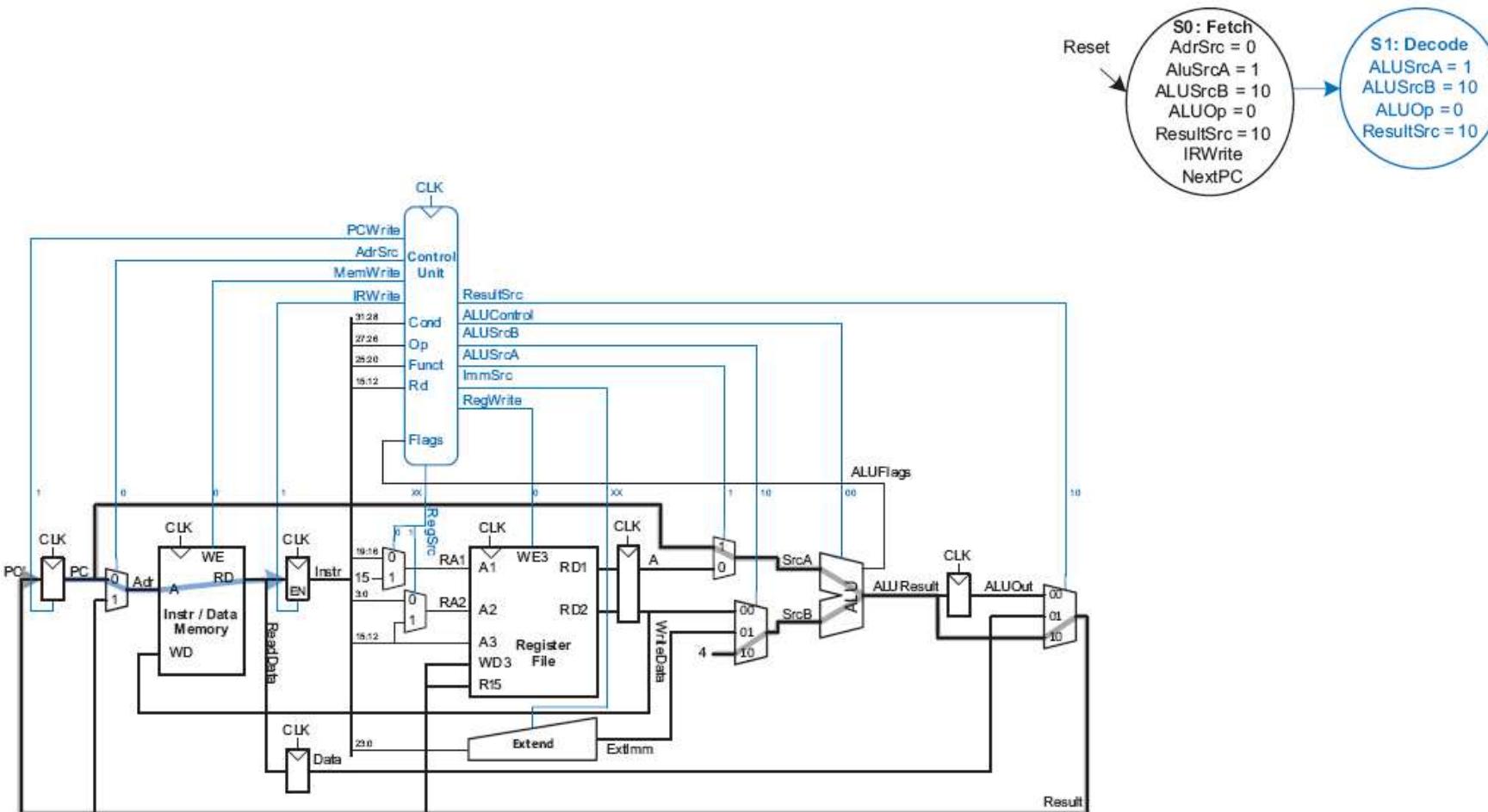
Per avere una visione più chiara di quali siano gli stati e di come vengano effettuate le transizioni da uno stato all'altro è utile considerare il **data flow** del processore.

In altri termini, data una istruzione osserviamo il comportamento del processore nei diversi cicli, in cui avvengono i quattro passi **fetch**, **decode**, **execute** e **store**.



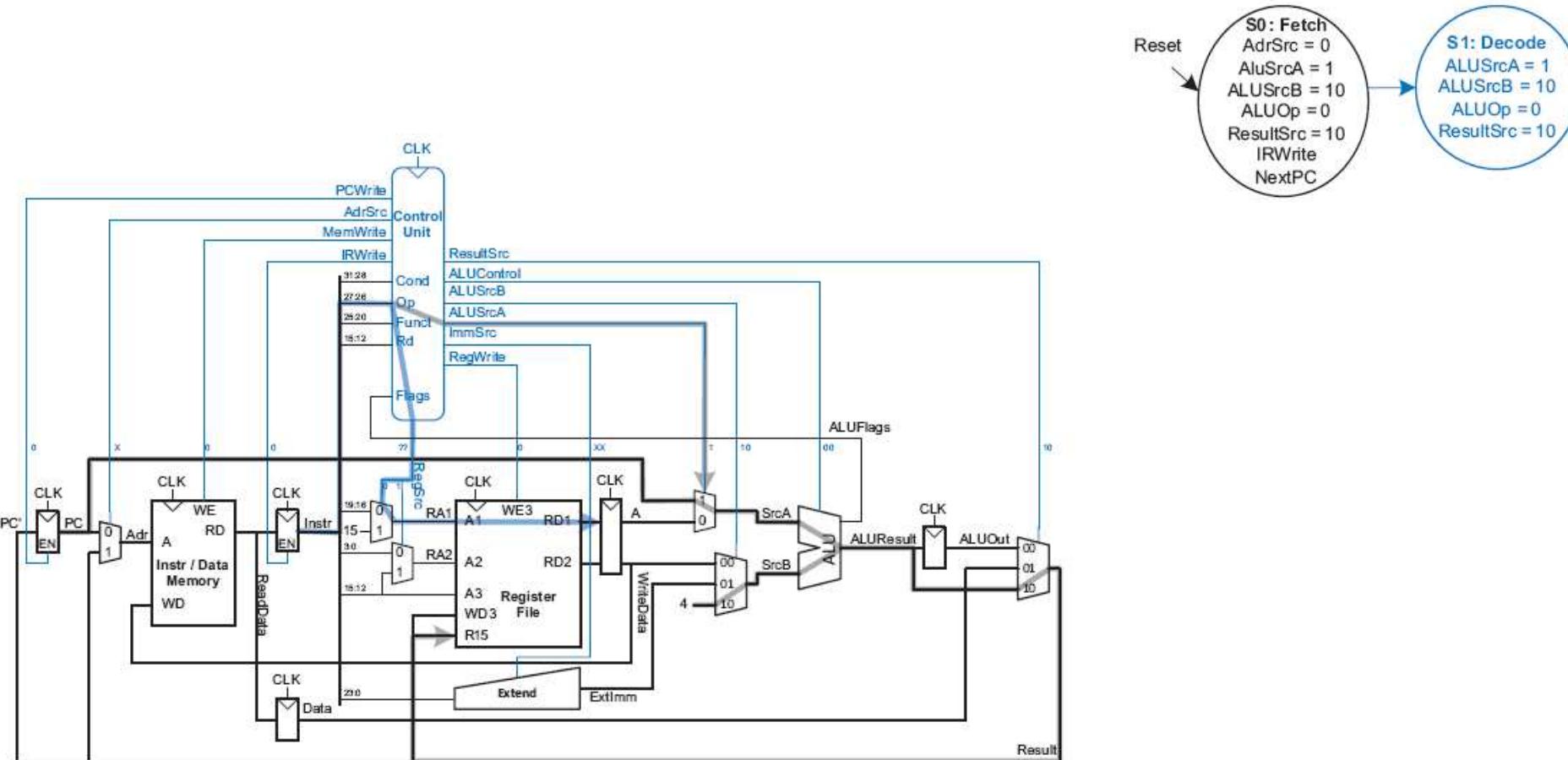
Dataflow di LDR

Operazione: **Fetch**



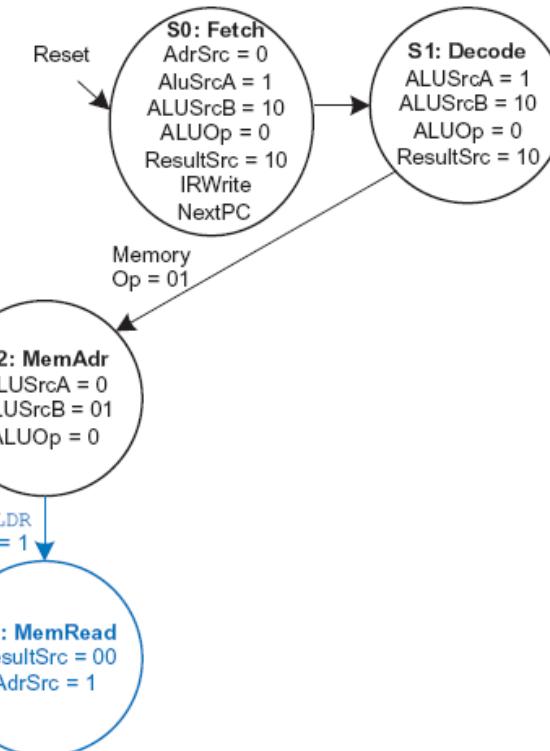
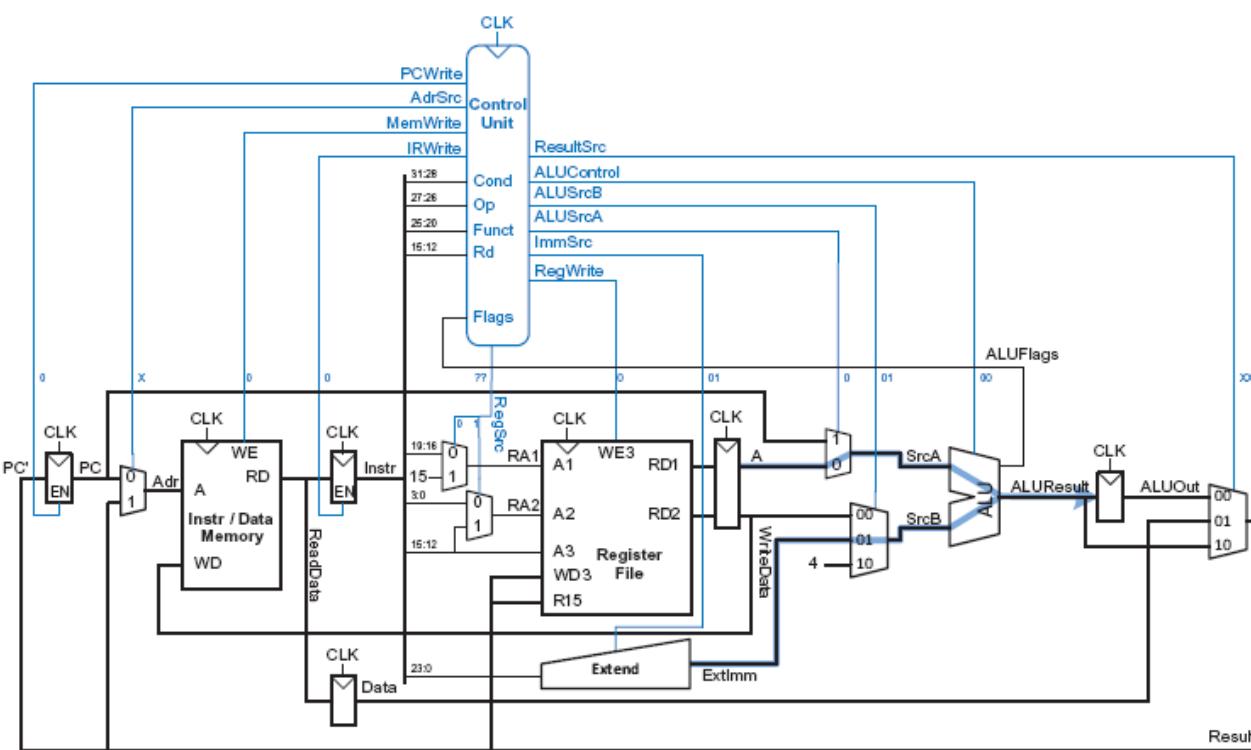
Dataflow di LDR

Operazione: **Decode**



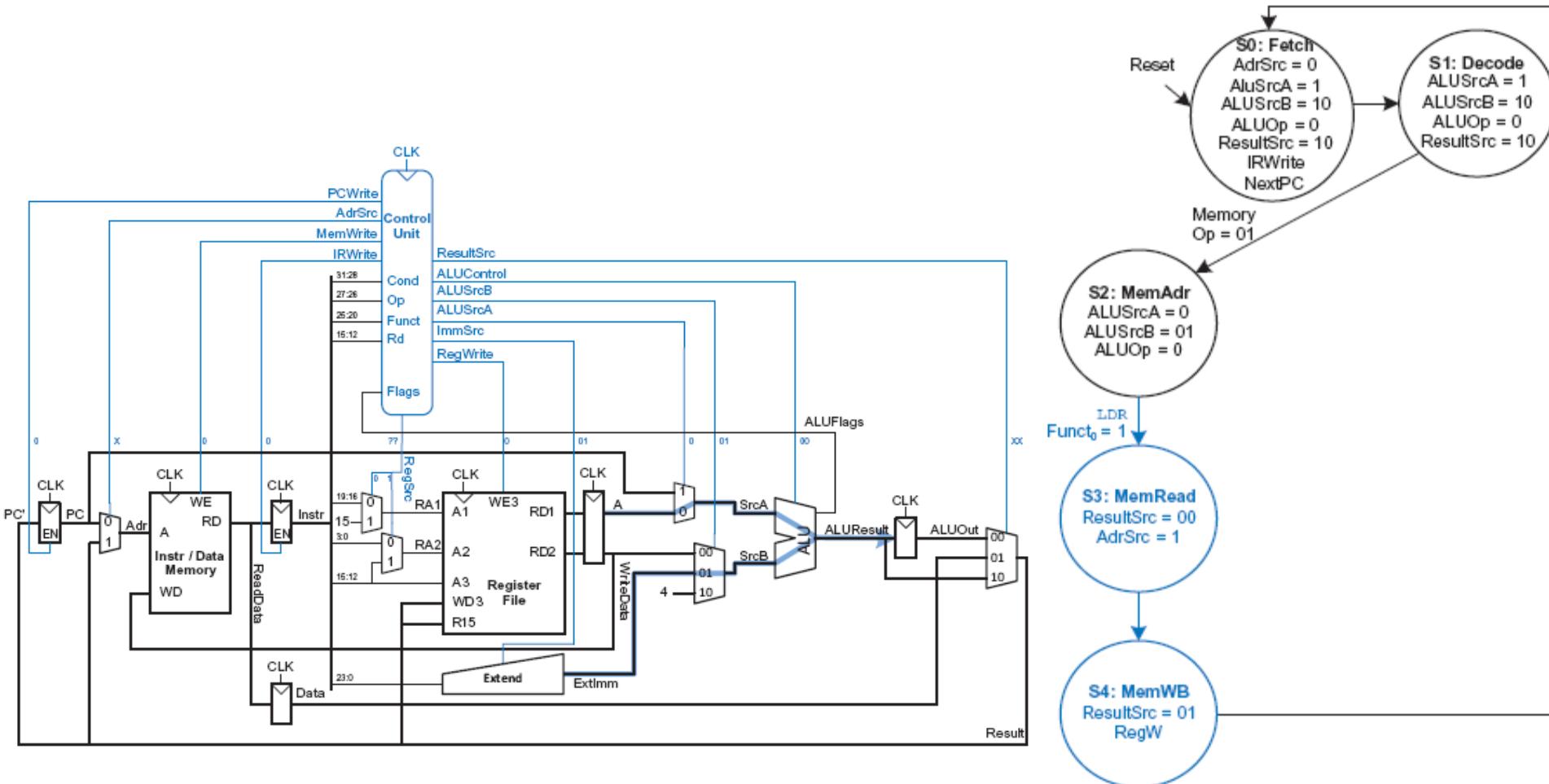
Dataflow di LDR

Operazione: **Execute** (memory address computation)



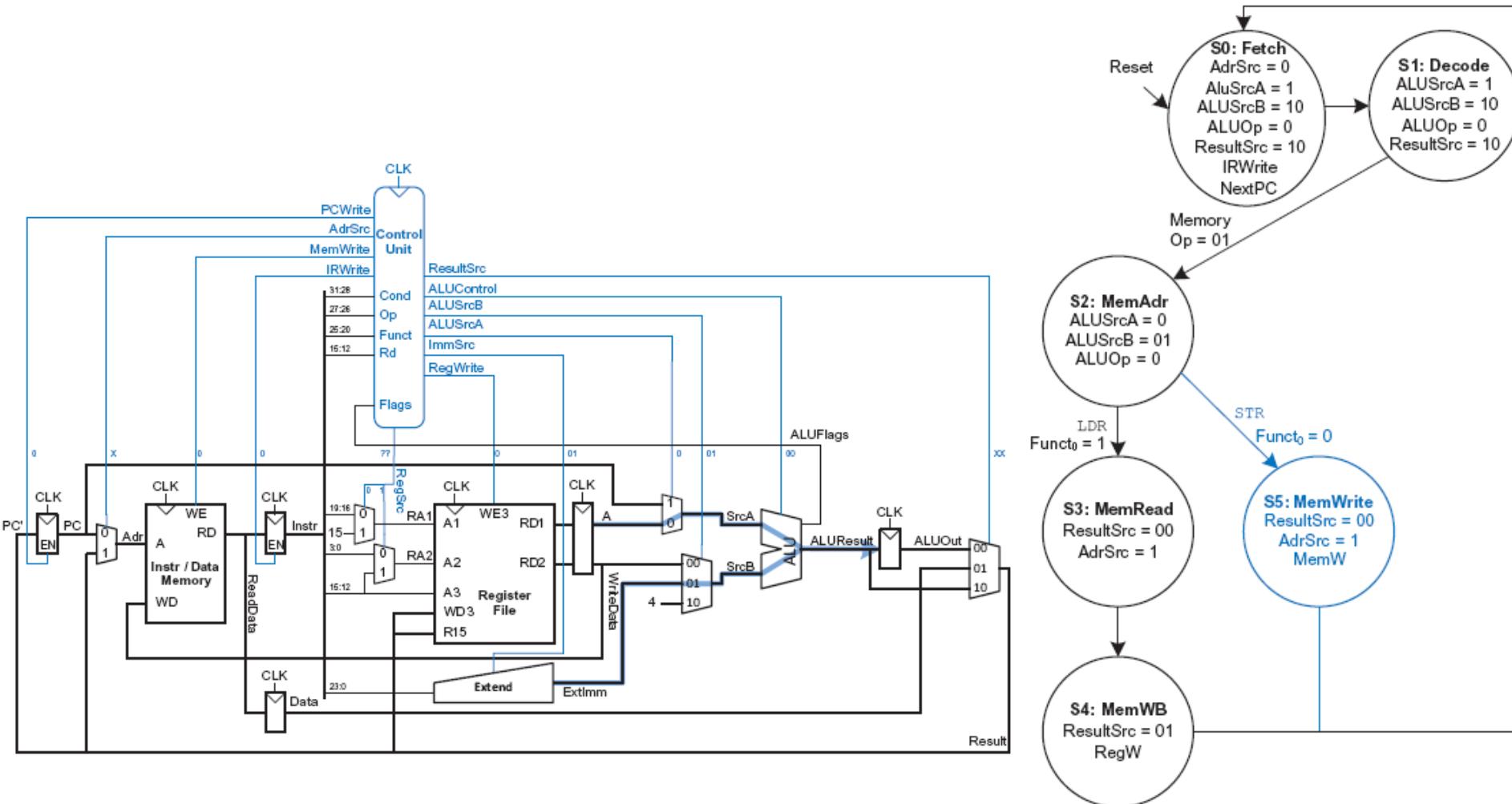
Dataflow di LDR

Operazione: **Store** (memory read)



Dataflow di STR

Operazione: **Execute (memory write)**



Analisi delle prestazioni

In un processore a ciclo multiplo, il tempo impiegato per eseguire una istruzione dipende dal numero di cicli di clock, di cui necessita e dalla durata di un singolo ciclo di clock.

Il numero di cicli di clock necessario ad eseguire le diverse istruzioni è di:

- ▶ **branch** – 3 cicli;
- ▶ **data processing** – 4 cicli;
- ▶ **memory store** – 4 cicli;
- ▶ **memory load** – 5 cicli;

Valutiamo le prestazioni del processore a ciclo multiplo rispetto al benchmark SPECINT2000, che prevede:

- ▶ **LDR** – 25%;
- ▶ **STR** – 10%;
- ▶ **B** – 13%;
- ▶ **Data Processing** – 52%;

Analisi delle prestazioni

Il numero medio di cicli per istruzione è dato da:

$$\begin{aligned}\text{CPI} &= (\text{perc. di B})(\#\text{cicli B}) + (\text{perc. di DP} + \text{perc. di STR})(\#\text{cili DP e STR}) + (\text{perc. di LDR})(\#\text{cili LDR}) = \\ &= (0.13) \quad (3) \quad + \quad (0.52 + 0.10) \quad (4) \quad + \quad (0.25) \quad (5) \quad = 4.12\end{aligned}$$

I percorsi critici nel datapath, che richiedono maggior tempo e che quindi sono predominanti, sono due:

- ▶ Dal **PC**, attraverso il multiplexer **SrcA**, attraverso l'ALU, attraverso il multiplexer **Result**, attraverso la porta **R15**, fino al registro **A**.
- ▶ Da **ALUOut**, attraverso il registro **Result**, attraverso il multiplexer **Adr**, attraverso la memoria (read), fino al registro **Data**.

- ▶ (t_{pcq_PC}) – caricamento di un nuovo indirizzo (PC) sul fronte di salita del clock;
- ▶ (t_{mux}) – selezione di un output da parte del multiplexer.
- ▶ (t_{ALU}) – l'ALU esegue su srcA e srcB una operazione.
- ▶ (t_{setup}) – viene impostato un segnale.

$$T_{c2} = t_{pcq_PC} + 2t_{mux} + \max[t_{ALU} + t_{mux}, t_{mem}] + t_{setup};$$

Analisi delle prestazioni

Domanda: qual è il tempo di esecuzione per un programma con 100 miliardi di istruzioni?

Risposta:

secondo l'equazione

$$T_{c2} = t_{pcq_PC} + 2t_{mux} + \max[t_{ALU} + t_{mux}, t_{mem}] + t_{setup};$$

il tempo di ciclo del processore a ciclo multiplo è

$$T_{c2} = 40 + 2(25) + 200 + 50 = 340 \text{ ps.}$$

Secondo l'equazione

$$\text{Tempo di esecuzione} = (\# \text{istruzioni}) \left(\frac{\text{cicli}}{\text{istruzione}} \right) \left(\frac{\text{secondi}}{\text{ciclo}} \right)$$

il tempo di esecuzione totale è

$$T_1 = (100 \times 10^9 \text{ istruzioni}) (4.12 \text{ cicli / istruzione}) (340 \times 10^{-12} \text{ s / ciclo}) = 140 \text{ secondi.}$$

Table 7.5 Delay of circuit elements

Element	Parameter	Delay (ps)
Register clk-to-Q	t_{pcq}	40
Register setup	t_{setup}	50
Multiplexer	t_{mux}	25
ALU	t_{ALU}	120
Decoder	t_{dec}	70
Memory read	t_{mem}	200
Register file read	t_{RFread}	100
Register file setup	$t_{RFsetup}$	60

Nota: il tempo impiegato dal processore a ciclo singolo per lo stesso benchmark era di 84 sec.

Considerazioni finali

Una delle motivazioni alla base della progettazione di un processore a ciclo multiplo è stata quella di evitare che il ciclo durasse quanto quello necessario all'istruzione più lenta.

Questo esempio dimostra che il processore a ciclo multiplo è più lento di quello a ciclo singolo a causa delle latenze di propagazione.

Infatti, sebbene l'istruzione più lenta (LDR) sia stata suddivisa in cinque fasi, la frequenza di ciclo del processore non è aumentata di cinque volte.

Questo in parte perché:

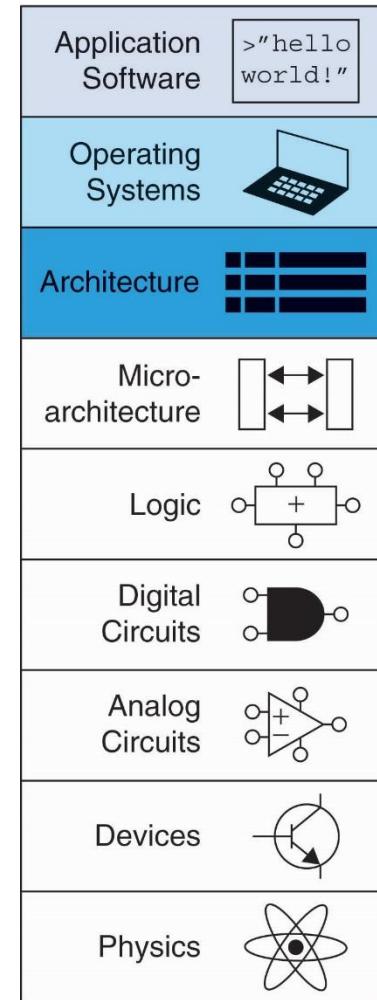
- ▶ non tutti i passaggi hanno esattamente la stessa lunghezza;
- ▶ i tempi di setup sono necessari ad ogni passo, e non solo la prima volta per l'intera istruzione.

In generale, ci si è resi conto che il fatto che alcuni calcoli siano più veloci di altri è difficile da sfruttare, a meno che le differenze sono grandi.

Architettura

Architettura: descrizione operazionale di computer

- Come un programmatore *vede* a basso livello un computer
- Definisce un insieme di istruzioni e di registri che fungono da operandi per tali istruzioni
- Istruzioni ARM
 - **Assembly language:** formato human-readable
 - **Machine language:** formato computer-readable
- Abbiamo già visto nelle lezioni precedenti come le diverse tipologie di istruzioni (memory, data-processing, e branch) sono tradotte in formato macchina



Locazione degli Operandi

- Operandi di una istruzione ARM
 - Registri
 - Costanti (*immediates*)
- ARM ha 16 registri a 32 bit (R0... R15) che sono fisicamente equivalenti fra loro, ma dal punto di vista logico sono usati con scopi specifici

Name	Use
R0	Argument / return value / temporary variable
R1-R3	Argument / temporary variables
R4-R11	Saved variables
R12	Temporary variable
R13 (SP)	Stack Pointer
R14 (LR)	Link Register
R15 (PC)	Program Counter

Generazione di costanti

E' possibile definire costanti con l'istruzione MOV:

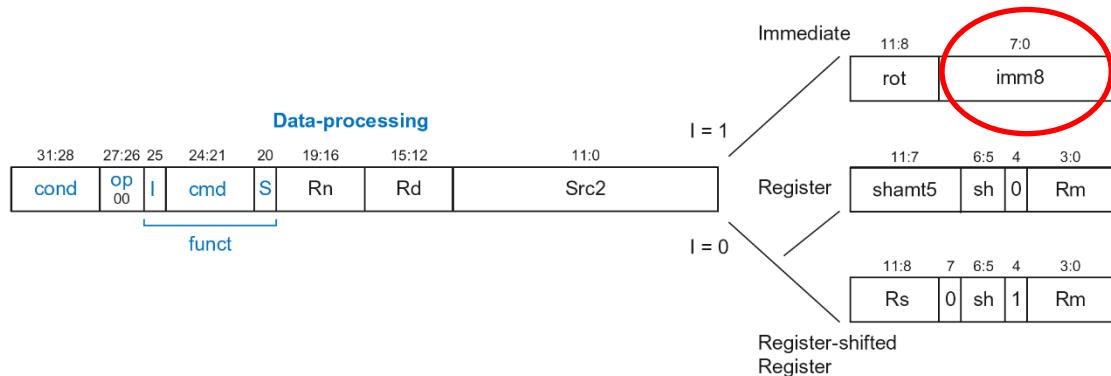
C Code

```
//int: 32-bit signed word  
int a = 23;  
int b = 0x45;
```

ARM Assembly Code

```
; R0 = a, R1 = b  
MOV R0, #23  
MOV R1, #0x45
```

Le costanti così generate hanno una precisione massima di 8 bits



Generazione di costanti

E' possibile definire costanti con l'istruzione MOV:

C Code

```
//int: 32-bit signed word  
int a = 23;  
int b = 0x45;
```

ARM Assembly Code

```
; R0 = a, R1 = b  
MOV R0, #23  
MOV R1, #0x45
```

Nota: MOV può anche essere usato per spostare il contenuto di un registro in un altro registro:

```
MOV R7, R9
```

Generazione di costanti

Generare costanti usando MOV e ORR:

C Code

```
int a = 0x7EDC8765;
```

ARM Assembly Code

```
# R0 = a
MOV R0, #0x7E000000
ORR R0, R0, #0xDC0000
ORR R0, R0, #0x8700
ORR R0, R0, #0x65
```

Istruzioni condizionali

C Code

ARM Assembly Code

```
; R0=f, R1=g, R2=h, R3=i, R4=j

if (i == j)      CMP R3, R4          ; set flags with R3-R4
    f = g + h;   BNE L1           ; if i!=j, skip if block
                  ADD R0, R1, R2 ; f = g + h

L1
f = f - i;       SUB R0, R0, R2 ; f = f - i
```

Nota: il codice assembly effettua il test opposto ($i \neq j$) rispetto a quello di alto livello ($i == j$)

Istruzioni condizionali

C Code

```
if (i == j)
    f = g + h;
f = f - i;
```

ARM Assembly Code

;R0=f, R1=g, R2=h, R3=i, R4=j

if (i == j)	CMP R3, R4 ; set flags with R3-R4
f = g + h;	ADDEQ R0, R1, R2 ; if (i==j) f = g + h
f = f - i;	SUB R0, R0, R2 ; f = f - i

Istruzioni condizionali

Codice alternativo per piccoli blocchi di codice:

Original

```
CMP R3, R4  
BNE L1  
ADD R0, R1, R2  
L1  
SUB R0, R0, R2
```

Alternate Assembly Code

```
;R0=f, R1=g, R2=h, R3=i, R4=j  
  
CMP R3, R4          ; set flags with R3-R4  
ADDEQ R0, R1, R2   ; if (i==j) f = g + h  
SUB R0, R0, R2     ; f = f - i
```

Istruzioni condizionali

C Code

```
if (i == j)  
    f = g + h;  
  
else  
    f = f - i;
```

ARM Assembly Code

```
;R0=f, R1=g, R2=h, R3=i, R4=j  
  
CMP R3, R4          ; set flags with R3-R4  
BNE L1              ; if i!=j, skip if block  
ADD R0, R1, R2      ; f = g + h  
B L2                ; branch past else block  
  
.L1  
SUB R0, R0, R2      ; f = f - i  
.L2
```

Istruzioni condizionali

Codice alternativo per piccoli blocchi di codice:

Original

```
CMP R3, R4  
BNE L1  
ADD R0, R1, R2  
B L2  
L1  
SUB R0, R0, R2  
L2
```

Alternate Assembly Code

```
;R0=f, R1=g, R2=h, R3=i, R4=j  
  
CMP R3, R4 ; set flags with R3-R4  
ADDEQ R0, R1, R2 ; if (i==j) f = g + h  
SUBNE R0, R0, R2 ; else f = f - i
```

Istruzioni iterative

C Code

```
// determines the power ; R0 = pow, R1 = x
// of x such that 2x = 128 MOV R0, #1 ; pow = 1
int pow = 1;           MOV R1, #0 ; x = 0
int x    = 0;
```

```
while (pow != 128) {
    pow = pow * 2;
    x  = x + 1;
}
```

ARM Assembly Code

```
; R0 = pow, R1 = x
MOV R0, #1 ; pow = 1
MOV R1, #0 ; x = 0

WHILE
    CMP R0, #128 ; R0-128
    BEQ DONE ; if (pow==128)
              ; exit loop
    LSL R0, R0, #1 ; pow=pow*2
    ADD R1, R1, #1 ; x=x+1
    B WHILE ; repeat loop

DONE
```

Il codice assembly verifica la condizione opposta (`pow == 128`) a quella del C (`pow != 128`).

Istruzioni iterative

```
for (initialization; condition; loop operation)  
    statement
```

- **initialization**: eseguita prima che il loop inizi
- **condition**: condizione di continuazione che è verificata all'inizio di ogni iterazione
- **loop operation**: eseguita alla fine di ogni iterazione
- **statement**: eseguito ad ogni iterazione, ovvero fintantoché la condizione di continuazione è verificata

Istruzioni iterative

C Code

```
// adds numbers from 1-9 ; R0 = i, R1 = sum
```

```
int sum = 0
```

```
for (i=1; i!=10; i=i+1)
```

```
    sum = sum + i;
```

ARM Assembly Code

```
; R0 = i, R1 = sum
```

```
MOV R0, #1 ; i = 1
```

```
MOV R1, #0 ; sum = 0
```

```
FOR
```

```
CMP R0, #10 ; R0-10
```

```
BEQ DONE ; if (i==10)  
          ; exit loop
```

```
ADD R1, R1, R0 ; sum=sum + i
```

```
ADD R0, R0, #1 ; i = i + 1
```

```
B FOR ; repeat loop
```

```
DONE
```

Istruzioni iterative

In ARM, i loop decrescenti fino a 0 sono più efficienti

C Code

```
// adds numbers from 1-9  
int sum = 0  
  
for (i=9; i!=0; i=i-1)  
    sum = sum + i;
```

ARM Assembly Code

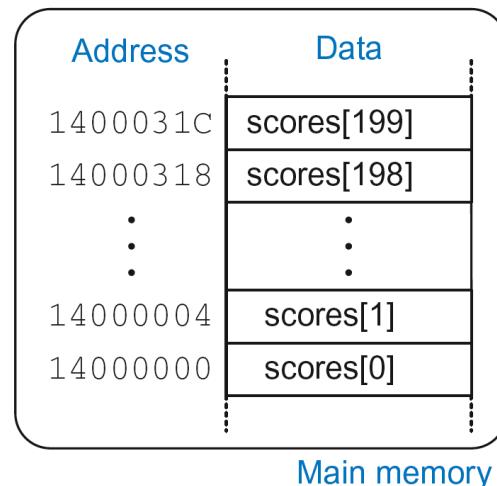
```
; R0 = i, R1 = sum  
MOV R0, #9 ; i = 9  
MOV R1, #0 ; sum = 0  
  
FOR  
    ADD R1, R1, R0 ; sum=sum + i  
    SUBS R0, R0, #1 ; i = i - 1  
                ; and set flags  
    BNE FOR ; if (i!=0)  
            ; repeat loop
```

Si risparmiano 2 istruzioni per ogni iterazione:

- Si accorpano decremento e comparazione: SUBS R0, R0, #1
- Solo un branch invece di due

Arrays

- Consente l'accesso ad una quantità di dati simili
 - **Index:** accesso ad un qualche elemento
 - **Size:** numero di elementi
- Esempio: array di parole (32bit)
 - **Base address** = 0x14000000 (indirizzo di scores[0])
 - Gli altri elementi sono raggiungibili a partire dal base address



Arrays

C Code

```
int array[5];
array[0] = array[0] * 8;
array[1] = array[1] * 8;
```

ARM Assembly Code

```
; R0 = array base address
MOV R0, #0x60000000          ; R0 = 0x60000000
LDR R1, [R0]                  ; R1 = array[0]
LSL R1, R1, 3                ; R1 = R1 << 3 = R1*8
STR R1, [R0]                  ; array[0] = R1
LDR R1, [R0, #4]              ; R1 = array[1]
LSL R1, R1, 3                ; R1 = R1 << 3 = R1*8
STR R1, [R0, #4]              ; array[1] = R1
```

Arrays

C Code

```
int array[200];
int i;

for (i=199; i >= 0; i = i - 1)
    array[i] = array[i] * 8;
```

ARM Assembly Code

```
; R0 = array base address, R1 = i
MOV R0, 0x60000000
MOV R1, #199          .

FOR
    LDR R2, [R0, R1, LSL #2]      ; R2 = array(i)
    LSL R2, R2, #3                ; R2 = R2<<3 = R2*8
    STR R2, [R0, R1, LSL #2]      ; array(i) = R2
    SUBS R1, R1, #1               ; i = i - 1
                                ; and set flags
    BPL FOR                      ; if (i>=0) repeat loop
```

Chiamate di funzioni

- **Caller:** funzione chiamante, in questo caso main
- **Callee:** funzione chiamata, in questo caso sum

C Code

```
void main()
{
    int y;
    y = sum(42, 7);
    ...
}

int sum(int a, int b)
{
    return (a + b);
}
```

Contratto

- **Caller:**
 - Passa gli argomenti al callee
 - Esegue un jump al callee
- **Callee:**
 - Esegue la funzione chiamata
 - ritorna il risultato al caller
 - ritorna nel punto di chiamata del caller
 - non deve sovrascrivere i registri e la memoria usata dal caller

Convenzioni ARM per chiamata a funzione

- **Chiamata a funzione:** branch and link BL
- **Return** da funzione: ripristina in PC il valore del link register
`MOV PC, LR`
- **Argomenti:** R0 – R3
- **Valore di ritorno:** R0

Esempio di chiamata a funzione

C Code

```
int main() {  
    simple();  
    a = b + c;  
}
```

```
void simple() {  
    return;  
}
```

ARM Assembly Code

```
0x00000200 MAIN      BL  SIMPLE  
0x00000204           ADD R4, R5, R6
```

...

```
0x00401020 SIMPLE   MOV PC, LR
```

void significa che **simple** non ritorna un valore

Esempio di chiamata a funzione

C Code

```
int main() {  
    simple();  
    a = b + c;  
}
```

```
void simple() {  
    return;  
}
```

ARM Assembly Code

0x00000200	MAIN	BL SIMPLE
0x00000204		ADD R4, R5, R6
...		

0x00401020	SIMPLE	MOV PC, LR
------------	--------	------------

BL

branches to SIMPLE

LR = PC + 4 = 0x00000204

MOV PC, LR

makes PC = LR

(the next instruction executed is at 0x00000204)

Argomenti e valore di ritorno

Convenzioni ARM:

- Argomenti: R0 - R3
- Valore di ritorno: R0

C Code

```
int main() {
    int y;
    ...
    y = diffofsums(2, 3, 4, 5); // 4 arguments
    ...
}

int diffofsums(int f, int g, int h, int i) {
    int result;
    result = (f + g) - (h + i);
    return result;           // return value
}
```

Argomenti e valore di ritorno

ARM Assembly Code

```
; R4 = y
MAIN
...
MOV R0, #2          ; argument 0 = 2
MOV R1, #3          ; argument 1 = 3
MOV R2, #4          ; argument 2 = 4
MOV R3, #5          ; argument 3 = 5
BL DIFFOFSUMS      ; call function
MOV R4, R0          ; y = returned value
...
; R4 = result
DIFFOFSUMS
ADD R8, R0, R1      ; R8 = f + g
ADD R9, R2, R3      ; R9 = h + i
SUB R4, R8, R9      ; result = (f + g) - (h + i)
MOV R0, R4          ; put return value in R0
MOV PC, LR          ; return to caller
```

Argomenti e valore di ritorno

ARM Assembly Code

```
; R4 = result
DIFFOFSUMS
    ADD R8, R0, R1      ; R8 = f + g
    ADD R9, R2, R3      ; R9 = h + i
    SUB R4, R8, R9      ; result = (f + g) - (h + i)
    MOV R0, R4          ; put return value in R0
    MOV PC, LR          ; return to caller
```

- diffofsums sovrascrive indebitamente i 3 registri R4, R8, R9 dedicati alle saved variables della funzione chiamante main
- diffofsums invece dovrebbe usare lo *stack* per memorizzare temporaneamente i valori di questi registri prima di operare su di essi

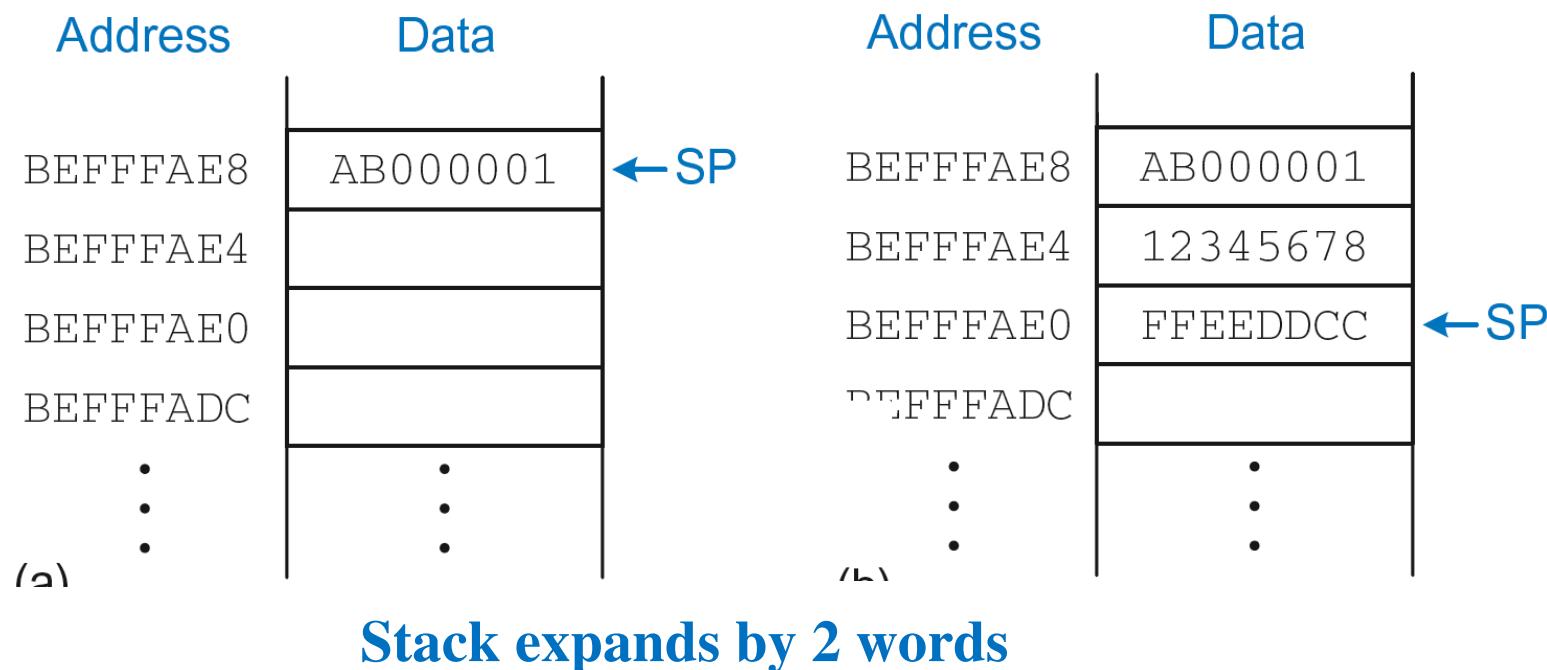
Lo stack delle funzioni

- Lo stack costituisce la memoria usata per salvare temporaneamente il valore delle variabili
- E' organizzato come una pila last-in-first-out (LIFO)
- Operazioni sullo stack
 - ***Expands***: usa più memoria quando necessaria (push)
 - ***Contracts***: usa meno memoria quando (pop)



Lo stack delle funzioni

- In termini di indirizzi di memoria, l'espansione dello stack avviene in senso decrescente (dagli indirizzi maggiori a quelli più piccoli)
- Stack pointer: SP *punta* al top dello stack



Come le funzioni usano lo stack

- Le funzioni chiamate non devono avere side effect inattesi
- Invece, nell esempio precedente diffofsuns sovrascrive indebitamente i 3 registri R4, R8, R9

ARM Assembly Code

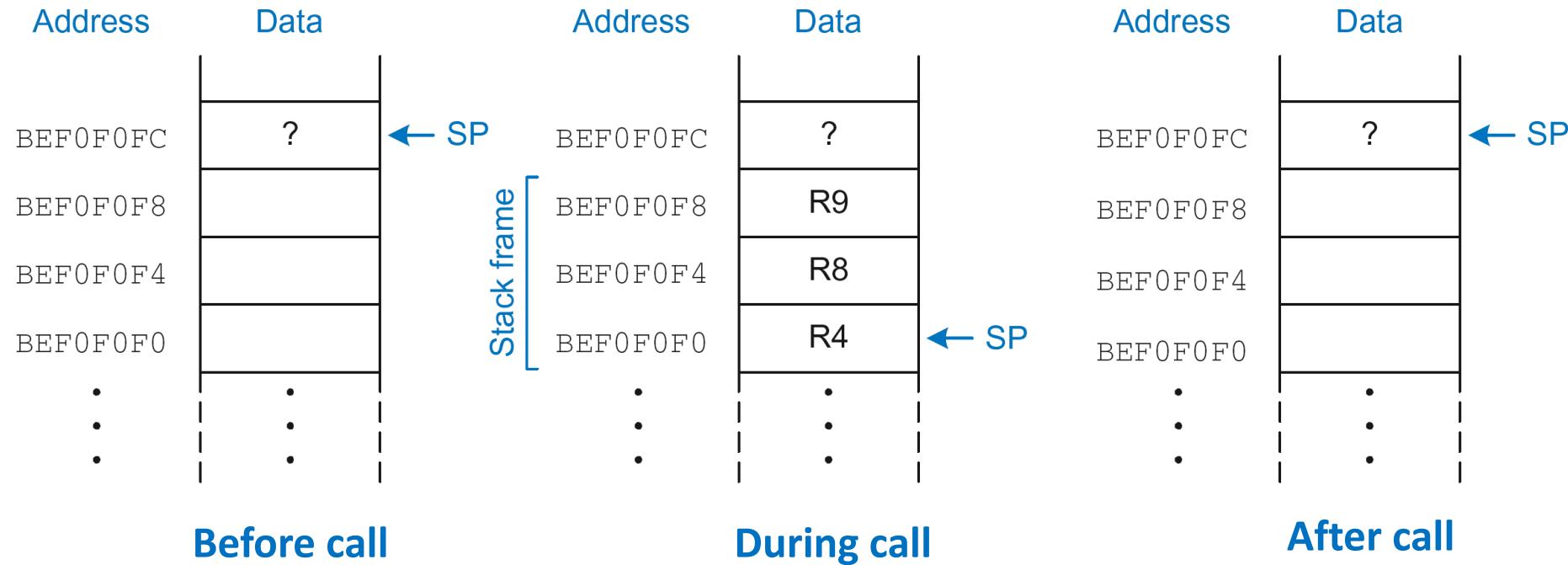
```
; R4 = result
DIFFOFSUMS
    ADD R8, R0, R1      ; R8 = f + g
    ADD R9, R2, R3      ; R9 = h + i
    SUB R4, R8, R9      ; result = (f + g) - (h + i)
    MOV R0, R4          ; put return value in R0
    MOV PC, LR          ; return to caller
```

Memorizzare i registri sullo stack

ARM Assembly Code

```
; R2 = result
DIFFOFSUMS
    SUB SP, SP, #12      ; make space on stack for 3 registers
    STR R4, [SP, #-8]    ; save R4 on stack
    STR R8, [SP, #-4]    ; save R8 on stack
    STR R9, [SP]          ; save R9 on stack
    ADD R8, R0, R1        ; R8 = f + g
    ADD R9, R2, R3        ; R9 = h + i
    SUB R4, R8, R9        ; result = (f + g) - (h + i)
    MOV R0, R4            ; put return value in R0
    LDR R9, [SP]          ; restore R9 from stack
    LDR R8, [SP, #-4]     ; restore R8 from stack
    LDR R4, [SP, #-8]     ; restore R4 from stack
    ADD SP, SP, #12       ; deallocate stack space
    MOV PC, LR            ; return to caller
```

Memorizzare i registri sullo stack



Registers

Preserved <i>Callee-Saved</i>	Nonpreserved <i>Caller-Saved</i>
R4-R11	R12
R14 (LR)	R0-R3
R13 (SP)	CPSR
stack above SP	stack below SP

Pop e push

- Salvare e ripristinare registri dallo stack è una operazione così frequente che ARM mette a disposizione delle istruzioni specifiche, Pop e Push, che consentono di avere un codice più succinto.
- Push consente di salvare in memoria più registri aggiornare consistentemente lo stack: PUSH {R4, R8, R9}
- Pop ripristina uno o più registri e incrementa consistentemente il valore dello stack: PUSH {R4, R8, R9}

PoP

Chiamate a funzioni innestate

C Code

```
int f1(int a, int b) {  
    int i, x;  
    x = (a + b) * (a - b);  
    for (i=0; i<a; i++)  
        x = x + f2(b+i);  
    return x;  
}  
  
int f2(int p) {  
    int r;  
    r = p + 5;  
    return r + p;  
}
```

ARM Assembly Code

```
; R0=a, R1=b, R4=i, R5=x      ; R0=p, R4=r  
F1  
    PUSH {R4, R5, LR}           PUSH {R4}  
    ADD   R5, R0, R1             ADD   R4, R0, 5  
    SUB   R12, R0, R1            ADD   R0, R4, R0  
    MUL   R5, R5, R12            POP   {R4}  
    MOV   R4, #0                 MOV   PC, LR  
FOR  
    CMP   R4, R0  
    BGE   RETURN  
    PUSH {R0, R1}  
    ADD   R0, R1, R4  
    BL    F2  
    ADD   R5, R5, R0  
    POP   {R0, R1}  
    ADD   R4, R4, #1  
    B    FOR  
RETURN  
    MOV   R0, R5  
    POP   {R4, R5, LR}  
    MOV   PC, LR
```

Chiamate a funzioni innestate

ARM Assembly Code

```
; R0=a, R1=b, R4=i, R5=x ; R0=p, R4=r
F1                                         F2
    PUSH {R4, R5, LR} ; save regs      PUSH {R4}          ; save regs
    ADD   R5, R0, R1 ; x = (a+b)     ADD   R4, R0, 5   ; r = p+5
    SUB   R12, R0, R1 ; temp = (a-b)
    MUL   R5, R5, R12 ; x = x*temp
    MOV   R4, #0       ; i = 0        POP   {R4}          ; restore regs
                                            MOV   PC, LR      ; return
FOR
    CMP   R4, R0       ; i < a?
    BGE  RETURN        ; no: exit loop
    PUSH {R0, R1}       ; save regs
    ADD   R0, R1, R4   ; arg is b+i
    BL    F2            ; call f2(b+i)
    ADD   R5, R5, R0   ; x = x+f2(b+i)
    POP   {R0, R1}       ; restore regs
    ADD   R4, R4, #1   ; i++
    B    FOR             ; repeat loop
RETURN
    MOV   R0, R5       ; return x
    POP   {R4, R5, LR} ; restore regs
    MOV   PC, LR      ; return
```

Address	Data
BEF7FF0C	?
BEF7FF08	
BEF7FF04	
BEF7FF00	
BEF7FEFC	
BEF7FEF8	
BEF7FEF4	
⋮	⋮
⋮	⋮
⋮	⋮

At beginning of f1

f1's stack frame

Address	Data
BEF7FF0C	?
BEF7FF08	LR
BEF7FF04	R5
BEF7FF00	R4
BEF7FEFC	R1
BEF7FEF8	R0
BEF7FEF4	
⋮	⋮
⋮	⋮
⋮	⋮

Just before calling f2

f1's stack frame

f2's stack frame

Address	Data
BEF7FF0C	?
BEF7FF08	LR
BEF7FF04	R5
BEF7FF00	R4
BEF7FEFC	R1
BEF7FEF8	R0
BEF7FEF4	R4
⋮	⋮
⋮	⋮
⋮	⋮

After calling f2

C Code

```
int factorial(int n) {  
    if (n <= 1)  
        return 1;  
    else  
        return (n * factorial(n-1));  
}
```

ARM Assembly Code

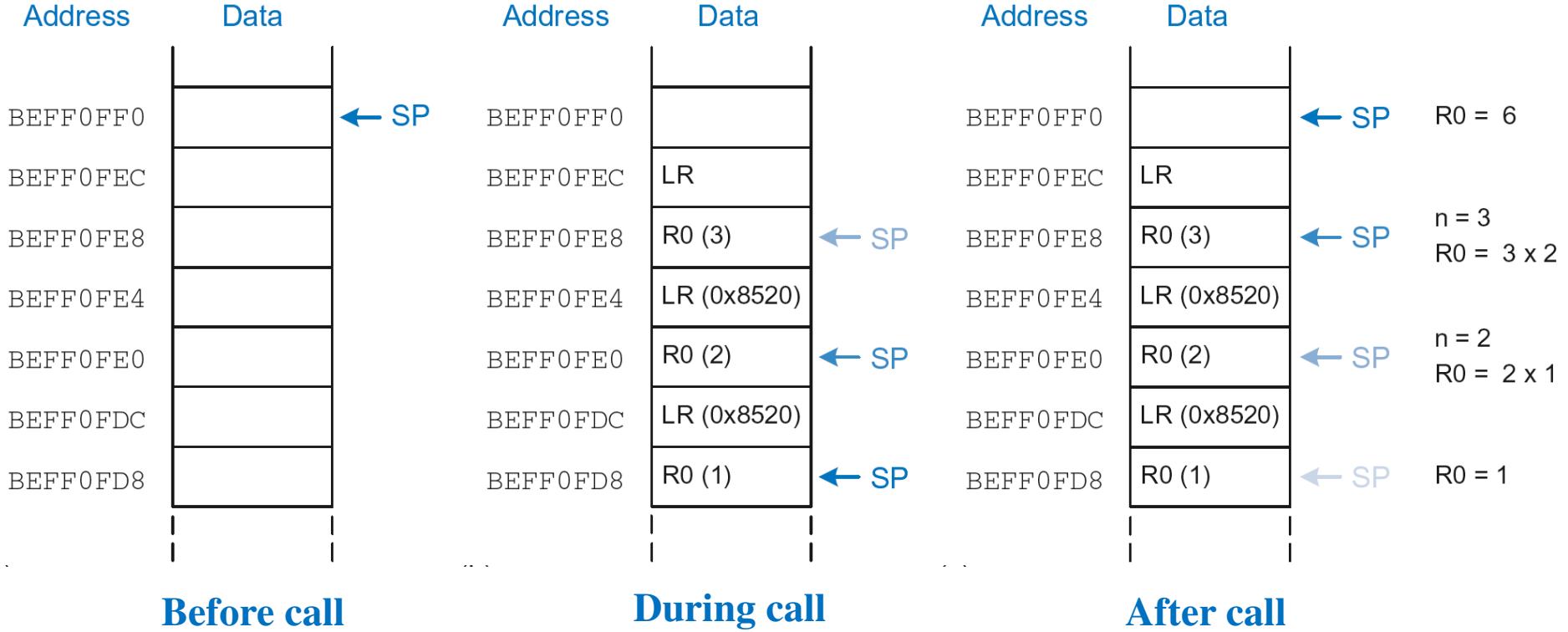
0x94	FACTORIAL	STR R0, [SP, #-4]!	; store R0 on stack
0x98		STR LR, [SP, #-4]!	; store LR on stack
0x9C		CMP R0, #2	; set flags with R0-2
0xA0		BHS ELSE	; if (r0>=2) branch to else
0xA4		MOV R0, #1	; otherwise return 1
0xA8		ADD SP, SP, #8	; restore SP 1
0xAC		MOV PC, LR	; return
0xB0	ELSE	SUB R0, R0, #1	; n = n - 1
0xB4		BL FACTORIAL	; recursive call
0xB8		LDR LR, [SP], #4	; restore LR
0xBC		LDR R1, [SP], #4	; restore R0 (n) into R1
0xC0		MUL R0, R1, R0	; R0 = n*factorial(n-1)
0xC4		MOV PC, LR	; return

C Code

```
int factorial(int n) {  
    if (n <= 1)  
        return 1;  
  
    else  
        return (n * factorial(n-1));  
}
```

ARM Assembly Code

```
0x94 FACTORIAL    STR R0, [SP, #-4]!  
0x98              STR LR, [SP, #-4]!  
0x9C              CMP R0, #2  
0xA0              BHS ELSE  
0xA4              MOV R0, #1  
0xA8              ADD SP, SP, #8  
0xAC              MOV PC, LR  
0xB0 ELSE         SUB R0, R0, #1  
0xB4              BL  FACTORIAL  
0xB8              LDR LR, [SP], #4  
0xBC              LDR R1, [SP], #4  
0xC0              MUL R0, R1, R0  
0xC4              MOV PC, LR
```

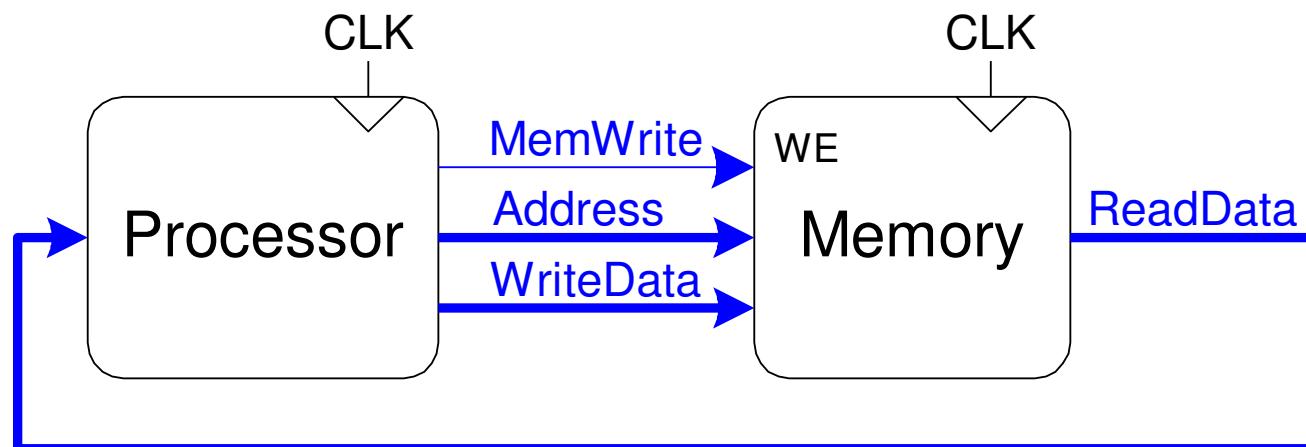


Memorie

Memorie vs CPU

Nell'architettura Von Neuman il canale di comunicazione tra la CPU e la memoria è il punto critico (collo di bottiglia) del sistema.

La tecnologia consente di realizzare CPU sempre più veloci e memorie sempre più grandi ma la velocità di accesso delle memorie non cresce così rapidamente come la velocità della CPU.



Memorie vs CPU

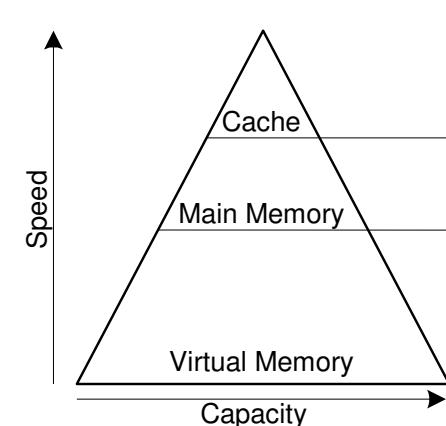
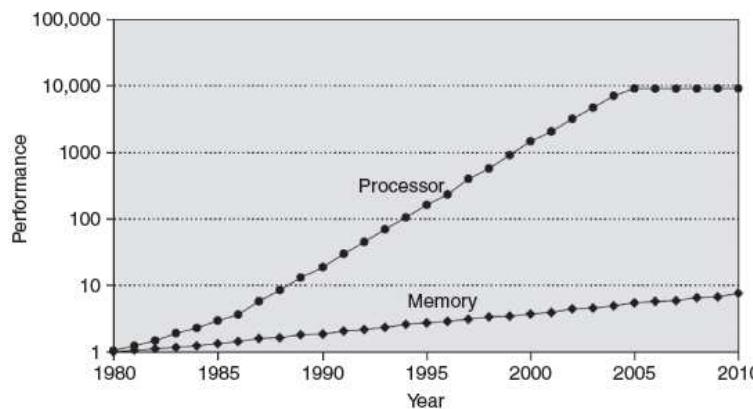
Storicamente le **CPU** sono sempre state più veloci delle **memorie**.

Al giorno d'oggi siamo in grado di produrre delle memorie veloci quanto una CPU moderna, il reale problema è dato da:

- queste memorie hanno un costo elevatissimo.
- le memorie dovrebbero essere piazzate in gran parte sugli stessi chip delle CPU, il che non è possibile.

Per ovviare a questo problema, gli ingegneri ricorrono ad uno schema chiamato “**a gerarchia di memoria**” in cui si combinano:

- una quantità molto **piccola** di memoria estremamente **veloce** (la cache)
- una quantità molto **grande** di memoria **lenta**.



Technology	Price / GB	Access Time (ns)	Bandwidth (GB/s)
SRAM	\$10,000	1	25+
DRAM	\$10	10 - 50	10
SSD	\$1	100,000	0.5
HDD	\$0.1	10,000,000	0.1

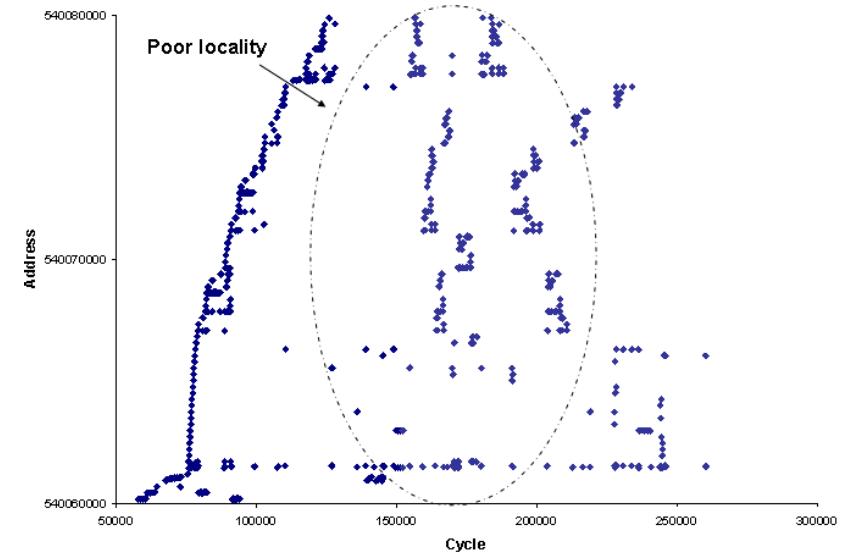
Inquadramento del Problema

La maggior parte del tempo di esecuzione di una CPU è, di solito, impegnato da procedure **in cui vengono eseguite ripetutamente le stesse istruzioni**.

In realtà, non è importante conoscere lo schema dettagliato della sequenza di istruzioni, il punto chiave è che molte istruzioni in aree ben localizzate del programma vengono eseguite ripetutamente in un determinato periodo, e si accede al resto del programma relativamente di rado.

Questa proprietà viene chiamata *località dei riferimenti*. Si manifesta in due modi: *località temporale* e *località spaziale*.

- La *località temporale* rappresenta la probabilità che un'istruzione eseguita di recente venga eseguita nuovamente, entro breve tempo.
- La *località spaziale* rappresenta invece la probabilità che istruzioni vicine ad un'istruzione eseguita di recente (dove la vicinanza è espressa in termini di indirizzi delle istruzioni) siano anch'esse eseguite nel prossimo futuro.



Utilità di avere una cache memory

La velocità con cui la memoria risponde alle richieste di istruzioni e dati della CPU ha un peso determinante sulle prestazioni di un sistema.

Se tra la memoria principale e la CPU si potesse interporre una memoria molto veloce, contenente le parti di programma e di dati che, volta per volta, interessano l'elaborazione, il tempo totale di esecuzione verrebbe ridotto in modo significativo.

Questa è esattamente la funzione della "*cache memory*" che in inglese significa letteralmente "*schermo della memoria*".

La struttura della CPU, infatti, non viene influenzata dalla presenza o meno di una *cache memory*. L'interfaccia verso la memoria applica il "*protocollo*" di trasferimento *dalla* memoria o *verso* la memoria ignorando la presenza di una struttura intermedia.

Agli albori di questa tecnica, infatti, la cache memory era solo un accessorio a pagamento.

Si tratta quindi di un elemento che inserito tra CPU e memoria principale impedisce alla prima di "*vedere*" i tempi di risposta reali della memoria.

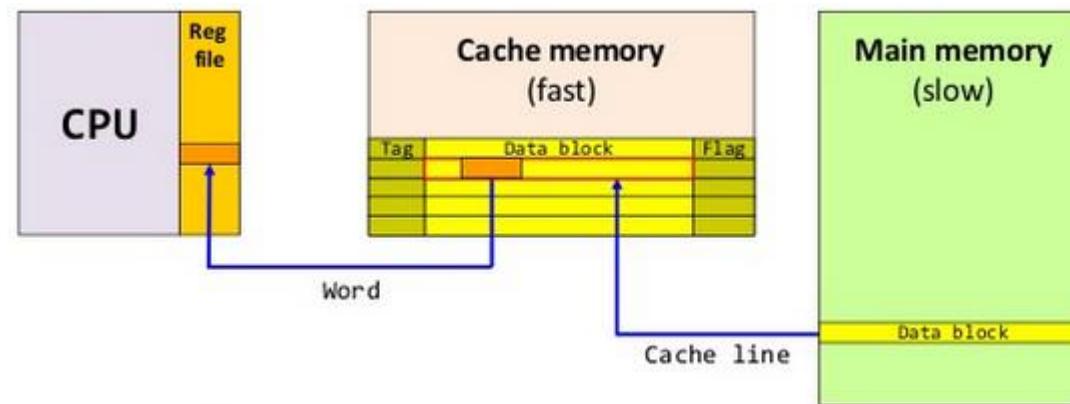
Una traduzione possibile in italiano del concetto è "*memoria tamponcino*".

Principi di funzionamento

Concettualmente, le operazioni svolte da una memoria cache sono molto semplici. I circuiti di controllo della memoria cache sono progettati per avvantaggiarsi della proprietà della località dei riferimenti.

L'aspetto temporale di questa proprietà suggerisce di portare un elemento (istruzioni o dati) nella cache quando viene richiesto per la prima volta, in modo tale che rimanga a disposizione nel caso di una nuova richiesta.

L'aspetto spaziale suggerisce che, invece di portare dalla memoria principale alla cache un elemento alla volta, è conveniente portare un insieme di elementi che risiedono in indirizzi adiacenti.



Si farà riferimento al termine *blocco* per indicare un insieme di indirizzi contigui di una qualche dimensione.

Un altro termine utilizzato di frequente per indicare un blocco della cache è *linea di cache*.

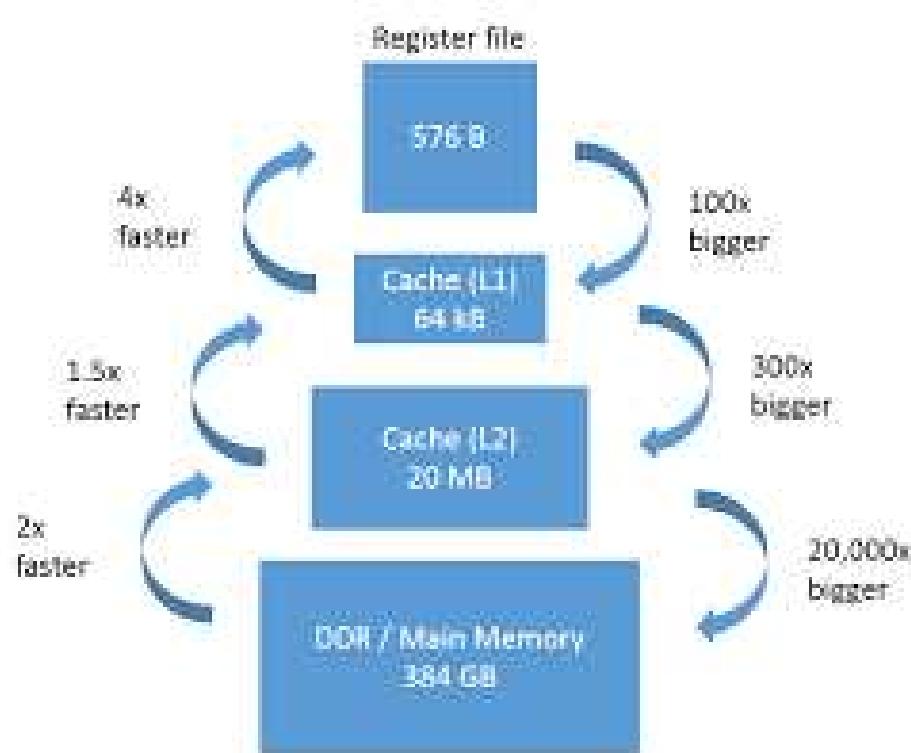
Dimensione della cache

Quando si riceve una richiesta di lettura fatta dalla CPU, il contenuto di un blocco di parole di memoria contenenti la locazione specificata viene trasferito nella cache, una o più parole alla volta.

In seguito, ognqualvolta il programma fa riferimento a una di queste locazioni del blocco, i valori desiderati vengono letti direttamente dalla cache.

Affinché la cache svolga efficacemente il suo compito deve avere tempi di accesso molto più brevi di quelli della memoria principale e ciò come vedremo impone che sia piccola.

Se è piccola non potrà che contenere una frazione ridotta delle istruzioni ed i dati della memoria principale.



Tecniche di gestione

- La corrispondenza tra i blocchi della memoria principale e quelli della cache è specificata dalla funzione di *posizionamento (mapping)*.
- Quando la cache è piena e si fa riferimento a una parola di memoria (istruzione o dato) che non è presente nella cache, l'hardware di controllo della cache deve decidere quale blocco della cache debba essere rimosso per far spazio al nuovo blocco, che contiene la parola a cui si fa riferimento.
- L'insieme di regole in base alle quali viene fatta questa scelta costituisce *l'algoritmo di sostituzione*.

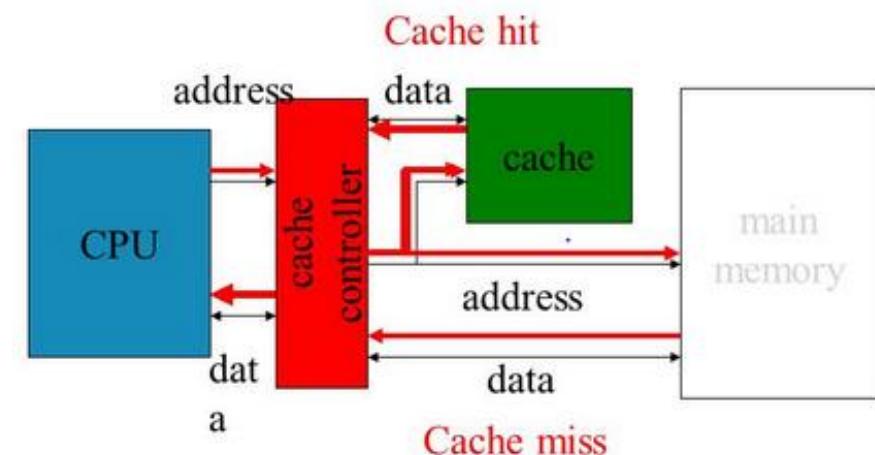
Tecniche di gestione

La struttura dei sistemi è, di solito, organizzata in modo che la presenza di una cache non influenzi il funzionamento della CPU.

Quest'ultima, infatti, nell'esecuzione del programma effettua le richieste di lettura e scrittura utilizzando gli indirizzi delle locazioni nella memoria principale.

E' la logica di controllo della cache che si fa carico di determinare se la parola richiesta è presente o meno nella cache.

Se è presente, viene effettuata l'operazione di lettura o scrittura della locazione di memoria appropriata. In questo caso si dice che l'accesso in lettura o in scrittura ha avuto successo (*read hit* o *write hit*).



Possibili gestioni di un “cache hit”

Se l'accesso alla cache ha avuto successo bisogna distinguere due tipi di situazioni:

Operazione di lettura: la memoria principale non viene coinvolta

Operazione di scrittura: il sistema può procedere in due modi:

- Nel primo (*write-through*), la locazione della cache e quella della memoria principale **vengono entrambe aggiornate**.
 - Il *write-through* è più semplice, ma implica inutili operazioni di scrittura nella memoria principale, se una parola viene aggiornata più volte durante il periodo in cui risiede nella cache.
- Nel secondo (*write-back*) si aggiorna **soltanto la locazione della memoria cache**, segnandola come aggiornata con un **bit di modifica** o *dirty*. La locazione della memoria principale viene aggiornata in seguito, quando il blocco contenente la parola marcata deve essere rimosso dalla memoria cache per far posto a un nuovo blocco.
 - Anche il protocollo *write-back* può causare inutili scritture nella memoria principale, visto che, quando si procede con la scrittura nella memoria principale di un blocco, **tutte le parole del blocco vengono scritte**, anche se solo una delle parole del blocco della cache è stata modificata.

Gestione di un “read miss”

Quando la parola indirizzata durante un'operazione di lettura non è presente nella cache si dice che l'accesso in lettura è fallito (*read miss*).

Il blocco di parole contenente la parola richiesta deve essere copiato dalla memoria principale nella memoria cache.

Due sono le possibilità:

Dopo aver caricato l'intero blocco nella memoria cache, la parola richiesta viene inviata alla CPU.

In alternativa, è possibile inviare immediatamente la parola alla CPU, non appena la si legge dalla memoria principale.

Quest'ultimo approccio, chiamato *load-through*, o anche *early restart*, riduce in qualche modo il tempo di attesa della CPU, a discapito di una maggiore complessità del circuito di controllo della cache.

Gestione di un “write miss”

Durante un'operazione di scrittura, se la parola indirizzata non è nella cache si dice che l'accesso in scrittura è fallito (*write miss*).

Anche in questo caso ci sono le due alternative,
se si utilizza un protocollo *write-through*, le informazioni vengono scritte direttamente
nella memoria principale,
nel caso invece di un protocollo *write-back*, il blocco, contenente la parola indirizzata,
viene prima caricato nella cache, poi viene soprascritto con le nuove informazioni.

Prestazioni

Hit Rate = # hits / # memory accesses
= 1 – Miss Rate

Miss Rate = # misses / # memory accesses
= 1 – Hit Rate

Average memory access time (AMAT): average time for processor to access data

$$\text{AMAT} = t_{\text{cache}} + MR_{\text{cache}}[t_{MM} + MR_{MM}(t_{VM})]$$

Prestazioni

- Un programma ha 2,000 loads e stores
- 1,250 di questi dati sono presenti nella cache
- Il resto è fornito da altri livelli nella gerarchia di memoria
- Quale è l'hit e il miss rate per la cache?

$$\text{Hit Rate} = 1250/2000 = 0.625$$

$$\text{Miss Rate} = 750/2000 = 0.375 = 1 - \text{Hit Rate}$$

Prestazioni

- Assumi che un processore ha 2 livelli di memoria: cache e main memory
- $t_{\text{cache}} = 1 \text{ cycle}$, $t_{MM} = 100 \text{ cycles}$
- Qual'è l'AMAT dati gli hit e miss rare precedenti?

$$\begin{aligned}\text{AMAT} &= t_{\text{cache}} + MR_{\text{cache}}(t_{MM}) \\ &= [1 + 0.375(100)] \text{ cycles} \\ &= 38.5 \text{ cycles}\end{aligned}$$

Terminologia Cache

- **Capacity (C):**
 - Numero di byte nella cache
- **Block size (b):**
 - Grandezza di un blocco, ovvero numero di bytes che sono trascritti nella cache per volta
- **Number of blocks ($B = C/b$):**
 - Numero di blocchi nella cache
- **Degree of associativity (M):**
 - Numero di blocchi in un set
- **Number of sets ($S = B/M$):**
 - Ogni indirizzo di memoria è mappato in esattamente un set della cache

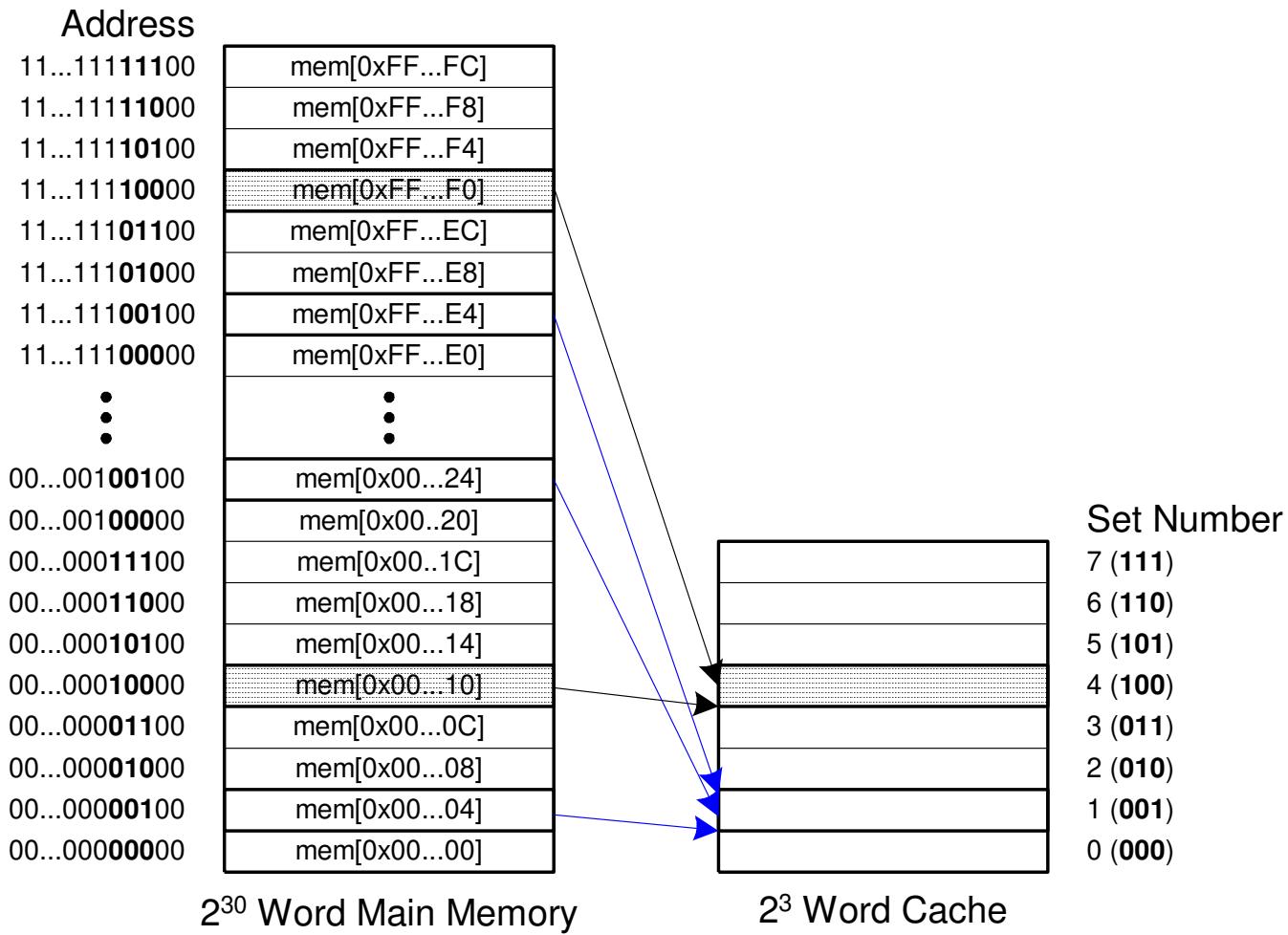
Tipologie di memorie cache

- La memoria Cache è organizzata in S set
 - Ogni indirizzo di memoria mappa in esattamente un unico set
 - Diverse categorie di memoria cache dipendono dal numero di blocchi che un set contiene:
 - **Direct mapped:** 1 blocco per ogni set
 - **N -way set associative:** N blocchi per set
 - **Fully associative:** tutti i blocchi di una cache sono in un unico set
 - Esaminiamo ogni tipologia con una cache con:
 - Capacity ($C = 8$ words)
 - Block size ($b = 1$ word)
 - Quindi, il numero di blocchi è ($B = 8$)
- Ovviamente una memoria del genere è ridicolmente piccola, serve solo per scopi didattici*

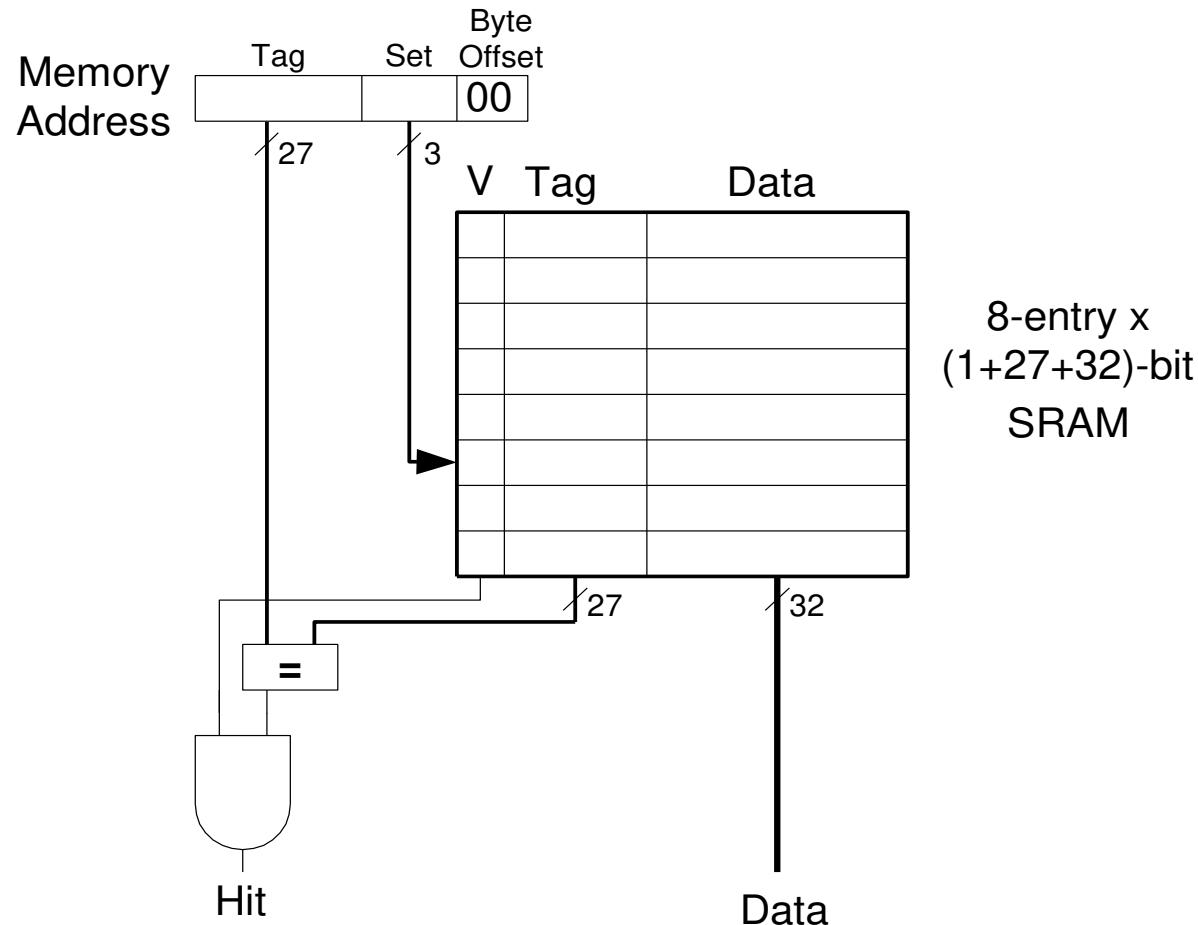
- Capacity: C
- Block size: b
- Number of blocks in cache: $B = C/b$
- Number of blocks in a set: N
- Number of sets: $S = B/N$

Organization	Number of Ways (N)	Number of Sets ($S = B/N$)
Direct Mapped	1	B
N-Way Set Associative	$1 < N < B$	B / N
Fully Associative	B	1

Direct Mapped Cache

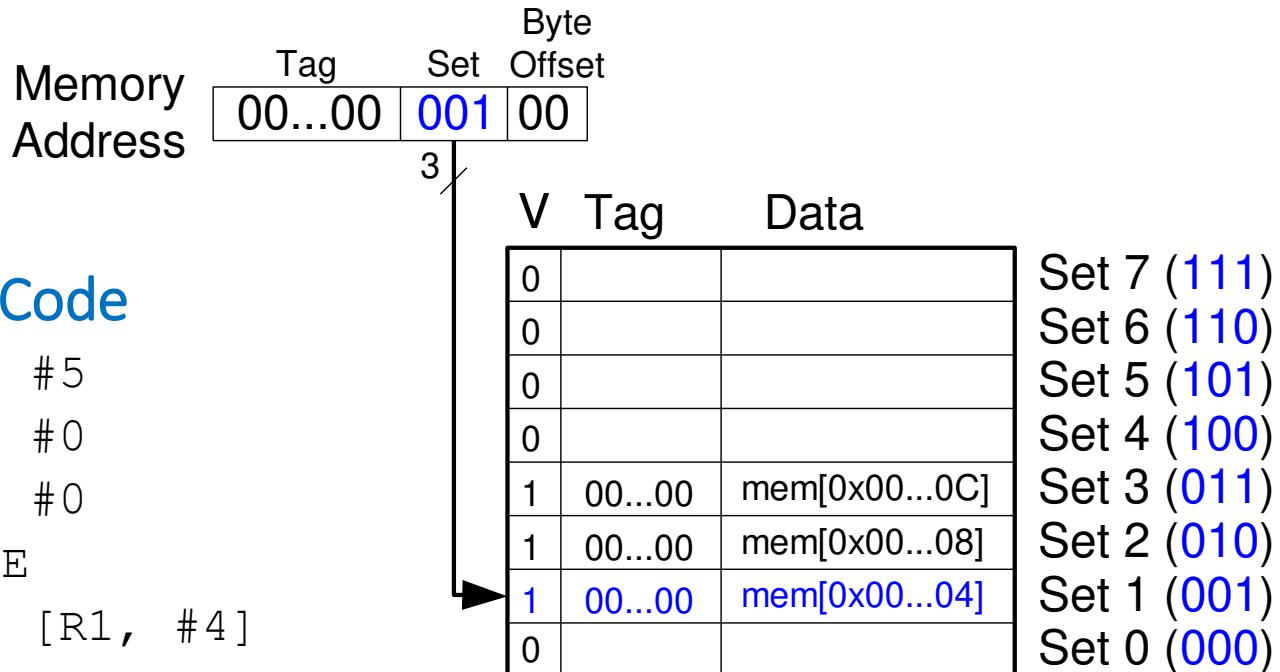


Direct Mapped Cache Hardware



ARM Assembly Code

```
MOV R0, #5  
MOV R1, #0  
LOOP  CMP R0, #0  
      BEQ DONE  
      LDR R2, [R1, #4]  
      LDR R3, [R1, #12]  
      LDR R4, [R1, #8]  
      SUB R0, R0, #1  
      B    LOOP  
  
DONE
```

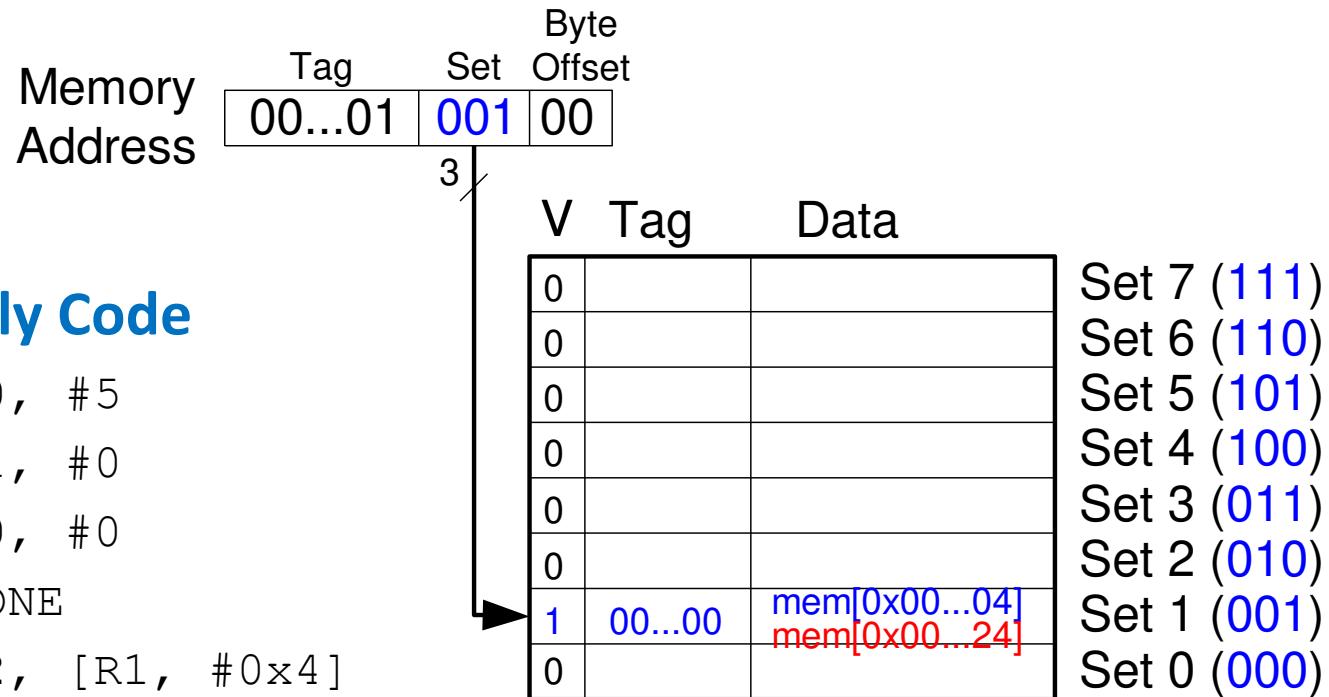


**Miss Rate = 3/15
= 20%**

Miss obbligatori

ARM Assembly Code

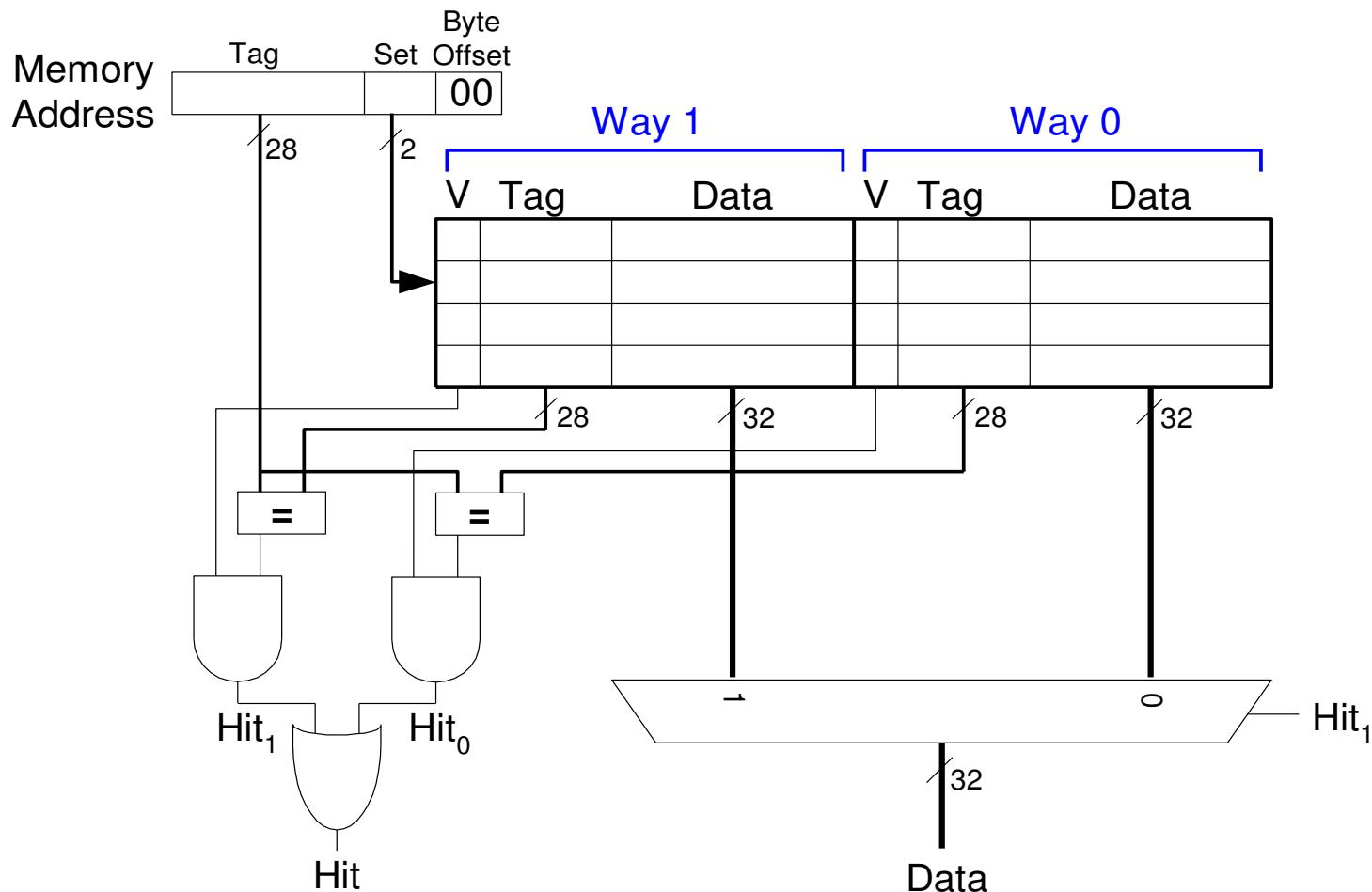
```
MOV R0, #5  
MOV R1, #0  
LOOP  CMP R0, #0  
      BEQ DONE  
      LDR R2, [R1, #0x4]  
      LDR R3, [R1, #0x24]  
      SUB R0, R0, #1  
      B    LOOP  
  
DONE
```



**Miss Rate = 10/10
= 100%**

Miss di conflitto

N -Way Set Associative Cache



ARM Assembly Code

```
MOV R0, #5  
MOV R1, #0  
LOOP  CMP R0, 0  
      BEQ DONE  
      LDR R2, [R1, #0x4]  
      LDR R3, [R1, #0x24]  
      SUB R0, R0, #1  
      B    LOOP
```

DONE

**Miss Rate = 2/10
= 20%**

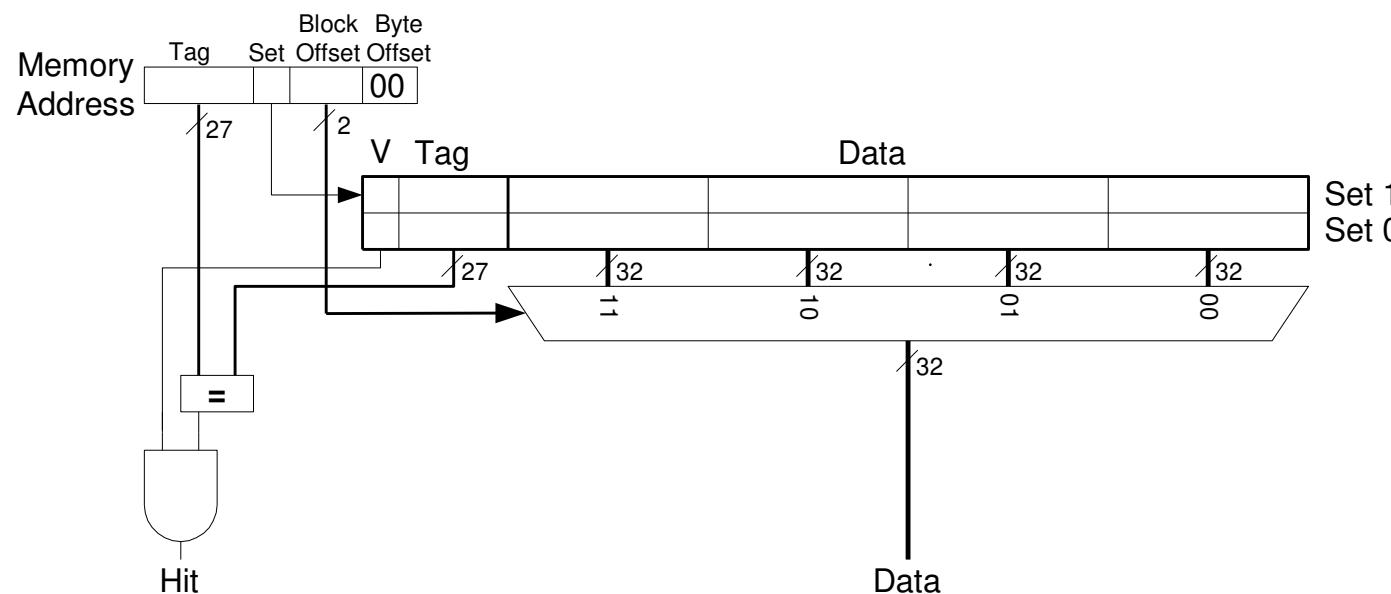
L'associatività riduce i miss di conflitto

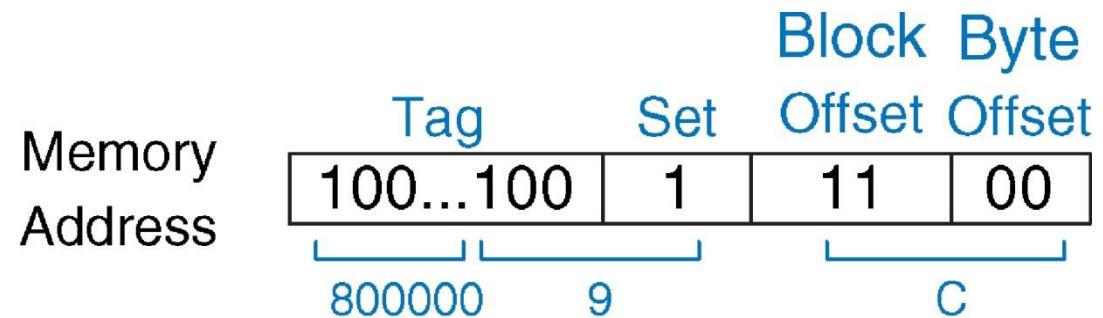
Way 1			Way 0		
V	Tag	Data	V	Tag	Data
0			0		
0			0		
1	00...10	mem[0x00...24]	1	00...00	mem[0x00...04]
0			0		

Set 3
Set 2
Set 1
Set 0

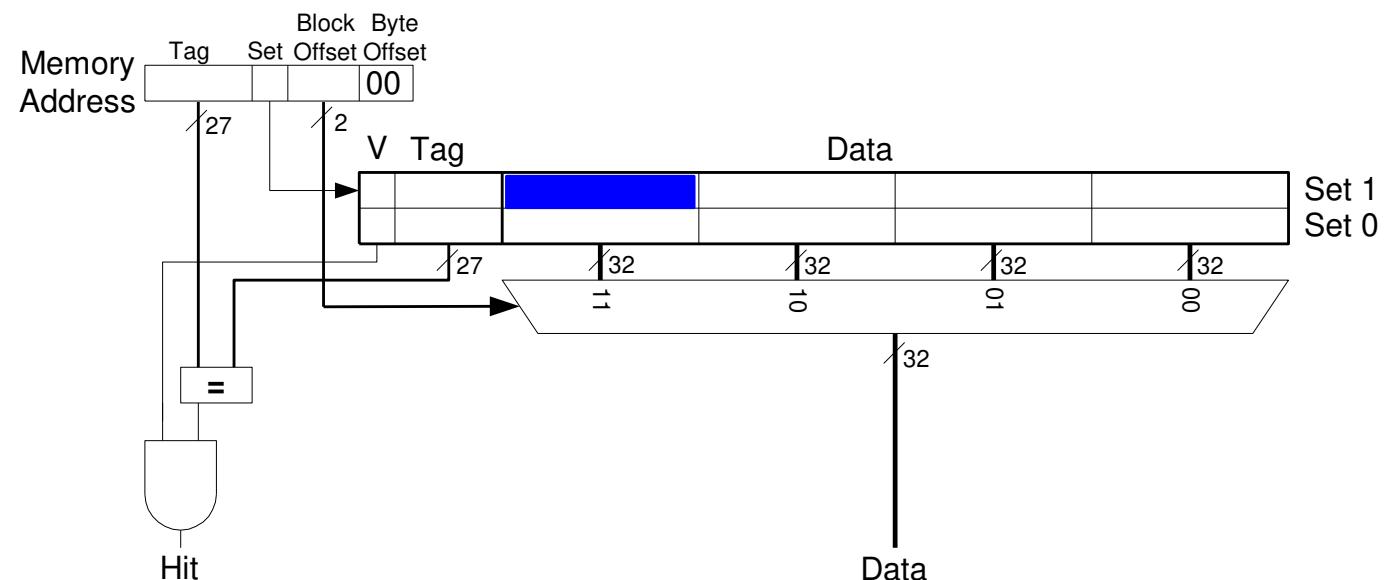
Come sfruttare la località spaziale?

- Incrementare il block size:
 - Block size, $b = 4$ words
 - $C = 8$ words
 - Direct mapped (1 block per set)
 - Number of blocks, $B = 2$ ($C/b = 8/4 = 2$)





© 2007 Elsevier, Inc. All rights reserved



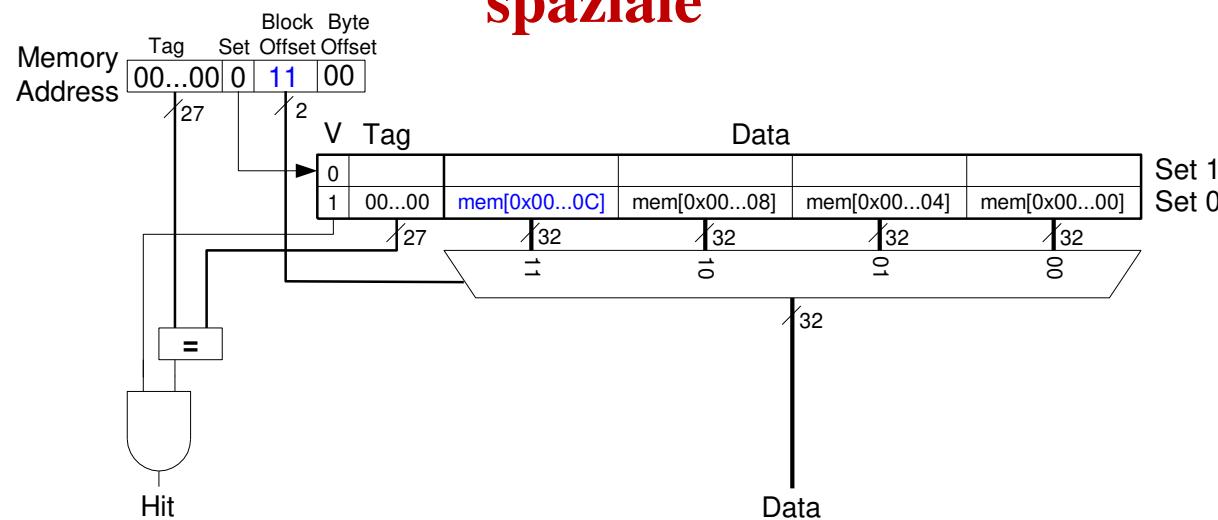
ARM assembly code

```
MOV R0, #5  
MOV R1, #0  
LOOP  CMP R0, 0  
      BEQ DONE  
      LDR R2, [R1, #4]  
      LDR R3, [R1, #12]  
      LDR R4, [R1, #8]  
      SUB R0, R0, #1  
      B    LOOP  
  
DONE
```

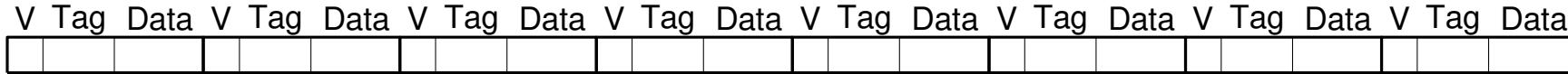
$$\text{Miss Rate} = 1/15$$

$$= 6.67\%$$

Blocchi più grandi riducono
i miss obbligatori
attraverso la località
spaziale



Fully Associative Cache



- Consente la massima flessibilità nella scelta della locazione nella cache in cui posizionare un blocco di memoria
- Lo spazio della cache può essere utilizzato in modo più efficiente e questo comporta una riduzione dei miss di conflitto
- Il costo della ricerca di un blocco in cache è notevole, mentre nel caso di indirizzamento diretto è praticamente nullo
 - Una operazione di questo tipo si chiama ricerca associativa o per contenuto, in quanto corrisponde all'operazione di ricerca di un item in un elenco
- E' più costosa da costruire

Il bit di validità

- Un ulteriore bit di controllo, chiamato ***bit di validità***, è necessario per ogni blocco.
- **Questo bit indica se il blocco contiene o meno dati validi.**
- Non deve però essere confuso con il ***bit di modifica***, menzionato in precedenza, il bit di modifica, che indica se il blocco è stato modificato o meno durante il periodo in cui è stato nella cache, serve soltanto nei sistemi che non fanno uso del metodo ***write-through***.

I bit di validità dei blocchi che si trovano nella cache vengono tutti posti a 0 nell'istante iniziale di accensione del sistema e, in seguito, ogni volta che nella memoria principale vengono caricati programmi e dati nuovi dal disco.

Il Direct Memory Access - DMA

I trasferimenti dal disco alla memoria principale vengono effettuati mediante un meccanismo detto di DMA (Direct Memory Access). Questa sigla specifica che durante il trasferimento la CPU non interviene, perché è la logica di controllo del disco che gestisce le linee indirizzo e le linee dati, fornendo i segnali di controllo necessari.

Come vedremo successivamente, in una operazione di questo genere vengono trasferite grandi quantità di istruzioni e dati organizzati in entità dette “*pagine*” ciascuna delle quali di solito contiene migliaia di *locazioni* e quindi centinaia di *blocchi*.

Normalmente, le pagine di programma e quelle di dati vanno nella memoria centrale senza coinvolgere la cache.

Gestione del bit di validità

Il bit di validità di un certo blocco della cache viene posto a “1” la prima volta che esso è chiamato a contenere istruzioni o dati trasferiti dalla memoria principale; poi, ogni volta che le locazioni di un blocco della memoria principale vengono interessate da un trasferimento di nuove istruzioni o di nuovi dati dal disco, viene effettuato un controllo per determinare se qualcuno dei blocchi che stanno per essere sovrascritti fossero o meno presenti nella cache.

Se nella cache c’è una copia del blocco, il suo bit di validità viene posto a “0”; in tal modo, si garantisce che nella cache non ci siano dati obsoleti.

Il bit di validità a “0” candida immediatamente il blocco ad essere sovrascritto.

Svuotamento (flush) della cache

Una situazione dello stesso tipo si determina quando viene effettuato un trasferimento tramite DMA dalla memoria al disco e la cache utilizza un protocollo write-back.

In questo caso, i dati nella memoria potrebbero non riflettere i cambiamenti che possono essere stati fatti sulla copia dei dati presenti nella cache.

Una possibile soluzione a tale problema consiste nello *svuotamento (flush)* della cache forzando i dati con il bit di modifica attivo a essere ricopiatati nella memoria principale prima che venga effettuato il trasferimento tramite DMA.

“Coerenza” della cache

Il sistema operativo è in grado di fare tutto tutte le operazioni necessarie in modo molto efficiente, senza peggiorare significativamente le prestazioni, visto che le operazioni di ingresso/uscita di scrittura non sono molto frequenti.

Questa necessità di garantire che due diverse entità (i sottosistemi della CPU e del DMA nel caso presente) utilizzino la stessa copia dei dati viene chiamata problema della ***coerenza della cache***.

Algoritmi di sostituzione (1 / 2)

- In una cache ad indirizzamento diretto, la posizione di ogni blocco è predefinita, quindi non esiste alcuna strategia di sostituzione.
- Nelle memorie cache *associative* e *set-associative*, esiste invece una certa flessibilità.

Quando un nuovo blocco deve essere portato nella cache, e tutte le posizioni che potrebbe occupare contengono dati validi, il controllore della cache deve decidere quale blocco, tra quelli esistenti, soprascrivere.

Questa è una decisione importante, perché la scelta potrebbe essere determinante per le prestazioni del sistema.

Algoritmi di sostituzione (2 / 2)

In generale, l'obiettivo è quello di **mantenere nella cache quei blocchi che hanno una maggior possibilità di essere nuovamente utilizzati nel prossimo futuro.** ma, non è facile prevedere i blocchi a cui si farà riferimento.

Una possibile strategia è di tener presente che la **località dei riferimenti** suggerisce che i blocchi a cui c'è stato accesso di recente hanno elavata probabilità di essere utilizzati nuovamente entro breve tempo.

Sostituzione con algoritmo LRU

Quando bisogna eliminare dalla cache un blocco, è sensato soprascrivere quello a cui non si accede da più tempo. Tale blocco prende il nome di blocco *utilizzato meno di recente* (*Least Recently Used, LRU*), e la tecnica si chiama *algoritmo di sostituzione LRU*.

Per utilizzare l'algoritmo LRU, il controllore della cache deve mantenere traccia di tutti gli accessi ai blocchi mentre l'elaborazione prosegue.

Il blocco LRU di una memoria cache *set-associativa* con insiemi di quattro blocchi, può, ad esempio, essere indicato da un contatore di 2 bit associato a ciascun blocco, il cui contenuto sia opportunamente gestito secondo un protocollo automatico

Gestione del contatore di accessi

Quando si ha un successo nell'accesso al blocco (*hit*):

- il contatore di quel blocco viene posto a 0. I contatori con valori originariamente inferiori a quelli del blocco a cui si accede, vengono incrementati di uno, mentre tutti gli altri rimangono invariati.

Quando, invece, l'accesso fallisce (*miss*) si aprono due possibilità:

- l'insieme non è pieno, il contatore associato al nuovo blocco caricato dalla memoria principale viene posto a 0 e il valore di tutti gli altri contatori viene incrementato di 1.
- l'insieme è pieno, si rimuove il blocco, il cui contatore ha valore 3, e si pone il nuovo blocco al suo posto, mettendo il contatore a 0. Gli altri tre contatori dell'insieme vengono incrementati di un'unità.

E' facile verificare che i valori dei contatori dei blocchi occupati sono sempre diversi.

Alternative all'algoritmo LRU

L'algoritmo LRU è stato utilizzato ampiamente, con buone prestazioni, in numerosi schemi di accesso. Ma in alcune situazioni, per esempio nel caso di accessi sequenziali a un vettore di elementi che è leggermente troppo grande per poter stare tutto nella cache, le prestazioni sono molto scadenti

Le prestazioni dell'algoritmo LRU possono essere migliorate introducendo una piccola dose di casualità nella scelta del blocco da sostituire.

Sono stati proposti molti altri algoritmi di sostituzione più o meno complessi, che richiedono talvolta anche notevoli complicazioni hardware, ma l'algoritmo più semplice, e spesso anche piuttosto efficace, consiste nello **scegliere in modo completamente casuale il blocco da sostituire**.

Tipi di Miss

- **Compulsory:** primo accesso ad un dato
- **Capacity:** la cache è troppo piccola per contenere tutti i dati di interesse
- **Conflict:** due dati sono indirizzati nella stessa locazione nella cache

Miss penalty: il tempo necessario per recuperare un blocco da un livello più basso nella gerarchia delle memorie

ARM Assembly Code

```
MOV R0, #0  
LDR R1, [R0, #4]  
LDR R2, [R0, #0x24]  
LDR R3, [R0, #0x54]
```

Way 1 Way 0

V	U	Tag	Data	V	Tag	Data
0	0			0		
0	0			0		
1	0	00...010	mem[0x00...24]	1	00...000	mem[0x00...04]
0	0			0		

Set 3 (11)
Set 2 (10)
Set 1 (01)
Set 0 (00)

(a)

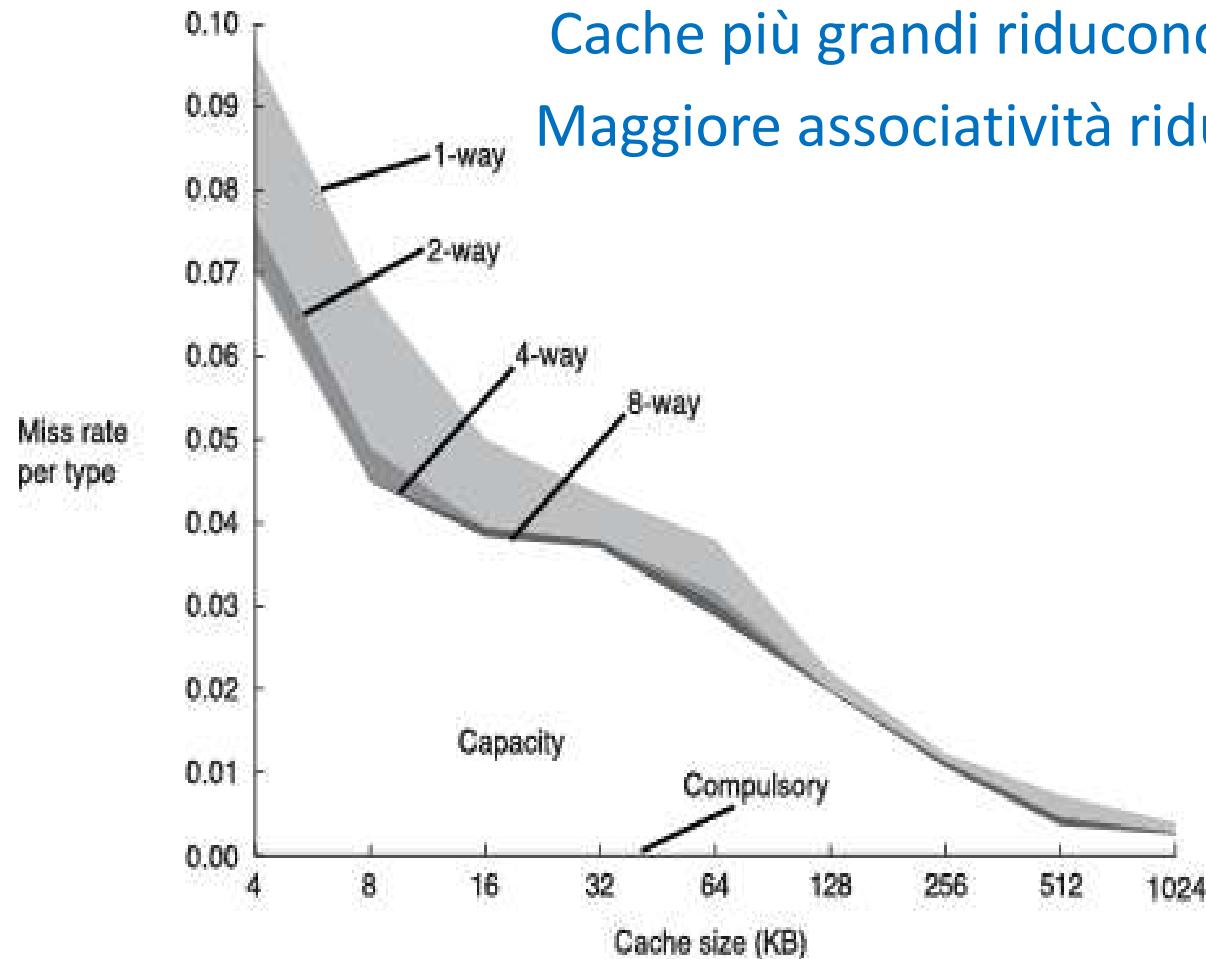
Way 1 Way 0

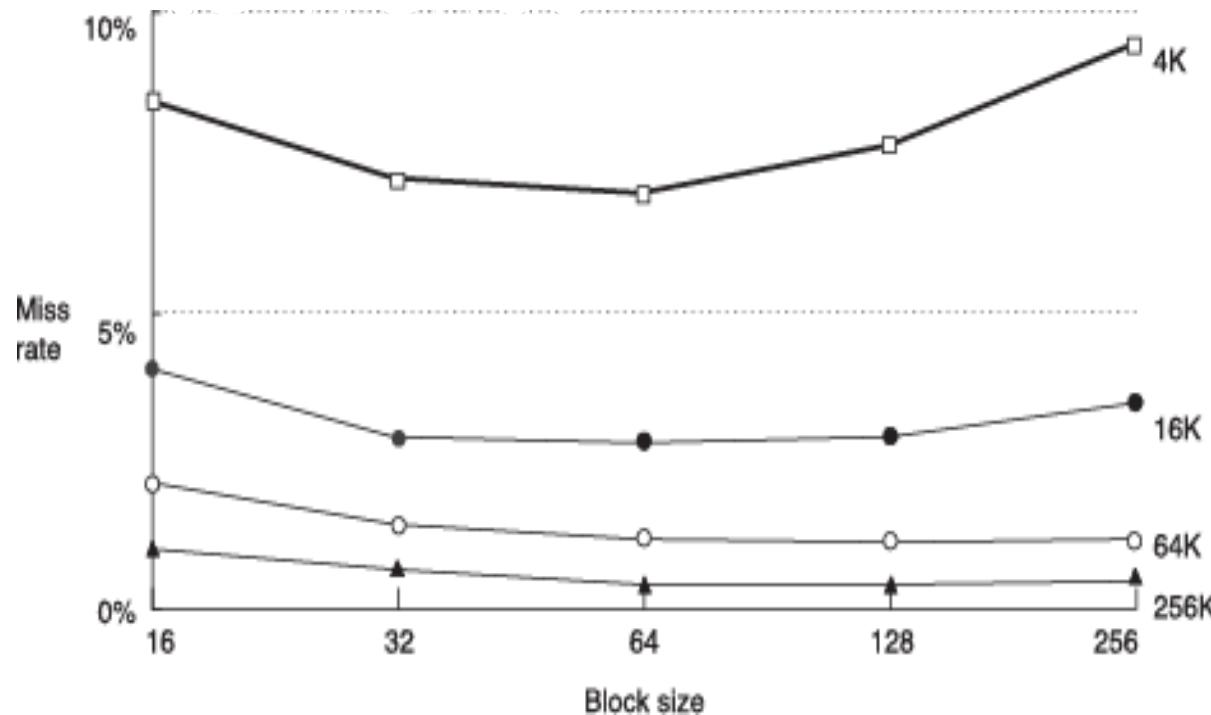
V	U	Tag	Data	V	Tag	Data
0	0			0		
0	0			0		
1	1	00...010	mem[0x00...24]	1	00...101	mem[0x00...54]
0	0			0		

Set 3 (11)
Set 2 (10)
Set 1 (01)
Set 0 (00)

(b)

Cache più grandi riducono i capacity miss
Maggiore associatività riduce i miss di conflitto





- Blocchi più grandi riducono i miss obbligatori sfruttando la località spaziale
- Data una cache di dimensioni fissate, all'aumentare del block size il numero di set decrementa ($S \times N_{xb} = C$). Quindi aumentano i miss di conflitto

Frazionamento della memoria in moduli

E utile che i trasferimenti da/verso le unità più veloci possano essere effettuati alla stessa frequenza dell'unità più veloce.

Ciò non può avvenire se si accede nello stesso modo all'unità più lenta e a quella più veloce.

Tale risultato può invece essere raggiunto se si sfrutta il parallelismo nell'organizzazione dell'unità più lenta.

Un modo efficiente per introdurre il parallelismo nella gestione della memoria principale consiste nell'impiego di un'organizzazione *interallacciata*.

Se la memoria principale di un calcolatore è strutturata come una collezione di moduli fisicamente separati, ognuno con il proprio registro degli indirizzi (Address Register, AR) e registro dei dati (Data Register, DR), le operazioni di accesso alla memoria possono procedere contemporaneamente in più di un modulo.

Così, si può incrementare la frequenza aggregata di trasmissione delle parole da o verso il sistema di memoria principale.

Distribuzione degli indirizzi nei moduli

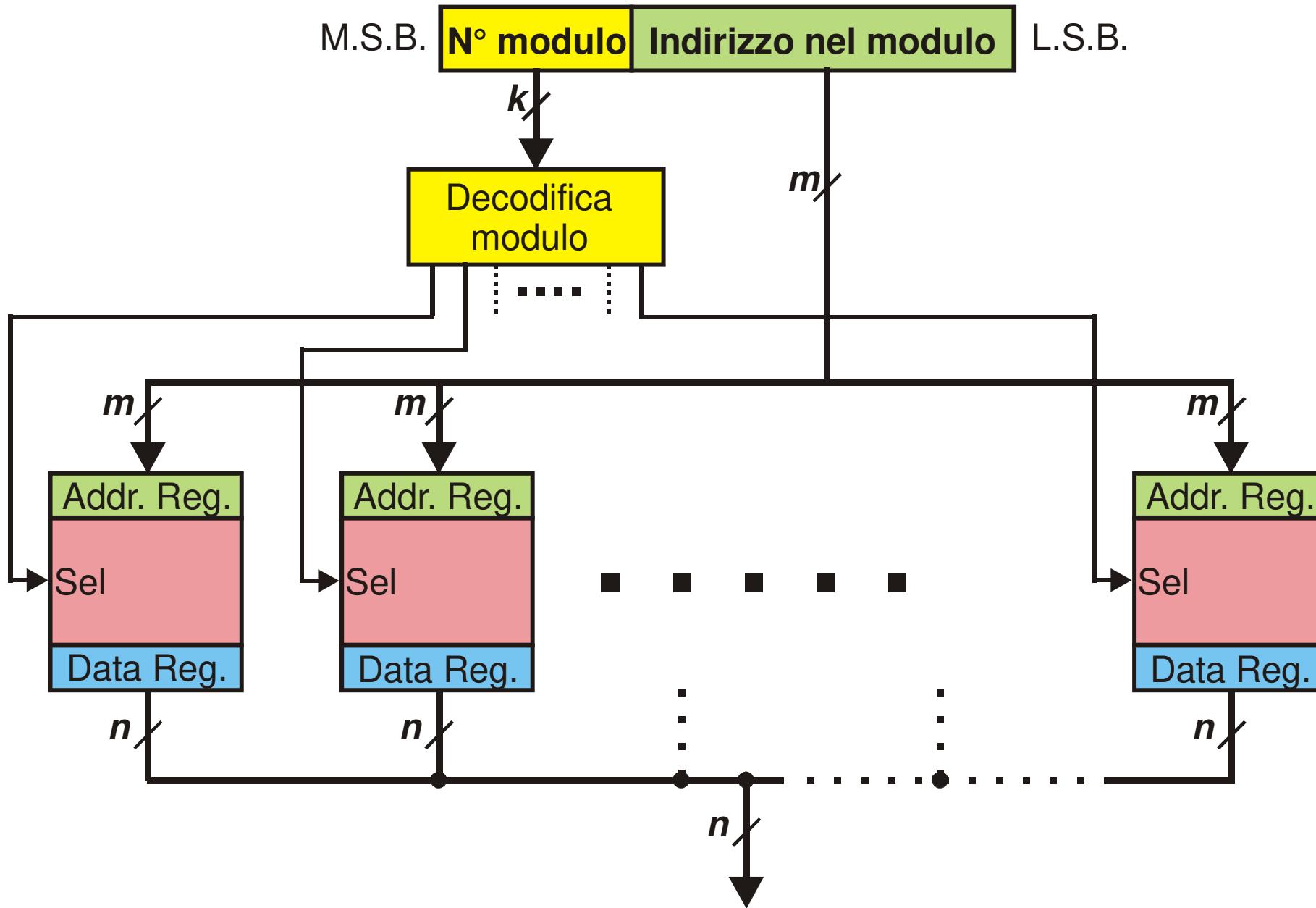
Il modo in cui gli indirizzi individuali sono distribuiti attraverso i moduli è importante per stabilire il numero medio di moduli che può essere tenuto occupato mentre le elaborazioni procedono.

Esistono due metodi di distribuzione degli indirizzi.

Nel primo, l'indirizzo di memoria generato dalla CPU viene decodificato nel seguente modo:

I k bit più significativi indicano uno degli n moduli, e gli m bit meno significativi indicano una particolare parola all'interno del modulo. Quando si accede a locazioni consecutive, come accade quando si copia un blocco di dati nella cache, viene coinvolto soltanto un modulo. Nello stesso tempo, tuttavia, dispositivi con possibilità di accesso diretto alla memoria (DMA) possono aver accesso alle informazioni contenute in altri moduli della memoria.

Indirizzo di memoria



Distribuzione degli indirizzi nei moduli

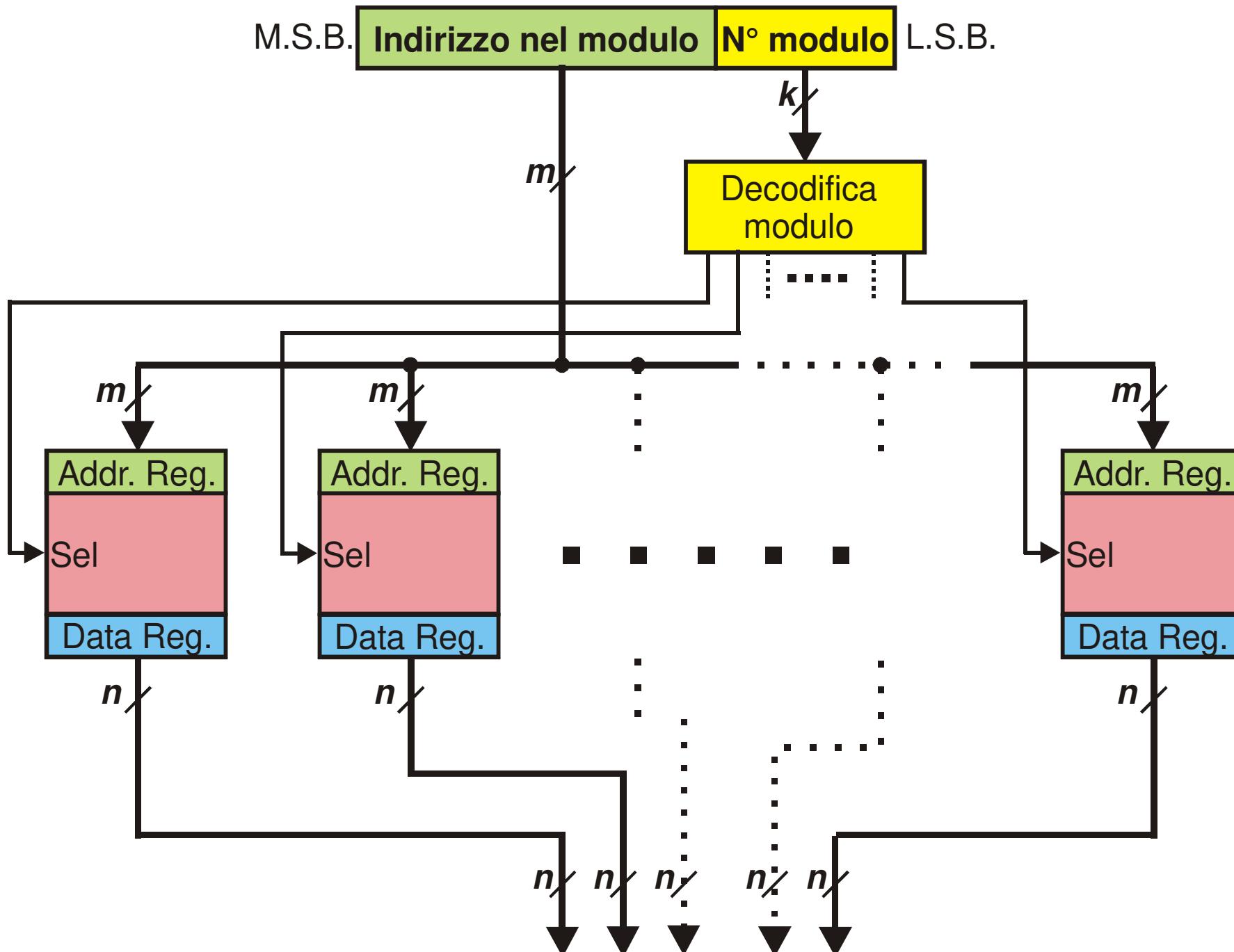
Il secondo e più efficiente modo di indirizzare i moduli è chiamato ***interallacciamento della memoria***.

I k bit meno significativi dell'indirizzo di memoria selezionano un modulo, mentre gli m bit più significativi indicano una locazione all'interno del modulo. In questo modo, indirizzi consecutivi sono posizionati in moduli successivi; così facendo, ogni componente del sistema che faccia richiesta di accesso a locazioni di memoria consecutive può tenere occupati vari moduli in un qualsiasi istante. Ciò consente di ottenere sia accessi a un blocco di dati più veloci, sia una più elevata media complessiva di utilizzo della memoria.

Per realizzare la struttura interallacciata, devono esserci 2^k moduli.

In caso contrario, ci sono degli stati vuoti associati a locazioni inesistenti nello spazio di indirizzamento della memoria.

Indirizzo di memoria

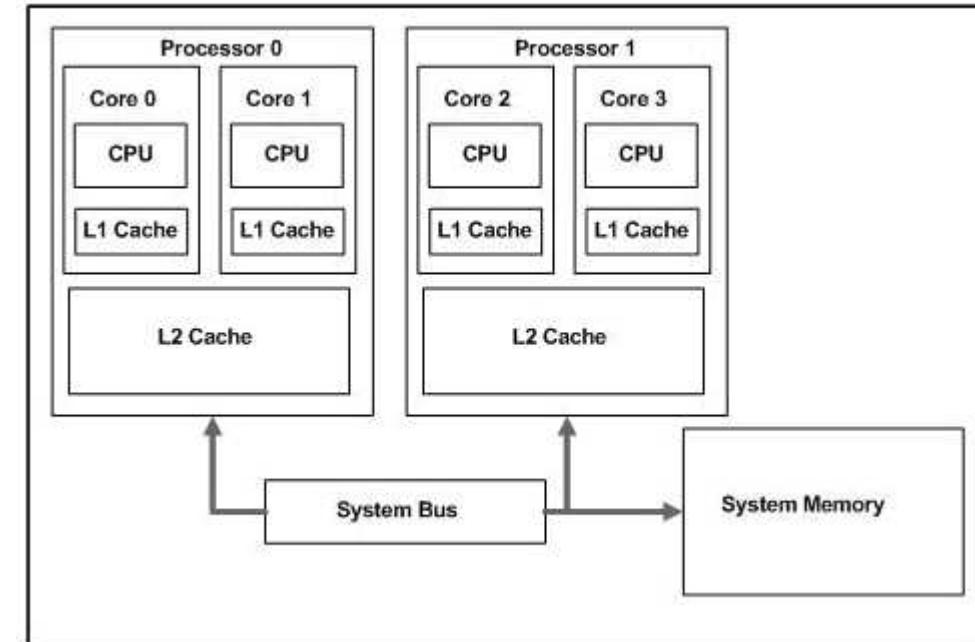


Cache multi livello

- Oggigiorno la memoria cache costituisce una gerarchia di 2 o 3 livelli: L1, L2, L3
- Vi possono essere due tipologie di cache multi livello: inclusiva o esclusiva
- Inclusiva: L2 contiene i dati presenti in L1 (similmente L2 è contenuta in L3)
 - L2 contiene sia i dati correnti di L1, sia quelli che erano presenti in precedenza e sono stati sovrascritti mediante l'algoritmo di sostituzione (copy back)
 - Per mantenere la coerenza L1 ha una politica di write-through verso L2, e così L2 verso L3
 - Miss in L1 e hit in L2: il blocco relativo viene copiato da L2 a L1 (in caso di sostituzione un dato va da L2 a L1 e un'altro da L1 a L2)
 - Miss in L1 e L2: il blocco dalla main memory viene copiato sia in L1 che L2
- Esclusiva: L2 contiene solo i dati di copy back da L1 (victim cache)
 - La memoria complessiva della cache è data dalla somma delle capacità dei vari livelli, in realtà nelle moderne architetture la capacità di L2 è un ordine di grandezza rispetto a quella di L1, e così fra L3 e L2, il guadagno rispetto alle cache inclusive non è molto significativo).
 - Miss in L1 e hit in L2: Le linee di cache di L1 e L2 vengono scambiati fra loro, cioè la linea di cache di L1 viene memorizzata in L2 e la linea di L2 in L1.
 - Miss in L1 e L2: Il dato letto dalla memoria è memorizzato direttamente in L1 e la linea di cache rimpiazzata di L1 (*victim data*) è trasferita in L2 rimpiazzando un'altra linea di cache secondo la politica di rimpiazzo usata.

Cache condivise

- Nelle architetture multi core L2 o L3 possono essere condivisi fra più core di uno stesso processore, questo consente:
 - Un uso più efficiente del livello condiviso
 - Se un core è inattivo, allora l'altro può utilizzare per se il livello condiviso
 - Nella programmazione multi-threading consente di utilizzare in maniera più efficiente dati condivisi
 - Un core può eseguire un pre-/post-processing di dati forniti dall'altro core
 - Riduce il front-side bus traffic
 - Un unico livello condiviso che si interfaccia con la main memory



La Memoria Virtuale

Agli albori dell'informatica la memoria centrale di un computer raramente superava le 4k parole ed i sistemi operativi erano, a dir poco, essenziali.

A quell'epoca, chiunque si disponesse a scrivere un programma doveva prima informarsi di quale fosse la disponibilità di spazio nella memoria centrale del sistema da adoperare, al netto delle necessità della CPU e della "parte residente" del sistema operativo.

Fatto il conteggio doveva scrivere un programma fatto di pezzi da mandare in esecuzione uno per volta (**overlay, sovrapposizione**), registrando i risultati parziali in gruppi di memoria i cui indirizzi fossero noti anche ai successivi pezzi di programma.

Questi li prelevavano per usarli ed, a loro volta registravano i risultati in locazioni prefissate.

Senza questa procedura il programma non poteva essere eseguito.

La Memoria Virtuale

Tutto questo è oggi completamente ignoto agli utilizzatori di computer.

Il fatto che talvolta siano indicati dei requisiti minimi del sistema per usare un certo programma, è un semplice suggerimento per ottenere dal programma un comportamento ragionevole, senza lunghissimi tempi di attesa tra due operazioni successive.

Oggi è assolutamente normale che programmi che tra occupazione diretta ed indiretta avrebbero bisogno di 25-30 Mbyte girino su macchine con memoria centrale da 16 Mbyte, grazie alla tecnica della “**memoria virtuale**”, inventata da un gruppo di ricercatori inglesi di Manchester nel 1961 ed adottata dall'IBM negli anni '70.

La tecnica si avvale di strutture hardware che è fondamentale che siano nel circuito integrato della CPU e ciò spiega perché essa è apparsa nei sistemi basati su microprocessori “single chip” relativamente di recente.

La gestione è affidata al sistema operativo

In estrema sintesi la tecnica della *memoria virtuale* prevede che il sistema operativo assegni una parte degli spazi liberi nella memoria centrale ai segmenti di un programma installato su disco.

Il sistema operativo seguirà criteri prefissati, assegnando uno spazio compreso tra un minimo ed un massimo, in funzione della dimensione del programma.

Gli spazi potranno essere anche non continui.

Il risultato è che programmi piccoli saranno trasferiti interamente ma, naturalmente, in funzione della mappa di occupazione corrente potranno essere dislocati in maniera frammentata in qualsiasi parte della memoria.

La memoria è gestita dinamicamente

Programmi di grandi dimensioni avranno a disposizione spazi insufficienti a contenerli completamente e saranno soggetti a continue operazioni di “**swapping**”, cioè di sostituzione dinamica dei loro segmenti nello spazio di memoria centrale assegnato.

L'operatore può imporre al sistema operativo di dedicare più spazio ad un programma, ma se esso usa una gran parte del campo d'indirizzamento del processore, non potrà in ogni caso essere completamente trasferito in memoria.

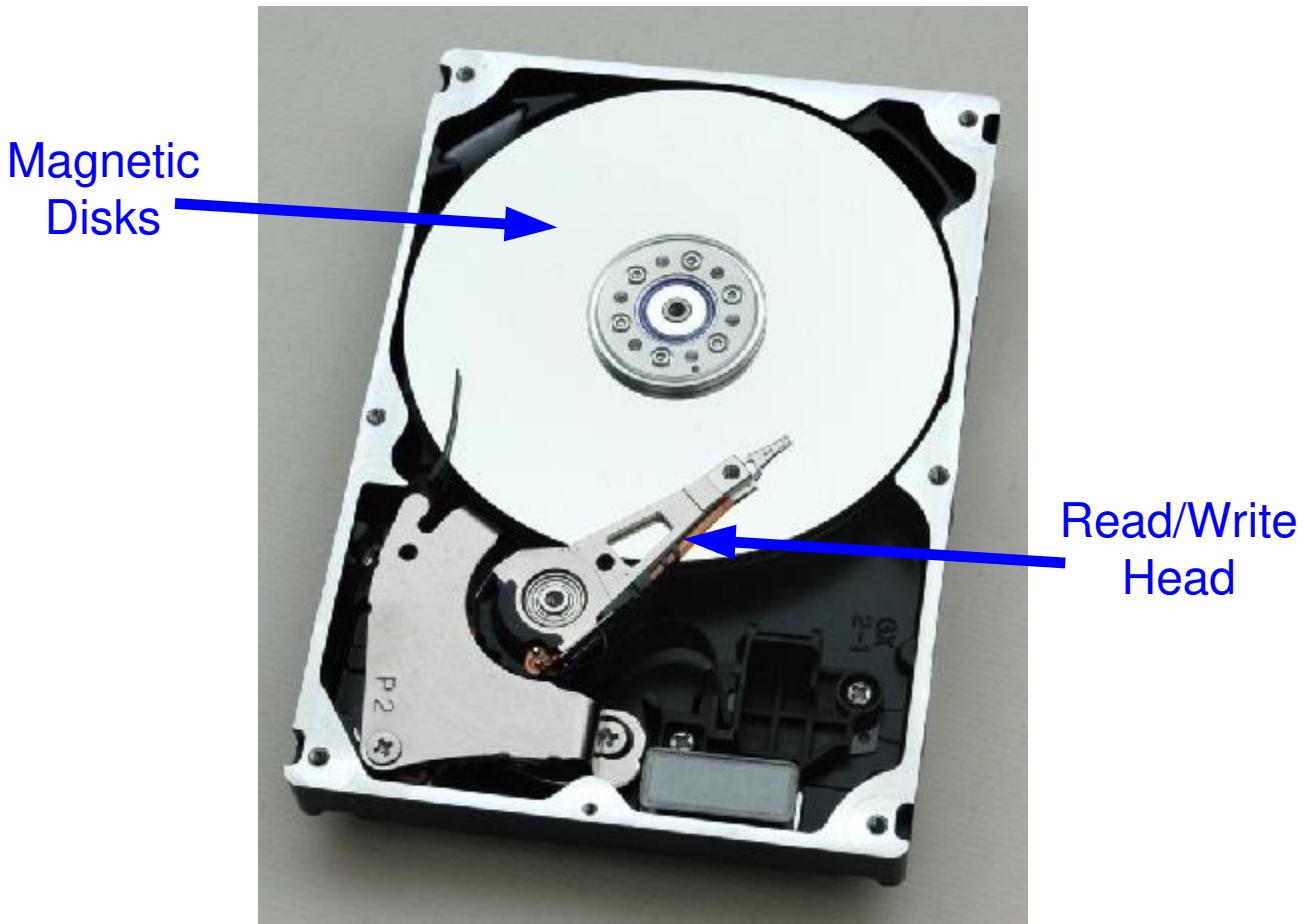
La tecnica di gestione degli spazi ha molti punti in comune con il funzionamento di una memoria cache.

Anche in questo caso, infatti, se gli spazi a disposizione nella memoria principale sono saturati, e il programma accede ad una locazione di memoria che non corrisponde ad un indirizzo esistente nella memoria centrale, si deve ricorrere ad un algoritmo di sostituzione.

Dimensioni della pagina di memoria

La parte di programma contenente l'indirizzo richiesto viene prelevato dal disco (trasferimento in DMA) e sostituita in memoria ad una delle parti che vi risiedevano.

Poiché il tempo di accesso al disco costituisce una penalità di fallimento molto più pesante, in termini di tempo rispetto a quella di una cache, l'entità minima da trasferire è presa molto più grande del blocco della cache ed è una *pagina di memoria* di solito compresa tra 2k e 16k; normalmente in un accesso al disco vengono scambiate molte pagine.



Takes milliseconds to *seek* correct location on disk

Overview

- **Indirizzi virtuali**

- I programmi usano indirizzi virtuali
- L'intero spazio degli indirizzi virtuali è memorizzato nell'hard drive (memoria di massa)
- Essendo la memoria fisica (main memory) più piccola della memoria di massa, solo un sottoinsieme degli indirizzi virtuali è presente nella main memory
- gli indirizzi virtuali vengono tradotti in indirizzi fisici della main memory
- I dati che non vengono ritrovati nella main memory sono prelevati dalla memoria di massa (operazione molto dispendiosa in termini di tempo)

- **Memory Protection**

- Il Sistema operativo associa ad ogni programma un suo spazio di indirizzi virtuali
- Così un programma può non essere consapevole che altri programmi sono in esecuzione

Analogie tra Cache e memoria virtuale

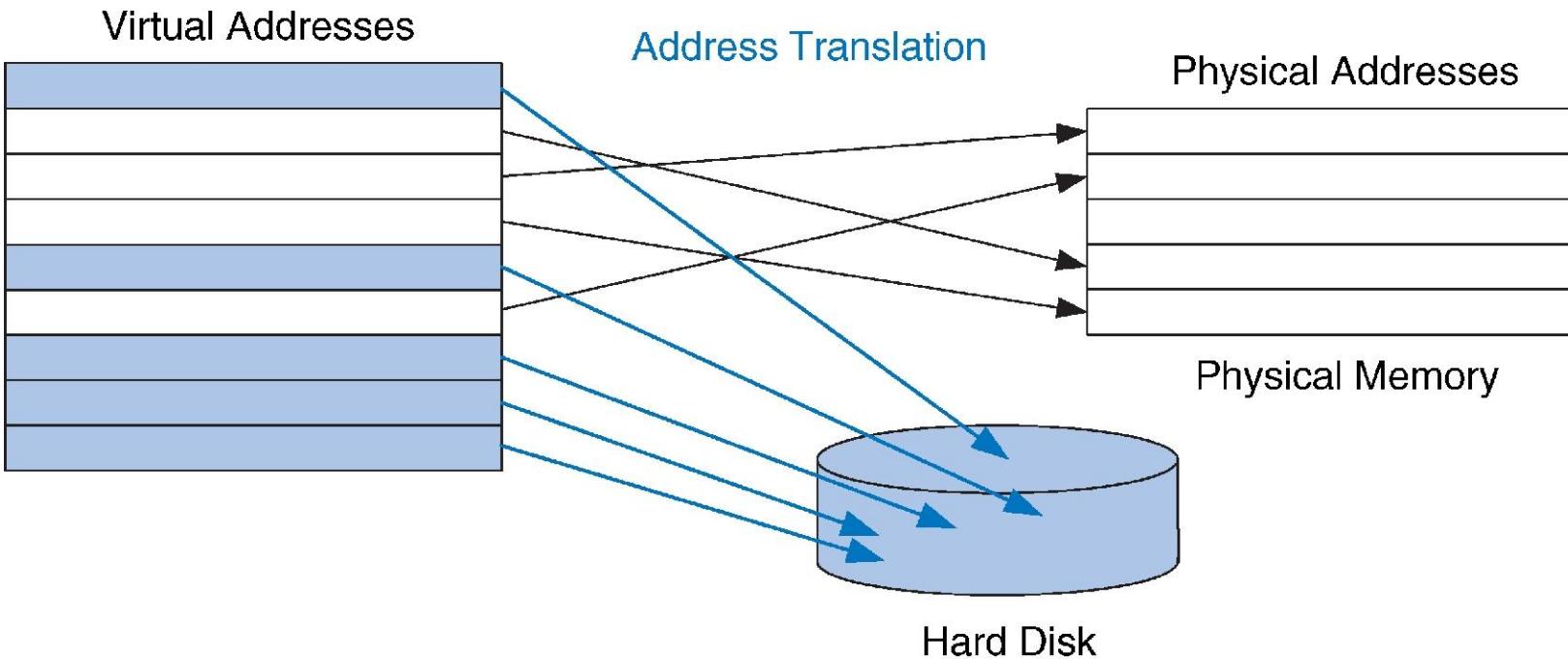
Cache	Virtual Memory
Block	Page
Block Size	Page Size
Block Offset	Page Offset
Miss	Page Fault
Tag	Virtual Page Number

La memoria fisica (main memory) agisce come cache per la memoria virtuale

Definizioni per la memoria virtuale

- **Page size:** dimensione dei segmenti di memoria trasferiti dalla memoria di massa alla main memory (2K-16K)
- **Address translation:** determinare l'indirizzo fisico nella main memory a partire da un indirizzo virtuale
- **Page table:** lookup table usata per tradurre gli indirizzi virtuali in indirizzi fisici

Indirizzi fisici e virtuali



- Data la grandezza delle pagine e la località spaziale e temporale durante l'esecuzione di un programma, la maggior parte degli accessi sono ritrovati nella main memory
- Ma i programmi sfruttano la maggior capacità della memoria virtuale

Esempio di memoria virtuale

System:

Virtual memory size: $2 \text{ GB} = 2^{31}$ bytes

Physical memory size: $128 \text{ MB} = 2^{27}$ bytes

Page size: $4 \text{ KB} = 2^{12}$ bytes

Organization:

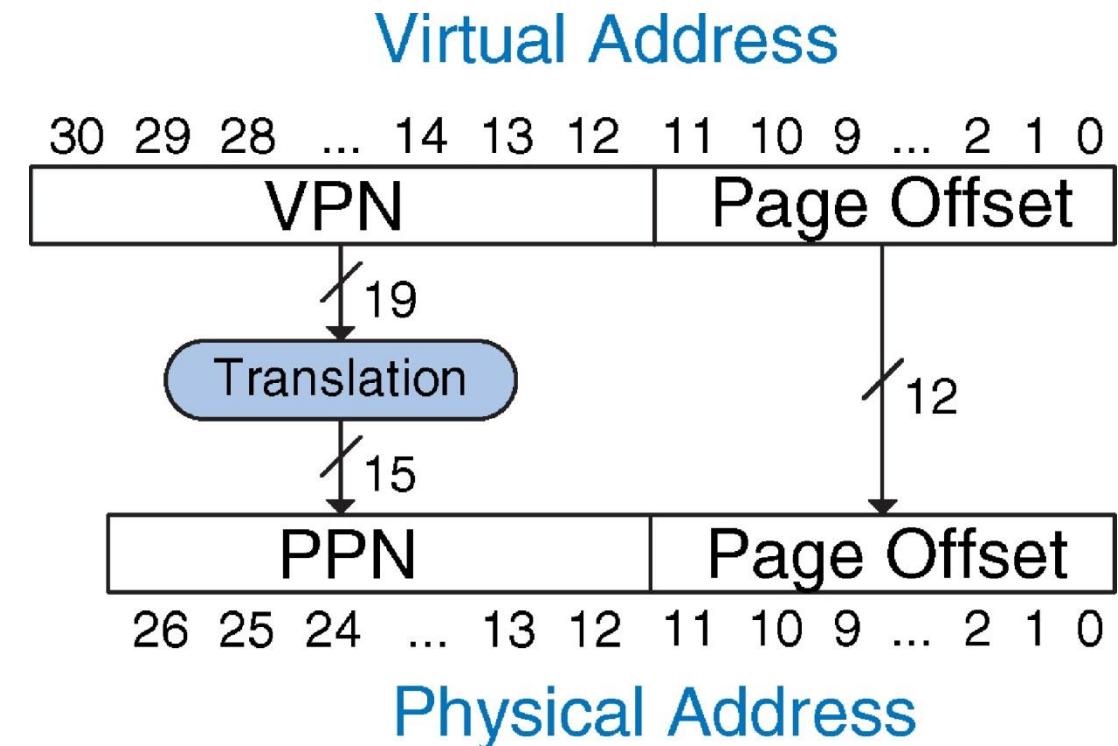
Virtual address: **31** bits

Physical address: **27** bits

Page offset: **12** bits

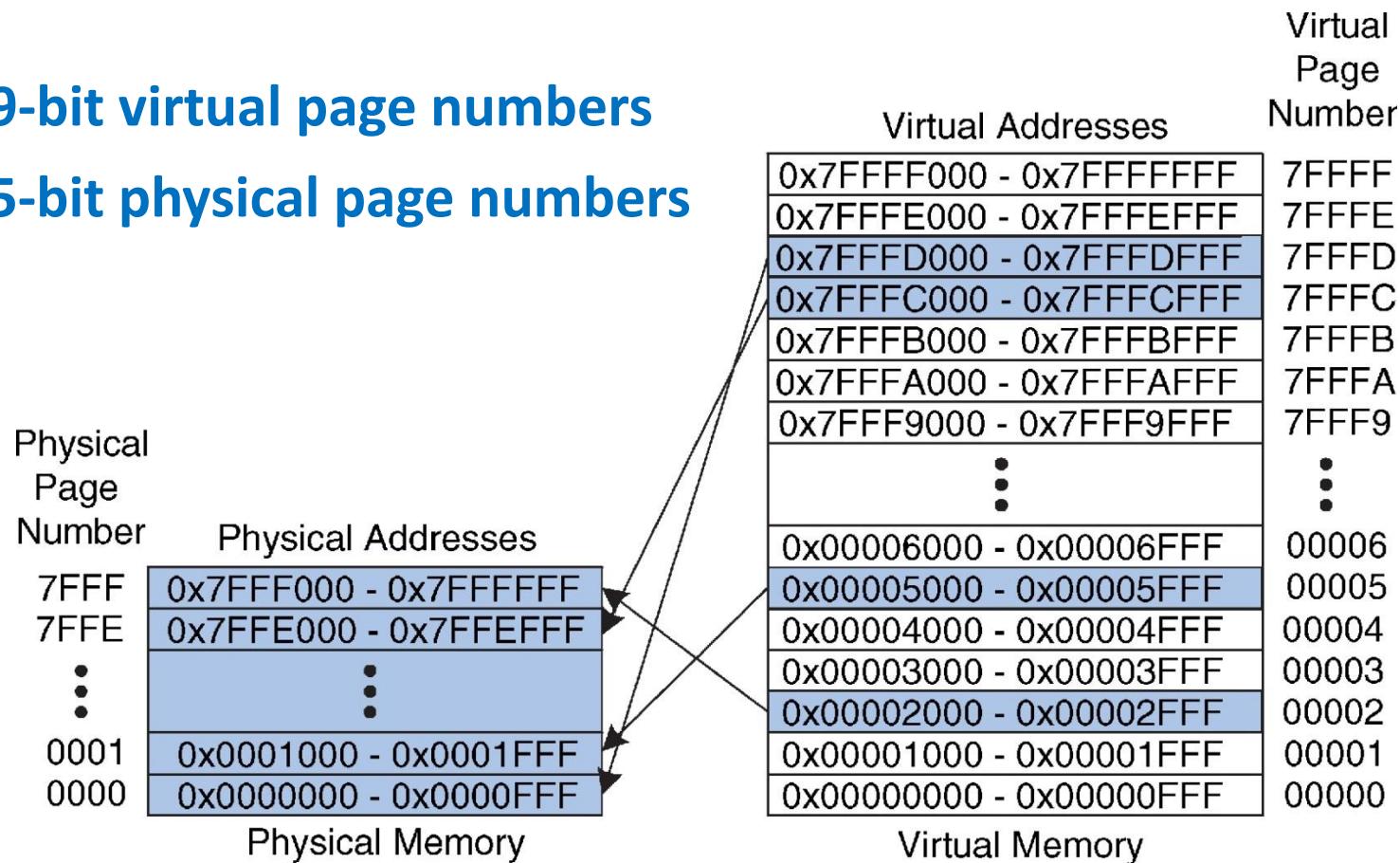
Virtual pages = $2^{31}/2^{12} = 2^{19}$ (VPN = **19** bits)

Physical pages = $2^{27}/2^{12} = 2^{15}$ (PPN = **15** bits)



Esempio di memoria virtuale

- 19-bit virtual page numbers
- 15-bit physical page numbers



Esempio di memoria virtuale

- 19-bit virtual page numbers
- 15-bit physical page numbers

Physical Page Number	Physical Addresses
7FFF	0x7FFF000 - 0x7FFFFFFF
7FFE	0x7FFE000 - 0x7FFEFFFF
⋮	⋮
0001	0x0001000 - 0x0001FFF
0000	0x0000000 - 0x0000FFF

Physical Memory

Virtual Memory size $2G = 31$ bit, quindi 8 esadecimali con il più significativo che arriva fino a 7

Virtual Addresses	Virtual Page Number
0x7FFF000 - 0x7FFFFFFF	7FFFF
0x7FFE000 - 0x7FFEFFFF	7FFFE
0x7FFFD000 - 0x7FFFDFFF	7FFFD
0x7FFFC000 - 0x7FFFCFFF	7FFFC
0x7FFFB000 - 0x7FFFBFFF	7FFFB
0x7FFFA000 - 0x7FFFAFFF	7FFFA
0x7FFF9000 - 0x7FFF9FFF	7FFF9
⋮	⋮
0x00006000 - 0x00006FFF	00006
0x00005000 - 0x00005FFF	00005
0x00004000 - 0x00004FFF	00004
0x00003000 - 0x00003FFF	00003
0x00002000 - 0x00002FFF	00002
0x00001000 - 0x00001FFF	00001
0x00000000 - 0x00000FFF	00000

Virtual Memory

Page 4K= 12 bit=3 esadecimali

Esempio memoria virtuale

Quale è l'indirizzo fisico
dell'indirizzo **0x247C**?

- VPN = **0x2**
- VPN 0x2 maps to PPN **0x7FFF**
- 12-bit page offset: **0x47C**
- Physical address = **0x7FFF47C**

Physical Page Number	Physical Addresses
7FFF	0x7FFF000 - 0x7FFFFFFF
7FFE	0x7FFE000 - 0x7FFEFFFF
:	:
0001	0x0001000 - 0x0001FFF
0000	0x0000000 - 0x0000FFF

Physical Memory

Virtual Page Number	Virtual Addresses
7FFFF	0x7FFFF000 - 0x7FFFFFFF
7FFFE	0x7FFFE000 - 0x7FFFEFFF
7FFFD	0x7FFF000 - 0x7FFF0FFF
7FFFC	0x7FFFC000 - 0x7FFFCFFF
7FFFB	0x7FFF000 - 0x7FFF0FFF
7FFFA	0x7FFF000 - 0x7FFF0FFF
7FFF9	0x7FFF000 - 0x7FFF0FFF
⋮	⋮
00006	0x00006000 - 0x00006FFF
00005	0x00005000 - 0x00005FFF
00004	0x00004000 - 0x00004FFF
00003	0x00003000 - 0x00003FFF
00002	0x00002000 - 0x00002FFF
00001	0x00001000 - 0x00001FFF
00000	0x00000000 - 0x00000FFF

Virtual Memory

La Memory Management Unit (MMU)

Quello a cui fa riferimento il processore nell'elaborazione è detto indirizzo **"virtuale"** o logico ed ha come unica limitazione di essere entro il campo di indirizzamento massimo del processore.

Il compito di tradurre questo indirizzo in un indirizzo che punti ad una locazione effettivamente esistente nel sistema (**indirizzo fisico**) è demandato ad una struttura logica contenuta nel chip del processore, che si chiama **Memory Management Unit (MMU)** e che opera sotto il controllo software del sistema operativo.

Traduzione degli indirizzi

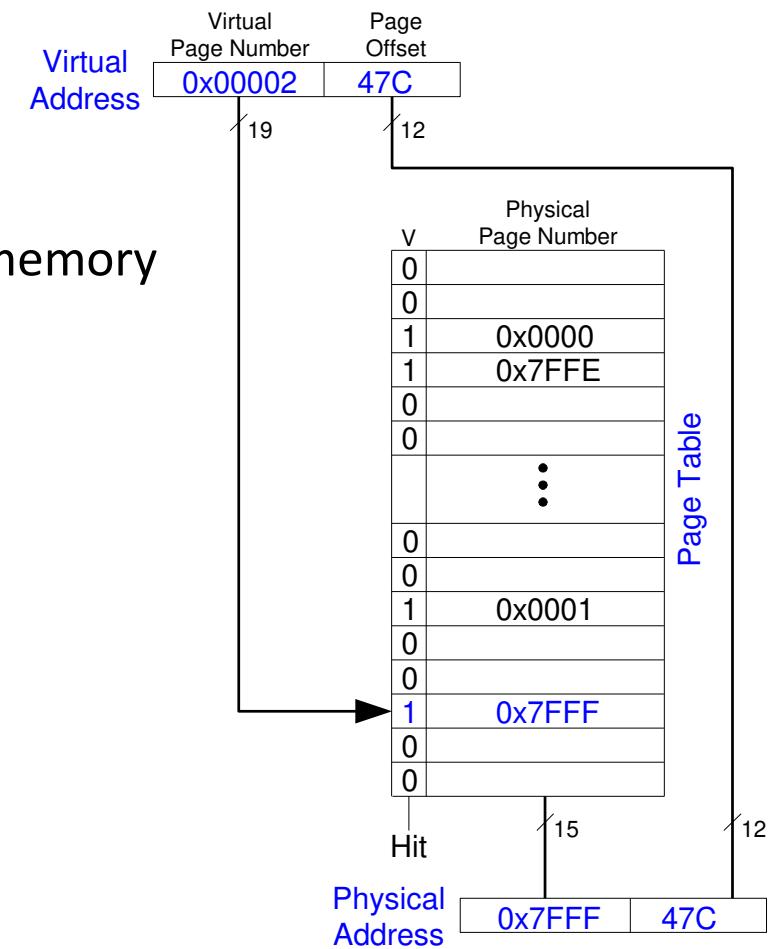
La memoria virtuale si comporta come una sorta di cache completamente associativa per il disco.

Tuttavia, se si gestisse la memoria virtuale come le cache, ossia tramite un comparatore sui bit più significativi per verificare la corrispondenza delle pagine, servirebbe un numero improponibile di comparatori, dato l'elevato numero di pagine in cui è organizzata la RAM.

In alternativa, il processo di traduzione degli indirizzi è gestito mediante **tabelle di paginazione**.

La page table

- Contiene un entry per ogni pagina virtuale
- Campi di una entry:
 - **Valid bit:** 1 se la pagina è presente nella main memory
 - **Physical page number:** dove la pagina è locata nella main memory



Esempio 1

Qual'è indirizzo fisico
dell'indirizzo virtuale
0x5F20?

- VPN = **5**
- Entry 5 in page table VPN
5 => physical page **1**
- Physical address: **0x1F20**

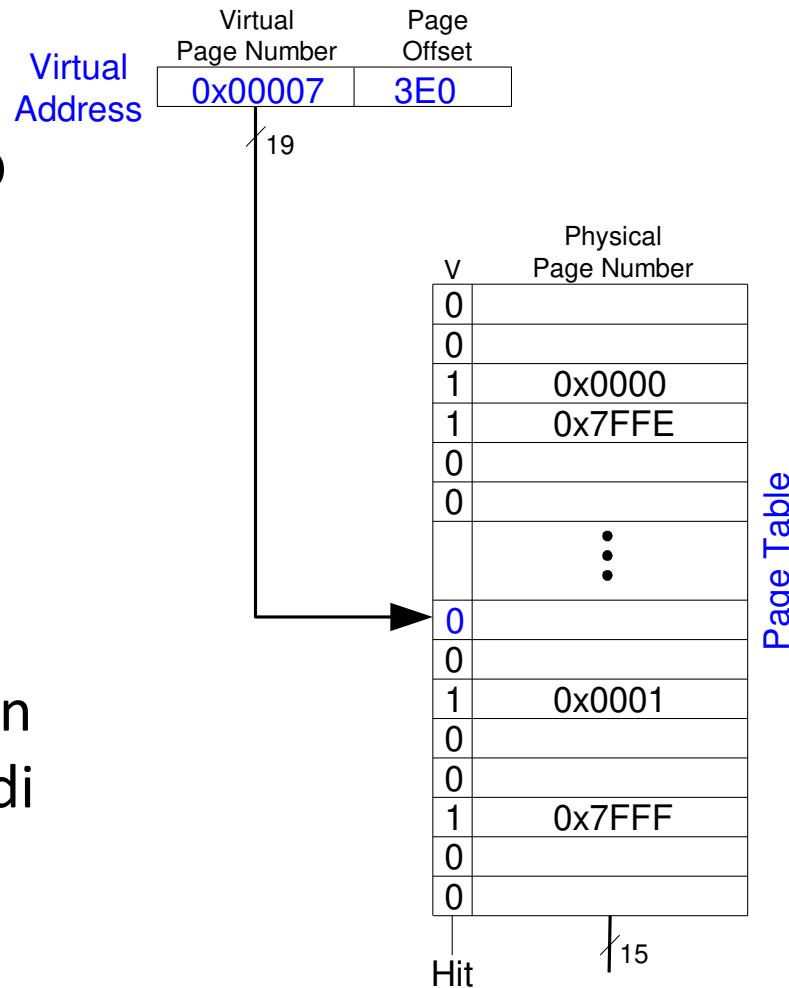
V	Physical Page Number
0	
0	
1	0x0000
1	0x7FFE
0	
0	
	⋮
0	
0	
1	0x0001
0	
0	
1	0x7FFF
0	
0	

Page Table

Esempio 2

Quale è l'indirizzo fisico
dell'indirizzo virtuale
0x73E0?

- VPN = **7**
- L'entry 7 non è valida
- La pagina virtuale deve essere caricata nella main memory dalla memoria di massa



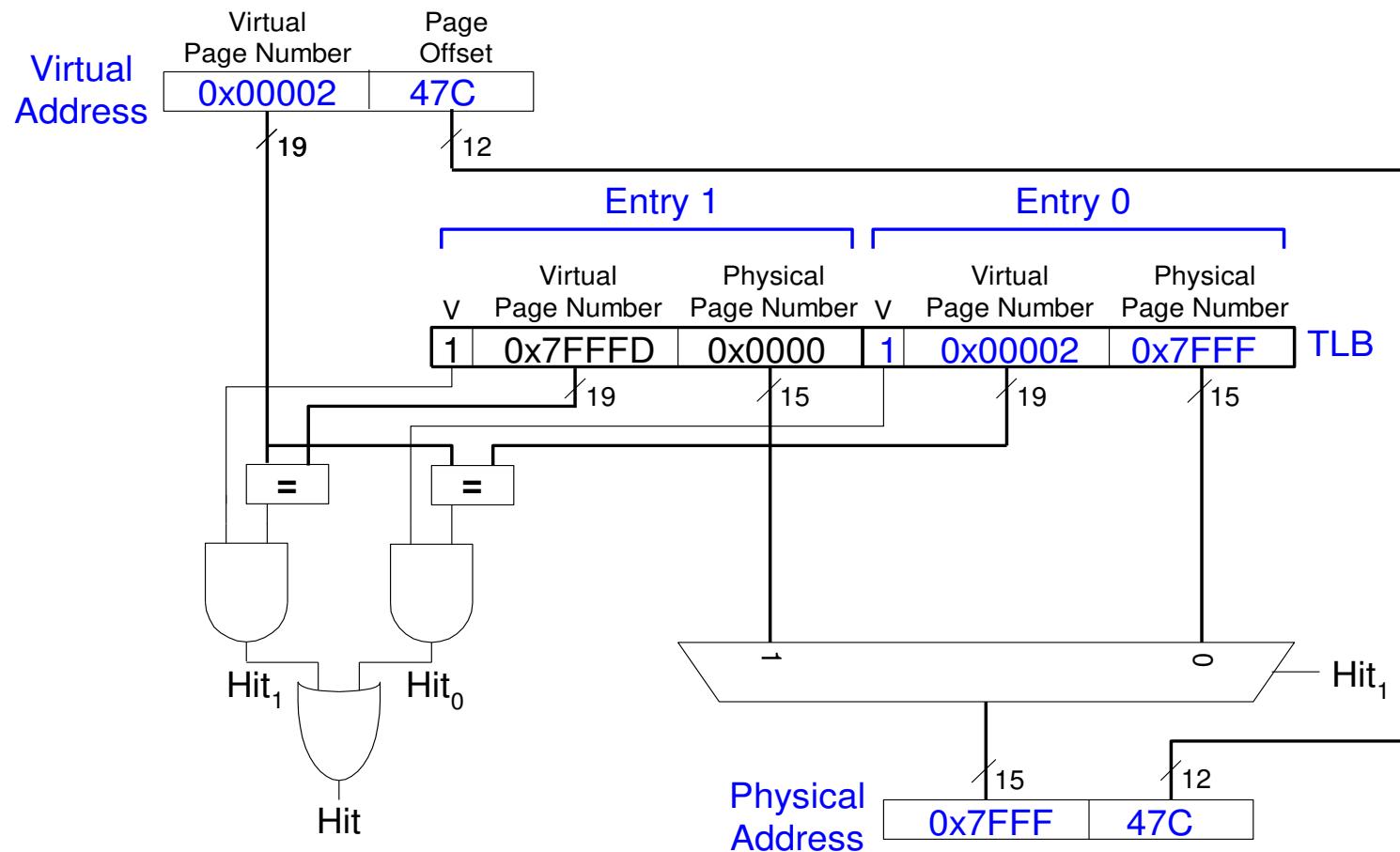
Caratteristiche della Page Table

- **La page table è solitamente voluminosa**
 - Di solito è essa stessa allocata nella main memory
- Le operazioni di load/store richiedono 2 accessi alla main memory:
 - Il primo per la traduzione (lettura della page table)
 - Il secondo è l'accesso vero e proprio al dato dopo la traduzione
- Questo dimezzerebbe le performance della main memory...

Translation Lookaside Buffer (TLB)

- Per migliorare le performance e far sì che la maggior parte delle operazioni di load e store avvenga con un solo accesso in memoria si usa una piccola cache dei traduzioni più recenti (TLB)
- TLB sfrutta il fatto che gli accessi alla page table godono di una elevate località temporale:
 - Essendo le dimensioni di una pagina relativamente grandi vi è un'alta probabilità che operazioni di load o store consecutive richiedano un accesso alla medesima pagina
- TLB
 - Veloce: accesso in < 1 cicli
 - Piccola: tipicamente 16 - 512 entry
 - Fully associative
 - Efficace: > 99 % hit rates

Esempio di 2-Entry TLB



Esercizi sulle cache

Domanda: Si ipotizzi che il 30% delle istruzioni in un programma tipico effettui un'operazione di scrittura o di lettura in RAM, e che la frequenza di successo di lettura in cache sia del 95% per le istruzioni e del 90% per i dati. Si supponga, inoltre, che la penalità di fallimento sia la stessa per operazioni di scrittura e operazioni di lettura. Si supponga che, in cicli di clock, una lettura diretta in memoria costi 10, una in cache costi 1 e una sia in cache che in memoria costi 16. Quale sarebbe il guadagno utilizzando la cache?

Esercizi sulle cache

Domanda: Si ipotizzi che il 30% delle istruzioni in un programma tipico effettui un'operazione di scrittura o di lettura in RAM, e che la frequenza di successo di lettura in cache sia del 95% per le istruzioni e del 90% per i dati. Si supponga, inoltre, che la penalità di fallimento sia la stessa per operazioni di scrittura e operazioni di lettura. Si supponga che, in cicli di clock, una lettura diretta in memoria costi 10, una in cache costi 1 e una sia in cache che in memoria costi 16. Quale sarebbe il guadagno utilizzando la cache?

Date n istruzioni, costo senza cache è: $(1 + 0,3) n = 10$

Costo con cache: $n (0,95 * 1 + 0,05 * 16) + 0,3 n (0,9 * 1 + 0,1 * 16)$

$$G = \frac{(1,3 * 10)n}{[(0,95+16 * 0,05)+(0,9+0,1*16)]n} = \frac{13}{2,5} = 5,2$$

Esercizi sulle cache

Domanda: Si ipotizzi che **4/5** delle istruzioni in un programma effettui un'operazione di scrittura o di lettura in RAM, e che la frequenza di successo di lettura in cache per le istruzioni sia **2/3** di quella per i dati. Si supponga, inoltre, che la penalità di fallimento sia la stessa per operazioni di scrittura e operazioni di lettura. Si supponga che, in cicli di clock, una lettura diretta in memoria costi **10**, una in cache costi **1** e che una in entrambe (sia in memoria, che in cache) costi **16**. Quale deve essere il tasso di successo dei dati per avere un guadagno maggiore di **1**?

Date n istruzioni, costo senza cache è: $(1 + 4/5) * n * 10 = 18 n$

Costo con cache: $n (2/3 * x * 1 + (1 - 2/3 * x) 16) + 4/5 n (x * 1 + (1-x) * 16)$

$$G = \frac{18}{[16 - 10x + 0,8(16 - 15x)]} = \frac{18}{28.8 - 22x} > 1$$

$$\begin{aligned} 22x &> 10.8 \\ x &> 0.49 \end{aligned}$$

Esercizi cache

Domanda 2 (Punti 5): Si consideri una memoria cache set-associativa da 2048 parole, suddivisa in 128 blocchi da 16 locazioni ciascuna, in cui si sceglie di avere 4 blocchi per insieme, ossia 32 insiemi. Sia data l'indirizzo 0x000244F4, in quale insieme della cache essa sarà inserita e con quale etichetta? Analogamente, quali sono insieme ed etichetta se i blocchi per insieme fossero 2?

0x000244F4 = 0000 0000 0000 0010 0100 0100 1111 0100

0000 0000 0000 0010 0100 0100 1111 0100

11 01 → Parola nel set = 13

10011 → Set = 19

0000 0000 0000 0010 0100 0 → Etichetta = 72

Esercizi cache

Domanda 2 (Punti 5): Si consideri una memoria cache set-associativa da 2048 parole, suddivisa in 64 blocchi da 32 locazioni ciascuna, in cui si sceglie di avere 2 blocchi per insieme, ossia 32 insiemi. Sia data la parola 0x0000D33C , in quale insieme della cache essa sarà inserita e con quale etichetta?

0x0000D33F = 0000 0000 0000 0000 1101 0011 0011 1100

011 11 → Parola nel set = 15

0011 0 → Set = 6

0000 0000 0000 0000 1101 → Etichetta = 13

Rappresentazione dell'informazione

Definire «informazione»

- Cos'è l'informazione? Proposta:
 - informazione è un qualunque insieme di segnali che condizionano l'evoluzione di un sistema
- Consideriamo, ad esempio, una caldaia con termostato. Lo scendere della temperatura al di sotto di una certa soglia genera un segnale di attivazione. Mentre, raggiunto un altro prefissato valore della temperatura, un secondo segnale provoca lo spegnimento della caldaia

Definire «informazione»

- Sembra una buona definizione ma
 - Prediamo ora come esempio una automobile: girando lo sterzo invia un segnale che modifica la traiettoria dell'auto facendola curvare
 - Esempi simili: ruotare la manovella di un crick modifica l'inclinazione di una macchina
- Così facendo qualsiasi cosa perturbi l'evoluzione di un sistema potrebbe essere considerato informazione
 - Problema: occorre definire cosa sia un segnale
 - Sebbene rispetto al termine informazione ci sembra più familiare e abbiamo più facilità ad associarlo a esempi concreti, definire nella sua generalità il termine segnale potrebbe essere non meno difficile di definire cosa sia un'informazione
 - Occorre poi stare attenti a non cadere in definizioni circolari: un segnale è una azione o evento che veicola una qualche informazione

Definire «informazione»

- Altri esempi/dubbi
 - Supponete di stare in macchina diretti verso il porto. Ad un certo punto dite al conducente che per il porto occorre svoltare a destra ma il conducente vi ignora e prosegue. Poiché quanto avete detto non ha condizionato l'evoluzione del sistema (la macchina ha continuato a tirare dritto) allora non può essere considerato un'informazione.
 - Sembra che il concetto di informazione quindi non dipenda necessariamente dalla effettiva evoluzione di un sistema, ne che sia indissolubilmente legato al concetto di segnale.
 - Allora la definizione è sbagliata?
 - In realtà se andate a cercare ad esempio sul vocabolario Treccani non ne troverete una migliore
 - Il problema è che la nozione di informazione, per la sua generalità, sembra essere una nozione primitiva
 - Non esiste un'unica definizione soddisfacente, ma ci si può accontentare di diverse nozioni, ognuna delle quali evidenzia un particolare aspetto
 - Una alternativa è quella di restringere il contesto in cui si intende una certa nozione per darne una definizione più precisa

Alfabeti, parole, linguaggi

- Per alfabeto A intenderemo un insieme finito e distinguibile di segni che chiameremo a seconda del contesto cifre, lettere, caratteri, simboli etc.
 - Le nove cifre dell'alfabeto decimale $A=\{'0',\dots,'9'\}$
 - Le ventisei lettere (minuscole) dell'alfabeto $A=\{'a',\dots,'z'\}$
 - I quattro simboli delle carte francesi $A=\{\clubsuit, \diamondsuit, \heartsuit, \spadesuit\}$
- Una parola (o stringa) su A è una sequenza finita di simboli dell'alfabeto A
 - “18945” ($A=\{'0',\dots,'9'\}$)
 - “pkwocod” ($A=\{'a',\dots,'z'\}$)
 - “ $\clubsuit \clubsuit \diamondsuit \spadesuit$ ” ($A=\{\clubsuit, \diamondsuit, \heartsuit, \spadesuit\}$)
- Con A^* indicheremo tutte le possibili parole generabili a partire dall'alfabeto A

Alfabeti, parole, linguaggi

Un linguaggio L sull'alfabeto A è un qualsiasi sottoinsieme di A^* .

Parole italiane

“pwfnfk”

“dog”

“siengs”

“casa”

“door”

“porta”

Parole inglesi

“pwfnfk”

“**dog**”

“siengs”

“casa”

“**door**”

“porta”

Alfabetti, parole, linguaggi

Assumiamo $A = \{0, \dots, 9\}$

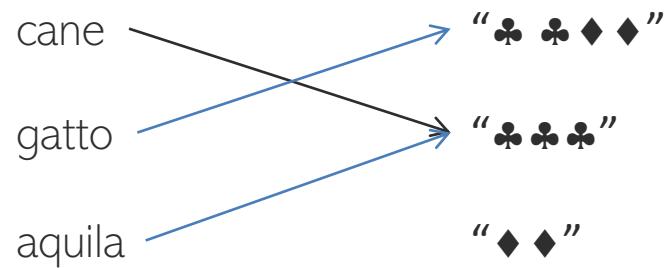
- Sia L l'insieme di parole che cominciano con '1':
 - "010", "123", "00237", "18923", "892003"
- Sia L l'insieme di parole in cui tutte le cifre di A occorrono lo stesso numero di volte
 - "02889", "", "22335", "4433", "0912837465", "09128374650918234765"
 - "4433" non è in L : la cifra '4' occorre 2 volte, la cifra '7' occorre 0 volte.
 - La parola vuota "" è in L : tutte le cifre occorrono 0 volte.
- Sia L l'insieme di parole per cui se due cifre occorrono almeno una volta allora occorrono lo stesso numero di volte.
 - "02889", "", "22335", "4433", "0912837465", "09128374650918234765"
- **Esercizio:** definire il linguaggio con cui usualmente denotiamo i numeri naturali

Codifica, decodifica

- Le parole di un linguaggio non sono altro che sequenze di simboli che non hanno alcun senso finché non forniamo un modo per interpretarle
 - “♣ ♣♦ ♠” ?
- Associare gli elementi di un insieme D alle parole di un linguaggio L viene detta codifica o rappresentazione di D.
- Ad esempio sia D l'insieme dei concetti cane, gatto e aquila (notate bene che sto parlando dei concetti non delle parole “cane”, “gatto” e “aquila”). Allora nella usuale codifica della lingua inglese
 - Cane → “dog”
 - Gatto → “cat”
 - Aquila → “eagle”
- Formalmente una codifica è una funzione totale $f: D \rightarrow L$
 - Se f non è suriettiva allora diremo che è ridondante
 - Se f non è iniettiva allora diremo che ambigua

Codifica, decodifica

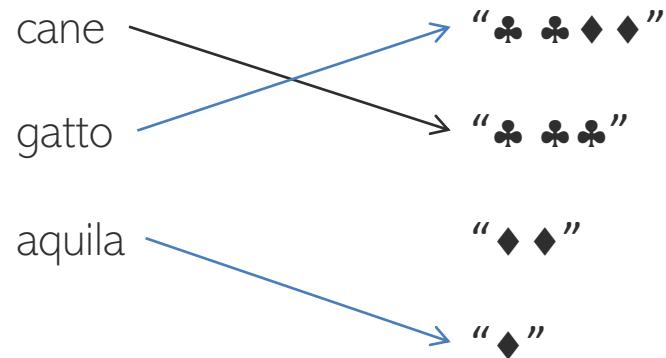
- Sia $D=\{\text{cane}, \text{gatto}, \text{aquila}\}$ e $L=\{\text{"♣ ♣ ♦ ♦"}, \text{"♣ ♣ ♣"}, \text{"♦ ♦"}\}$ e sia f rappresentata graficamente



- f è ambigua e ridondante

Codifica, decodifica

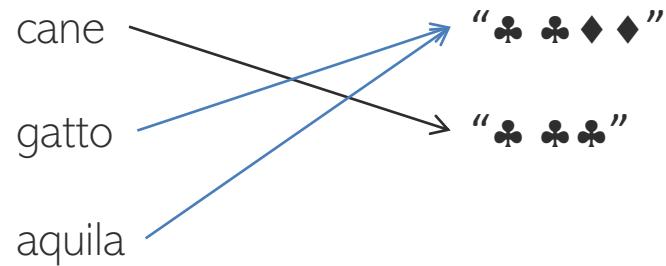
- Sia $D=\{\text{cane}, \text{gatto}, \text{aquila}\}$ e $L=\{\text{"♣ ♣ ♦ ♦"}, \text{"♣ ♣ ♣"}, \text{"♦ ♦"}, \text{"♦"}$ e sia f rappresentata graficamente



- f non è ambigua ma è ridondante

Codifica, decodifica

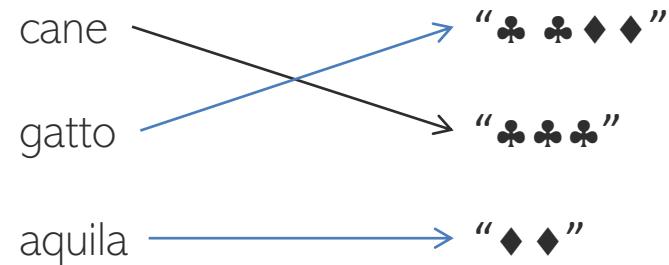
- Sia $D=\{\text{cane, gatto, aquila}\}$ e $L=\{\text{"♣ ♣ ♦ ♦"}, \text{"♣ ♣ ♣"}\}$ e sia f rappresentata graficamente



- f non è ridondante ma è ambigua

Codifica, decodifica

- Sia $D=\{\text{cane}, \text{gatto}, \text{aquila}\}$ e $L=\{\text{"♣ ♣ ♦ ♦"}, \text{"♣ ♣ ♣"}, \text{"♦ ♦"}\}$ e sia f rappresentata graficamente



- f non è ambigua e non è ridondante

Codifica, decodifica

- Sia N_D la cardinalità di D e N_L la cardinalità di L
 - Se $N_D > N_L$ qualsiasi codifica $f: D \rightarrow L$ è ambigua
 - Se $N_D < N_L$ qualsiasi codifica $f: D \rightarrow L$ è ridondante
- Inoltre è bene notare che
 - Non ambiguo significa che f è iniettiva (ad elementi distinti di D corrispondono elementi distinti di L)
 - Non ridondante significa che f è suriettiva (ogni elemento di L è la codifica di almeno un elemento di D)
 - Quindi se f è non ambigua e non ridondante allora f è iniettiva e suriettiva, quindi è una funzione biunivoca
- Una funzione $g: L \rightarrow D$ è invece detta una decodifica di D
- Se f è non ambigua (ovvero iniettiva) allora è invertibile. Quindi, induce naturalmente una decodifica $g = f^{-1}$

Proprietà delle codifiche

- Abbiamo già visto
 - ambigua/non ambigua
 - ridondante/non ridondante
- Altre proprietà
 - Economicità: numero di simboli utilizzati per unità di informazione
 - Semplicità nell'operazione di codifica e decodifica
 - Semplicità nell'eseguire operazioni sull'informazione codificata

Rappresentazione dei numeri naturali

- Occupiamoci ora delle possibili codifiche dei numeri naturali $0, 1, 2, \dots$
- Un codice ovvio è dato dal sistema di numerazione decimale basato su $A = \{'0', \dots, '9'\}$.
- Tale sistema è un sistema posizionale ovvero ad ogni cifra di una parola è assegnato un peso differente a seconda della posizione nella sequenza. Ad esempio dato "4456"
 - '6' rappresenta sei unità (cifra meno significativa)
 - '5' rappresenta cinque decine
 - '4' rappresenta quattro centinaia
 - '4' rappresenta quattro migliaia (cifra più significativa)

Fin'ora ho estensivamente usato gli apici per distinguere le parole ("4456") da quello che rappresentano (il numero 4456). E' chiaro che usare gli apici ogni volta per rimarcare la distinzione fra numerali (parole) e numeri (concetti) può rendere molto pesante la trattazione. Quindi in seguito inizierò ad omettere gli apici qualora il contesto non generi ambiguità.

Rappresentazione in base 10

- Decomponendo in potenze di 10, il numerale 1024 rappresenta il numero

$$1 \times 10^3 + 0 \times 10^2 + 2 \times 10^1 + 4 \times 10^0$$

- Generalizzando, un numerale $c_{m-1}c_{m-2}\dots c_0$ rappresenta

$$\sum_{i=0}^{m-1} c_i \cdot 10^i$$

- Il sistema decimale è quindi una codifica posizionale su base 10. Tuttavia non è l'unica, ad esempio i babilonesi utilizzavano un sistema di numerazione su base 60 (sessagesimale)
- I dispositivi digitali per elaborare e memorizzare l'informazione possono trovarsi in due possibili stati che rappresentano la cifra '0' e la cifra '1'. Il sistema corrispondente è quello in base 2 (binario)

Rappresentazione in una base generica

- Ad ogni naturale $b > 1$ corrisponde una codifica in base b .
- L'alfabeto A_b consiste in b simboli distinti che corrispondono ai numeri $0, 1, \dots, b-1$.
- Analogamente al sistema decimale, un numerale di cifre di A_b rappresenta il numero $s_{m-1} \dots s_0$

$$\sum_{i=0}^{m-1} s_i \cdot b^i$$

- Consideriamo ad esempio il sistema ottale (base 8). A_8 consiste nelle cifre '0', '1', ..., '7'

$$13 \longrightarrow 1 \cdot 8^1 + 3 \cdot 8^0 = 8 + 3 = 11$$

$$5201 \longrightarrow 5 \cdot 8^3 + 2 \cdot 8^2 + 0 \cdot 8^1 + 1 \cdot 8^0 = 5 \cdot 8^3 + 2 \cdot 8^2 + 1 \cdot 8^0 = 2560 + 128 + 1 = 2689$$

Rappresentazione in una base generica

A questo punto potrebbero sorgere delle ambiguità. Se qualcuno vi dicesse: 11 è un numero pari. Pensereste che sia impazzito. Lui potrebbe ribattere: certo! infatti in base 5

$$11 \rightarrow 1 \cdot 5^1 + 1 \cdot 5^0 = 5 + 1 = 6$$

E' chiaro che bisogna mettersi d'accordo su quale sistema di numerazione si adotta di volta in volta. Per questo useremo delle notazioni standard

- n denota un (generico) numero naturale, a prescindere dalla sua rappresentazione
- n_b denota un naturale rappresentato in base b
- Una sequenza di cifre decimali rappresenta un particolare naturale espresso in base dieci. Ad esempio 236 denota il numero duecentotrentasei
- Una sequenza di cifre seguite da un pedice b rappresenta un numero espresso in base b.
- esempio:

$$1001_2 = 2^3 + 1 = 9$$

Basi maggiori di 10

- Per basi $b < 10$ possiamo chiaramente ri-usare le usuali cifre '0', ..., '9'. Ad esempio, la codifica in base 6 utilizza le cifre '0', ..., '5' mentre quella in base 3 le cifre '0', '1' e '2' e così via.
- E per basi $b > 10$?
 - Si prendono in prestito le lettere dell'alfabeto
 - Per $b=16$ le cifre adottate sono '0', ..., '9', 'a', ..., 'f', dove:

$$a_{16} = 10 \quad b_{16} = 11$$

$$c_{16} = 12 \quad d_{16} = 13$$

$$e_{16} = 14 \quad f_{16} = 15$$

- Esempio:

$$b3c_{16} = 11 \cdot 16^2 + 3 \cdot 16^1 + 12 \cdot 16^0 = 2816 + 48 + 12 = 2876$$

Basi maggiori di 10

- E se le lettere dell'alfabeto non dovessero bastare?

γ 1	Ϙ 11	Ϛ 21	Ϛ 31	Ϛ 41	Ϛ 51
ϙ 2	ϙ 12	Ϛ 22	Ϛ 32	Ϛ 42	Ϛ 52
Ϛ 3	ϙ 13	Ϛ 23	Ϛ 33	Ϛ 43	Ϛ 53
Ϛ 4	ϙ 14	Ϛ 24	Ϛ 34	Ϛ 44	Ϛ 54
Ϛ 5	ϙ 15	Ϛ 25	Ϛ 35	Ϛ 45	Ϛ 55
Ϛ 6	ϙ 16	Ϛ 26	Ϛ 36	Ϛ 46	Ϛ 56
Ϛ 7	ϙ 17	Ϛ 27	Ϛ 37	Ϛ 47	Ϛ 57
Ϛ 8	ϙ 18	Ϛ 28	Ϛ 38	Ϛ 48	Ϛ 58
Ϛ 9	ϙ 19	Ϛ 29	Ϛ 39	Ϛ 49	Ϛ 59
ϙ 10	Ϛ 20	Ϛ 30	Ϛ 40	Ϛ 50	

Lunghezza di n rispetto ad una base

- Chiamiamo lunghezza di n rispetto a b il numero di cifre che occorrono per rappresentare n in base b
 - la lunghezza di 101 rispetto a 2 è 7: $1100101_2 = 101$
 - la lunghezza di 101 rispetto a 10 è 3: $101_{10} = 101$
 - la lunghezza di 101 rispetto a 16 è 2: $65_{16} = 101$
- La lunghezza di un numerale decresce al crescere della base di codifica

Valore massimo rappresentabile

Fissato un certo numero di cifre, qual è il massimo numero rappresentabile in una data base?

Riferendoci alla base 10

- Con 1 cifra rappresentiamo i numeri da 0 a 9
- Con 2 cifre i numeri da 0 a 99
- Con 3 cifre i numeri da 0 a 999
- Con m cifre i numeri da 0 a $10^m - 1$

Quindi se indichiamo con v_{max} il massimo numero rappresentabile con m cifre in base 10 abbiamo

$$v_{max} = 10^m - 1$$

Valore massimo rappresentabile

- E per una base diversa da 10, quale è il massimo valore rappresentabile con m cifre?
- Consideriamo, ad esempio, il caso $b=2$ e $m=4$, il massimo valore rappresentabile corrisponde al numerale 1111

$$1111_2 = 2^3 + 2^2 + 2^1 + 2^0 = 15 = 2^4 - 1$$

- Per una generico b e m il massimo numero rappresentabile si ottiene concatenando m volte la cifra “più alta” $(b-1)$. Quindi:

$$\begin{aligned} v_{max} &= (b-1)b^{m-1} + (b-1)b^{m-2} + \dots + (b-1)b^1 + (b-1)b^0 = \\ &= (b \cdot b^{m-1} - b^{m-1}) + (b \cdot b^{m-2} - b^{m-2}) + \dots + (b \cdot b^1 - b^1) + (b \cdot b^0 - b^0) = \\ &= b^m - b^{m-1} + b^{m-1} - b^{m-2} + \dots + b^2 - b + b - 1 = b^m - 1 \end{aligned}$$

Cifre necessarie per rappresentare n

- Nella slide precedente avevamo il numero di cifre m e volevamo sapere quale è il *massimo numero rappresentabile* in una certa base b .
- Consideriamo il problema inverso: abbiamo un valore n e ci chiediamo quante *cifre m* occorrono per rappresentarlo.
- Chiaramente il massimo numero rappresentabile con m cifre dovrà essere maggiore o uguale a n

$$b^m - 1 \geq n$$

$$b^m \geq n + 1$$

$$m \geq \log_b(n + 1)$$

- In particolare cerchiamo il più piccolo m tale che $m \geq \log_b(n + 1)$

$$m = \lceil \log_b(n + 1) \rceil$$

Cifre necessarie per rappresentare n

- Notazione: dato un reale x
 - $\lceil x \rceil$ denota l'approssimazione intera per eccesso
 - $\lfloor x \rfloor$ denota l'approssimazione intera per difetto
- Esempi:
 - $\lceil 14.78 \rceil = 15$
 - $\lfloor 9.88 \rfloor = 9$
- In qualche testo potrete trovare $m = \lfloor \log_b n \rfloor + 1$
 - Qualcuno si sbagliato?
- **Esercizio:** dimostrare che per ogni n e b

$$\lceil \log_b(n + 1) \rceil = \lfloor \log_b n \rfloor + 1$$

- Esercizio: dimostrare che per ogni n e b $\lceil \log_b(n+1) \rceil = \lfloor \log_b n \rfloor + 1$
- Prima osservazione:

$$\log_b n \quad \log_b(n+1)$$

Se $x < y$ allora $\lfloor x \rfloor + 1 \leq \lceil y \rceil$

Quindi $\lfloor \log_b n \rfloor + 1 \leq \lceil \log_b(n+1) \rceil$

- Supponiamo per assurdo che $\lfloor \log_b n \rfloor + 1 < \lceil \log_b(n+1) \rceil$
~~~~~  
↳ CHIUSO
- Essendo  $c$  un intero allora  $c$  deve essere compreso fra

$$\log_b n \quad c \quad \log_b(n+1)$$

- Ma allora avremmo:

$$\log_b n < c < \log_b (n+1)$$

↓ ↓

$b^c$   $\times$

$$b^{\log_b n} < b^c < b^{\log_b (n+1)}$$

$$n < b^c < n+1$$

$$\begin{aligned} b &\in \mathbb{N} \\ c &\in \mathbb{N} \end{aligned} \Rightarrow b^c \in \mathbb{N}$$

- Quindi esisterebbe un numero naturale fra  $n$  e  $n+1$ , assurdo!

# Relazione fra due basi diverse

- Consideriamo due sistemi di numerazione nelle basi  $a$  e  $b$ . Dato un naturale  $n$ , quale relazione intercorre fra  $m_a$  e  $m_b$ ?
- Per ogni base  $c$   $\lfloor \log_c n \rfloor + 1 - \log_c n \leq 1$
- Quindi considereremo le approssimazioni

$$\begin{aligned}m_a &\simeq \log_a n \\m_b &\simeq \log_b n\end{aligned}$$

- Ricordando che

$$\log_a n = \frac{\log_b n}{\log_b a}$$

- il rapporto fra le lunghezze  $m_a$  e  $m_b$  è costante (indipendente da  $n$ ) e pari a  $\log_a b$

$$m_a \simeq \log_a n = \frac{\log_b n}{\log_b a} = \frac{m_b}{\log_b a}$$

# Digressione: base unaria

- Ricordate che quando abbiamo introdotto i sistemi di numerazione abbiamo assunto la base  $b \geq 2$ .
- Perché non è possibile considerare una numerazione unaria? Certo! Ma ci sono delle controindicazioni...
  - La base unaria consta di una solo cifra I (detta in gergo matematico *mazzarella* 😊 )
  - Una mazzarella rappresenta 0, due mazzarelle 1, ...,  $n+1$  mazzarelle rappresentano  $n$
  - $\text{||||}_1 = 5$
  - Notate che a prescindere dalla posizione ogni mazzarella vale 1, quindi questo sistema non è posizionale
  - Non potrebbe essere altrimenti visto che 1 elevato ad una qualsiasi potenza è sempre uguale a 1!

# Digressione: base unaria

Andiamo a confrontare le lunghezze dei numerali con una base  $b \geq 2$

$$m_1 = n + 1 = b^{\log_b n} + 1 \simeq b^{m_b} + 1 \simeq b^{m_b}$$

Questo vuol dire che la codifica di  $n$  in base unaria cresce esponenzialmente rispetto alla codifica in base  $b$ !

La base unaria non è un buon sistema di rappresentazione

# Codifica ottimale

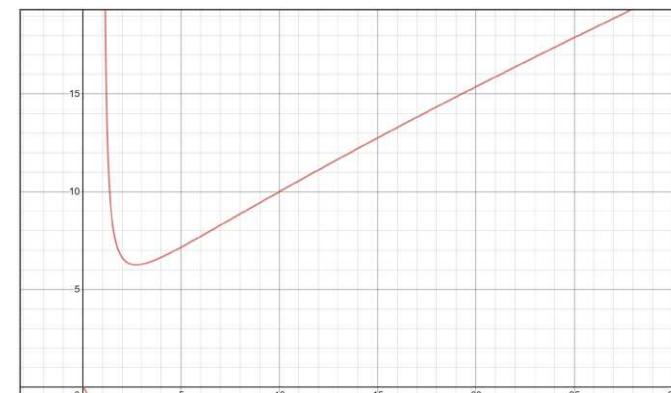
- Ritorniamo alle basi ammissibili ( $b \geq 2$ )
- Abbiamo visto che più grande è la base  $b$ , minore è il numero di cifre che occorrono per rappresentare un numero
- Quindi il codice binario è il sistema di codifica meno economico fra quelli ammissibili
  - Per esempio  $\log_2 10 \cong 3.32$  questo vuol dire che per rappresentare un numero  $n$  in binario occorrono *circa il triplo* delle cifre che occorrono per rappresentare lo stesso  $n$  nel sistema decimale
  - *Perché i computer adottano la codifica binaria?*

# Codifica ottimale

- Non bisogna tener presente solo la lunghezza di codifica ma anche il fatto che un calcolatore per operare in una certa base  $b$  deve poter rappresentare tutte le cifre di quella base, *quindi gli servono  $b$  differenti stati*.
- Una nozione di costo di una codifica che tiene conto anche di questo fattore è data dal prodotto

$$b \cdot m_b \simeq b \cdot \log_b n$$

- Si può mostrare che il valore di  $b$  che minimizza tale costo è il numero di Nepero e  $\approx 2.7$ , quindi le codifiche che si avvicinano di più a tale valore ottimale sono quelle in base 2 e 3



# Cambiamento di base

- Problema: dato  $n$  rappresentato in base  $a$ , quale è la sua rappresentazione in base  $b$ ?
- Supponiamo di saper fare le quattro operazioni in base  $a$
- L'algoritmo di cambiamento di base consiste nel dividere ripetutamente  $n$  (espresso in base  $a$ ) per  $b$  finché il quoziente non risulti uguale a zero. La sequenza di resti ottenuti (compresi tra 0 e  $b-1$ ) è la codifica dalla cifra meno significativa a quella più significativa di  $n$  in base  $b$
- Esempio: codificare  $251_{10}$  in base 3

$$251/3 = 83 \text{ resto: } 2$$

$$83/3 = 27 \text{ resto: } 2$$

$$27/3 = 9 \text{ resto: } 0$$

$$9/3 = 3 \text{ resto: } 0$$

$$3/3 = 1 \text{ resto: } 0$$

$$1/3 = 0 \text{ resto: } 1$$

$$251_{10} = 100022_3$$

# Cambiamento di base

- Esempio: codificare  $333_7$  in base 9
- Problema: non siamo molto allenati con la divisione in base 7. Meglio affrontare il problema in due passi:

- Codifico  $333_7$  in base 10

$$333_7 = 3 \cdot 7^2 + 3 \cdot 7^1 + 3 \cdot 7^0 = 171$$

- Codifico  $171_{10}$  in base 9

$$171/9 = 19 \quad \text{resto: 0}$$

$$19/9 = 2 \quad \text{resto: 1}$$

$$2/9 = 0 \quad \text{resto: 2}$$

$$333_7 = 210_9$$

# Codifica binaria e esadecimale

- Abbiamo detto che i componenti digitali operano in codice binario.
- Tuttavia la rappresentazione in binario non è molto human-friendly perché genera codici piuttosto lunghi
  - $599 = 1001010111_2$
- Si potrebbe pensare di usare la nostra base naturale (10). Questo comporta usare il precedente algoritmo per codifica/decodifica
  - Non proprio agevole
  - Non proprio velocissimo
  - Il problema è che 10 non è una potenza di 2

# Codifica binaria e esadecimale

- La codifica/decodifica in una base  $m$  potenza di 2 permette invece di usare qualche trucchetto che facilita la codifica e decodifica.
- le cifre utilizzate nel sistema esadecimale sono '0',..., '9', 'a',..., 'f'.
- Inoltre, poiché  $16=2^4$  è possibile codificare ogni cifra del sistema esadecimale mediante 4 bit
- Viceversa 4 bit del sistema binario corrispondono ad una cifra esadecimale

# Codifica binaria e esadecimale

- Codifica delle cifre esadecimali in binario

$$0 \rightarrow 0000 \qquad \qquad 8 \rightarrow 1000$$

$$1 \rightarrow 0001 \qquad \qquad 9 \rightarrow 1001$$

$$2 \rightarrow 0010 \qquad \qquad a \rightarrow 1010$$

$$3 \rightarrow 0011 \qquad \qquad b \rightarrow 1011$$

$$4 \rightarrow 0100 \qquad \qquad c \rightarrow 1100$$

$$5 \rightarrow 0101 \qquad \qquad d \rightarrow 1101$$

$$6 \rightarrow 0110 \qquad \qquad e \rightarrow 1110$$

$$7 \rightarrow 0111 \qquad \qquad f \rightarrow 1111$$

- Notate che questa codifica corrisponde alla codifica dei rispettivi numeri con possibili 0 non significativi

$$0111_2 = 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 7$$

# Da esadecimale a binario

- Dato un numerale esadecimale per codificarlo in binario è sufficiente giustapporre le codifiche delle singole cifre (eliminando eventualmente zeri non significativi)
- Esempi:
  - $4c9f_{16} \rightarrow 0100\ 1100\ 1001\ 1111 \rightarrow 10011001001111_2$
  - $b2a_{16} \rightarrow 1011\ 0010\ 1010 \rightarrow 101100101010_2$
  - $157_{16} \rightarrow 0001\ 0101\ 0111 \rightarrow 10101011_2$
- Nota bene che  $157_{16} = 1 \cdot 16^2 + 5 \cdot 16^1 + 7 \cdot 16^0 = 343$

# Da binario a esadecimale

- Per il passaggio di base da binario a esadecimale si effettua la codifica inversa avendo cura di aggiungere eventuali zeri non significativi in modo che la lunghezza del numerale in binario sia multipla di 4
- Esempi:
  - $10_2 \rightarrow 0010 \rightarrow 2_{16}$
  - $10011_2 \rightarrow 0001\ 0011 \rightarrow 13_{16}$
  - $111110010101100_2 \rightarrow 1111\ 1001\ 0101\ 1100 \rightarrow f95c_{16}$
- Nota: comunemente un numerale esadecimale viene indicato con il prefisso 0x
  - $f95c_{16} \rightarrow 0xf95c$

# Rappresentazione registri

- Un computer le grandezze numeriche sono elaborate mediante sequenze di simboli di lunghezza fissa dette parole
- Poichè una cifra esadecimale codifica 4 bit:
  - 1 byte (8 bit) → 2 cifre esadecimali
  - 4 byte (32 bit) → 8 cifre esadecimali
  - 8 byte (64 bit) → 16 cifre esadecimali

# Esercizi

- Convertire da base 2 a base 10:
  - 0110011
  - 10101100
  - 1100110011
- Convertire in base 10 i seguenti numeri:
  - $102210_3$
  - $431204_5$
  - $5036_7$
  - $198A1_{12}$

# Esercizi

- Convertire da base 10 alla base indicata i seguenti numeri:
  - 7562 base 8
  - 1938 base 16
  - 205 base 16
  - 175 base 2
- In un registro a 32 bit è memorizzato il valore 0xF3A7C2A4. Esprimere il contenuto del registro in base 2
- Convertire da base 2 a base 16:
  - 10010
  - 11010101
  - 10010011
- Scrivere in babilonese il numero 4000

# Numeri con parole di lunghezza fissa

- I registri dei moderni calcolatori sono tipicamente parole di 32 o 64 bit
- Con una parola di lunghezza fissa sono rappresentabili un numero finito di naturali.

|           |           |  |  |  |  |  |  |  |       |       |
|-----------|-----------|--|--|--|--|--|--|--|-------|-------|
| $2^{m-1}$ | $2^{m-2}$ |  |  |  |  |  |  |  | $2^1$ | $2^0$ |
|-----------|-----------|--|--|--|--|--|--|--|-------|-------|

- Nel caso di parole a 64 bit sono rappresentabili i numeri da 0 a  $2^{64}-1$
- Chiaramente, poiché ogni numero deve essere rappresentato dallo stesso numero di cifre, occorre ricorrere necessariamente a zeri non significativi

# Numeri con parole di lunghezza fissa

- Supponiamo di avere una macchina che opera con parole di 16 bit (come l'Atari St di quando ero piccolo)
  - il numero 9 sarà quindi rappresentato come: 0000 0000 0000 1001
- Operazioni come l'addizione o la moltiplicazione possono produrre numeri troppo grandi per essere rappresentati con un lunghezza fissata. In questo caso parleremo di trabocco o *overflow*
- Torniamo alla nostra macchina a 16 bit e supponiamo di voler elevare 1024 al quadrato. Questo produrrà un trabocco, infatti:

$$1024^2 = 1024 \cdot 1024 = 2^{10} \cdot 2^{10} = 2^{20} > 2^{16} - 1$$

# Somma in binario

L'operazione di somma in binario è algoritmicamente analoga a quella decimale con la differenza che il riporto si ha quando si eccede 1 (invece che 9)

$$\begin{array}{r}
 & 11 \\
 & 4277 \\
 + & 5499 \\
 \hline
 & 9776
 \end{array}$$

(a)

$$\begin{array}{r}
 & 11 \\
 & 1011 \\
 + & 0011 \\
 \hline
 & 1110
 \end{array}$$

(b)

Overflow: al termine della somma ho un riporto di 1

$$\begin{array}{r}
 111 \\
 1101 \\
 + 0101 \\
 \hline
 10010
 \end{array}$$

# Moltiplicazione in binario

Anche la moltiplicazione in binario è analoga a quella decimale

$$\begin{array}{r} 0101 \times \\ 0011 = \\ \hline & 0101 \\ & 0101= \\ & 0000== \\ & 0000==== \\ \hline & 0001111 \end{array}$$

# Rappresentazione di interi

- Occupiamoci ora della rappresentazione di numeri interi ..., -2,-1,0,1,2,... mediante parole di lunghezza fissata
- Chiaramente dobbiamo codificarne non solo il valore assoluto ma anche il segno
- Le principali rappresentazioni di numeri interi sono:
  - Rappresentazione con segno
  - Complemento all'intervallo (complemento a due)

# Rappresentazione con segno

- Si supponga che le parole siano di lunghezza  $m$  e la base sia  $b$
- In tale rappresentazione la cifra più significativa di una parola rappresenta il segno. Per convenzione 0 rappresenta il segno più, 1 rappresenta il segno meno
- Le rimanenti  $m-1$  cifre sono usate per rappresentare il valore assoluto di un intero

|     |           |  |  |  |  |  |  |       |       |
|-----|-----------|--|--|--|--|--|--|-------|-------|
| +/- | $2^{m-2}$ |  |  |  |  |  |  | $2^1$ | $2^0$ |
|-----|-----------|--|--|--|--|--|--|-------|-------|

- Ad esempio si considerino parole binarie di lunghezza 4
  - 0100  $\rightarrow$  +100  $\rightarrow$  4
  - 1011  $\rightarrow$  -011  $\rightarrow$  -3
  - Il più grande numero rappresentabile è 0111 (ovvero 7)
  - Il più piccolo numero rappresentabile è 1111 (ovvero -7)
  - Lo zero ha due possibili rappresentazioni: 0000 e 1000

# Rappresentazione con segno

- Poiché  $m-1$  cifre sono utilizzate per rappresentare il valore assoluto, il range di numeri codificabili è

$$-(b^{m-1} - 1), \dots, 0, \dots, b^{m-1} - 1$$

- Svantaggio: nelle operazioni di addizione o sottrazione occorre controllare il segno e i valori assoluti dei due operandi per determinare il segno del risultato.
  - 0010 + 0001
    - sono entrambi positivi quindi il segno sarà positivo → 0011
  - 0101 + 1010
    - il primo è positivo mentre il secondo è negativo, il valore assoluto del primo è maggiore del valore assoluto del secondo quindi il segno sarà positivo → 0011
  - 1100+0011
    - il primo è negativo mentre il secondo è positivo, il valore assoluto del primo è maggiore di quello del secondo, quindi il segno sarà negativo → 1001
  - 1011-1010
    - Entrambi sono negativi ma il valore assoluto del secondo è minore di quello del primo. Quindi occorre sottrarre 010 da 011 cambiando il bit del segno → 1001

# Complemento a 2

- La rappresentazione è analoga a quella unsigned con la differenza che il bit più significativo corrisponde a  $-2^{m-1}$  invece che  $2^{m-1}$

|            |           |  |  |  |  |  |  |  |       |       |
|------------|-----------|--|--|--|--|--|--|--|-------|-------|
| $-2^{m-1}$ | $2^{m-2}$ |  |  |  |  |  |  |  | $2^1$ | $2^0$ |
|------------|-----------|--|--|--|--|--|--|--|-------|-------|

- Consideriamo ad esempio la parola di 8 bit 

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |
|---|---|---|---|---|---|---|---|

 In complemento a due rappresenta il numero:

|      |    |    |     |    |    |    |    |        |
|------|----|----|-----|----|----|----|----|--------|
| -128 | +0 | +0 | +16 | +8 | +4 | +0 | +0 | = -100 |
|------|----|----|-----|----|----|----|----|--------|

# Complemento a 2

- Lo zero ha una unica rappresentazione

|   |   |  |  |  |  |  |  |   |   |
|---|---|--|--|--|--|--|--|---|---|
| 0 | 0 |  |  |  |  |  |  | 0 | 0 |
|---|---|--|--|--|--|--|--|---|---|

- Minimo valore rappresentabile:  $-2^{m-1} \rightarrow$

|   |   |   |  |  |  |  |   |   |   |
|---|---|---|--|--|--|--|---|---|---|
| 1 | 0 | 0 |  |  |  |  | 0 | 0 | 0 |
|---|---|---|--|--|--|--|---|---|---|

- Massimo valore rappresentabile:  $2^{m-1} - 1 \rightarrow$

|   |   |   |  |  |  |  |   |   |   |
|---|---|---|--|--|--|--|---|---|---|
| 0 | 1 | 1 |  |  |  |  | 1 | 1 | 1 |
|---|---|---|--|--|--|--|---|---|---|

- Il range di rappresentazione è  $[-2^{m-1}, 2^{m-1} - 1]$

- Essendo  $-2^{m-1}$  in valore assoluto il «peso» più grande, se un numero inizia con 1 allora è negativo altrimenti è positivo

# Complemento a 2

- La somma avviene come per i numeri unsigned. Consideriamo ad esempio parole di 4 bit
  - $-2+1 = 1110 + 0001 = 1111 = -1$
  - $-7+7 = 1001 + 0111 = 0000$  (con riporto 1) = 0
- Per complementare un numero occorre invertire ogni cifra e sommare 1
  - Rappresentare il numero -2 e -7 con 4 bit
  - $2=0010 \rightarrow -2=1101 + 0001 = 1110$
  - $7=0111 \rightarrow -7=1000 + 0001 = 1001$
  - Attenzione! questo non vale per  $-2^{m-1}$  il cui complemento non è rappresentabile con m bit (i numeri positivi arrivano a  $2^{m-1} - 1$ ):
  - $-2^3 = 1000 \rightarrow 0111 + 0001 = 1000$

# Overflow nel complemento a 2

- Sommare un numero negativo e uno positivo non genera overflow
- L'esempio  $-7+7$  mostra come l'overflow *non avviene come per gli unsigned* quando ho riporto finale di 1
- L'overflow avviene quando entrambi gli operandi sono negativi (primo bit=1) o entrambi positivi (primo bit=0) è il risultato ha segno opposto
  - $4+5=0100+0101 =1001 =\textcolor{red}{-7}$
- Per estendere un numero in una rappresentazione con più bit basta riprodurre a sinistra il bit più significativo
  - $5=0101 \rightarrow 00000101$
  - $-4=1100 \rightarrow 11111100$

# Esercizi

- Fornire la rappresentazione binaria in complemento a 2 ad 8 bit del numero -15
- Fornire la rappresentazione binaria in complemento a 2 ad 8 bit del numero -109
- Eseguire la somma dei numeri 5 e -32 espressa in completamento a 2 ad 8 bit
- Convertire in esadecimale il numero -53248 espresso in completamento a 2 ad 16 bit

# Binary Coded Decimal

- L'elettronica degli elaboratori è binaria mentre la mente umana è abituata a ragionare in decimale.
- I codici Binary Coded Decimal hanno lo scopo di fornire una naturale rappresentazione binaria del sistema numerico decimale.
- Essendo 10 i simboli da codificare ('0',..., '9') avremo bisogno di 4 bit.
- Notate che con 4 bit consentono di avere  $2^4=16$  combinazioni che in binario puro corrispondono ai numeri 0,1,2,...9,10,...,15
- Poiché le combinazioni da 10 a 15 non si usano, la codifica BCD è *ridondante*

# Binary Coded Decimal

| Cifra<br>decimale | Cifra<br>BCD |
|-------------------|--------------|
| 0                 | 0000         |
| 1                 | 0001         |
| 2                 | 0010         |
| 3                 | 0011         |
| 4                 | 0100         |
| 5                 | 0101         |
| 6                 | 0110         |
| 7                 | 0111         |
| 8                 | 1000         |
| 9                 | 1001         |

i codici 1010, 1011, 1100, 1101, 1110, 1111 non sono utilizzati (codifica ridondante)

Numeri decimali a più cifre di codificano giustapponendo le codifiche cifra per cifra

Esempio:

$$23 = 0010\ 0011_{BCD}$$

Nota che

$$23 = 00100011_{BCD} \neq 00100011_2 = 35$$

Anche le somme differiscono:

$$1000_{BCD} + 0011_{BCD} = 8 + 3 = 11 = 00010001_{BCD}$$

e non 1011 (che è un codice non utilizzato)

# Binary Coded Decimal

- Naturalmente, la codifica BCD viene usata solo in funzione di interfaccia per rendere comprensibile ad operatori umani i risultati di una elaborazione numerica binaria. La codifica BCD del numero è, infatti, una operazione propedeutica alla sua visualizzazione su un display numerico decimale (es. display a sette segmenti). I quattro bit di ciascuna cifra codificata BCD vengono inviati ad un circuito di decodifica che provvederà ad attivare i segmenti della cifra corrispondente.
- E' opportuno sottolineare che, mentre la codifica binaria è il primo passo per approdare ad un sistema numerico che porta poi all'aritmetica binaria, non esiste una aritmetica BCD.

# Numeri con virgola

- Vediamo come rappresentare (approssimazioni di numeri) reali.
- Consideriamo un numero con virgola nella base naturale 10

$$c_{m-1}c_{m-2}\cdots c_0, c_{-1}\cdots c_{-k} = c_{m-1} \cdot 10^{m-1} + c_{m-2} \cdot 10^{m-2} + \cdots + c_0 \cdot 10^0 + c_{-1} \cdot 10^{-1} + \cdots + c_{-k} \cdot 10^{-k}$$

- Per una generica base  $b$  abbiamo la generalizzazione

$$\sum_{i=-k}^{h-1} c_i \cdot b^i$$

# Cambiamenti di base con virgola

- Per il cambiamento di un numero  $x$  da una base  $a$  ad una  $b$  si procede separatamente per la parte intera e per quella frazionaria
- Per la parte intera l'algoritmo chiaramente è quello visto in precedenza
- Per la parte frazionaria in procedimento è l'inverso:
  - si moltiplica la parte frazionaria di  $x$  per  $b$  (entrambi codificati in base  $a$ ). La parte intera del risultato sarà un numero da 0 a  $b-1$  che, convertito in base  $b$ , costituisce la prima cifra frazionaria di  $x$  in base  $b$ .
  - La parte frazionaria del risultato  $f$  si moltiplica ancora per  $b$  e la nuova parte intera, convertita in base  $b$ , costituisce la seconda cifra frazionaria.
  - Si procede iterativamente finché la parte frazionaria del risultato è zero o si è raggiunta la precisione desiderata

# Cambiamenti di base con virgola

convertire in binario 0,625

$$0,625 \cdot 2 = 1,250 \quad i=1 \quad f=0,250 \quad c_{-1}=1$$

$$0,250 \cdot 2 = 0,500 \quad i=0 \quad f=0,500 \quad c_{-2}=0$$

$$0,500 \cdot 2 = 1,000 \quad i=1 \quad f=0,000 \quad c_{-3}=1$$

$$0,625 = 0,101_2$$

Convertire  $0,65_8$  in base 7 fino alla 3 cifra significativa

- Convertiamo prima in decimale
  - $6 \times 8^{-1} + 5 \times 8^{-2} = 6 \times 0,125 + 5 \times 0,015625 = 0,75 + 0,078125 = 0,828125$
- Convertiamo  $0,828125$  in base 7:  $0,554_7$ ,
  - $0,828125 \times 7 = 5,796875 \quad 5$
  - $0,796875 \times 7 = 5,578125 \quad 5$
  - $0,578125 \times 7 = 4,046875 \quad 4$

# Rappresentazione in virgola fissa

- Ci occupiamo ora di rappresentare numeri positivi frazionari con parole (binarie) di lunghezza fissata  $m$
- Nella rappresentazione in virgola fissa si suddividono gli  $m$  bit in due sottoparole
  - i primi  $h$  bit (con  $h < m$ ) sono dedicati alla codifica della parte intera
  - i rimanenti  $k = m - h$  bit rappresentano la parte frazionaria
- Supponiamo di far uso di parole a 32 bit e di dedicare 20 bit per la parte intera e 12 per la parte frazionaria:
  - Massimo intero codificabile:  $2^{20} - 1$
  - Con 12 bit per la parte frazionaria si codificano circa 3 cifre decimali (ricordate  $\log_2 10 = 3,32$ )

# Esercizi

- Fornire la rappresentazione binaria in virgola fissa del numero 7.25 con 4 bit per la parte intera e 4 bit di parte frazionaria
- Riportare in codice esadecimale la rappresentazione binaria in virgola fissa del numero 2.33 con 4 bit per la parte intera e 4 bit di parte frazionaria, trascurando l'eventuale resto
- Fornire la rappresentazione binaria in virgola fissa del numero 55.4121 con 8 bit per la parte intera e 4 bit di parte frazionaria, trascurando l'eventuale resto
- Eseguire la somma  $12.25+5.5$  nella rappresentazione binaria in virgola fissa con 5 bit per la parte intera e 3 per quella frazionaria
- Eseguire la somma  $9.875+10.5$  nella rappresentazione binaria in virgola fissa con 5 bit per la parte intera e 3 per quella frazionaria

# Rappresentazione in virgola mobile

- Nella rappresentazione in virgola fissa disponendo di parole a 64 bit e supponendo di dedicarle tutte per la rappresentazione della parte frazionaria riusciremmo a codificare circa 18 cifre decimali
- Alcune applicazioni scientifiche operano con valori ancora più piccoli, altre invece con valori molto grandi
- In generale, fissare a priori la lunghezza della parte intera  $h$  e di quella frazionaria  $k$  costituisce una scelta rigida
- Per ovviare a queste difficoltà è stata introdotta una rappresentazione detta in virgola mobile

# Rappresentazione in virgola mobile

Floating-Point: non-uniform distribution (variable precision)



IQ Fractions: uniform distribution (same precision everywhere)



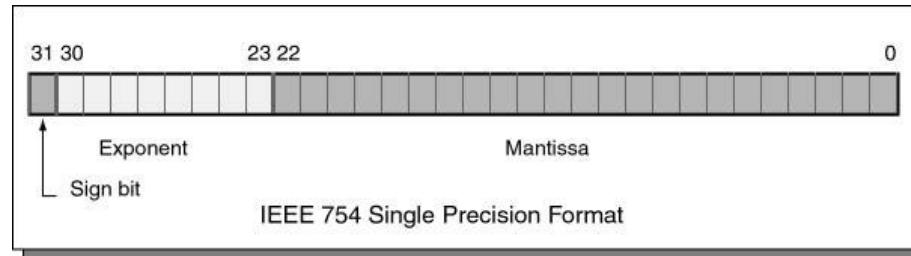
- ◆ Both floating-point and IQ formats have  $2^{32}$  possible values on the number line
- ◆ It's how each distributes these values that differs

# Rappresentazione in virgola mobile

- Questa sfrutta la rappresentazione di un numero in una data base b:  
$$x = (-1)^s m b^e$$
  - $s$  determina il segno: 0 positivo, 1 negativo
  - $m$  è detta mantissa
  - $e$  è detto esponente
- Un numero reale è quindi rappresentato da una triple  $(s,m,e)$ . Notate però che vi sarebbero infiniti modi di rappresentare  $x$ . Ad esempio, per 34,67 in base 10:
  - $3467 \times 10^{-2}$
  - $3,467 \times 10^3$
  - $0,3467 \times 10^4$
- Useremo la rappresentazione scientifica in cui  $b > m \geq 1$
- In binario questo vuol dire che la mantissa è sempre del tipo  $1,xyz$ 
  - Chiaramente nella rappresentazione della mantissa non sprecherò memoria per rappresentare il primo bit “1” e lo considero sottointeso

# Standard IEEE 754

- Una rappresentazione largamente adottata è quella dell'Institute of Electrical and Electronical Engineering (IEEE)
- Prevede 4 diversi formati per calcoli in singola o doppia precisione di tipo semplice o esteso (raramente usato)
- Descriveremo i tipi semplici
  - Singola precisione: 32 bit totali, 1 per il segno, 23 per la mantissa e 8 per l'esponente



- Doppia precisione: 64 bit totali, 1 per il segno, 52 per la mantissa e 11 per l'esponente

# Dettagli dello Standard IEEE

- Riguardo all'esponente, il complemento a 2 inverte l'ordine fra positivi e negativi (i numerali associati ai negativi sono *maggiori* di quelli associati ai positivi)
- Per ovviare a questo difetto si usa nello standard IEEE la “rappresentazione polarizzata”
- Dati  $k$  bit assegnati all'esponente, il più grande esponente rappresentabile (in complemento a 2) è  $P = 2^{k-1} - 1$
- In rappresentazione polarizzata, un valore  $e$  (compreso tra 0 e  $2^k - 1$ ) codifica l'esponente  $e'$   $= e - P$
- Ad esempio sia  $k=3$  ( $P = 2^2 - 1 = 3$ ):

$$\begin{array}{lll} e = 0 [000] \rightarrow e' = -3 & e = 1 [001] \rightarrow e' = -2 & e = 2 [010] \rightarrow e' = -1 \\ e = 4 [100] \rightarrow e' = 1 & e = 5 [101] \rightarrow e' = 2 & e = 6 [110] \rightarrow e' = 3 \\ & & e = 7 [111] \rightarrow e' = 4 \end{array}$$

- Quindi ora gli esponenti sono codificati in ordine crescente da  $-3$  ( $000_2$ ) a  $4$  ( $111_2$ )

# Dettagli dello Standard IEEE

- Per la mantissa si adotta la rappresentazione scientifica  $1,xyz\dots$
- Riassumendo ad  $(s, m, e)$  è associato il numero

$$x = (-1)^s \cdot 1,m \cdot 2^{e-P}$$

- IEEE 754 precisione singola: 8 bit per l'esponente
  - $P = 2^7 - 1 = 127$
- IEEE 754 precisione doppia: 11 bit per l'esponente
  - $P = 2^{10} - 1 = 1023$
- Problema con questa rappresentazione non riesco a rappresentare lo 0. Per questo lo standard IEEE considera delle eccezioni.

# Dettagli dello Standard IEEE

precisione singola

|     | e=0 ( $0000000_2$ )                 | e=1,...,254                          | e=255 ( $1111111_2$ ) |
|-----|-------------------------------------|--------------------------------------|-----------------------|
| m=0 | $(-1)^s \times 0$                   | $(-1)^s \times 1,0 \times 2^{e-127}$ | $(-1)^s \infty$       |
| m≠0 | $(-1)^s \times 0,m \times 2^{-126}$ | $(-1)^s \times 1,m \times 2^{e-127}$ | NaN (indefinito)      |

Per  $m = 0$  ed  $e = 0$ , si hanno due rappresentazioni dello 0, a seconda che  $s$  sia 1 (segno negativo) o 0 (segno positivo).

Le combinazioni che corrispondono ad  $m = 0$  ed  $e = 255$  sono la rappresentazione di  $\pm \infty$ .

Per  $m \neq 0$  ed  $e = 0$  è prevista una rappresentazione per numeri molto vicini allo 0.

Ricordate che  $m$  non è vincolato a iniziare con 1. Quindi se usassimo la rappresentazione usuale avrei, assumendo  $s=0$ , otterrei numeri maggiori di  $2^{-127}$ :  $1,m \times 2^{-127} > 2^{-127}$

Invece per  $m=00\dots01$ :  $0,0\dots01 \times 2^{-126} = 2^{-23} \times 2^{-126} = 2^{-149}$

# Esempi standard IEEE

- Il numero decimale 1021 è rappresentato dalla tripla (s, m, e):  
(0; 111 1111 0100 0000 0000 0000; 1000 1000)

s= 0 perché il numero è positivo.

La rappresentazione binaria di 1021 è:

$$1\ 111\ 1111\ 01 = 1,111\ 1111\ 01 \times 2^9$$

$$m = 111\ 1111\ 0100\ 0000\ 0000\ 0000$$

Avendo 8 bit a disposizione per l'esponente,  $P = 2^{8-1} - 1 = 2^7 - 1 = 127$ .

L'esponente  $e$  si ricava invertendo la relazione di definizione della costante di polarizzazione:

$$e = e' + P = 9 + 127 = 136$$

che, convertito in binario, da 1000 1000

# Esercizi

- Rappresentare nel formato IEEE 754
  - 1.25
  - 10
  - -0.625
  - 1007
  - 3.875

potrete verificare la correttezza degli esercizi svolti tramite  
<https://www.h-schmidt.net/FloatConverter/IEEE754.html>

- -0,625
- Convertire in binario

$$0,625 \times 2 = 1,25 \quad 1$$

$$0,25 \times 2 = 0,5 \quad 0$$

$$0,5 \times 2 = 1,0 \quad 1$$

$$0,625 = 0,101_2$$

- Normalizzare nel formato scientifico  $1,xyz\dots$

$$0,101 = 1,01 \times 2^{-1}$$

- Quindi ottengo la rappresentazione binaria in virgola mobile

$$(-1)^1 \ 1,01 \ 2^{-1}$$

- Conversione nello standard IEEE 754

- s=1
- m= 010 0000 0000 0000 0000 0000
- e = 127 - 1 = 126 = 0111 1110

# Operazioni in virgola mobile

- La moltiplicazione fra due numeri  $n_1$  ed  $n_2$  rappresentati in virgola mobile dalle triple  $(s_1, m_1, e_1)$  ed  $(s_2, m_2, e_2)$  ha per risultato il numero rappresentato dalla tripla:  $(s, e, m)$  in cui:
  - $s = 0$  se  $s_1 = s_2$  oppure:  $s = 1$  se  $s_1 \neq s_2$
  - $e = e_1 + e_2$
  - $m = m_1 \times m_2$
  - Dopo l'operazione di solito è necessaria la normalizzazione del risultato
- La divisione si effettua con regole analoghe.
- L'addizione e la sottrazione sono più complesse perché prima di effettuarle bisogna rendere uguali gli esponenti.
  - Durante questa operazione se i numeri sono uno molto grande ed uno molto piccolo, per effetto dello scorrimento delle mantisse per pareggiare gli esponenti, si possono perdere cifre significative.

# Perdita di cifre significative

- Vediamo con un esempio perché si possono perdere cifre significative effettuando una somma. Per semplicità, considereremo una coppia di numeri decimali espressi attraverso una mantissa di 4 cifre ed un esponente di una sola cifra:

$$n_1 = 1,3435 \times 10^3 \text{ ed } n_2 = 1,9970 \times 10^5.$$

- Per effettuare la somma si può portare l'esponente di  $n_1$  da 3 a 5. Siccome ciò equivale a moltiplicare di fattore 100, per mantenere il valore costante occorre contemporaneamente dividere per 100 la mantissa:

$$1,3435 \times 10^3 = 0,013435 \times 10^5$$

- Poiché le cifre della mantissa sono 4, occorre effettuare un arrotondamento. Per difetto otterremmo 0,0134.
- La somma dei due numeri risulta:  $2,0104 \times 10^5$  (invece di  $2,010435 \times 10^5$ )

# Codifica caratteri alfa-numerici

- I calcolatori, nonostante il nome italiano (in francese si chiamano *ordinateur*) sono spesso utilizzati per manipolare informazioni non numeriche.
- Si parla di caratteri "alfanumerici" per sottolineare che in un testo sono presenti:
  - caratteri alfabetici (a,b,c,d,...)
  - caratteri numerici (0,...,9)
  - segni di punteggiatura (!,?,...)
  - simboli particolari vario tipo (€, &, @, ...)
- I processori moderni non hanno istruzioni specifiche per testi. Quindi, si usano codifiche da testo a numeri interi
- Un testo è una sequenza di caratteri. I codici associano un numero intero ad ogni carattere.

# Codifiche di caratteri

**1968 ASCII.**

Codice a 7 bit: 95 caratteri stampabili e 33 di controllo.

**1980 Extended ASCII.**

Varie estensioni a 8 bit, con simboli grafici e lettere accentate.

**1991 Unicode.**

Codice a 21 bit (1 milione di simboli). Attualmente (v. 9.0) definiti circa 128.000 caratteri! Viene ulteriormente codificato in **UTF-8**.

**1992 UTF-8.**

Codifica di Unicode a lunghezza variabile (da 1 a 4 byte). Retro-compatibile con ASCII. UTF-8 è la codifica consigliata per XML e HTML.

# ASCII TABLE

| Decimal | Hex | Char                   | Decimal | Hex | Char    | Decimal | Hex | Char | Decimal | Hex | Char  |
|---------|-----|------------------------|---------|-----|---------|---------|-----|------|---------|-----|-------|
| 0       | 0   | [NULL]                 | 32      | 20  | [SPACE] | 64      | 40  | @    | 96      | 60  |       |
| 1       | 1   | [START OF HEADING]     | 33      | 21  | !       | 65      | 41  | A    | 97      | 61  | a     |
| 2       | 2   | [START OF TEXT]        | 34      | 22  | "       | 66      | 42  | B    | 98      | 62  | b     |
| 3       | 3   | [END OF TEXT]          | 35      | 23  | #       | 67      | 43  | C    | 99      | 63  | c     |
| 4       | 4   | [END OF TRANSMISSION]  | 36      | 24  | \$      | 68      | 44  | D    | 100     | 64  | d     |
| 5       | 5   | [ENQUIRY]              | 37      | 25  | %       | 69      | 45  | E    | 101     | 65  | e     |
| 6       | 6   | [ACKNOWLEDGE]          | 38      | 26  | &       |         |     | F    |         |     | f     |
| 7       | 7   | [BELL]                 | 39      | 27  | '       |         |     | G    |         |     | g     |
| 8       | 8   | [BACKSPACE]            | 40      | 28  | (       | 72      | 48  | H    | 104     | 68  | h     |
| 9       | 9   | [HORIZONTAL TAB]       | 41      | 29  | )       | 73      | 49  | I    | 105     | 69  | i     |
| 10      | A   | [LINE FEED]            | 42      | 2A  | *       | 74      | 4A  | J    | 106     | 6A  | j     |
| 11      | B   | [VERTICAL TAB]         | 43      | 2B  | +       | 75      | 4B  | K    | 107     | 6B  | k     |
| 12      | C   | [FORM FEED]            | 44      | 2C  | ,       | 76      | 4C  | L    | 108     | 6C  | l     |
| 13      | D   | [CARRIAGE RETURN]      | 45      | 2D  | -       | 77      | 4D  | M    | 109     | 6D  | m     |
| 14      | E   | [SHIFT OUT]            | 46      | 2E  | .       | 78      | 4E  | N    | 110     | 6E  | n     |
| 15      | F   | [SHIFT IN]             | 47      | 2F  | /       | 79      | 4F  | O    | 111     | 6F  | o     |
| 16      | 10  | [DATA LINK ESCAPE]     | 48      | 30  | 0       | 80      | 50  | P    | 112     | 70  | p     |
| 17      | 11  | [DEVICE CONTROL 1]     | 49      | 31  | 1       | 81      | 51  | Q    | 113     | 71  | q     |
| 18      | 12  | [DEVICE CONTROL 2]     | 50      | 32  | 2       | 82      | 52  | R    | 114     | 72  | r     |
| 19      | 13  | [DEVICE CONTROL 3]     | 51      | 33  | 3       | 83      | 53  | S    | 115     | 73  | s     |
| 20      | 14  | [DEVICE CONTROL 4]     | 52      | 34  | 4       | 84      | 54  | T    | 116     | 74  | t     |
| 21      | 15  | [NEGATIVE ACKNOWLEDGE] |         |     | 5       | 85      | 55  | U    | 117     | 75  | u     |
| 22      | 16  | [SYNCHRONOUS IDLE]     |         |     | 6       | 86      | 56  | V    | 118     | 76  | v     |
| 23      | 17  | [ENG OF TRANS. BLOCK]  | 55      | 37  | 7       | 87      | 57  | W    | 119     | 77  | w     |
| 24      | 18  | [CANCEL]               | 56      | 38  | 8       | 88      | 58  | X    | 120     | 78  | x     |
| 25      | 19  | [END OF MEDIUM]        | 57      | 39  | 9       | 89      | 59  | Y    | 121     | 79  | y     |
| 26      | 1A  | [SUBSTITUTE]           | 58      | 3A  | :       | 90      | 5A  | Z    | 122     | 7A  | z     |
| 27      | 1B  | [ESCAPE]               | 59      | 3B  | ;       | 91      | 5B  | [    | 123     | 7B  | {     |
| 28      | 1C  | [FILE SEPARATOR]       | 60      | 3C  | <       | 92      | 5C  | \    | 124     | 7C  |       |
| 29      | 1D  | [GROUP SEPARATOR]      | 61      | 3D  | =       | 93      | 5D  | ]    | 125     | 7D  | }     |
| 30      | 1E  | [RECORD SEPARATOR]     | 62      | 3E  | >       | 94      | 5E  | ^    | 126     | 7E  | ~     |
| 31      | 1F  | [UNIT SEPARATOR]       | 63      | 3F  | ?       | 95      | 5F  | _    | 127     | 7F  | [DEL] |

"0": 011 0000

A: 100 0001

a: 110 0001

# Codice ASCII

- Ai primi 32 numerali sono assegnati caratteri di controllo
  - null indica la fine di una stringa
  - carriage return porta il cursore su una nuova riga (andata a capo)
  - horizontal tab è l'usuale carattere tab
- Altro tipo:
  - Bell dovrebbe far suonare un cicalino

# Alcuni caratteri Unicode

| <b>codice</b> | <b>carattere</b>                                                                    | <b>significato</b>                 |
|---------------|-------------------------------------------------------------------------------------|------------------------------------|
| U+0434        | Д                                                                                   | Lettera cirillica “de”             |
| U+6328        | ؑ                                                                                   | Lettera araba “ba”                 |
| U+0E10        | ກ                                                                                   | Lettera tailandese “tho than”      |
| U+1F63A       |    | “Smiling cat face with open mouth” |
| U+1F382       |  | Torta di compleanno                |

# Il principio di UTF-8

- UTF-8 codifica ciascun carattere Unicode con una sequenza lunga da 1 a 4 byte . Il primo byte di un carattere indica quanto è lunga la sequenza:

| Primo byte | Byte totali | Bit a disposizione del carattere |
|------------|-------------|----------------------------------|
| 0xxx xxxx  | 1           | 7                                |
| 110x xxxx  | 2           | 11                               |
| 1110 xxxx  | 3           | 16                               |
| 1111 0xxx  | 4           | 21                               |

- Tutti i byte successivi nella sequenza hanno il formato 10xx xxxx

# Esempio di UTF-8

- Consideriamo il simbolo dell'euro “€”, codice Unicode U+20AC
- E' un codice di 16 bit, quindi richiede 3 byte in UTF-8
- Vediamo come i 16 bit vengono distribuiti su 3 byte da UTF-8:

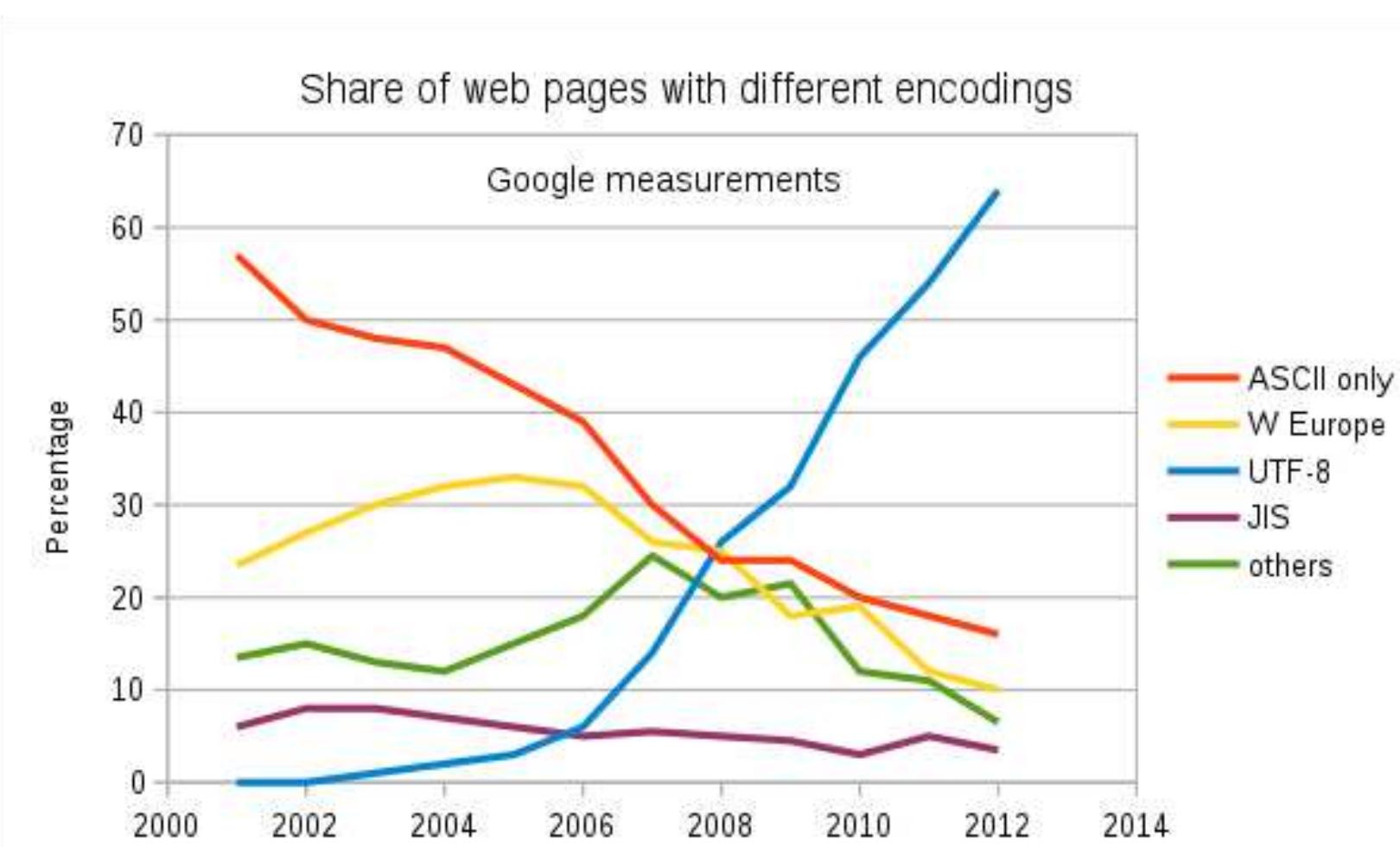
0x20AC = 0010 0000 1010 1100

- Codifica UTF-8:

1110 0010 | 1000 0010 | 1010 1100



# Il successo di UTF-8



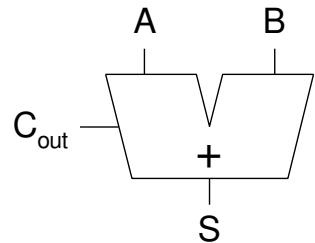
- Rappresentare il numero  $345,22_6$  in base 5 con una precisione di 3 cifre
- Quanti bit sono necessari e sufficienti per rappresentare il numero 2367 in rappresentazione binaria senza segno? E in complemento a due?
- Rappresentare nello standard IEEE 754 in singola precisione il numero 43,75
- Date parole di 8 bit, si esegua la somma in complemento a due di 56 e -27. Rappresentare il risultato a 8 bit in esadecimale
- Rappresentare il carattere Unicode U+235D in UTF-8

- 1022,143<sub>5</sub>
- 12 e 13
- 0 10000100 010111000000000000000000
- 1110 0010 1000 1101 1001 1101

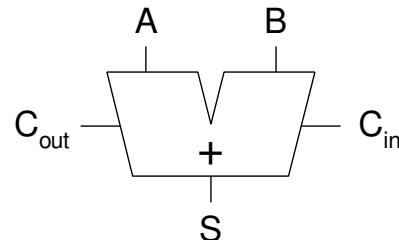
# Circuiti aritmetici e memorie

# 1-Bit Adders

Half  
Adder



Full  
Adder



| A | B | C <sub>out</sub> | S |
|---|---|------------------|---|
| 0 | 0 | 0                | 0 |
| 0 | 1 | 0                | 1 |
| 1 | 0 | 0                | 1 |
| 1 | 1 | 1                | 0 |

$$S = A \oplus B$$

$$C_{out} = AB$$

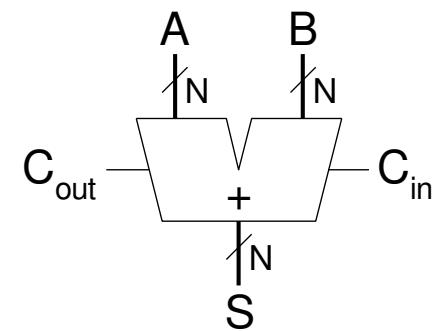
| C <sub>in</sub> | A | B | C <sub>out</sub> | S |
|-----------------|---|---|------------------|---|
| 0               | 0 | 0 | 0                | 0 |
| 0               | 0 | 1 | 0                | 1 |
| 0               | 1 | 0 | 0                | 1 |
| 0               | 1 | 1 | 1                | 0 |
| 1               | 0 | 0 | 0                | 1 |
| 1               | 0 | 1 | 1                | 0 |
| 1               | 1 | 0 | 1                | 0 |
| 1               | 1 | 1 | 1                | 1 |

$$S = A \oplus B \oplus C_{in}$$

$$C_{out} = AB + AC_{in} + BC_{in}$$

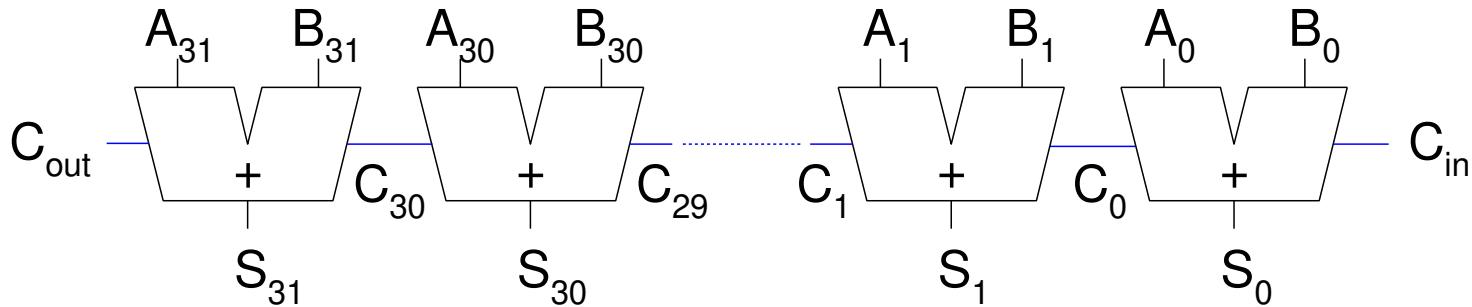
# Multibit Adders (CPAs)

- Tipologie carry propagate adders (CPAs):
  - Ripple-carry (lento)
  - Carry-lookahead (veloce)
- Carry-lookahead adders offrono prestazioni migliori ma richiedono più hardware



# Ripple-Carry Adder

- 1-bit adder concatenate insieme
- Il riporto si propaga lungo la catena
- Svantaggio: **lento**



# Ritardo del Ripple-Carry

$$t_{\text{ripple}} = N t_{FA}$$

$t_{FA}$  è il ritardo di un singolo 1-bit full adder

N è il numero di 1-bit full adders

# Carry-Lookahead Adder

Calcolare  $C_{out}$  di un blocco di  $k$ -bit usando le funzioni *generate* e *propagate*

| $C_{in}$ | $A_i$ | $B_i$ | $C_{out}$ |
|----------|-------|-------|-----------|
| 0        | 0     | 0     | 0         |
| 1        |       |       | 0         |
| 0        | 0     | 1     | 0         |
| 1        |       | 1     | 1         |
| 0        | 1     | 0     | 0         |
| 1        |       | 1     | 1         |
| 0        | 1     | 1     | 1         |
| 1        |       | 1     | 1         |

# Carry-Lookahead Adder

| $C_{in}$ | $A_i$ | $B_i$ | $C_{out}$ |
|----------|-------|-------|-----------|
| 0        | 0     | 0     | 0         |
| 1        |       |       | 0         |
| 0        | 0     | 1     | 0         |
| 1        |       |       | 1         |
| 0        | 1     | 0     | 0         |
| 1        |       |       | 1         |
| 0        | 1     | 1     | 1         |
| 1        |       |       | 1         |

# Carry-Lookahead Adder

- Il bit *i-esimo* produce un riporto (carry out) per generazione o per propagazione di un riporto del bit *(i-1)-esimo* (carry in)
- **Generate:** Il bit *i-esimo* genera un carry out se  $A_i$  e  $B_i$  sono entrambi 1.

$$G_i = A_i B_i$$

- **Propagate:** Il bit *i-esimo* propaga un carry al carry out se  $A_i$  o  $B_i$  sono 1.

$$P_i = A_i + B_i$$

- **Carry out:** il carry *i* ( $C_i$ ) è dato da:

$$C_i = A_i B_i + (A_i + B_i) C_{i-1} = G_i + P_i C_{i-1}$$

# Block Propagate and Generate

Ora dobbiamo estendere le funzioni Propagate and Generate a blocchi di  $k$ -bits, ovvero:

- calcolare se un blocco di  $k$ -bit ( $i, i+1, \dots, i+k-1$ ) propaga il carry out del blocco precedente (ovvero da  $C_{i-1}$  a  $C_{i+k-1}$ )
- calcolare se un blocco di  $k$ -bit ( $i, i+1, \dots, i+k$ ) genera un carry out (in  $C_{i+k-1}$ )

# Esempi

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |

# Esempi

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |   |

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |   |

.

# Esempi

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |   |

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |   |

# Block Propagate and Generate Signals

- Block propagate and generate per un blocco di 4-bit ( $P_{3:0}$  and  $G_{3:0}$ ):

$$P_{3:0} = P_3 P_2 P_1 P_0$$

$$\begin{aligned} G_{3:0} &= G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 = \\ &G_3 + P_3 (G_2 + P_2 (G_1 + P_1 G_0)) \end{aligned}$$

- In generale,

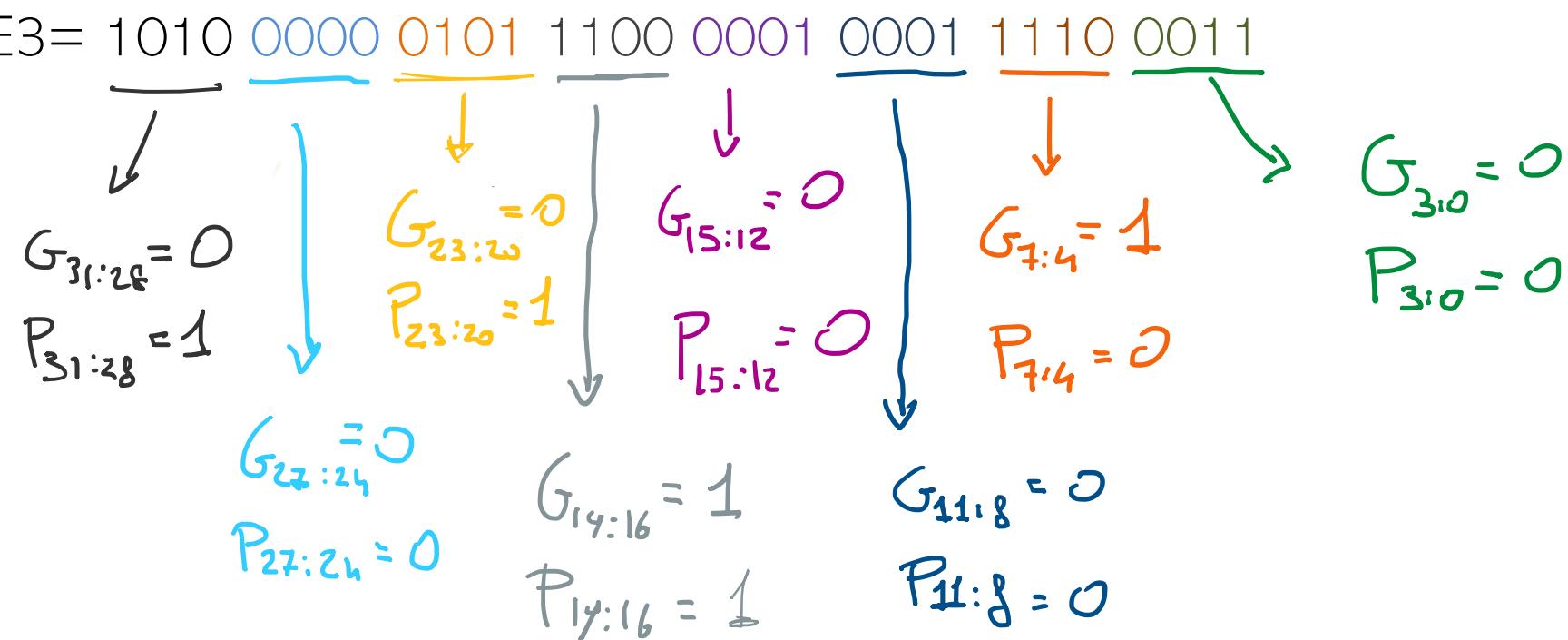
$$P_{i:j} = P_i P_{i-1} P_{i-2} \dots P_j$$

$$G_{i:j} = G_i + P_i (G_{i-1} + P_{i-1} (G_{i-2} + P_{i-2} (\dots G_j) \dots))$$

$$C_i = G_{i:j} + P_{i:j} C_{j-1}$$

# Esempio

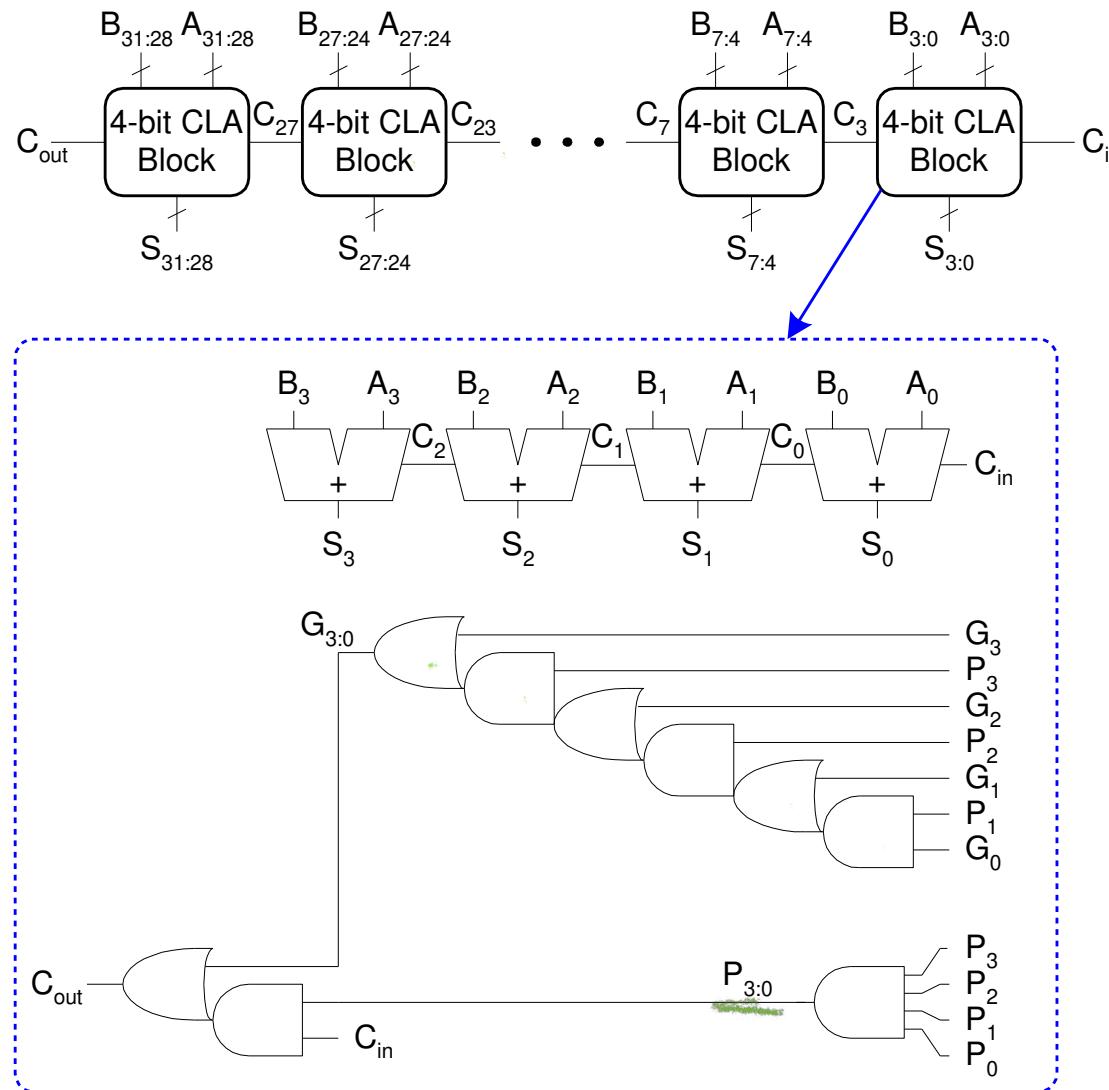
- Consideriamo le due parole a 32 bit
  - 0x52A703C1
  - 0xA05C11E3
- $0x52A703C1 = 0101\ 0010\ 1010\ 0111\ 0000\ 0011\ 1100\ 0001$
- $0xA05C11E3 = 1010\ \underline{0000}\ \underline{0101}\ \underline{1100}\ \underline{0001}\ \underline{0001}\ \underline{1110}\ \underline{0011}$



# Carry-Lookahead Addition

- Step 1: calcola tutti i  $G_i$  and  $P_i$
- Step 2: calcola  $G$  and  $P$  per blocchi di  $k$ -bit
- Step 3:  $C_{in}$  si propaga mediante la logica propagate/generate dei vari blocchi di  $k$ -bit

# 32-bit CLA with 4-bit Blocks



# Ritardo del Carry-Lookahead Adder

Per un  $N$ -bit CLA con blocchi di  $k$  bit:

$$t_{CLA} = t_{pg} + t_{pg\_block} + (N/k - 1)t_{AND\_OR} + kt_{FA}$$

- $t_{pg}$  : ritardo per generare  $P_i, G_i$
- $t_{pg\_block}$  : ritardo per generare  $P_{i:j}, G_{i:j}$
- $t_{AND\_OR}$  : ritardo delle porte AND/OR a monte della logica propagate/generate, questo ritardo si propaga da  $C_{in}$  a  $C_{out}$  che si propaga lungo i  $N/k - 1$  blocchi

Un  $N$ -bit carry-lookahead adder è generalmente più veloce di un ripple-carry adder per  $N > 16$

# Confronto

- 32-bit ripple-carry vs 32-bit CLA con blocchi di 4 bit
- 2-input gate delay = 100 ps; full adder delay = 300 ps

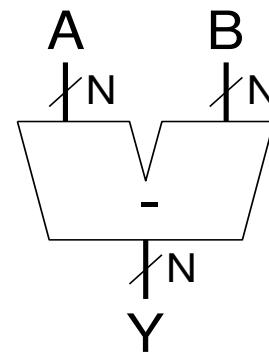
$$\begin{aligned} t_{\text{ripple}} &= Nt_{FA} = 32(300 \text{ ps}) \\ &= \mathbf{9.6 \text{ ns}} \end{aligned}$$

$$\begin{aligned} t_{\text{CLA}} &= t_{pg} + t_{pg\_block} + (N/k - 1)t_{\text{AND\_OR}} + kt_{FA} \\ &= [100 + 600 + (7)200 + 4(300)] \text{ ps} \\ &= \mathbf{3.3 \text{ ns}} \end{aligned}$$

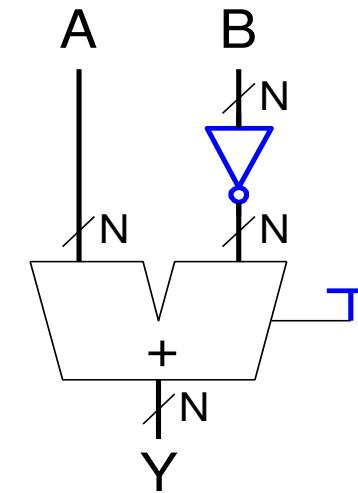
# Sottrattore

- Nella rappresentazione a complemento a 2 per complementare un numero occorre invertire ogni cifra e sommare 1
  - Es:  $2 = 0010_2 \rightarrow -2 = 1101_2 + 0001_2 = 1110_2$

## Symbol

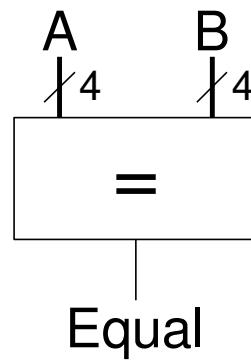


## Implementation

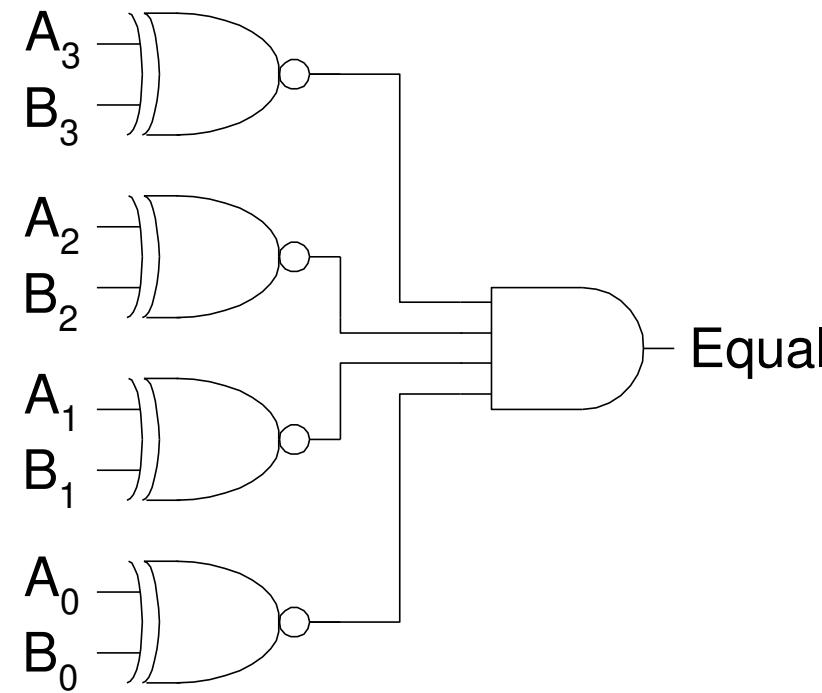


# Comparatore: uguaglianza

## Symbol

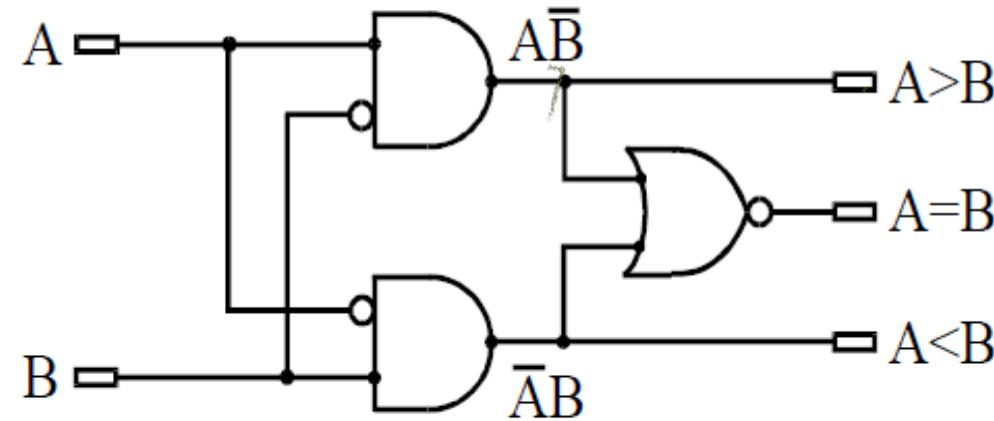


## Implementation

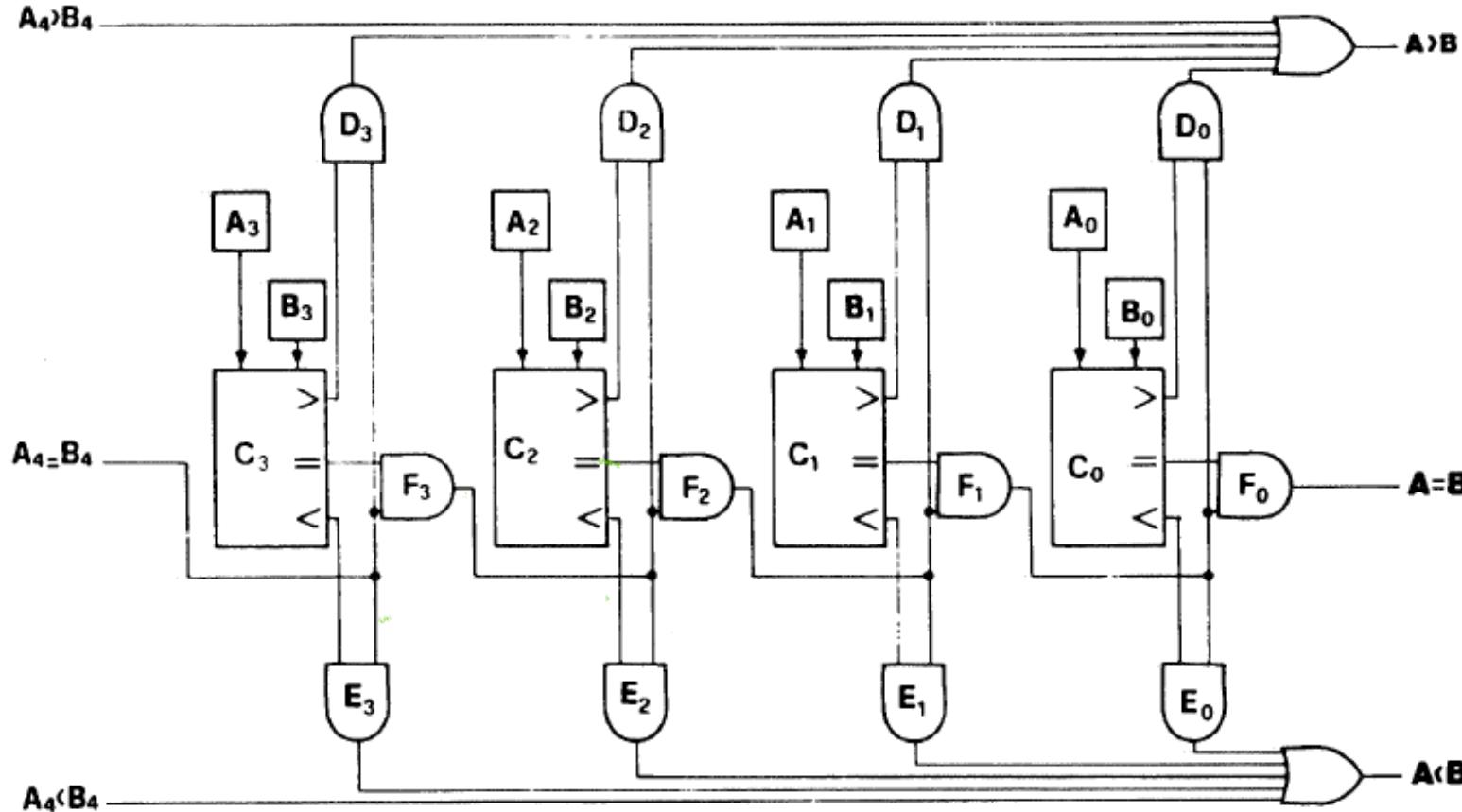


# Comparatore completo a un bit

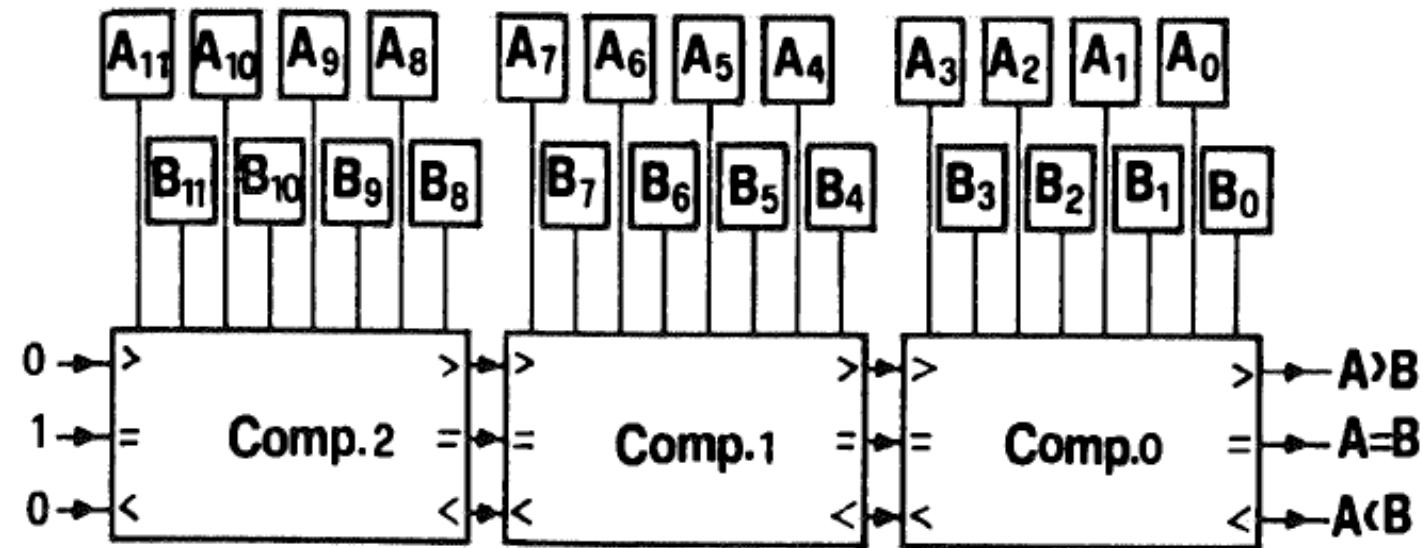
| A | B | $A > B$ | $A = B$ | $A < B$ |
|---|---|---------|---------|---------|
| 0 | 0 | 0       | 1       | 0       |
| 0 | 1 | 0       | 0       | 1       |
| 1 | 0 | 1       | 0       | 0       |
| 1 | 1 | 0       | 1       | 0       |



# Comparatore completo a 4 bit



# Comparatore completo a 12 bit



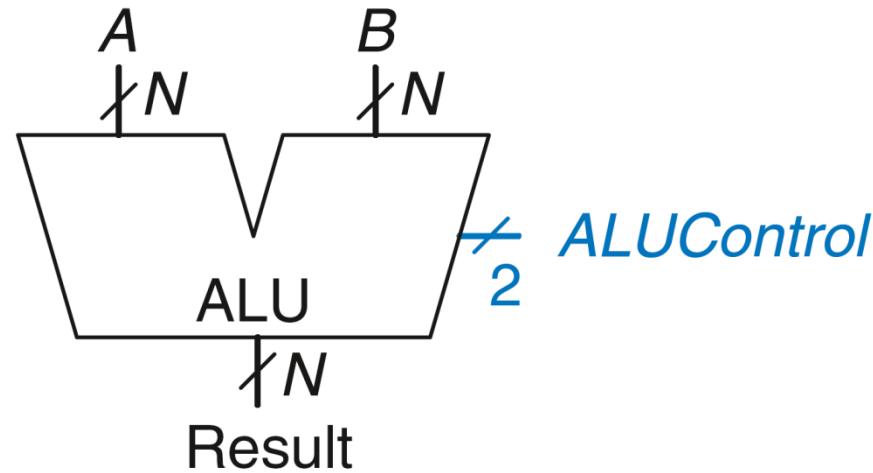
# ALU: Arithmetic Logic Unit

Operazioni che ALU tipicamente esegue:

- Addizione
- Sottrazione
- AND (bit a bit)
- OR (bit a bit)

# ALU: Arithmetic Logic Unit

| ALUControl <sub>1:0</sub> | Function |
|---------------------------|----------|
| 00                        | Add      |
| 01                        | Subtract |
| 10                        | AND      |
| 11                        | OR       |

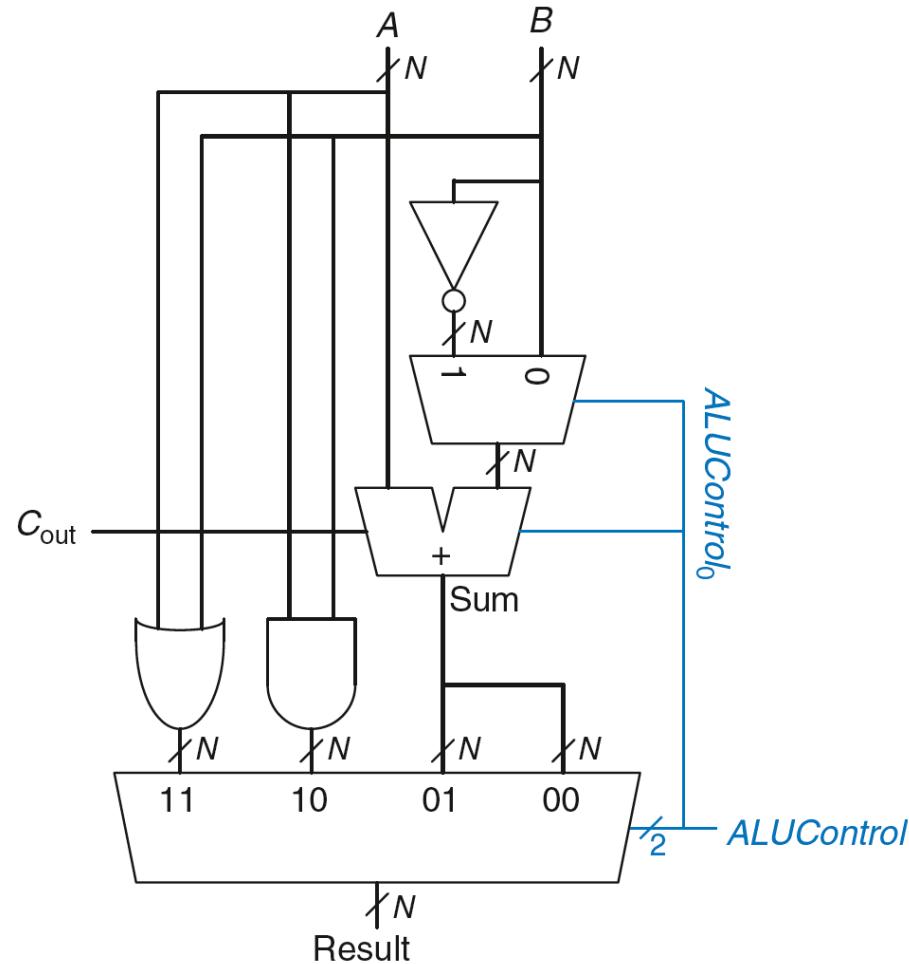


# ALU: Arithmetic Logic Unit

| ALUControl <sub>1:0</sub> | Function |
|---------------------------|----------|
| 00                        | Add      |
| 01                        | Subtract |
| 10                        | AND      |
| 11                        | OR       |

**Esempio: A OR B**

$ALUControl_{1:0} = 11$



# ALU: Arithmetic Logic Unit

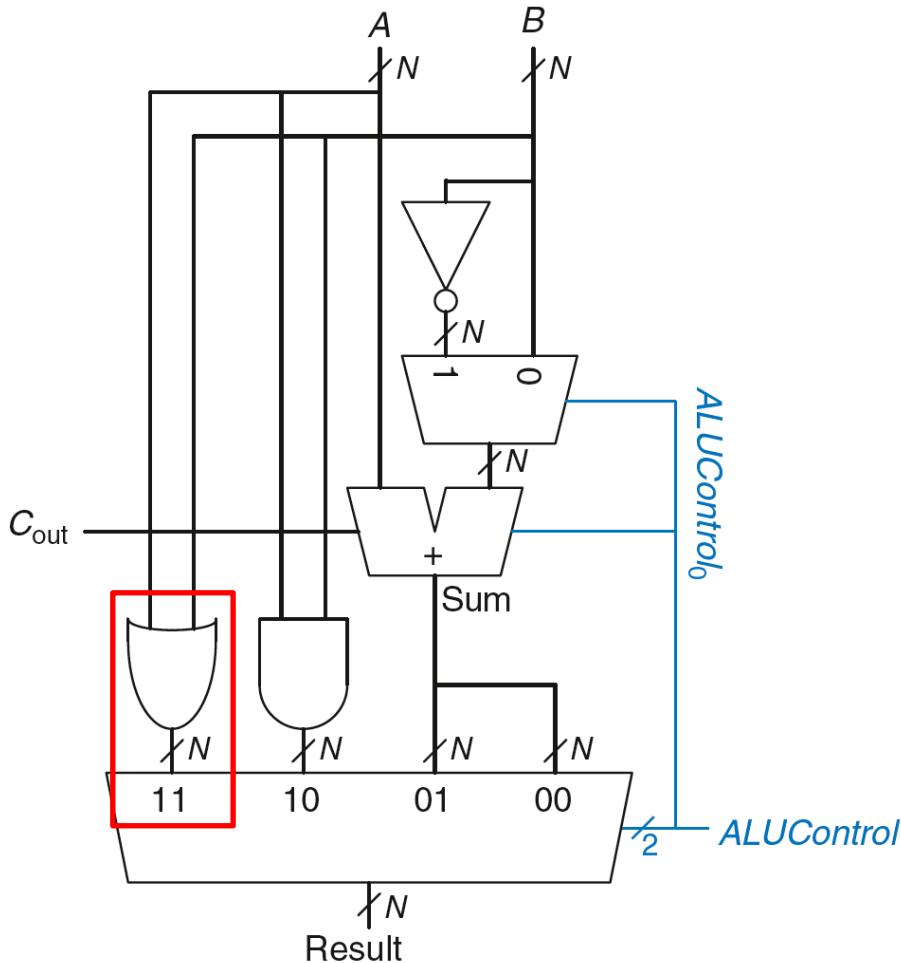
| ALUControl <sub>1:0</sub> | Function |
|---------------------------|----------|
| 00                        | Add      |
| 01                        | Subtract |
| 10                        | AND      |
| 11                        | OR       |

**Esempio: A OR B**

$ALUControl_{1:0} = 11$

Mux seleziona l'output della porta OR

**Result = A OR B**



# ALU: Arithmetic Logic Unit

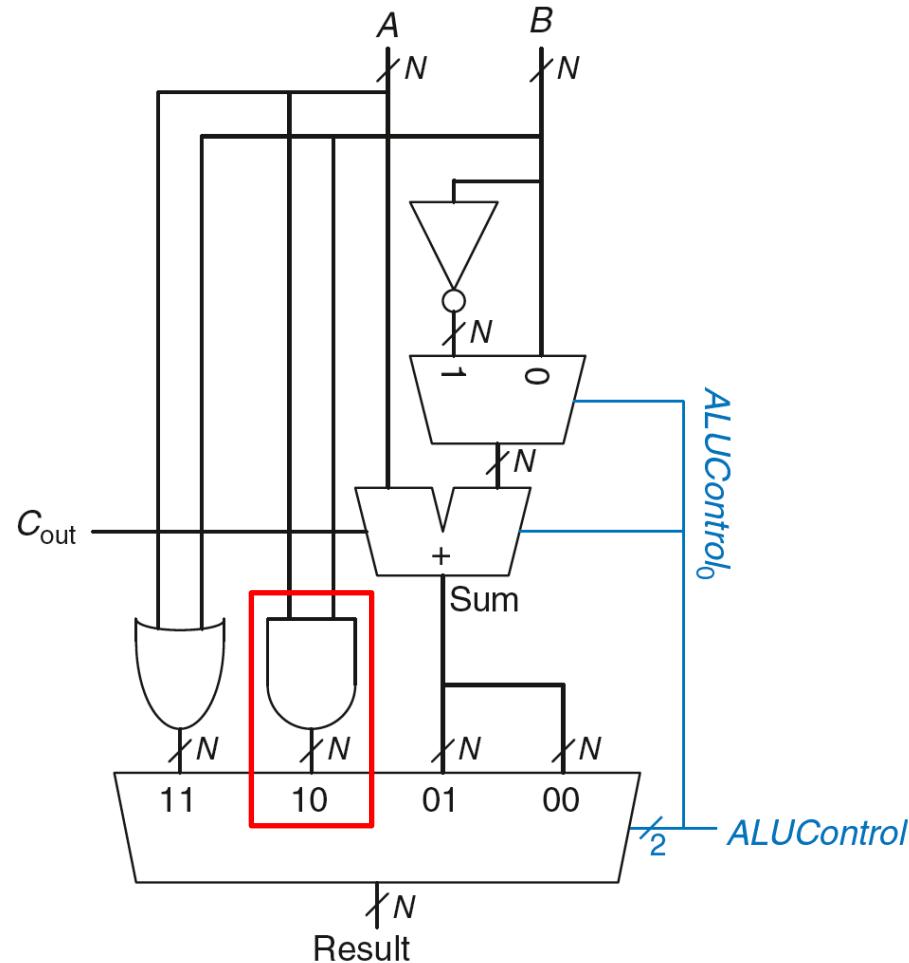
| ALUControl <sub>1:0</sub> | Function |
|---------------------------|----------|
| 00                        | Add      |
| 01                        | Subtract |
| 10                        | AND      |
| 11                        | OR       |

**Esempio: A AND B**

$ALUControl_{1:0} = 10$

Mux seleziona l'output della porta AND

**Result = A AND B**

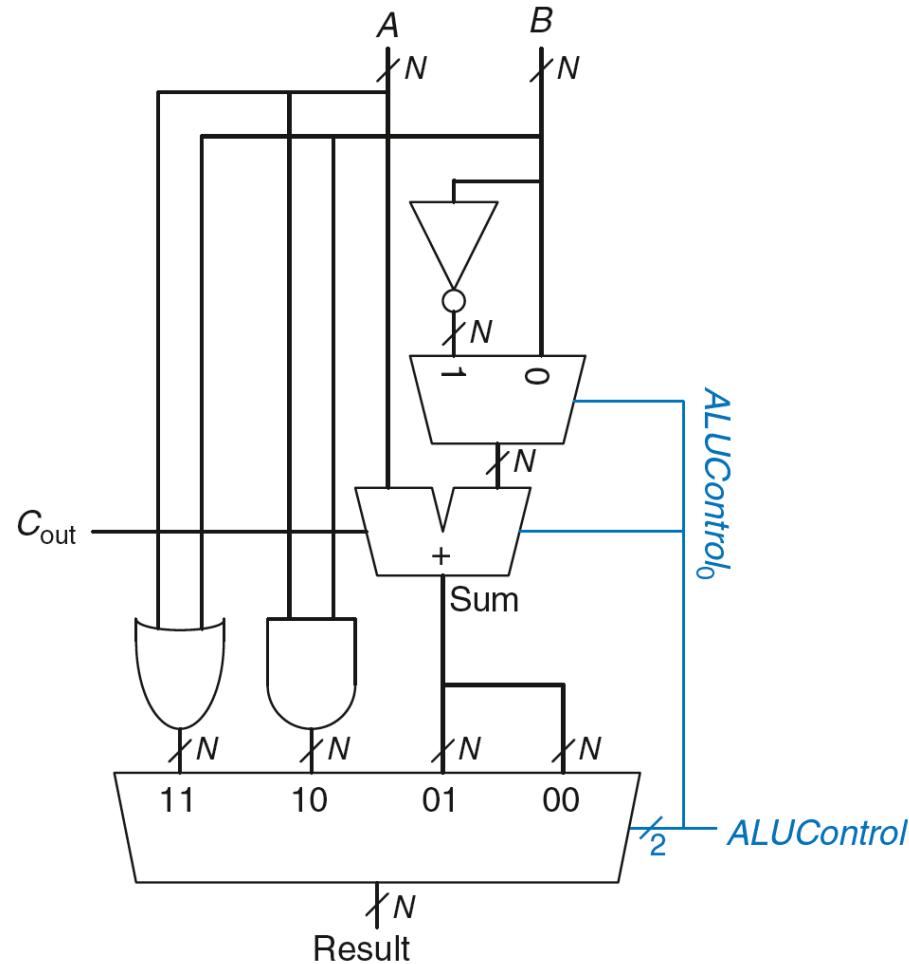


# ALU: Arithmetic Logic Unit

| ALUControl <sub>1:0</sub> | Function |
|---------------------------|----------|
| 00                        | Add      |
| 01                        | Subtract |
| 10                        | AND      |
| 11                        | OR       |

**Esempio:  $A + B$**

$ALUControl_{1:0} = 00$



# ALU: Arithmetic Logic Unit

| ALUControl <sub>1:0</sub> | Function |
|---------------------------|----------|
| 00                        | Add      |
| 01                        | Subtract |
| 10                        | AND      |
| 11                        | OR       |

**Esempio:  $A + B$**

$ALUControl_{1:0} = 00$

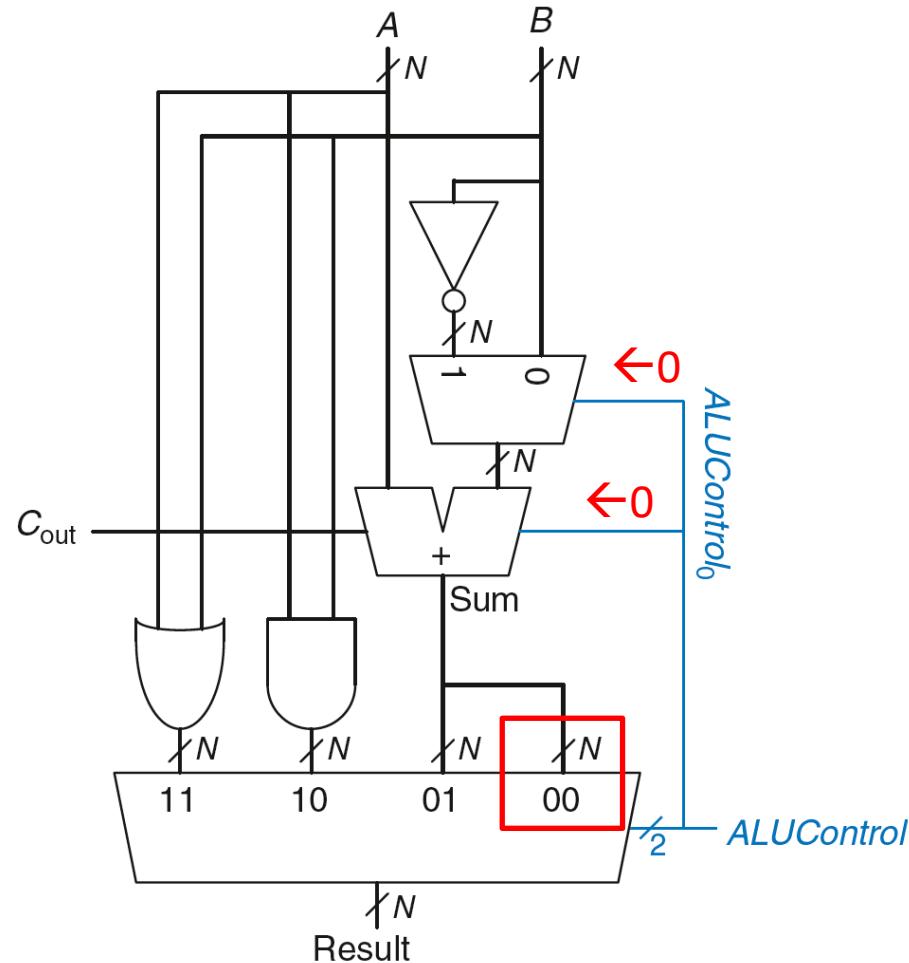
$ALUControl_0 = 0$ , quindi:

Cin = 0

il 2<sup>nd</sup> input dell'adder è  $B$

Mux seleziona *Sum* come *Result*

**$Result = A + B$**



# ALU: Arithmetic Logic Unit

| ALUControl <sub>1:0</sub> | Function |
|---------------------------|----------|
| 00                        | Add      |
| 01                        | Subtract |
| 10                        | AND      |
| 11                        | OR       |

**Esempio:  $A - B$**

$ALUControl_{1:0} = 01$

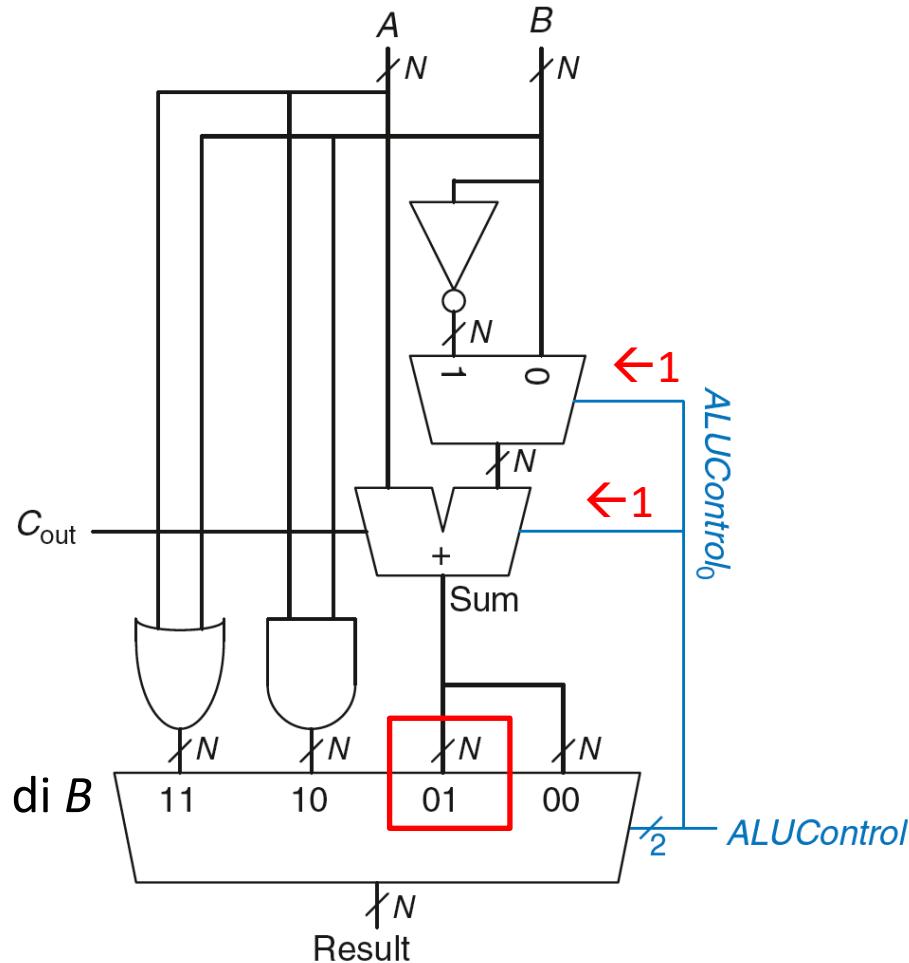
$ALUControl_0 = 1$ , quindi:

Cin = 1

2<sup>nd</sup> input dell'adder è il complemento di B

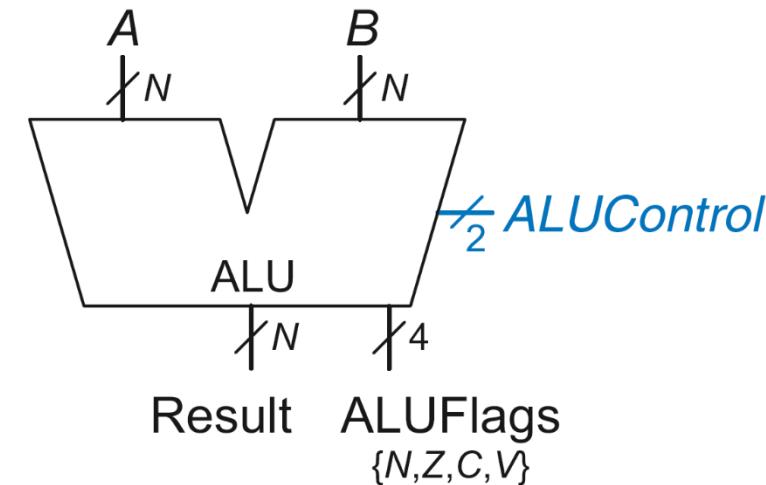
Mux seleziona Sum come Result

**Result =  $A - B$**

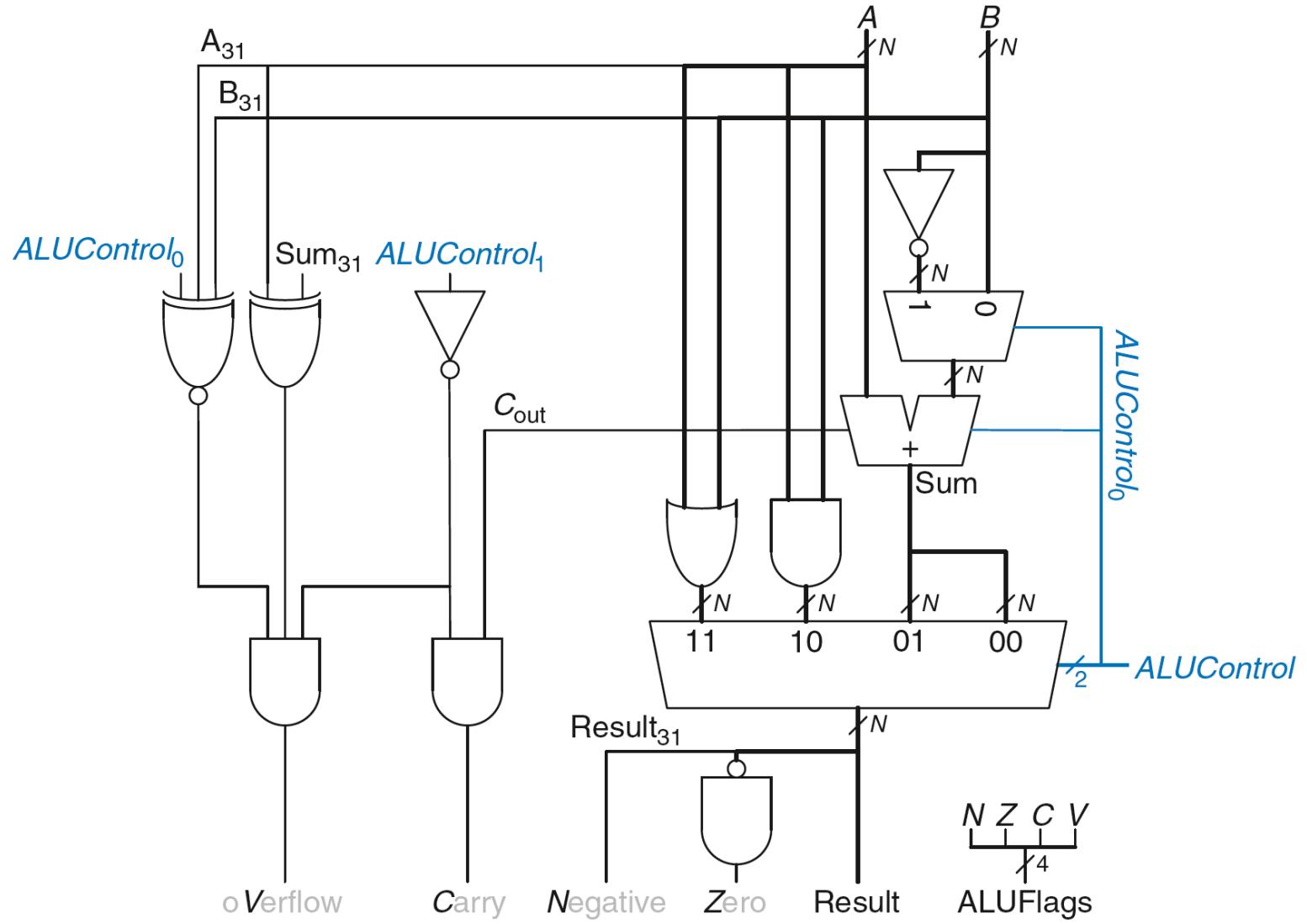


# ALU con flags di stato

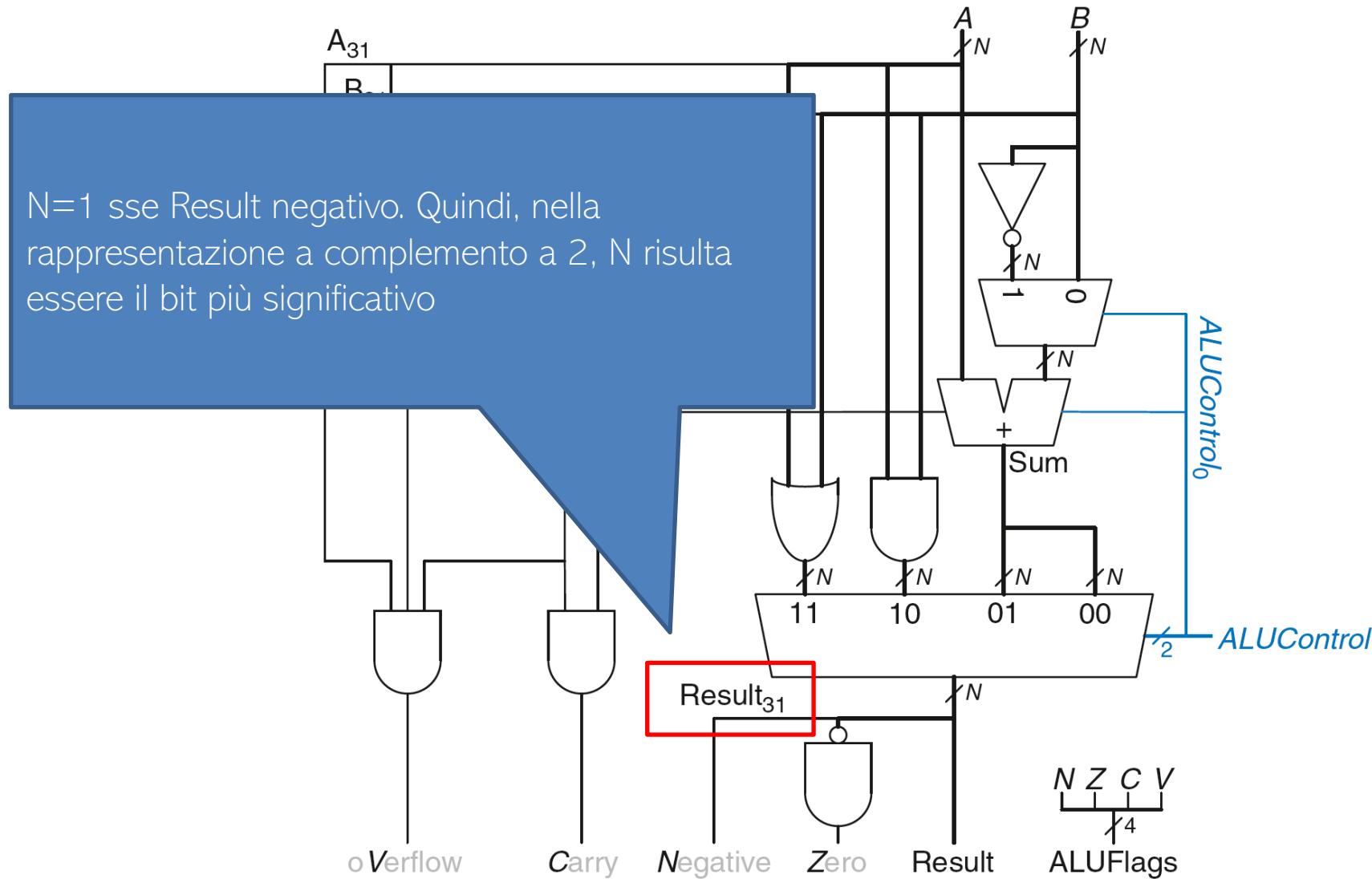
| Flag | Description               |
|------|---------------------------|
| $N$  | <i>Result</i> is Negative |
| $Z$  | <i>Result</i> is Zero     |
| $C$  | Adder produces Carry out  |
| $V$  | Adder oVerflowed          |



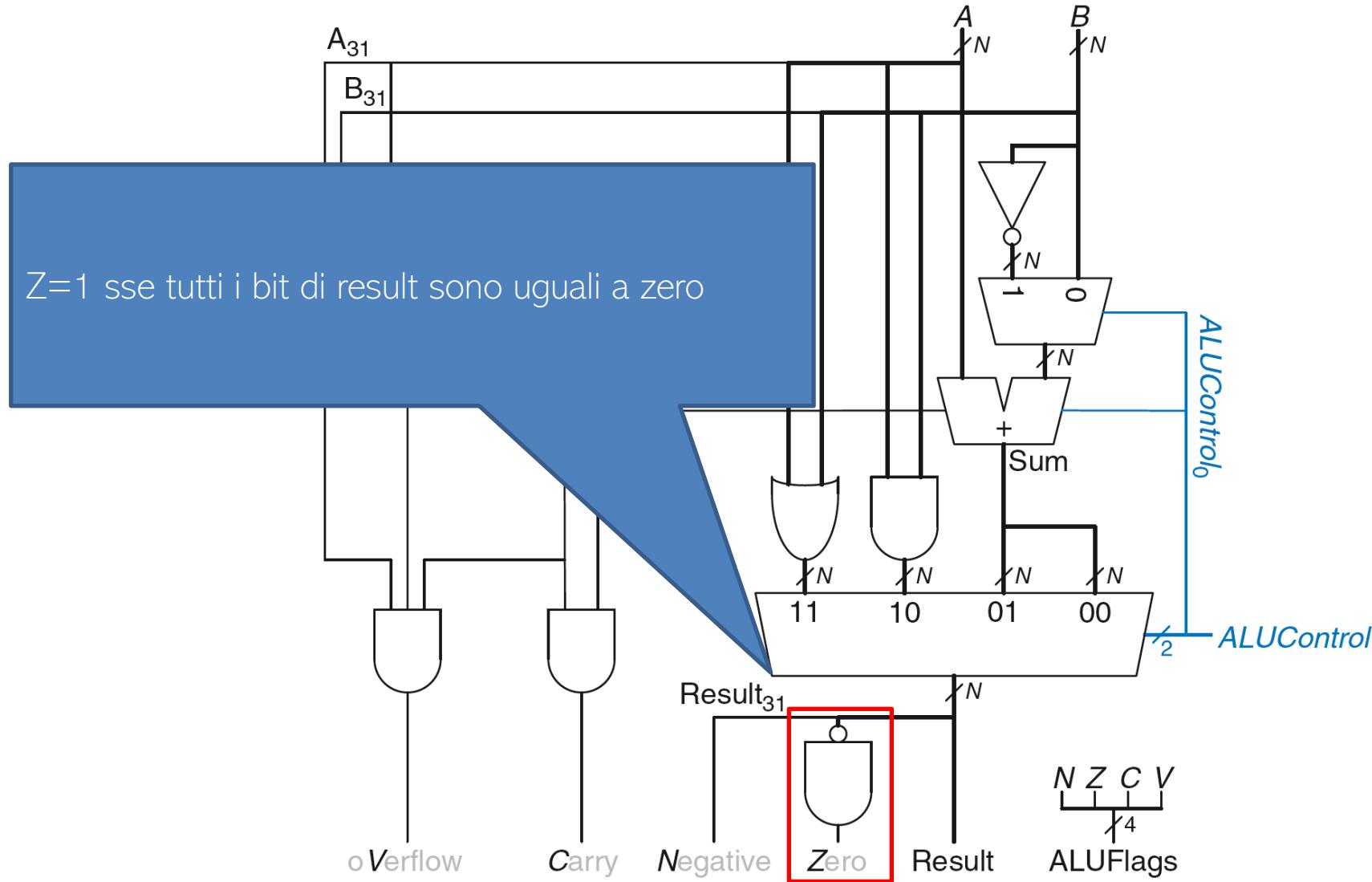
# ALU con flags di stato



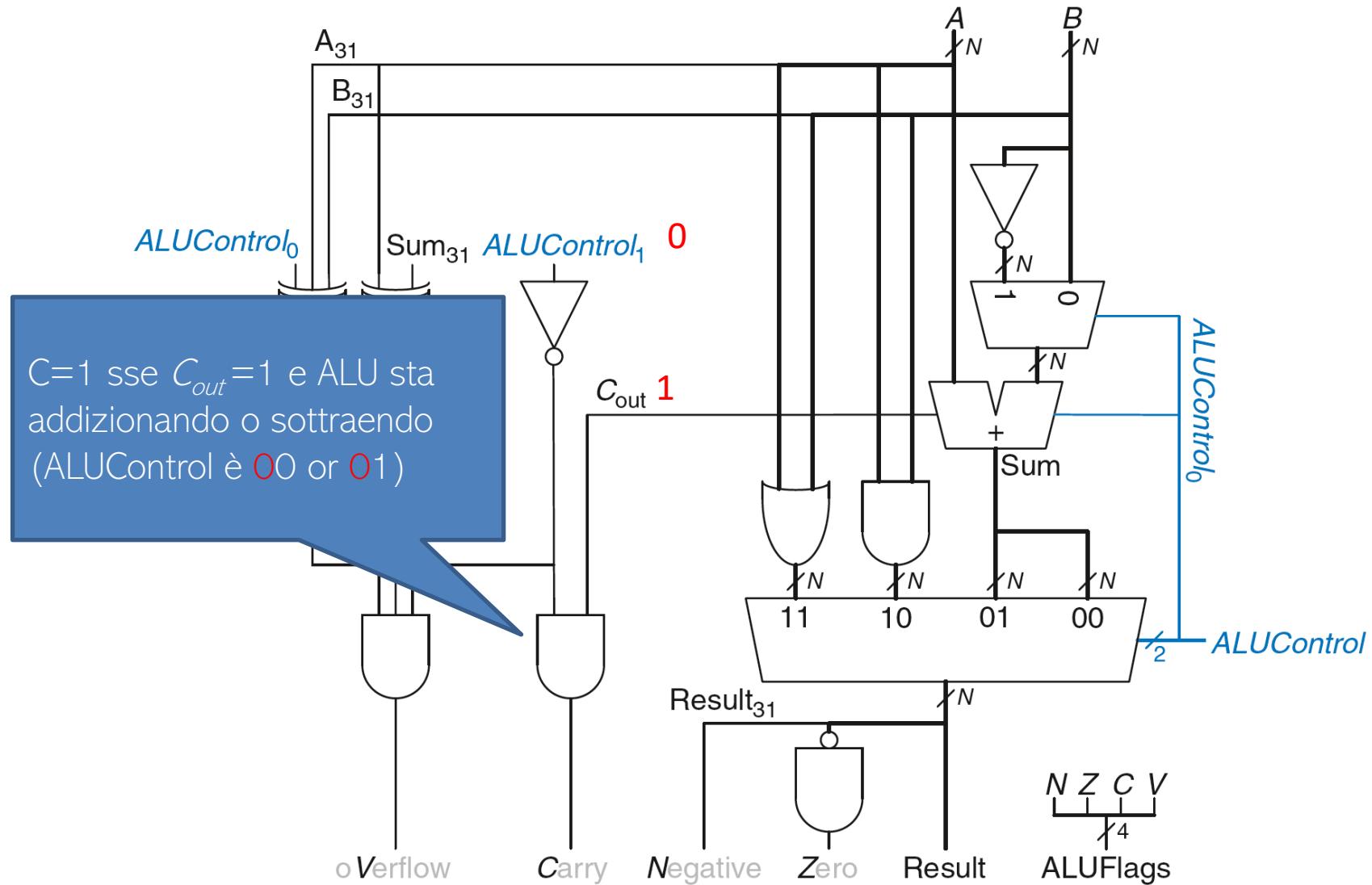
# ALU con flags di stato: Negative



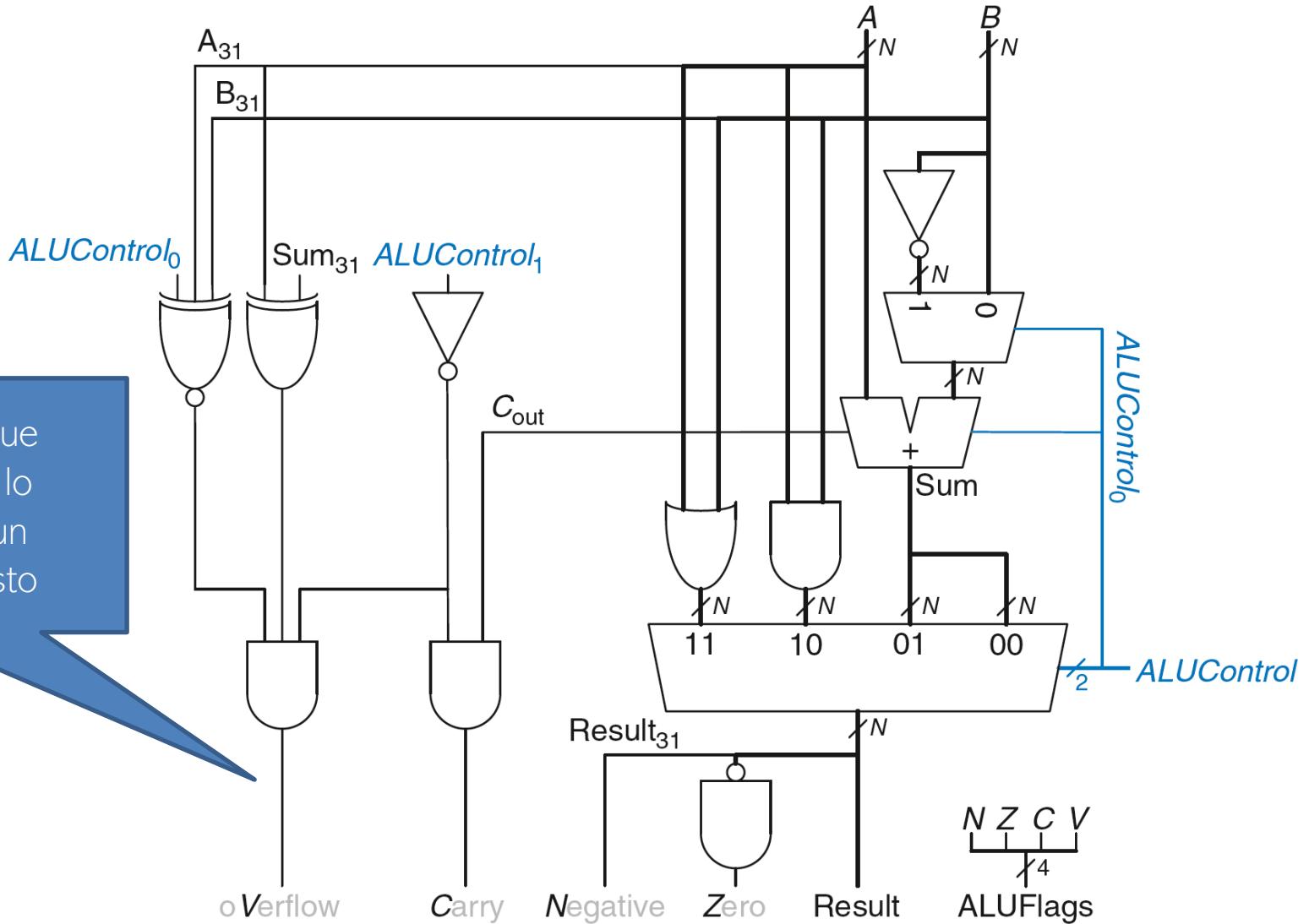
# ALU con flags di stato: Zero



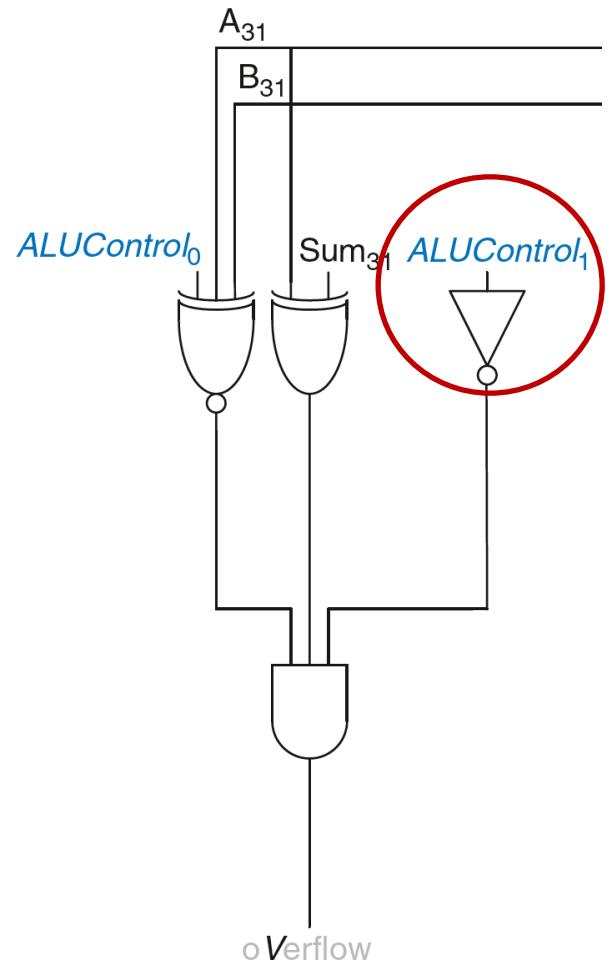
# ALU con flags di stato: Carry



# ALU con flags di stato: Overflow



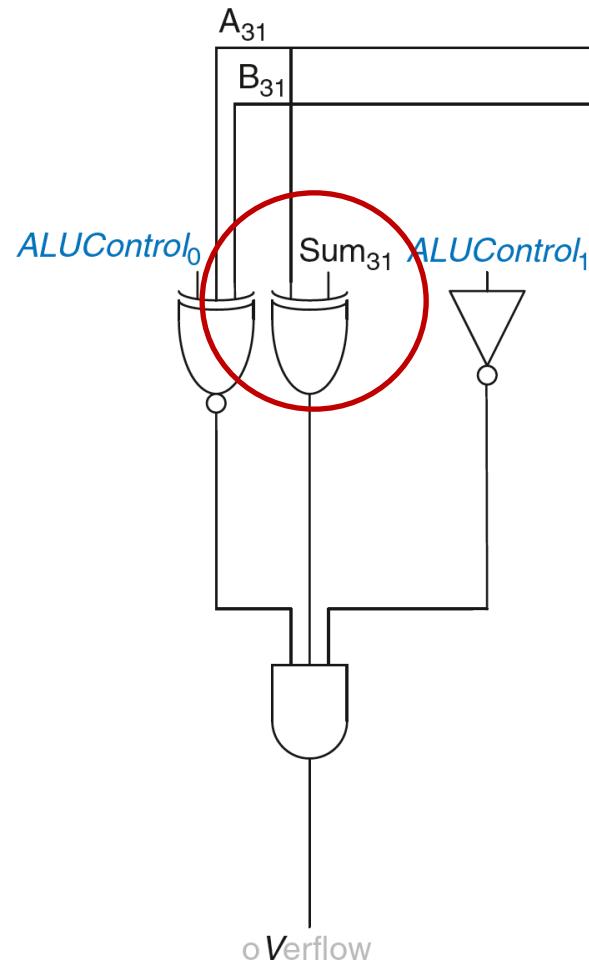
# ALU con flags di stato: Overflow



**V = 1** sse:

ALU esegue una addizione o sottrazione  
(*ALUControl*<sub>1</sub> = 0)

# ALU con flags di stato: Overflow



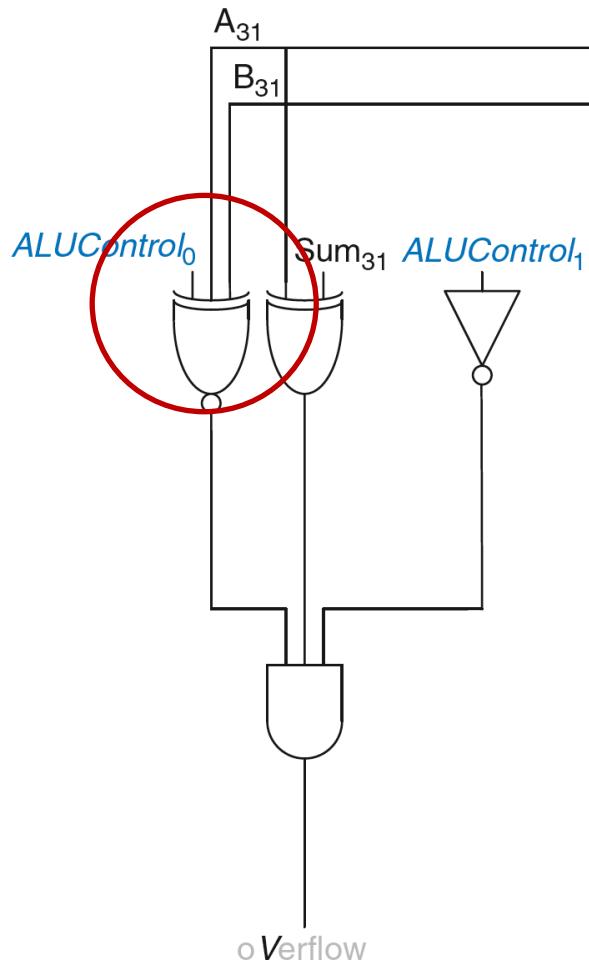
**V = 1** sse:

ALU esegue una addizione o sottrazione  
( $ALUControl_1 = 0$ )

**AND**

A e Sum hanno segno opposto

# ALU con flags di stato: Overflow



**V = 1** sse:

ALU esegue una addizione o sottrazione  
( $ALUControl_1 = 0$ )

**AND**

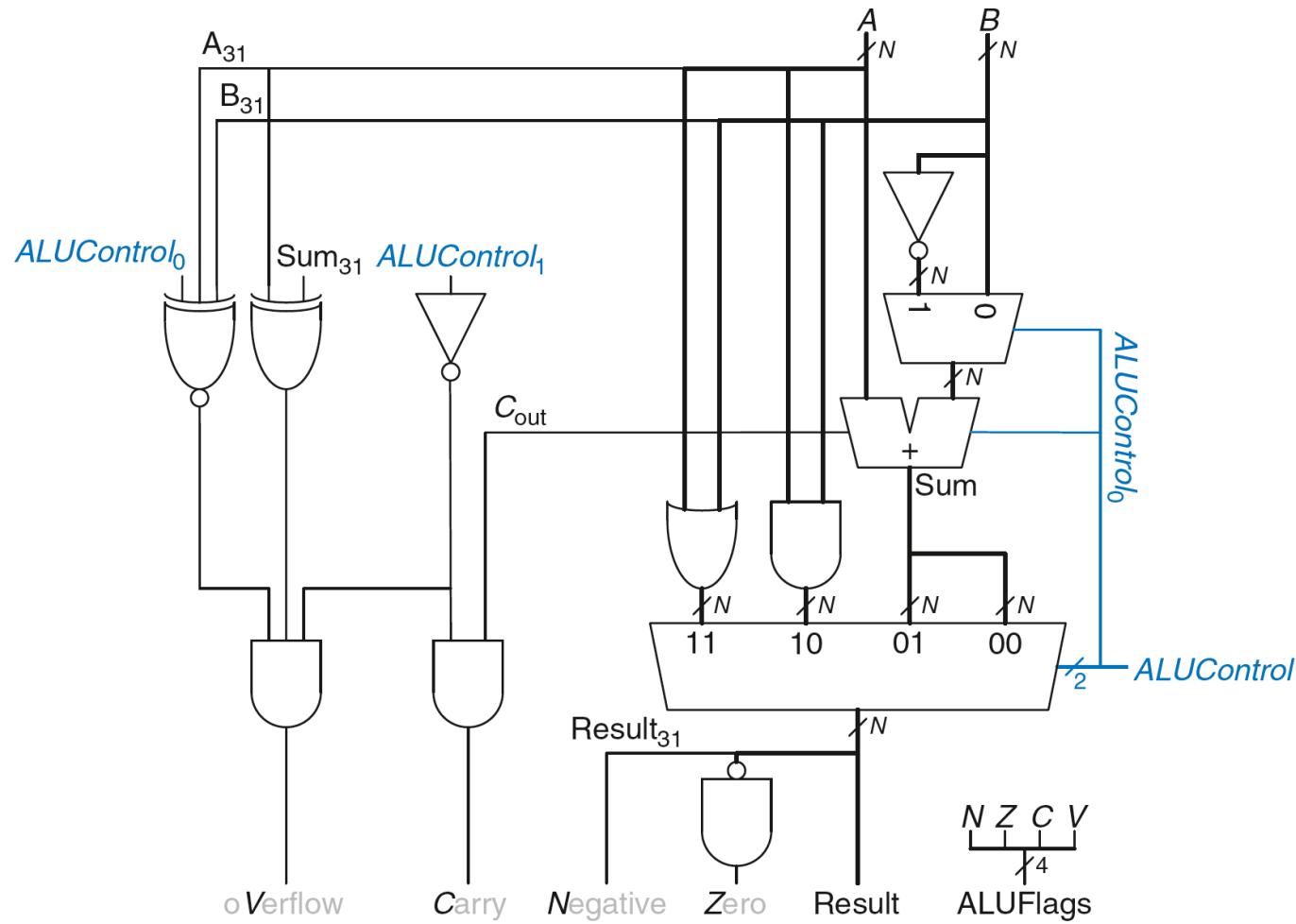
A e Sum hanno segno opposto

**AND**

A e B hanno lo stesso segno sotto addizione  
( $ALUControl_0 = 0$ ) **OR**

A e B hanno segni differenti sotto sottrazione  
( $ALUControl_0 = 1$ )

# ALU con flags di stato



# Shifters

**Logical shifter:** trasla i bit a sinistra o a destra e riempie gli spazi vuoti con degli 0

**Arithmetic shifter:** come il logical shifter a sinistra, nella traslazione a destra invece riempie gli spazi vuoti con il bit più significativo (msb)

**Rotator:** ruota i bits in cerchio, i bits che “escono” da un lato rientrano dall’ “altro”

# Shifters

## Logical shifter:

- Ex:  $11001 \gg 2 = 00110$
- Ex:  $11001 \ll 2 = 00100$

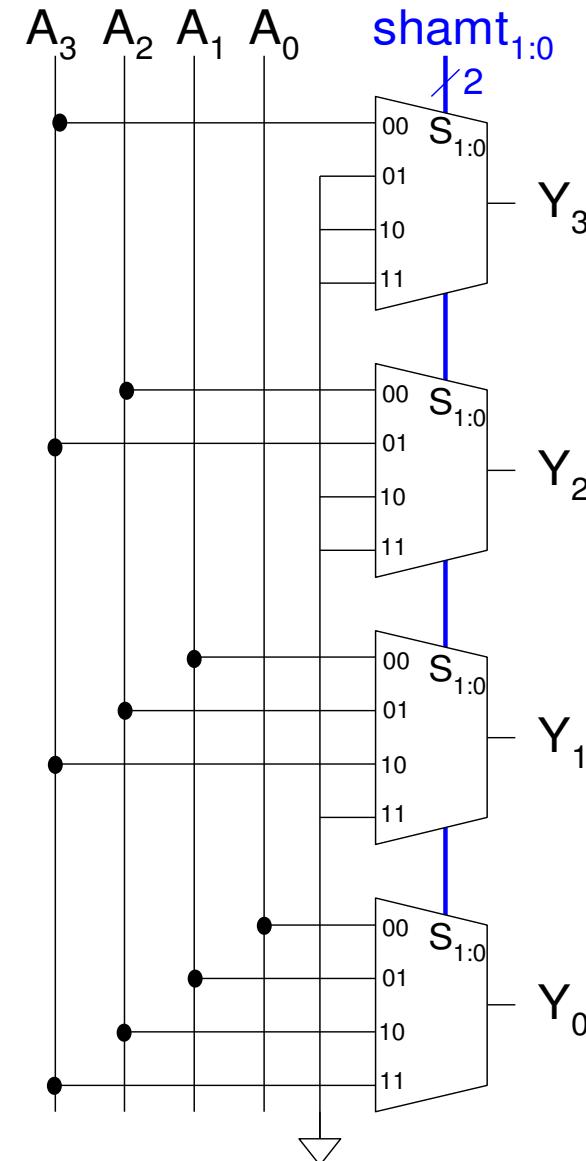
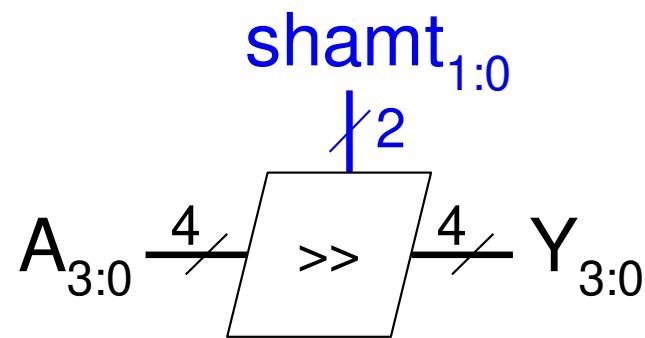
## Arithmetic shifter:

- Ex:  $11001 \ggg 2 = 11110$
- Ex:  $11001 \lll 2 = 00100$

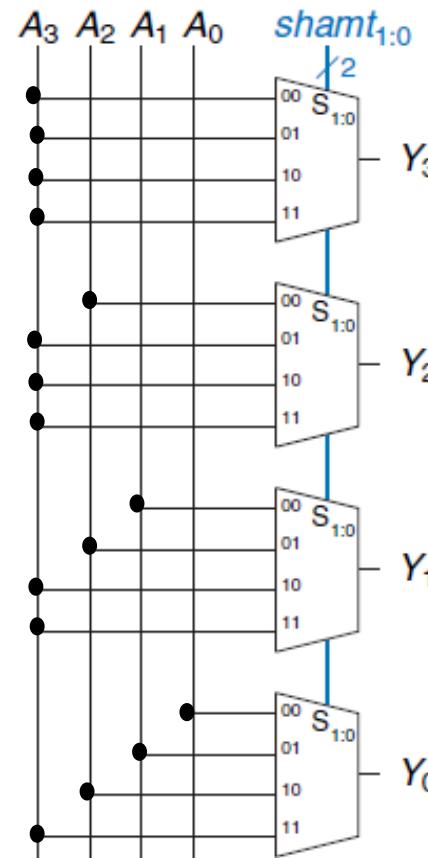
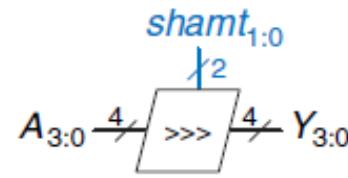
## Rotator:

- Ex:  $11001 \text{ ROR } 2 = 01110$
- Ex:  $11001 \text{ ROL } 2 = 00111$

# Shifter Design



# Shifter Design



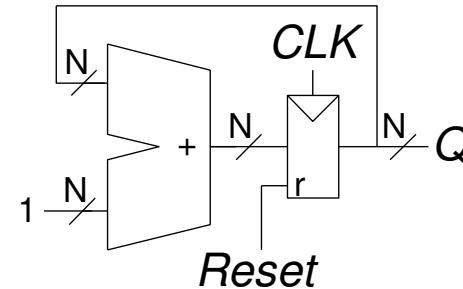
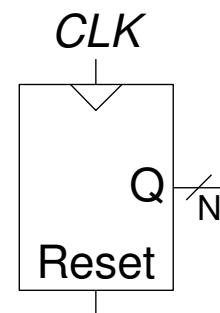
# Shifters as Multipliers, Dividers

- $A \lll N = A \times 2^N$ 
  - Example:  $00001 \ll 2 = 00100$  ( $1 \times 2^2 = 4$ )
  - Example:  $11101 \ll 2 = 10100$  ( $-3 \times 2^2 = -12$ )
- $A \ggg N = A \div 2^N$ 
  - Example:  $01000 \ggg 2 = 00010$  ( $8 \div 2^2 = 2$ )
  - Example:  $10000 \ggg 2 = 11100$  ( $-16 \div 2^2 = -4$ )

# Counters

- Incrementa ad ogni clock
- Usato per eseguire cicli su numeri:  
000, 001, 010, 011, 100, 101, 110, 111, 000, 001...
- Si usa:
  - Come orologio digitale
  - Program counter: tiene traccia della istruzione corrente da eseguire

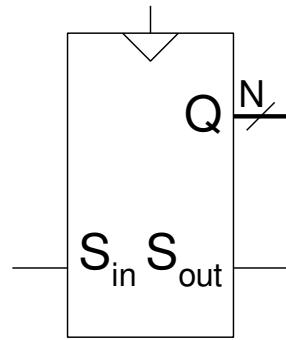
## Symbol      Implementation



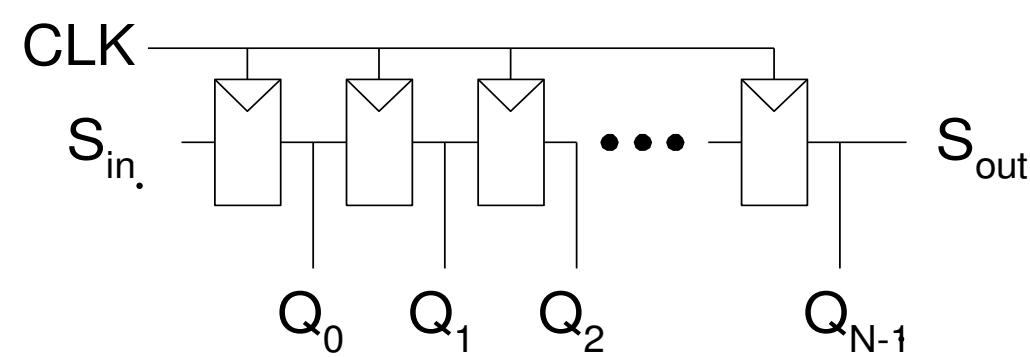
# Shift Registers

- Trasla di un bit ad ogni clock
- Ritorna un bit in uscita ( $S_{out}$ ) ad ogni clock
- *Serial-to-parallel converter*: converte un input seriale ( $S_{in}$ ) in un output parallelo ( $Q_{0:N-1}$ )

**Symbol:**

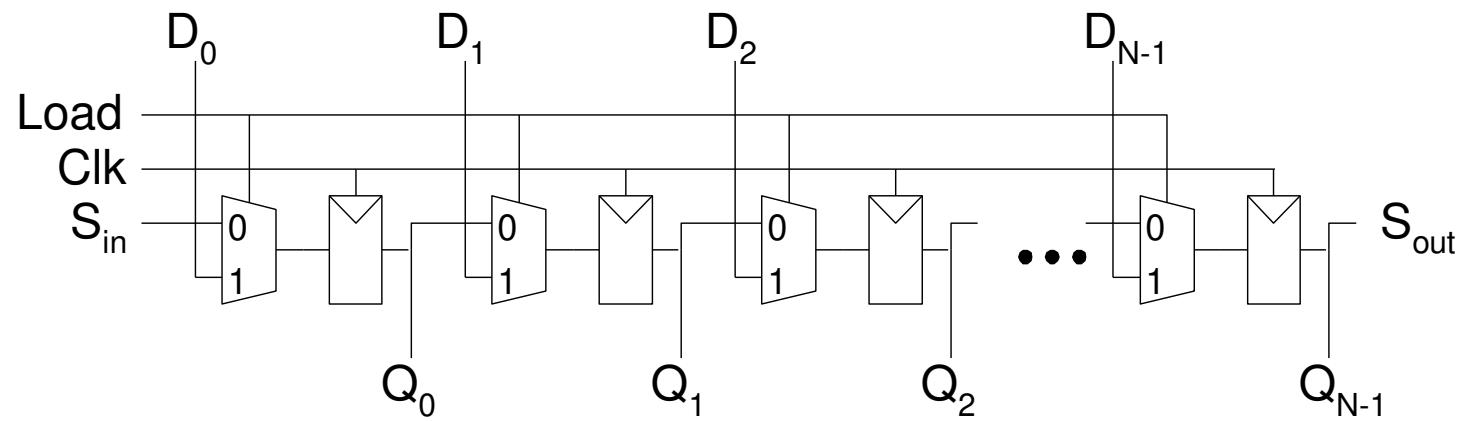


**Implementation:**



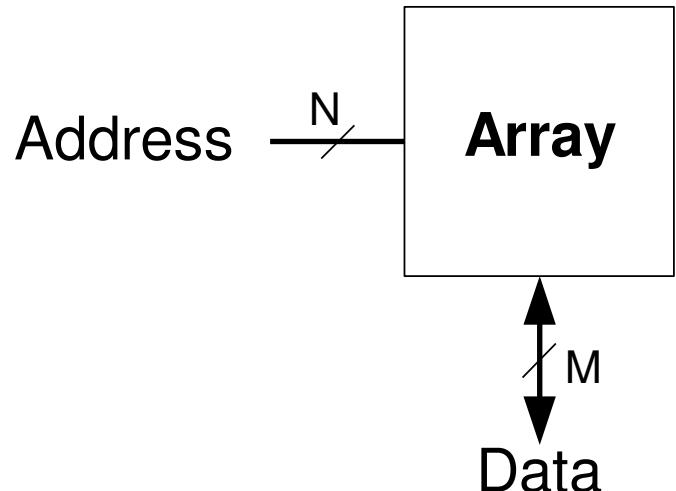
# Shift Register con load parallelo

- Quando  $Load = 1$ , funziona come un usuale registro a  $N$ -bit
- Quando  $Load = 0$ , agisce da shift register
- Quindi può agire da convertitore *seriale-parallelo* (da  $S_{in}$  a  $Q_{0:N-1}$ ) o da convertitore *parallelo-seriale* ( $D_{0:N-1}$  to  $S_{out}$ )



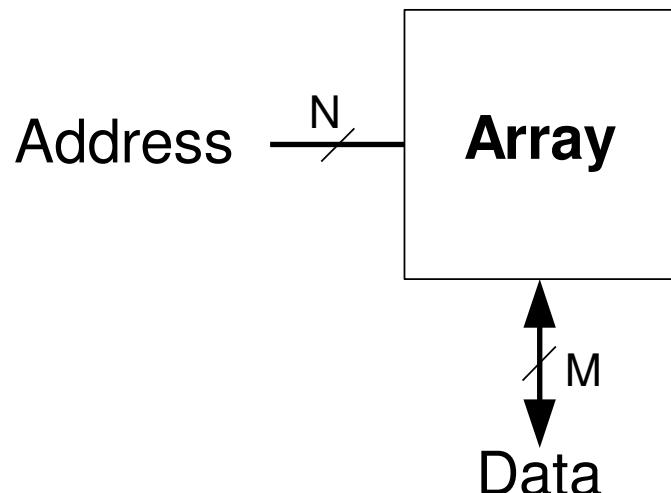
# Memory Arrays

- Memorizzare efficacemente una grossa quantità di dati
- 3 tipologie:
  - Dynamic random access memory (DRAM)
  - Static random access memory (SRAM)
  - Read only memory (ROM)
- dato a  $M$ -bit letto/scritto su di un unico indirizzo a  $N$ -bit



# Memory Arrays

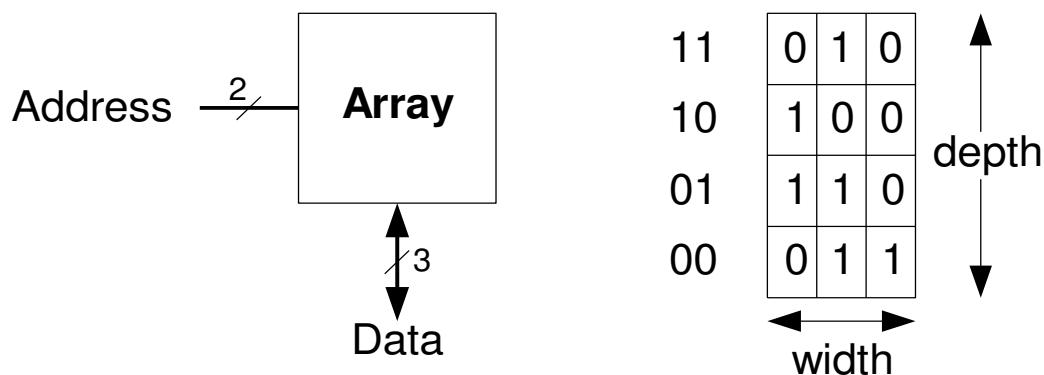
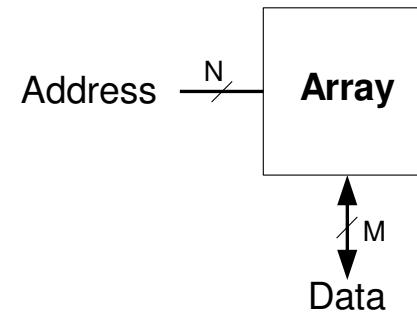
- Memorizzare efficacemente una grossa quantità di dati
- 3 tipologie:
  - Dynamic random access memory (DRAM)
  - Static random access memory (SRAM)
  - Read only memory (ROM)
- dato a  $M$ -bit letto/scritto su di un unico indirizzo a  $N$ -bit



Tipicamente  
 $M=8$  (un byte) e  
 $N=32$  (o 64)

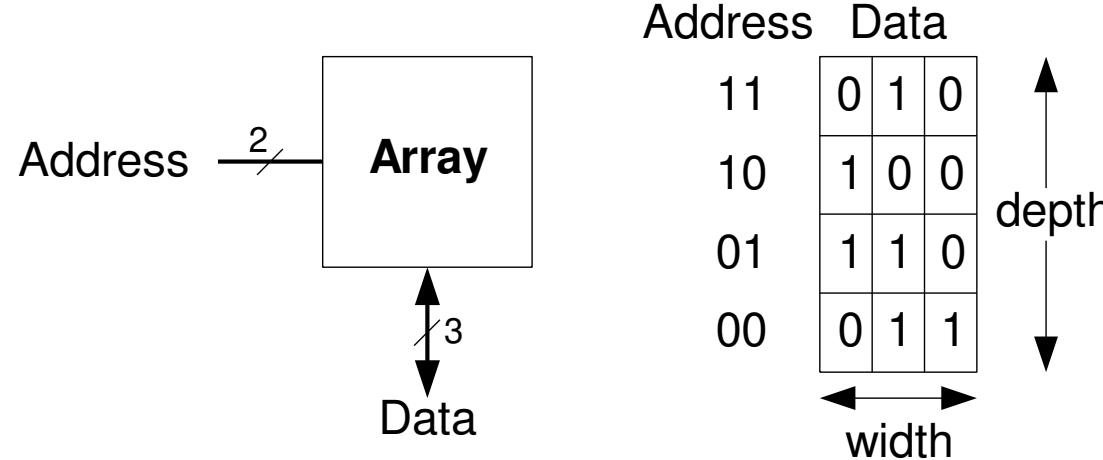
# Memory Arrays

- Array bidimensionali di celle di memoria
- Ogni cella memorizza un bit
- Con  $N$  bit di indirizzo e  $M$  bits data:
  - $2^N$  righe e a  $M$  colonne
  - **Depth:** numero di righe (parole)
  - **Width:** numero di colonne (lunghezza di una parola)
  - **Dimensione Array :** depth  $\times$  width =  $2^N \times M$

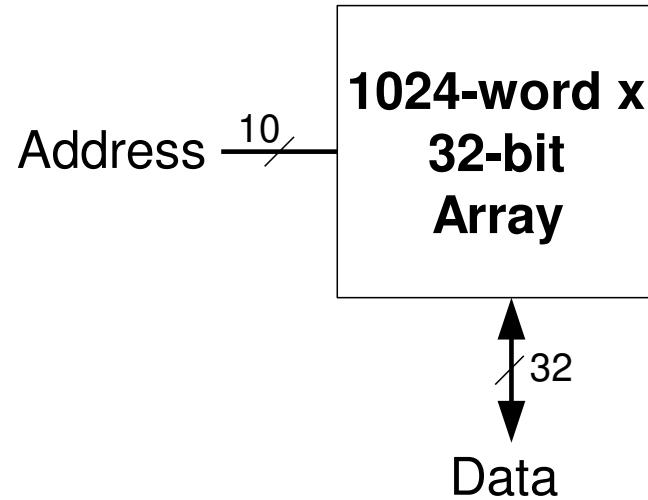


# Esempio

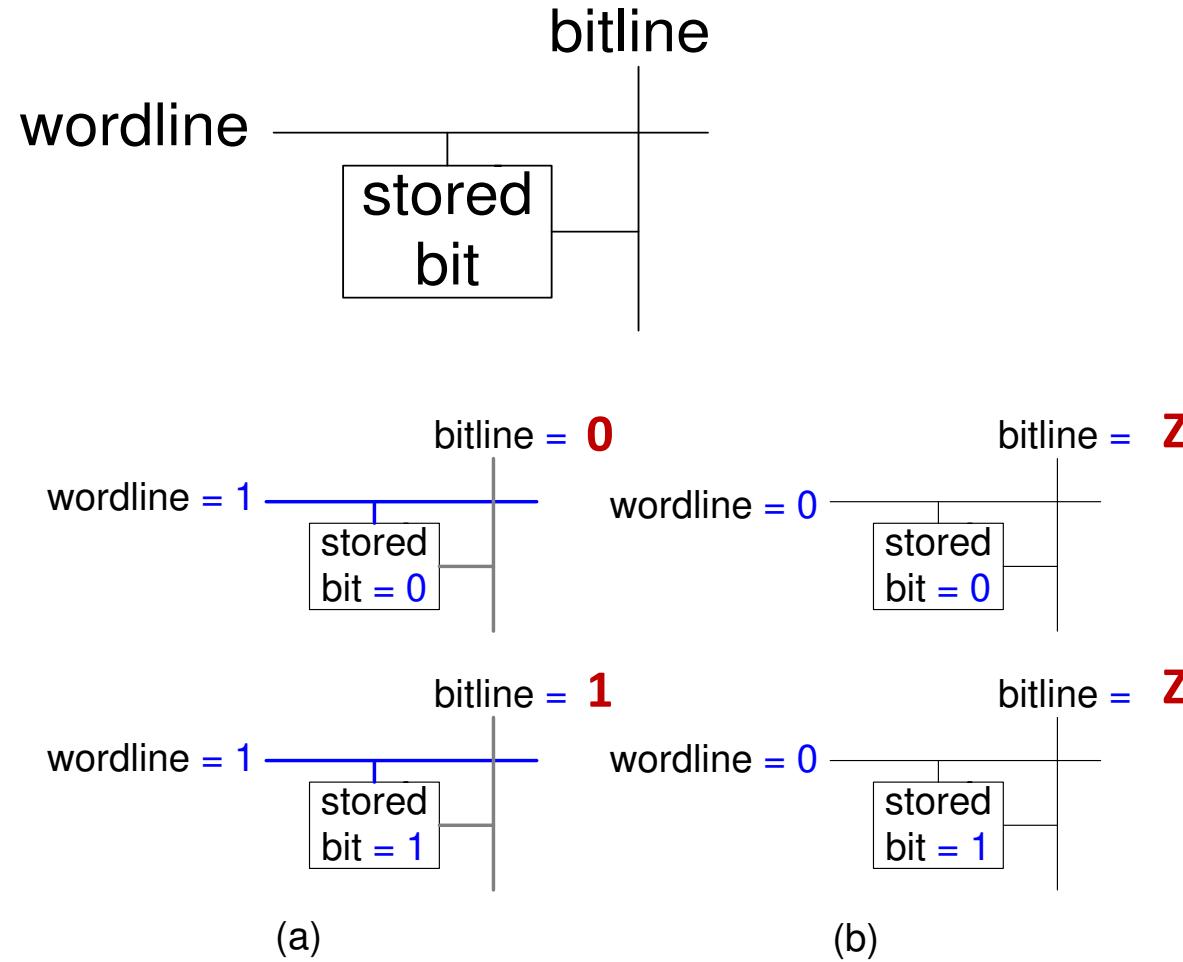
- $2^2 \times 3$ -bit array
- Numero parole: 4
- Lunghezza parole: 3-bits
- All'indirizzo 10 corrisponde la parola 100



# Memory Arrays



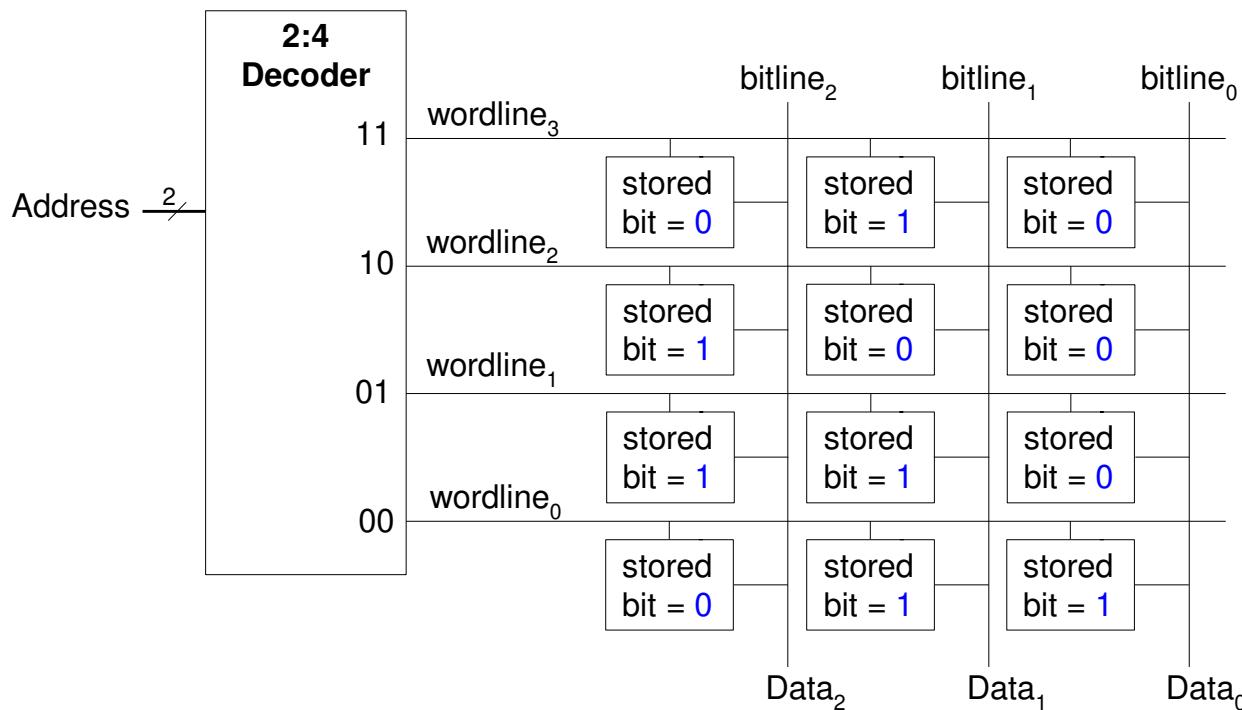
# Memory Array Bit Cells



# Memory Array

## ■ Wordline:

- Agisce come un enable
- Seleziona una riga nella memoria
- Corrisponde ad un unico indirizzo
- Solo una wordline per volta è attivata



# Tipi di memoria

- Random access memory (RAM): **volatile**
- Read only memory (ROM): **nonvolatile**

# RAM: Random

- **Volatile:** i dati sono persi quando il computer è spento
- Le operazioni di lettura e scrittura sono più veloci (rispetto alle ROM)
- E' la memoria primaria di un computer (DRAM)

E' chiamata *random access memory* perché il tempo di accesso ad una parola è lo stesso per ogni indirizzo (a differenza delle memorie ad accesso sequenziale come i nastri che si usavano ai tempi del C64)

# ROM: Read Only Memory

- **Non volatile:** mantiene i dati anche quando il computer è spento
- La lettura è relativamente veloce ma la scrittura è lenta o semplicemente non è possibile scrivere
- Drives, flash memory, Bios etc. sono ROMs

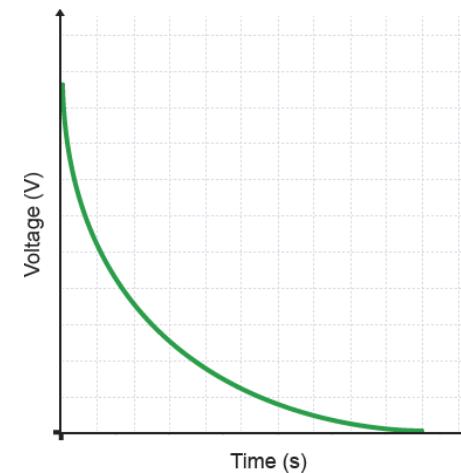
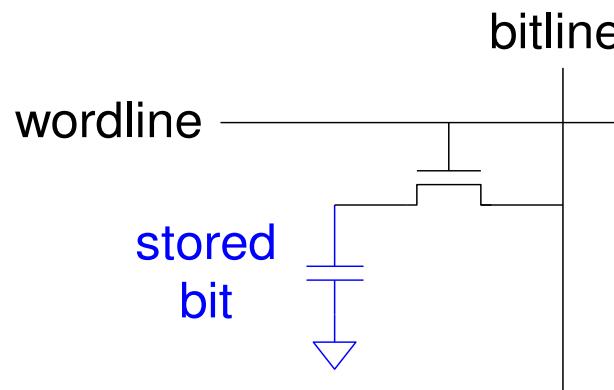
ROM sta per *read only* memory perché inizialmente questo tipo di memorie venivano “scritte” bruciando dei fusibili. Quindi, una volta configurate, non erano più manipolabili. Chiaramente questo non è più il caso per Flash memory e altri tipi di ROMs.

# Tipi di RAM

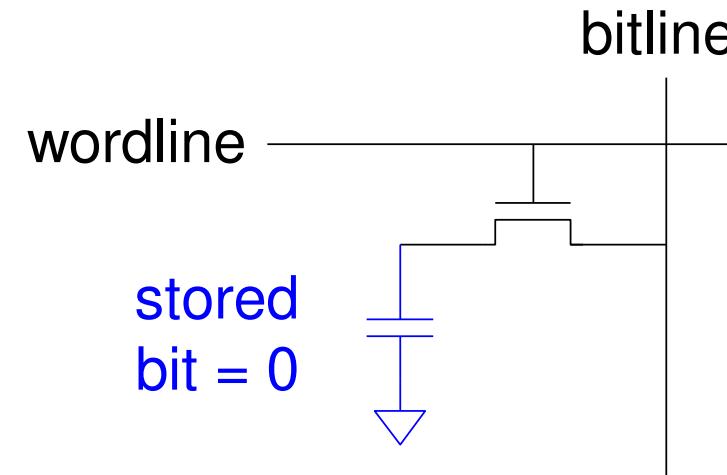
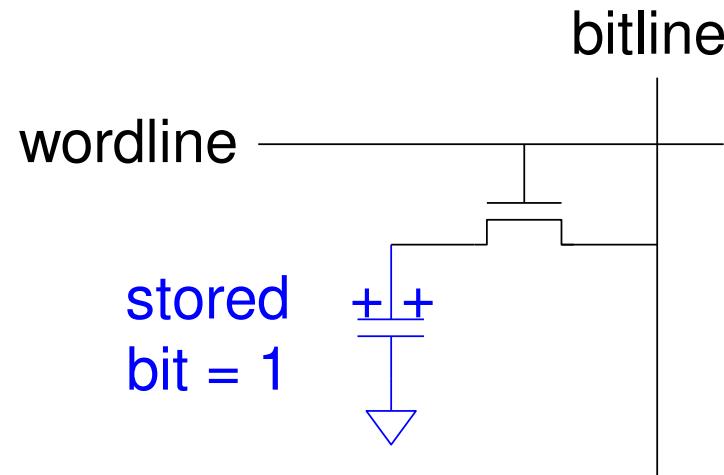
- **DRAM** (Dynamic random access memory)
- **SRAM** (Static random access memory)
- Si differenziano per le componenti usate per memorizzare i dati:
  - DRAM usa delle capacità
  - SRAM usa invertitori

# DRAM

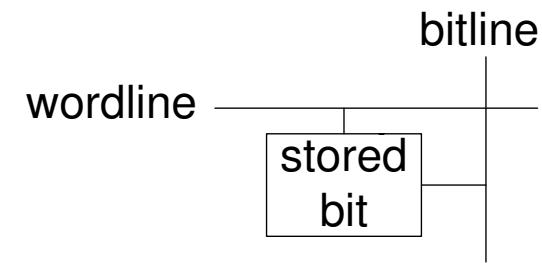
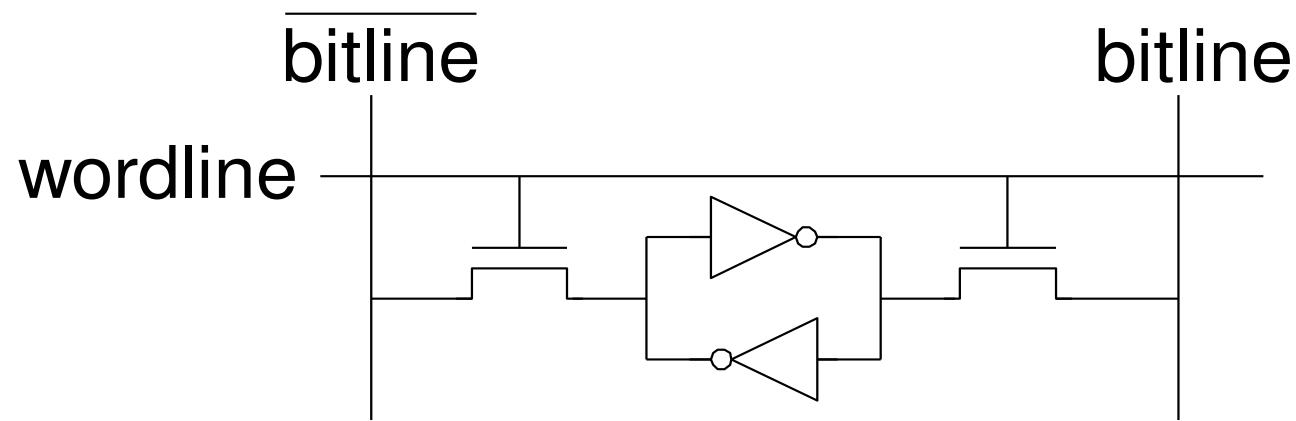
- i bit data sono immagazzinati in delle capacità
- *Dynamic* perché il valore di un bit deve essere “refreshed” (riscritto) periodicamente e anche dopo che è stato letto:
  - La perdita di carica di una capacità degrada il valore memorizzato
  - La lettura distrugge il valore letto



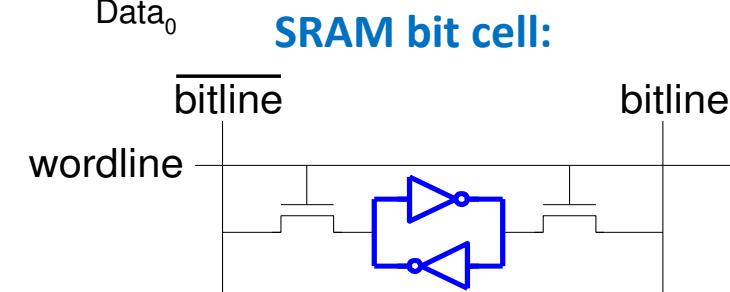
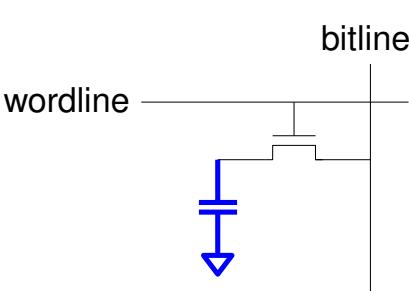
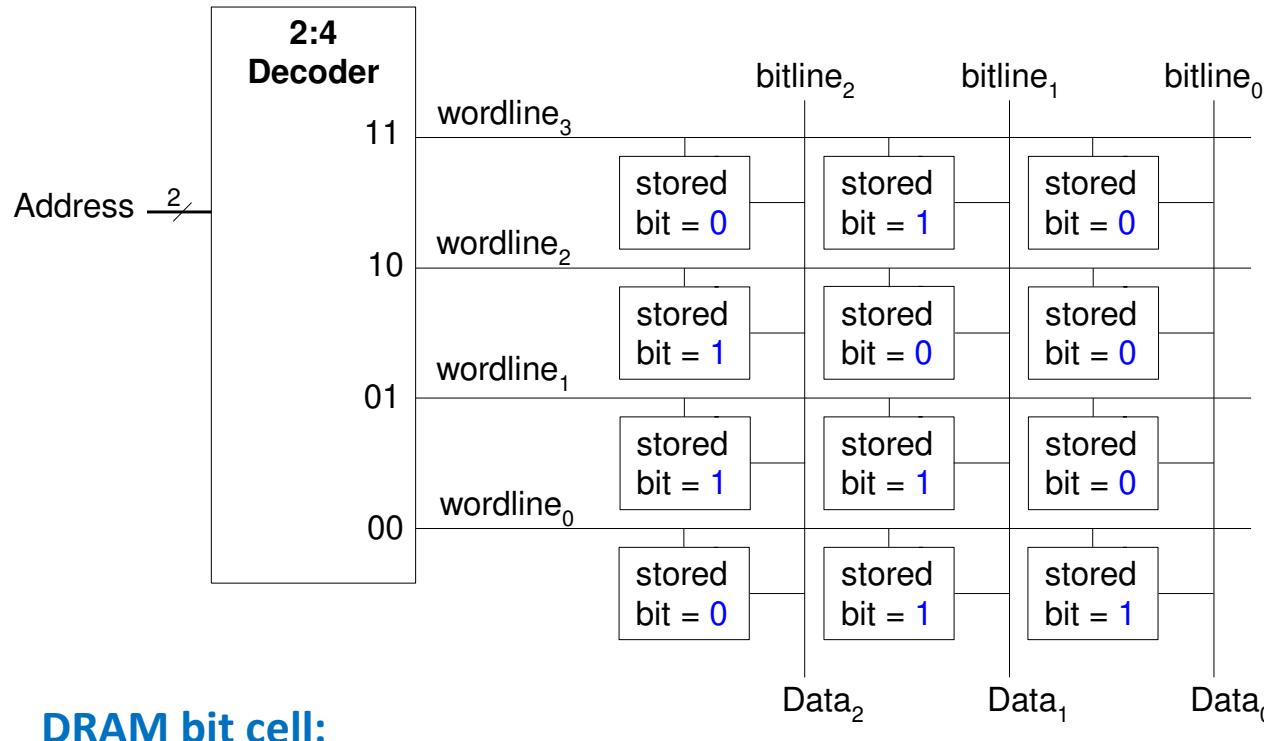
# DRAM



# SRAM



# Memory Arrays Review



# DRAM vs SRAM

- DRAM è più lenta
- Scrittura e refresh (millisecondi) inducono un consumo maggiore di energia
- DRAM, più economiche

| Memory Type | Transistors per Bit Cell | Latency |
|-------------|--------------------------|---------|
| flip-flop   | ~20                      | fast    |
| SRAM        | 6                        | medium  |
| DRAM        | 1                        | slow    |

# DRAM vs SRAM

- DRAM è più lenta
- Scrittura e refresh (millisecondi) inducono un consumo maggiore di energia
- DRAM, più economiche

| Memory Type | Transistors per Bit Cell | Latency |
|-------------|--------------------------|---------|
| flip-flop   | ~20                      | fast    |
| SRAM        | 6                        | medium  |
| DRAM        | 1                        | slow    |



A blue callout bubble points from the SRAM row in the table towards the word "registri". The word "registri" is written in white text inside the blue bubble.

registri

# DRAM vs SRAM

- DRAM è più lenta
- Scrittura e refresh (millisecondi) inducono un consumo maggiore di energia
- DRAM, più economiche

| Memory Type | Transistors per Bit Cell | Latency |
|-------------|--------------------------|---------|
| flip-flop   | ~20                      | fast    |
| SRAM        | 6                        | medium  |
| DRAM        | 1                        | slow    |

The diagram illustrates the memory hierarchy. It features a blue rounded rectangle labeled "cache" containing the SRAM row from the table. An arrow points from the word "cache" to the SRAM row, indicating that SRAM is positioned between the fast flip-flop and the slow DRAM.

# DRAM vs SRAM

- DRAM è più lenta
- Scrittura e refresh (millisecondi) inducono un consumo maggiore di energia
- DRAM, più economiche

| Memory Type | Transistors per Bit Cell | Latency |
|-------------|--------------------------|---------|
| flip-flop   | ~20                      | fast    |
| SRAM        | 6                        | medium  |
| DRAM        | 1                        | slow    |

memoria centrale

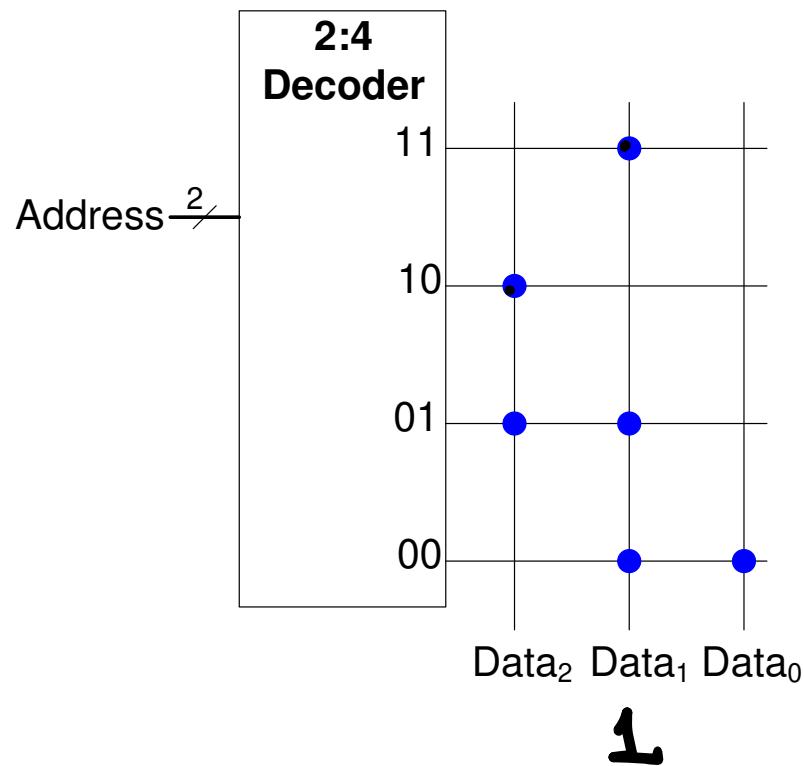
# DDR SDRAM

- (DDR) Double data rate (S) synchronous (DRAM) dynamic random access memory
- Le operazioni di lettura e scrittura sono sincronizzate da un clock
- Le trasmissioni avvengono sia nel rising-edge del clock ( $0 \rightarrow 1$ ) che nel falling-edge ( $1 \rightarrow 0$ )
- In questo modo il data-bandwidth doppio rispetto alla frequenza di clock (double pumping)

# DDR SDRAM

| Standard name | Memory clock<br>(MHz) | Cycle time <sup>[4]</sup><br>(ns) | Data rate<br>(MT/s) | Peak transfer rate<br>(MB/s) |
|---------------|-----------------------|-----------------------------------|---------------------|------------------------------|
| DDR-200       | 100                   | 10                                | 200                 | 1600                         |
| DDR-266       | 133.33                | 7.5                               | 266.67              | 2133.33                      |
| DDR-333       | 166.67                | 6                                 | 333.33              | 2666.67                      |
| DDR-400       | 200                   | 5                                 | 400                 | 3200                         |

# ROM Storage

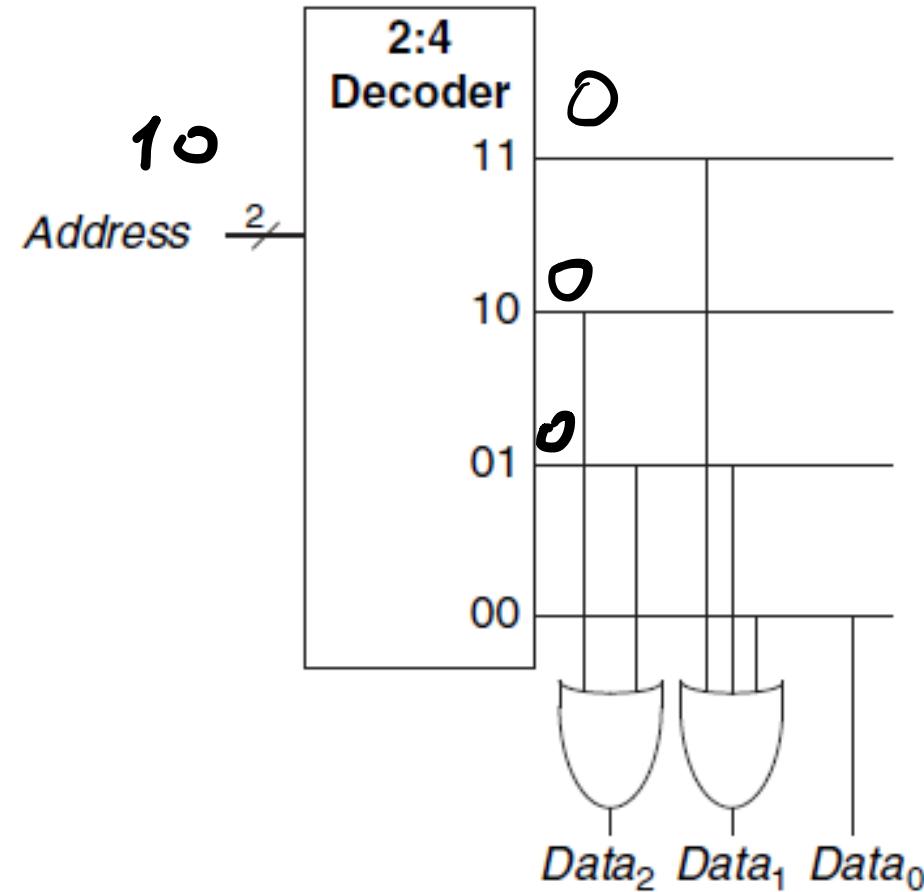


| Address | Data |   |   |
|---------|------|---|---|
| 11      | 0    | 1 | 0 |
| 10      | 1    | 0 | 0 |
| 01      | 1    | 1 | 0 |
| 00      | 0    | 1 | 1 |

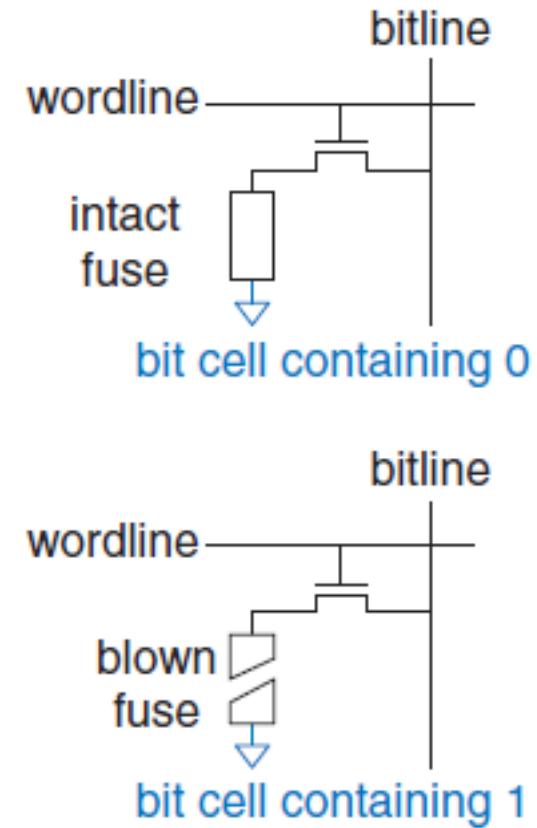
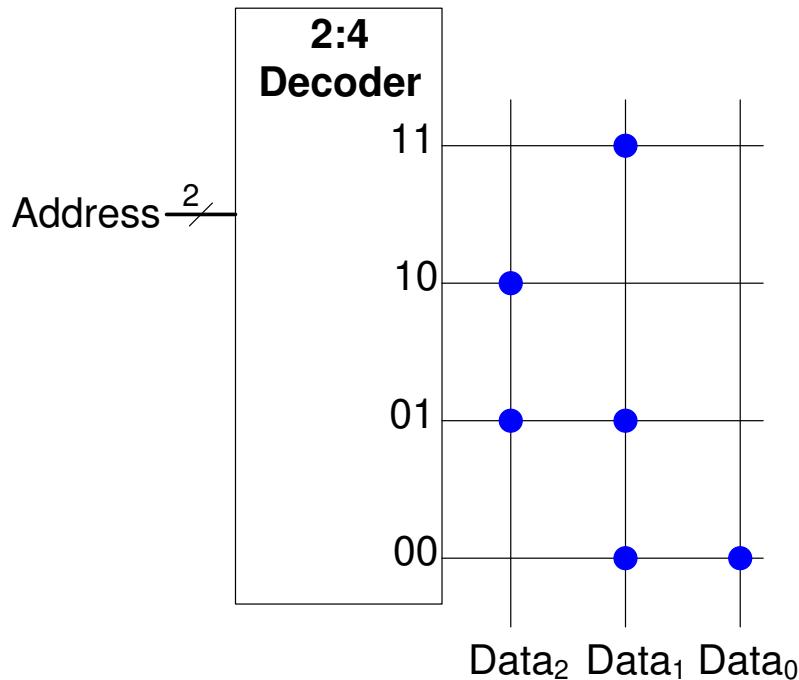
depth

width

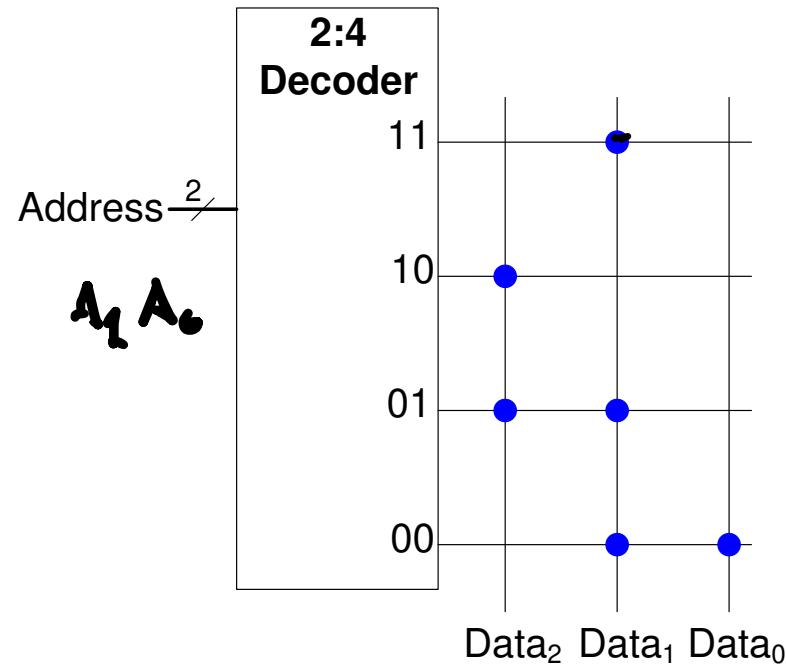
# ROM via porte logiche



# Fuse Programmable ROM



# ROM Logic



$$Data_2 = A_1 \oplus A_0$$

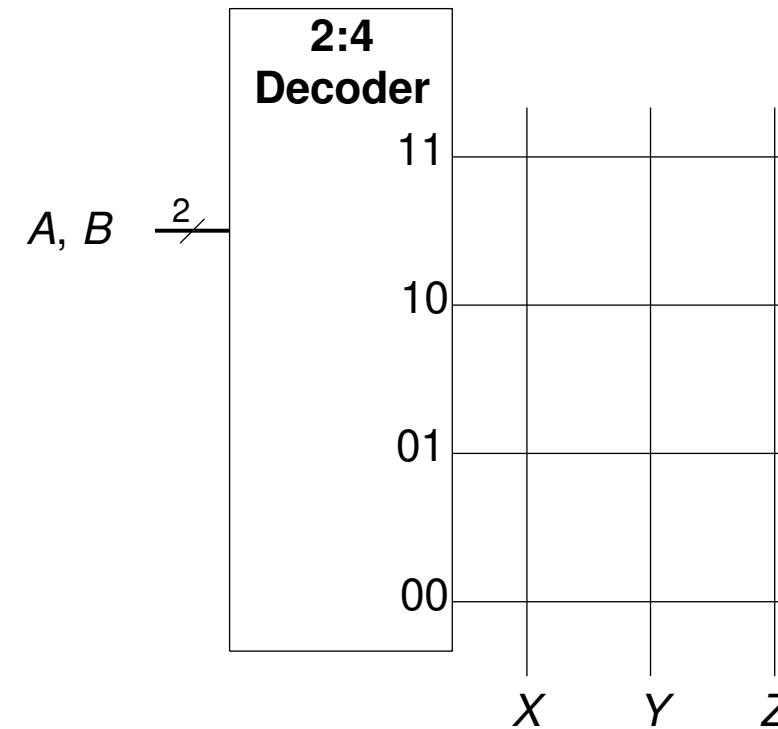
$$Data_1 = \overline{A}_1 + A_0$$

$$Data_0 = \overline{\overline{A}}_1 \overline{\overline{A}}_0$$

# Esempio

Usare un  $2^2 \times 3$ -bit ROM per implementare funzioni booleane:

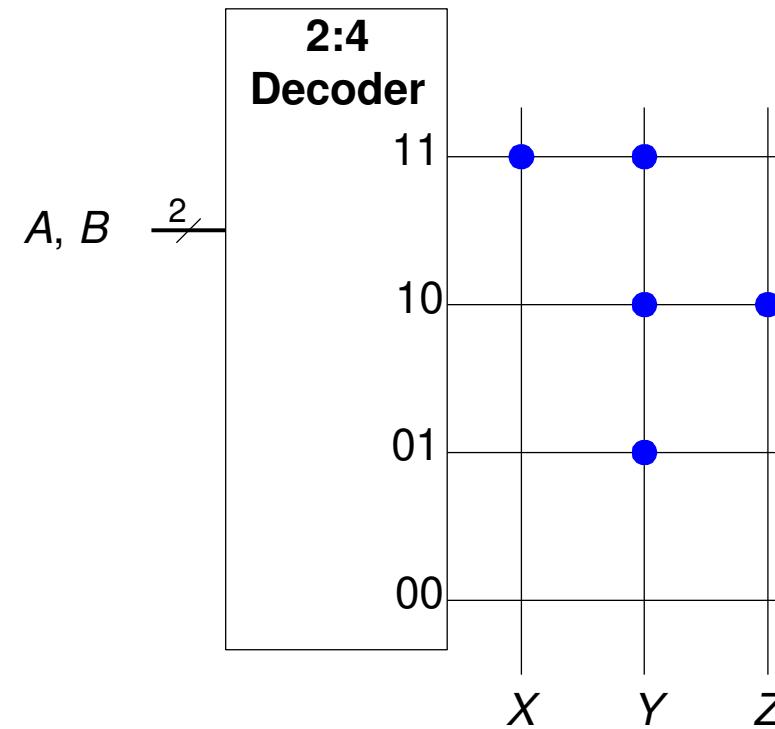
- $X = AB$
- $Y = A + B$
- $Z = A \bar{B}$



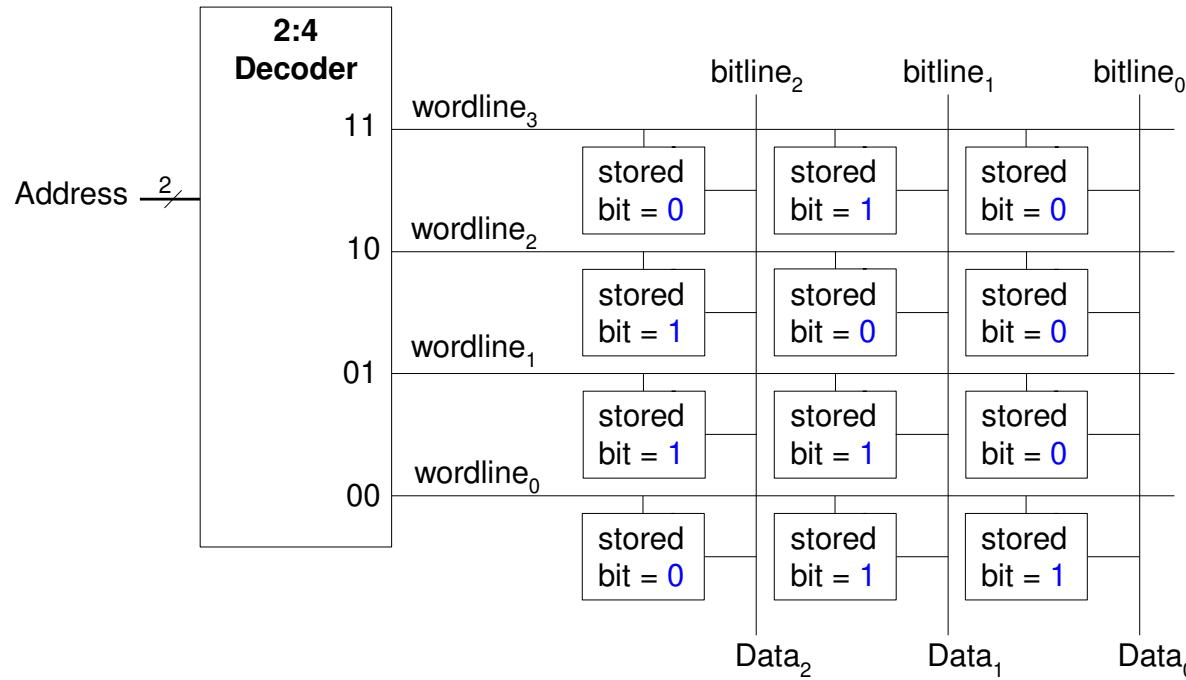
# Esempio

Usare un  $2^2 \times 3$ -bit ROM per implementare funzioni booleane:

- $X = AB$
- $Y = A + B$
- $Z = A \bar{B}$



# *Lookup tables (LUTs)*



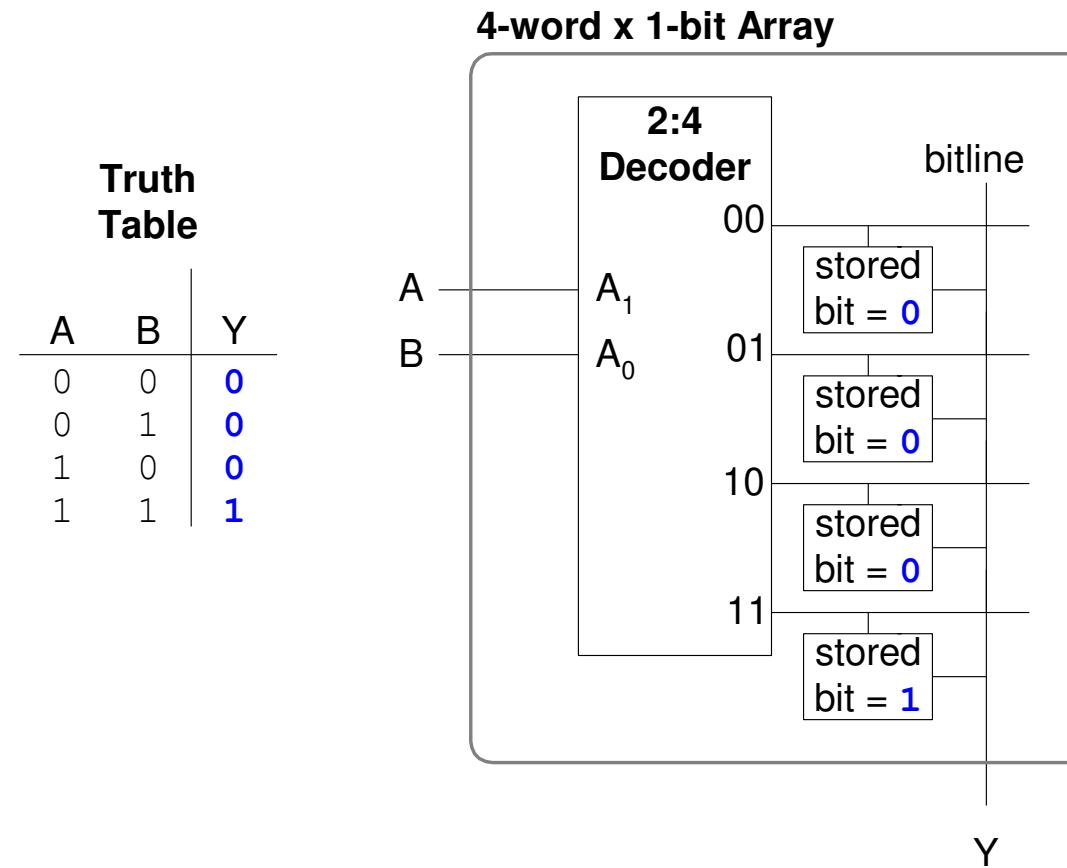
$$Data_2 = A_1 \oplus A_0$$

$$Data_1 = \bar{A}_1 + A_0$$

$$Data_0 = \bar{A}_1 \bar{A}_0$$

# Logiche con memory array

Sono chiamate *lookup tables* (LUTs): si “guardano” gli output per ogni combinazione di input (indirizzo)

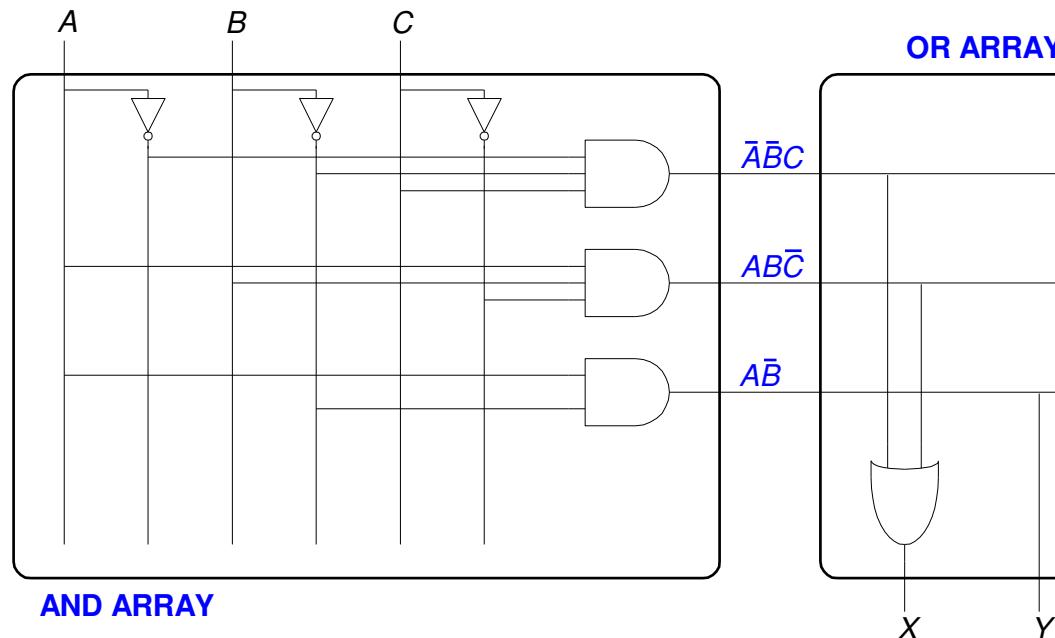
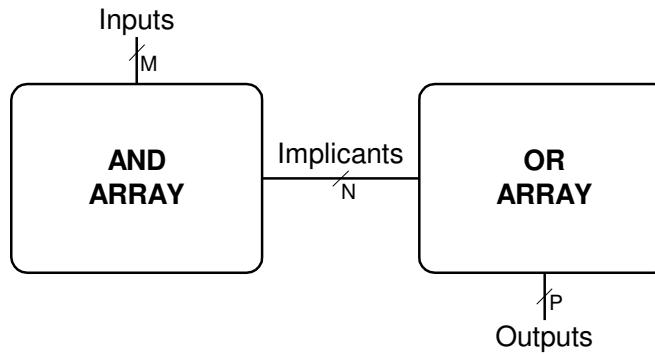


# Logic Arrays

- **PLAs** (Programmable logic arrays)
  - AND array seguito da un OR array
  - solo circuiti combinatori (SOP)
  - Le connessioni interne sono fisse
- **FPGAs** (Field programmable gate arrays)
  - Array elementi logici (LE)
  - Circuiti combinatori e sequenziali
  - Programmable internal connections

# PLAs

- $X = \overline{A}\overline{B}C + A\overline{B}\overline{C}$
- $Y = AB$



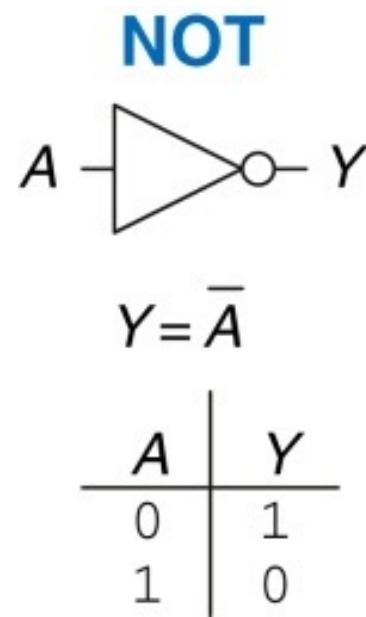
# Algebra di Boole e reti combinatorie

# Porte logiche

- Come accennato, un calcolatore può essere visto come un complesso sistema digitale che manipola e memorizza informazioni rappresentate in codice *binario*.
- I componenti digitali che costituiscono i mattoni fondamentali di un calcolatore sono le *porte logiche*
- Le porte logiche realizzano delle semplici operazioni che prendono uno o più input e producono un output
- Gli input sono indicati generalmente con le prime lettere dell'alfabeto A,B,C,D,... mentre gli output con le ultime X,Y,...
- Poiché gli input e gli output possono assumere in generale sia il valore 0 che 1 allora essi costituiscono delle variabili (dette variabili booleane)

# Porta NOT

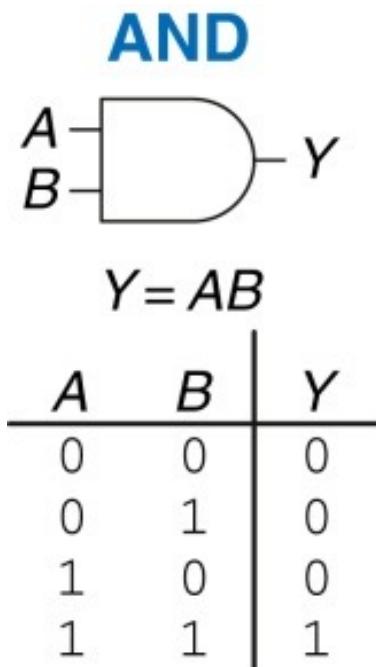
- La porta NOT restituisce in output il *complemento* dell'input:



- Y è uguale a 1 se e solo se A **non** è uguale a 1:  $Y=1-A$
- Altri simboli per NOT sono  $\neg A$  o  $\sim A$  (usati soprattutto dai logici)

# Porta AND

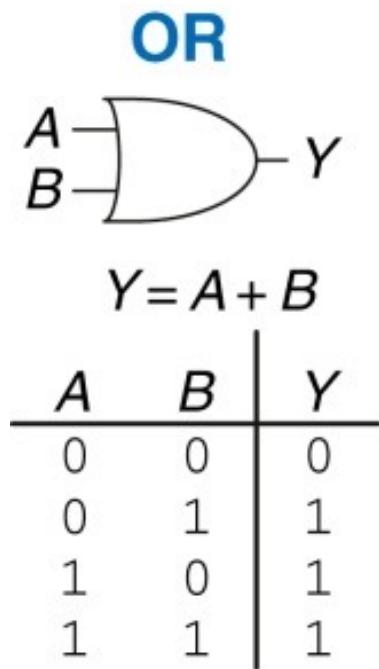
- La porta AND restituisce in output la *congiunzione* degli input:



- Y è uguale a 1 se e solo se A **e** B sono entrambi uguali a 1
- L'operatore AND è anche rappresentato con  $A \cdot B$  o  $A \wedge B$

# Porta OR

- La porta OR restituisce in output la *disgiunzione* degli input:

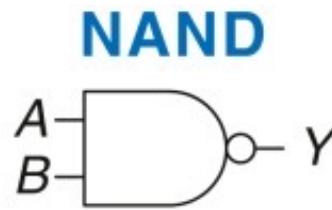


- Y è uguale a 1 se e solo se A **o** B è uguale a 1
- L'operatore OR è anche rappresentato con  $A+B$  o  $A \vee B$

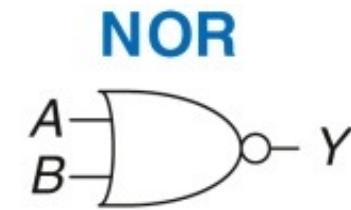
# Altre porte logiche



$$Y = A \oplus B$$



$$Y = \overline{AB}$$



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

| A | B | Y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

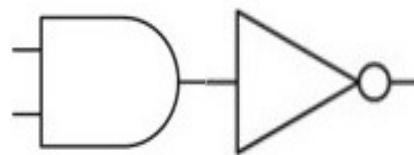
| A | B | Y |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

| A | B | Y |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

- XOR:  $Y = 1$  se e solo se A **oppure** B è uguale a 1 (equiv.  $A \neq B$ )
- NAND: Y è uguale a 1 se e solo se A **o** B **non** sono uguali a 1
- NOR: Y è uguale a 1 se e solo se A **e** B **non** sono uguali a 1

# Altre porte logiche

- La porta NAND è un elemento circuitale che viene prodotto direttamente, tuttavia può essere ottenuto complementando una porta AND, ovvero mettendo *in serie* una porta AND e una NOT:



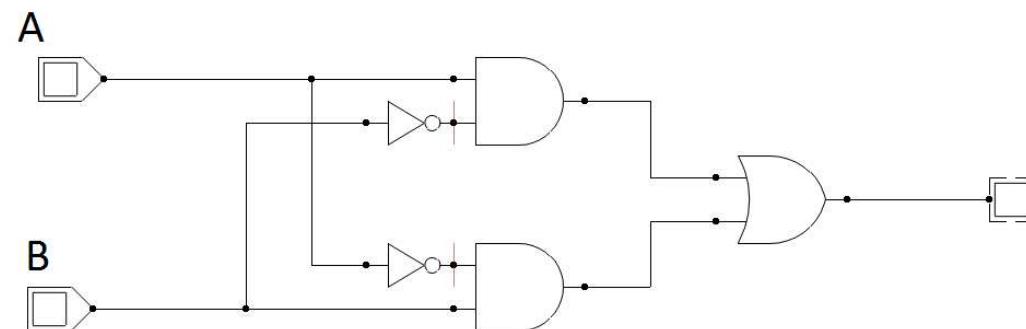
| A | B | $A \cdot B$ | $\overline{A \cdot B}$ |
|---|---|-------------|------------------------|
| 0 | 0 | 0           | 1                      |
| 0 | 1 | 0           | 1                      |
| 1 | 0 | 0           | 1                      |
| 1 | 1 | 1           | 0                      |

- Analogamente la porta NOR si può ottenere mettendo *in serie* una porta OR e una NOT:



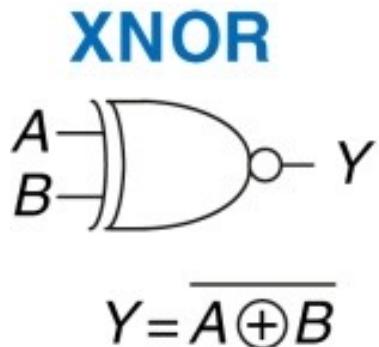
# Altre porte logiche

- Anche la porta XOR può essere ottenuta mediante porte AND, OR e NOT.
  - XOR:  $Y = 1$  sse  $A \neq B$
  - $A \neq B$  sse  $(A=1$  e  $B=0)$  o  $(A=0$  e  $B=1)$
  - $Y=(A \cdot \bar{B}) + (\bar{A} \cdot B)$



# XNOR

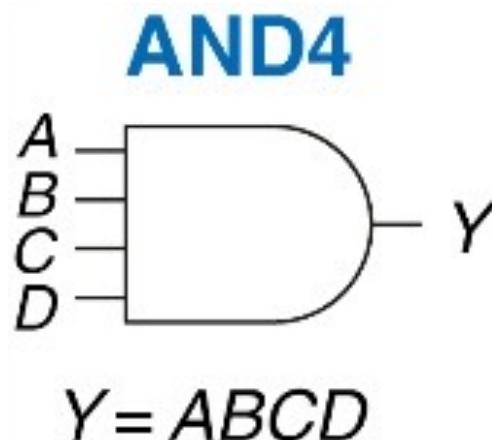
- Problema: quale è la tabella di verità della porta XNOR?



| $A$ | $B$ | $A \oplus B$ | $\overline{A \oplus B}$ |
|-----|-----|--------------|-------------------------|
| 0   | 0   | 0            | 1                       |
| 0   | 1   | 1            | 0                       |
| 1   | 0   | 1            | 0                       |
| 1   | 1   | 0            | 1                       |

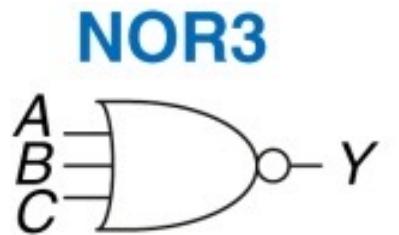
# Porte logiche con più linee di input

- Le porte logiche AND, OR, NAND,... possono avere anche più di 2 linee di ingresso.



| A | B | C | D | Y |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 |

# Porte logiche con più linee di input



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

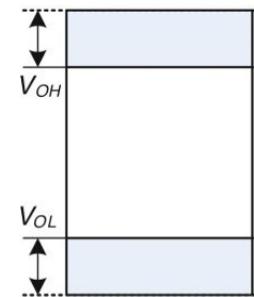
| <i>A</i> | <i>B</i> | <i>C</i> | <i>Y</i> |
|----------|----------|----------|----------|
| 0        | 0        | 0        | 1        |
| 0        | 0        | 1        | 0        |
| 0        | 1        | 0        | 0        |
| 0        | 1        | 1        | 0        |
| 1        | 0        | 0        | 0        |
| 1        | 0        | 1        | 0        |
| 1        | 1        | 0        | 0        |
| 1        | 1        | 1        | 0        |

# Cosa c'è sotto?

- Finora abbiamo descritto le porte logiche in termini astratti in cui gli ingressi e le uscite assumono i valori binari 0 o 1.
- Chiaramente, per realizzare queste porte in concreto, questi valori devono corrispondere a una certa grandezza fisica.
- Nei calcolatori elettronici (questo laptop per intenderci) la grandezza fisica che *reifica* i valori logici 0 e 1 è il potenziale elettrico.
- Daremo per scontato che abbiate le nozioni minime di base su cosa sia il potenziale elettrico (o che almeno non mettiate le dita in una presa da 220V).
- Il valore logico 0 è rappresentato dal valore di potenziale di 0V (GRD)
- Il valore logico 1 è rappresentato da un valore di potenziale  $V_{DD}$  fornito dal generatore.
  - Fino agli anni ottanta  $V_{DD} = 5V$ , l'avvento di portatili, tablet e smartphone ha reso necessario operare con valori di potenziale più bassi  $V_{DD} \leq 1,5V$

# Cosa c'è sotto?

- Notate che il potenziale elettrico è una grandezza fisica *continua* con cui rappresentiamo dei valori discreti (0 e 1) mediante le grandezze nominali GRD e  $V_{DD}$ .
- Come ogni sistema fisico reale, un circuito digitale è soggetto a del *rumore* che può alterare i valori nominali su cui operiamo. Per questo i componenti digitali hanno delle soglie di tolleranza  $GRD - V_{OL}$  e  $V_{OH} - V_{DD}$ .



- Per un corretto funzionamento di un circuito, il rumore non deve eccedere le soglie di tolleranza di tutti i suoi componenti.

# Funzioni booleane

- Le porte logiche esaminate finora costituiscono delle specifiche *funzioni booleane*

$$f:\{0,1\}^N \rightarrow \{0,1\}$$

- Domanda: quante funzioni booleane di N variabili esistono?

| $A_N A_{N-1} \dots A_1 A_0$ | $y$       |
|-----------------------------|-----------|
| 0 0 ... 0 0                 | $y_0$     |
| 0 0 ... 0 1                 | $y_1$     |
| 0 0 ... 1 0                 | $y_2$     |
| 0 0 ... 1 1                 | $y_3$     |
| .                           | .         |
| .                           | .         |
| .                           | .         |
| 1 1 ... 1 1                 | $y_{2^N}$ |

$2^N$  righe

Una generica  $f$  assegna «liberamente» valori 0 o 1 alle  $y_i$ .  
Quindi il numero di funzioni booleane di  $N$  variabili è pari al numero di parole binarie di lunghezza  $2^N$ , ovvero:

$$2^{2^N}$$

# Funzioni booleane di 2 variabili

Se le variabili sono 2 allora ottengo  $2^4 = 16$  possibili funzioni booleane

| A B | f <sub>0</sub> | f <sub>1</sub> | f <sub>2</sub> | f <sub>3</sub> | f <sub>4</sub> | f <sub>5</sub> | f <sub>6</sub> | f <sub>7</sub> | f <sub>8</sub> | f <sub>9</sub> | f <sub>10</sub> | f <sub>11</sub> | f <sub>12</sub> | f <sub>13</sub> | f <sub>14</sub> | f <sub>15</sub> |
|-----|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 0 0 | 0              | 0              | 0              | 0              | 0              | 0              | 0              | 0              | 1              | 1              | 1               | 1               | 1               | 1               | 1               | 1               |
| 0 1 | 0              | 0              | 0              | 0              | 1              | 1              | 1              | 1              | 0              | 0              | 0               | 0               | 1               | 1               | 1               | 1               |
| 1 0 | 0              | 0              | 1              | 1              | 0              | 0              | 1              | 1              | 0              | 0              | 1               | 1               | 0               | 0               | 1               | 1               |
| 1 1 | 0              | 1              | 0              | 1              | 0              | 1              | 0              | 1              | 0              | 1              | 0               | 1               | 0               | 1               | 0               | 1               |

AND                    XOR                    OR                    NOR                    XNOR                    NAND

# Funzioni booleane di 2 variabili

Se le variabili sono 2 allora ottengo  $2^4 = 16$  possibili funzioni booleane

| A B | $f_0$ | $f_1$ | $f_2$ | $f_3$ | $f_4$ | $f_5$ | $f_6$ | $f_7$ | $f_8$ | $f_9$ | $f_{10}$ | $f_{11}$ | $f_{12}$ | $f_{13}$ | $f_{14}$ | $f_{15}$ |
|-----|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|----------|----------|----------|----------|----------|----------|
| 0 0 | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 1     | 1     | 1     | 1        | 1        | 1        | 1        | 1        |          |
| 0 1 | 0     | 0     | 0     | 1     | 1     | 1     | 1     | 0     | 0     | 0     | 0        | 1        | 1        | 1        | 1        |          |
| 1 0 | 0     | 0     | 1     | 1     | 0     | 0     | 1     | 1     | 0     | 0     | 1        | 1        | 0        | 0        | 1        |          |
| 1 1 | 0     | 1     | 0     | 1     | 0     | 1     | 0     | 1     | 0     | 1     | 0        | 1        | 0        | 1        | 0        |          |

Funzione costante  
 $Y=0$

Funzione costante  
 $Y=1$

# Funzioni booleane di 2 variabili

Se le variabili sono 2 allora ottengo  $2^4 = 16$  possibili funzioni booleane

| A          | B | $f_0$ | $f_1$ | $f_2$ | $f_3$    | $f_4$ | $f_5$    | $f_6$ | $f_7$ | $f_8$ | $f_9$ | $f_{10}$ | $f_{11}$ | $f_{12}$ | $f_{13}$ | $f_{14}$ | $f_{15}$ |
|------------|---|-------|-------|-------|----------|-------|----------|-------|-------|-------|-------|----------|----------|----------|----------|----------|----------|
| <b>0 0</b> |   | 0     | 0     | 0     | <b>0</b> | 0     | <b>0</b> | 0     | 0     | 1     | 1     | <b>1</b> | 1        | 1        | 1        | 1        | 1        |
| <b>0 1</b> |   | 0     | 0     | 0     | <b>0</b> | 1     | <b>1</b> | 1     | 1     | 0     | 0     | <b>0</b> | 0        | 1        | 1        | 1        | 1        |
| <b>1 0</b> |   | 0     | 0     | 1     | <b>1</b> | 0     | <b>0</b> | 1     | 1     | 0     | 0     | <b>1</b> | 1        | 0        | 0        | 1        | 1        |
| <b>1 1</b> |   | 0     | 1     | 0     | <b>1</b> | 0     | <b>1</b> | 0     | 1     | 0     | 1     | <b>0</b> | 1        | 0        | 1        | 0        | 1        |

Y=A                          Y=B                          Y=̄A  
Y=̄B

# Funzioni booleane di 2 variabili

Se le variabili sono 2 allora ottengo  $2^4 = 16$  possibili funzioni booleane

| A B | f <sub>0</sub> | f <sub>1</sub> | f <sub>2</sub> | f <sub>3</sub> | f <sub>4</sub> | f <sub>5</sub> | f <sub>6</sub> | f <sub>7</sub> | f <sub>8</sub> | f <sub>9</sub> | f <sub>10</sub> | f <sub>11</sub> | f <sub>12</sub> | f <sub>13</sub> | f <sub>14</sub> | f <sub>15</sub> |
|-----|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 0 0 | 0              | 0              | 0              | 0              | 0              | 0              | 0              | 0              | 1              | 1              | 1               | 1               | 1               | 1               | 1               | 1               |
| 0 1 | 0              | 0              | 0              | 0              | 1              | 1              | 1              | 1              | 0              | 0              | 0               | 0               | 1               | 1               | 1               | 1               |
| 1 0 | 0              | 0              | 1              | 1              | 0              | 0              | 1              | 1              | 0              | 0              | 1               | 1               | 0               | 0               | 1               | 1               |
| 1 1 | 0              | 1              | 0              | 1              | 0              | 1              | 0              | 1              | 0              | 1              | 0               | 1               | 0               | 1               | 0               | 1               |

$$Y = A\bar{B}$$

$$Y = \bar{A}B$$

$$Y = \overline{\bar{A}\bar{B}} = A + \bar{B} \quad B \leq A$$

$$Y = \overline{AB} = \bar{A} + B \quad A \leq B$$

# Funzioni booleane di 2 variabili

Se le variabili sono 2 allora ottengo  $2^4 = 16$  possibili funzioni booleane

| A B | f <sub>0</sub> | f <sub>1</sub> | f <sub>2</sub> | f <sub>3</sub> | f <sub>4</sub> | f <sub>5</sub> | f <sub>6</sub> | f <sub>7</sub> | f <sub>8</sub> | f <sub>9</sub> | f <sub>10</sub> | f <sub>11</sub> | f <sub>12</sub> | f <sub>13</sub> | f <sub>14</sub> | f <sub>15</sub> |
|-----|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 0 0 | 0              | 0              | 0              | 0              | 0              | 0              | 0              | 1              | 1              | 1              | 1               | 1               | 1               | 1               | 1               | 1               |
| 0 1 | 0              | 0              | 0              | 0              | 1              | 1              | 1              | 1              | 0              | 0              | 0               | 0               | 1               | 1               | 1               | 1               |
| 1 0 | 0              | 0              | 1              | 1              | 0              | 0              | 1              | 1              | 0              | 0              | 1               | 1               | 0               | 0               | 1               | 1               |
| 1 1 | 0              | 1              | 0              | 1              | 0              | 1              | 0              | 1              | 0              | 1              | 0               | 1               | 0               | 1               | 0               | 1               |

Y=A

Y=  $\bar{B}$

Y=  $\overline{\bar{A}B} = A + \bar{B}$

# Funzioni booleane di 2 variabili

Se le variabili sono 2 allora ottengo  $2^4 = 16$  possibili funzioni booleane

| A B | f <sub>0</sub> | f <sub>1</sub> | f <sub>2</sub> | f <sub>3</sub> | f <sub>4</sub> | f <sub>5</sub> | f <sub>6</sub> | f <sub>7</sub> | f <sub>8</sub> | f <sub>9</sub> | f <sub>10</sub> | f <sub>11</sub> | f <sub>12</sub> | f <sub>13</sub> | f <sub>14</sub> | f <sub>15</sub> |
|-----|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 0 0 | 0              | 0              | 0              | 0              | 0              | 0              | 0              | 0              | 1              | 1              | 1               | 1               | 1               | 1               | 1               | 1               |
| 0 1 | 0              | 0              | 0              | 0              | 1              | 1              | 1              | 1              | 0              | 0              | 0               | 0               | 1               | 1               | 1               | 1               |
| 1 0 | 0              | 0              | 1              | 1              | 0              | 0              | 1              | 1              | 0              | 0              | 1               | 1               | 0               | 0               | 1               | 1               |
| 1 1 | 0              | 1              | 0              | 1              | 0              | 1              | 0              | 1              | 0              | 1              | 0               | 1               | 0               | 1               | 0               | 1               |

Y=B

Y=  $\bar{A}$

Y=  $\overline{AB} = \bar{A} + B$

# Funzioni booleane di 2 variabili

- Se le variabili sono 2 allora ottengo 16 possibili funzioni booleane

| A B | $f_0$ | $f_1$ | $f_2$ | $f_3$ | $f_4$ | $f_5$ | $f_6$ | $f_7$ | $f_8$ | $f_9$ | $f_{10}$ | $f_{11}$ | $f_{12}$ | $f_{13}$ | $f_{14}$ | $f_{15}$ |
|-----|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|----------|----------|----------|----------|----------|----------|
| 0 0 | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 1     | 1     | 1     | 1        | 1        | 1        | 1        | 1        | 1        |
| 0 1 | 0     | 0     | 0     | 0     | 1     | 1     | 1     | 1     | 0     | 0     | 0        | 0        | 1        | 1        | 1        | 1        |
| 1 0 | 0     | 0     | 1     | 1     | 0     | 0     | 1     | 1     | 0     | 0     | 1        | 1        | 0        | 0        | 1        | 1        |
| 1 1 | 0     | 1     | 0     | 1     | 0     | 1     | 0     | 1     | 0     | 1     | 0        | 1        | 0        | 1        | 0        | 1        |

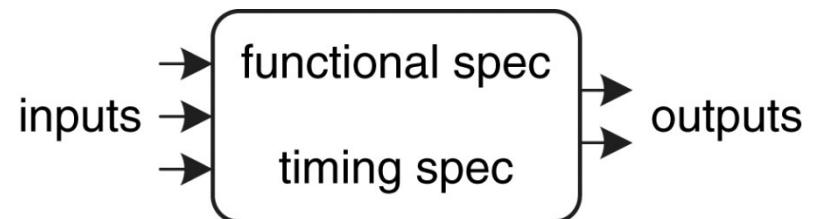
$$Y = (A \leq B) \cdot (B \leq A) \quad A \equiv B$$

$$Y = B \leq A$$

$$Y = A \leq B$$

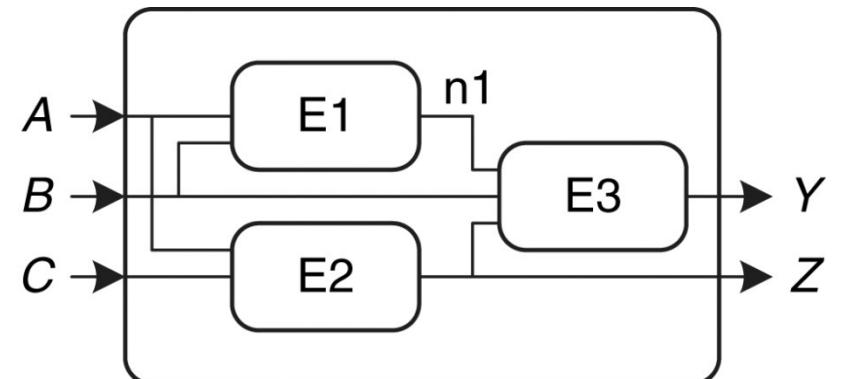
# Circuiti digitali

- In generale un circuito digitale è una rete che elabora segnali discreti (rappresentati da variabili booleane). Prescindendo dalla sua configurazione interna, un circuito può essere visto come una *black-box* con
  - Uno o più input
  - Uno o più output
  - Una *specifica funzionale* che rappresenta la relazione fra input e output
  - Una *specifica temporale* che descrive il ritardo che intercorre affinché i segnali di input si propaghino nel circuito fino agli output.



# Circuiti digitali

- La struttura interna di un circuito è composta da *elementi* e *nodi*.
- Un elemento è esso stesso un circuito digitale.
- Un nodo è una connessione che trasporta il segnale (e.g. filo elettrico). I nodi si distinguono in nodi di input, output e interni
  - input: riceve il segnale dal mondo esterno
  - output: riporta il segnale al mondo esterno
  - Nodo interno: connette due elementi

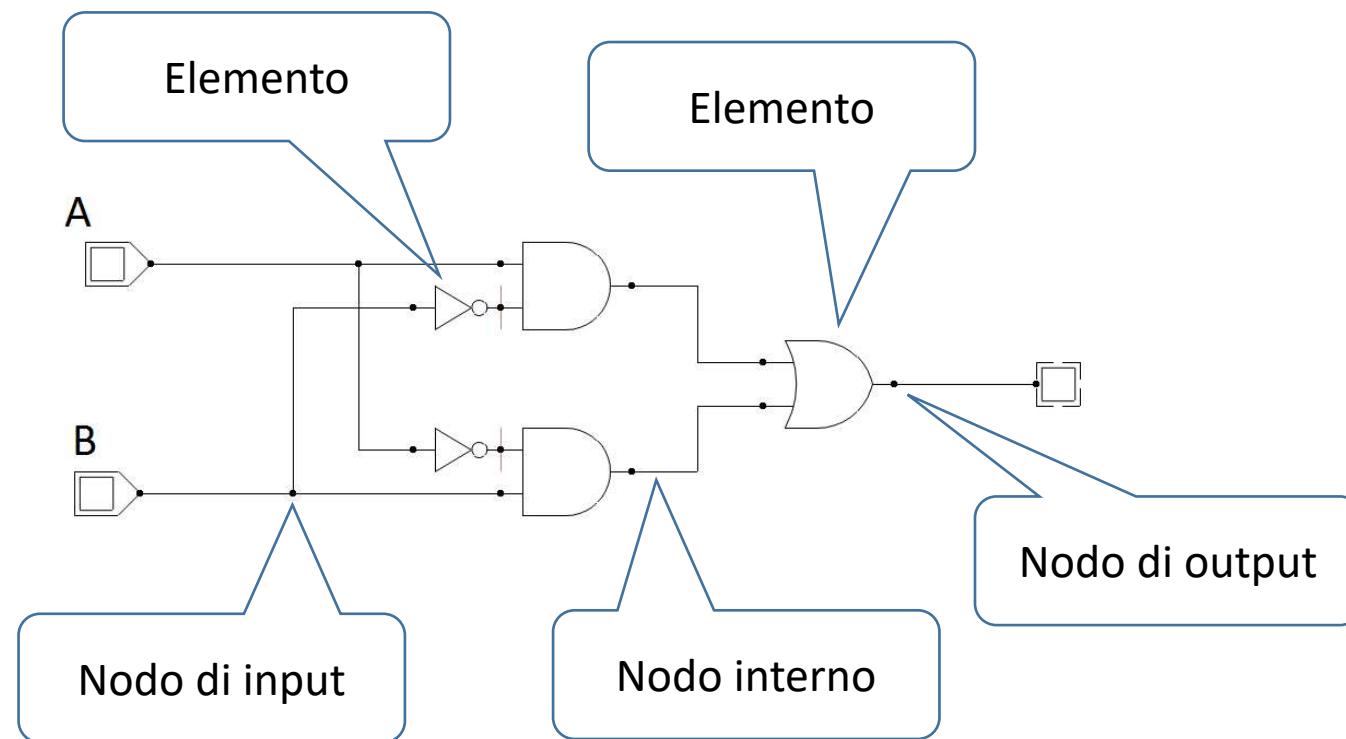


# Reti combinatorie e sequenziali

- Vi sono due grandi categorie di circuiti digitali: le reti combinatorie e le reti sequenziali.
- In una rete combinatoria, i valori degli output dipendono esclusivamente dal valore corrente degli input (al netto dei ritardi di propagazione). In tal senso, le reti combinatorie si dicono memoryless, ovvero non hanno memoria della “*storia*” precedente del circuito
- In un rete sequenziale, invece, gli output dipendono non solo dal valore corrente degli input, ma anche dai valori precedenti. Si dice quindi che il circuito ha memoria.

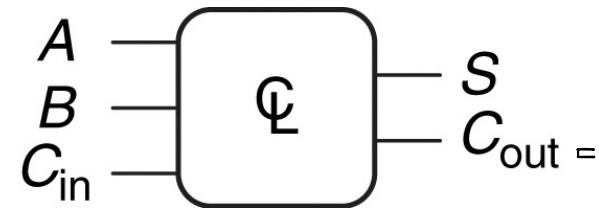
# Reti combinatorie

- Le porte logiche viste finora sono un esempio di rete combinatoria.
- Un'altro esempio di rete combinatoria l'abbiamo visto in precedenza



# Full adder

- Quando non siamo interessati alla struttura interna allora una rete combinatoria è descritta come segue

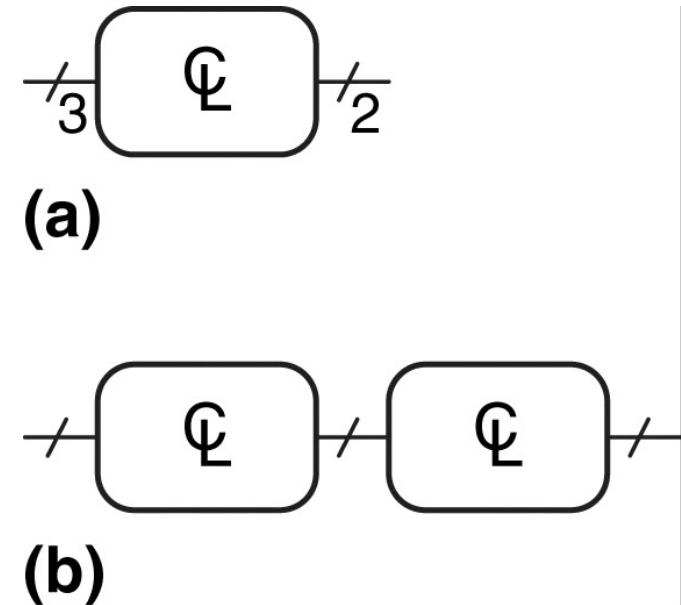


$$S = A \oplus B \oplus C_{\text{in}}$$
$$C_{\text{out}} = AB + AC_{\text{in}} + BC_{\text{in}}$$

- Questa rete combinatoria rappresenta un full adder di cui parleremo in seguito.
- Come avrete già capito, vi sono molti modi differenti per realizzare le funzioni che corrispondono al output S e C<sub>out</sub>

# Input e output multipli

- Per semplificare la rappresentazione grafica, si usa la notazione qui di fianco per indicare ingressi ed uscite multiple.
- In figura (a) un generico circuito combinatorio con 3 input e 2 output
- Il numero di input e output può essere omesso quando chiaro dal contesto o non rilevante. In figura (b) due generiche reti combinatorie in sequenza. Notate che il numero di output della prima rete deve essere uguale al numero di input della seconda.

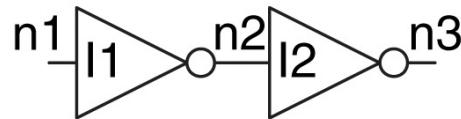


# Regole di composizione di reti combinatorie

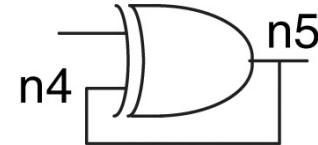
- Come abbiamo visto reti combinatorie possono essere composte per formare reti più grandi. Cosa ci garantisce che il risultato sia ancora una rete combinatoria? Ecco un insieme di regole che costituiscono delle condizioni sufficienti (ma non necessarie)
  - Ogni elemento è esso stesso una rete combinatoria
  - Ogni nodo che non è un input connette esattamente un output di un elemento
  - Il circuito non contiene cicli, ogni cammino interno alla rete visita un nodo al più una volta

# Regole di composizione di reti combinatorie

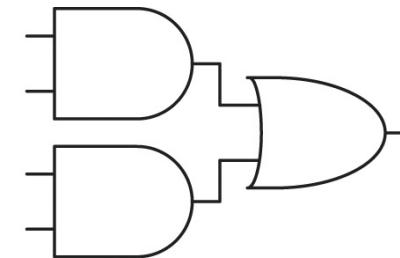
- Quali dei seguenti circuiti costituisce, secondo le regole di composizione, una rete combinatoria?



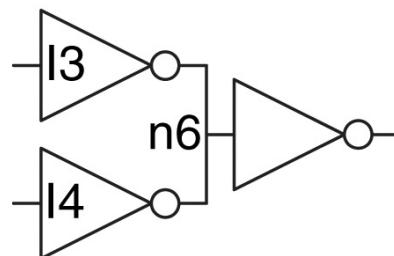
(a)



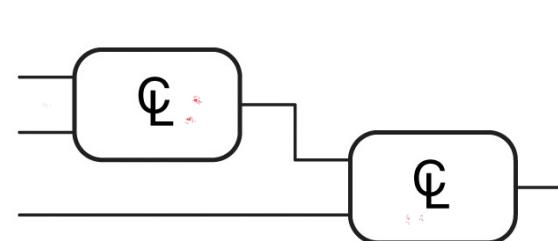
(b)



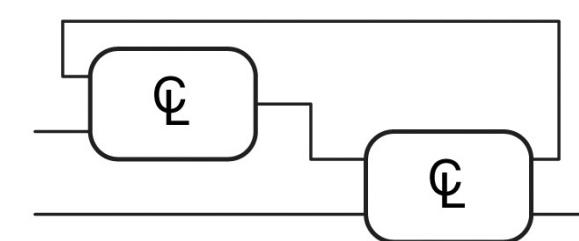
(c)



(d)



(e)



(f)

# Tabella di verità e espressioni booleane

- Le specifiche funzionali di un circuito combinatorio sono descritte tramite tabelle di verità o espressioni booleane
- In generale, più espressioni booleane corrispondono alla stessa tabella di verità
  - Per esempio abbiamo visto che  $\overline{AB}$  e  $\bar{A} + B$  corrispondono alla stessa tabella di verità, ovvero  $\overline{AB} = \bar{A} + B$ .
- Come si ricava la tabella di verità di una data espressione?
- Viceversa, a partire da una tabella di verità, come si ricava una espressione corrispondente?

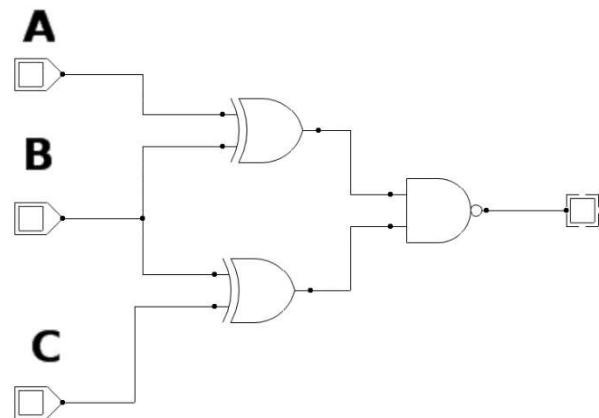
# Dalle espressioni alle tavole

- Il calcolo di una tabella di verità a partire una espressione booleana non è dissimile concettualmente dal calcolo di una espressione aritmetica. Si calcolano a ritroso le tavole di verità delle sue sotto espressioni.
- Consideriamo l'espressione  $(\bar{A}B) + (B \oplus C)$

| A | B | C | $\bar{A}$ | $\bar{A} \cdot B$ | $B \oplus C$ | $(\bar{A}B) + (B \oplus C)$ |
|---|---|---|-----------|-------------------|--------------|-----------------------------|
| 0 | 0 | 0 | 1         | 0                 | 0            | 0                           |
| 0 | 0 | 1 | 1         | 0                 | 1            | 1                           |
| 0 | 1 | 0 | 1         | 1                 | 1            | 1                           |
| 0 | 1 | 1 | 1         | 1                 | 0            | 1                           |
| 1 | 0 | 0 | 0         | 0                 | 0            | 0                           |
| 1 | 0 | 1 | 0         | 0                 | 1            | 1                           |
| 1 | 1 | 0 | 0         | 0                 | 1            | 1                           |
| 1 | 1 | 1 | 0         | 0                 | 0            | 0                           |

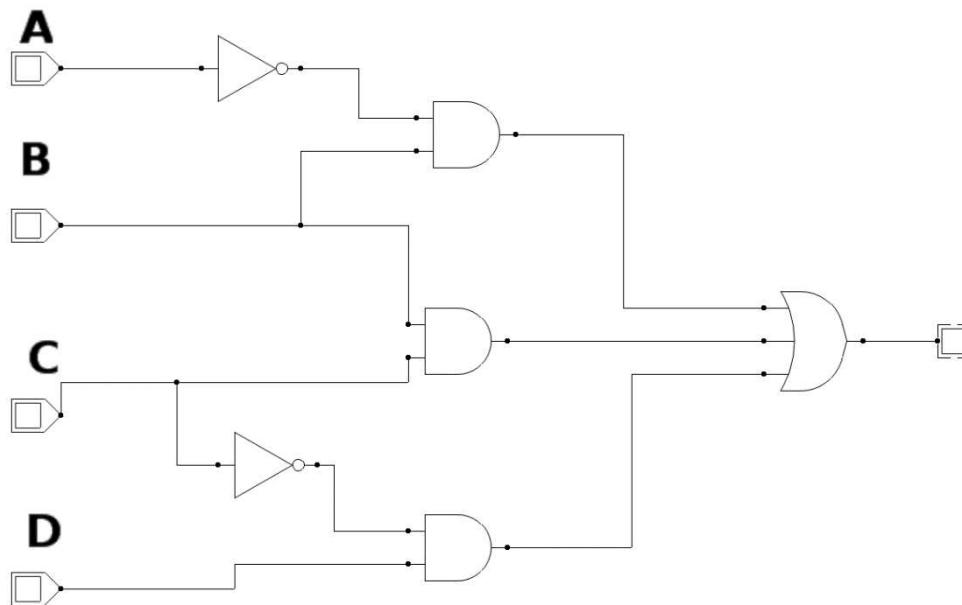
# Esercizi

- Dimostrare che  $(A \cdot \bar{B}) + (\bar{A} \cdot B)$  è equivalente alla porta XOR calcolandone la tabella di verità
- Scrivere l'espressione booleana e la relativa tabella di verità della rete:



# Esercizi

- Scrivere l'espressione booleana e la relativa tabella di verità della rete:



# Dalle tavole alle espressioni

- ricavare da una tabella di verità un'espressione booleana
- forma sintattica ben precisa: **SOP** (somma di prodotti)
- Definizioni preliminari:
  - Una variabile booleana  $A$  o la sua negata  $\bar{A}$  sono detti **letterali**
  - Un prodotto (AND) di letterali è detto **implicante**:  $\bar{A}B$ ,  $\bar{A}B\bar{C}$ ,  $B$  sono tutti implicanti
  - Dato un insieme  $K$  di variabili booleane, un **mintermine** di  $K$  è un implicante che comprende (positive oppure negative) tutte le variabili in  $K$ .
    - Ad esempio, se  $K = \{A, B, C\}$  allora  $\bar{A}B\bar{C}$  è un mintermine per  $K$ . Invece,  $\bar{A}B$  è un implicante ma non un mintermine.
    - Analogamente, un **maxtermine** di  $K$  è una somma di letterali in cui occorrono tutte le variabili in  $K$ .
      - Ad esempio  $\bar{A} + \bar{B} + \bar{C}$  è un maxtermine di  $K = \{A, B, C\}$ .
  - ordine di precedenza NOT → AND → OR
    - $\bar{A}B\bar{C} + CB = (\bar{A}B\bar{C}) + (CB)$

# La forma SOP

- Ognuna delle  $2^N$  righe di una tabella di verità è caratterizzata da un mintermone

| A B | f <sub>0</sub> | f <sub>1</sub> | f <sub>2</sub> | f <sub>3</sub> | f <sub>4</sub> | f <sub>5</sub> | f <sub>6</sub> | f <sub>7</sub> | f <sub>8</sub> | f <sub>9</sub> | f <sub>10</sub> | f <sub>11</sub> | f <sub>12</sub> | f <sub>13</sub> | f <sub>14</sub> | f <sub>15</sub> |
|-----|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 0 0 | 0              | 0              | 0              | 0              | 0              | 0              | 0              | 0              | 1              | 1              | 1               | 1               | 1               | 1               | 1               | 1               |
| 0 1 | 0              | 0              | 0              | 0              | 1              | 1              | 1              | 1              | 0              | 0              | 0               | 0               | 1               | 1               | 1               | 1               |
| 1 0 | 0              | 0              | 1              | 1              | 0              | 0              | 1              | 1              | 0              | 0              | 1               | 1               | 0               | 0               | 1               | 1               |
| 1 1 | 0              | 1              | 0              | 1              | 0              | 1              | 0              | 1              | 0              | 1              | 0               | 1               | 0               | 1               | 0               | 1               |

Diagram showing the extraction of minterms from the truth table:

- Minterm  $f_0$  corresponds to  $Y = AB$
- Minterm  $f_4$  corresponds to  $Y = A\bar{B}$
- Minterm  $f_9$  corresponds to  $Y = \bar{A}B$
- Minterm  $f_{15}$  corresponds to  $Y = \bar{A}\bar{B}$

# La forma SOP

- I mintermini sono enumerati rigo dopo rigo a partire da 0,1 e così via. Quindi ogni mintermine è denotato dal numero binario della configurazione di input corrispondente.

| A | B | Y | minterm           | minterm name |
|---|---|---|-------------------|--------------|
| 0 | 0 | 0 | $\bar{A} \bar{B}$ | $m_0$        |
| 0 | 1 | 1 | $\bar{A} B$       | $m_1$        |
| 1 | 0 | 0 | $A \bar{B}$       | $m_2$        |
| 1 | 1 | 0 | $A B$             | $m_3$        |

# La forma SOP

- Quindi ad ogni tabella di verità corrisponde una espressione booleana ottenuta sommando tutti i mintermini per cui il valore dell'output Y è pari a 1

| A | B | Y | minterm           | minterm name |                     |
|---|---|---|-------------------|--------------|---------------------|
| 0 | 0 | 0 | $\bar{A} \bar{B}$ | $m_0$        | $Y = \bar{A}B + AB$ |
| 0 | 1 | 1 | $\bar{A} B$       | $m_1$        | $Y = \Sigma(1,3)$   |
| 1 | 0 | 0 | $A \bar{B}$       | $m_2$        |                     |
| 1 | 1 | 1 | $A B$             | $m_3$        |                     |

# La forma SOP

- Consideriamo un esempio a tre variabili

| A | B | C | Y |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 |

$$Y = \bar{A}\bar{B}\bar{C} + A\bar{B}\bar{C} + A\bar{B}C$$

$$Y = \Sigma(0,4,5)$$

# La forma POS

- Una forma duale per rappresentare una funzione booleana è in forma **POS** (prodotto di somme).
- Ad ogni riga di una tabella di verità corrisponde un maxtermine che è uguale a 0 solo per quella riga

| A | B | f <sub>0</sub> | f <sub>1</sub> | f <sub>2</sub> | f <sub>3</sub> | f <sub>4</sub> | f <sub>5</sub> | f <sub>6</sub> | f <sub>7</sub> | f <sub>8</sub> | f <sub>9</sub> | f <sub>10</sub> | f <sub>11</sub> | f <sub>12</sub> | f <sub>13</sub> | f <sub>14</sub> | f <sub>15</sub> |
|---|---|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 0 | 0 | 0              | 0              | 0              | 0              | 0              | 0              | 0              | 0              | 1              | 1              | 1               | 1               | 1               | 1               | 1               | 1               |
| 0 | 1 | 0              | 0              | 0              | 0              | 1              | 1              | 1              | 1              | 0              | 0              | 0               | 0               | 1               | 1               | 1               | 1               |
| 1 | 0 | 0              | 0              | 1              | 1              | 0              | 0              | 1              | 1              | 0              | 0              | 1               | 1               | 0               | 0               | 1               | 1               |
| 1 | 1 | 0              | 1              | 0              | 1              | 0              | 1              | 0              | 1              | 0              | 1              | 0               | 1               | 0               | 1               | 0               | 1               |

Y = A + B      Y = A +  $\bar{B}$       Y =  $\bar{A}$  + B      Y =  $\bar{A}$  +  $\bar{B}$

# La forma POS

- Anche i maxtermini sono enumerati come i mintermini.

| A | B | Y | maxterm             | maxterm name |
|---|---|---|---------------------|--------------|
| 0 | 0 | 0 | $A + B$             | $M_0$        |
| 0 | 1 | 1 | $A + \bar{B}$       | $M_1$        |
| 1 | 0 | 0 | $\bar{A} + B$       | $M_2$        |
| 1 | 1 | 1 | $\bar{A} + \bar{B}$ | $M_3$        |

- La forma normale POS di una funzione booleana si ottiene come prodotto dei maxtermini per cui la funzione ritorna 0

- $Y = (A + B) \cdot (\bar{A} + B)$
- $Y = \prod(0,2)$

# La forma POS

- $Y = (A + B) \cdot (\bar{A} + B)$

| A B | f <sub>0</sub> | f <sub>1</sub> | f <sub>2</sub> | f <sub>3</sub> | f <sub>4</sub> | f <sub>5</sub> | f <sub>6</sub> | f <sub>7</sub> | f <sub>8</sub> | f <sub>9</sub> | f <sub>10</sub> | f <sub>11</sub> | f <sub>12</sub> | f <sub>13</sub> | f <sub>14</sub> | f <sub>15</sub> |
|-----|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 0 0 | 0              | 0              | 0              | 0              | 0              | 0              | 0              | 0              | 1              | 1              | 1               | 1               | 1               | 1               | 1               | 1               |
| 0 1 | 0              | 0              | 0              | 0              | 1              | 1              | 1              | 1              | 0              | 0              | 0               | 0               | 1               | 1               | 1               | 1               |
| 1 0 | 0              | 0              | 1              | 1              | 0              | 0              | 1              | 1              | 0              | 0              | 1               | 1               | 0               | 0               | 1               | 1               |
| 1 1 | 0              | 1              | 0              | 1              | 0              | 1              | 0              | 1              | 0              | 1              | 0               | 1               | 0               | 1               | 0               | 1               |

$\text{Y} = (A + B) \cdot (\bar{A} + B)$

$\text{Y} = A + B$

$\text{Y} = \bar{A} + B$

# Forme SOP & POS

## SOP – sum-of-products

| O | C | E | minterm           |
|---|---|---|-------------------|
| 0 | 0 | 0 | $\bar{O} \bar{C}$ |
| 0 | 1 | 0 | $\bar{O} C$       |
| 1 | 0 | 1 | $O \bar{C}$       |
| 1 | 1 | 0 | $O C$             |

$$E = O\bar{C}$$

$$= \Sigma(2)$$

## POS – product-of-sums

| O | C | E | maxterm             |
|---|---|---|---------------------|
| 0 | 0 | 0 | $O + C$             |
| 0 | 1 | 0 | $O + \bar{C}$       |
| 1 | 0 | 1 | $\bar{O} + C$       |
| 1 | 1 | 0 | $\bar{O} + \bar{C}$ |

$$E = (O + C)(O + \bar{C})(\bar{O} + C)$$
$$= \Pi(0, 1, 3)$$

# Forme SOP & POS

- Se la tabella di verità ha pochi 1 allora la forma SOP è più succinta della forma POS
- Se la tabella di verità ha pochi 0 allora la forma POS è più succinta della forma SOP
- Nel caso in cui il numero di 0 e 1 è pressappoco lo stesso le due forme si equivalgono

# Algebra di Boole

- Come abbiamo visto la medesima funzione può essere descritta da espressioni booleane distinte
- Alcune di queste possono essere più semplici di altre

| A | B | Y | minterm           | minterm name |
|---|---|---|-------------------|--------------|
| 0 | 0 | 0 | $\bar{A} \bar{B}$ | $m_0$        |
| 0 | 1 | 1 | $\bar{A} B$       | $m_1$        |
| 1 | 0 | 0 | $A \bar{B}$       | $m_2$        |
| 1 | 1 | 1 | $A B$             | $m_3$        |

$$Y = \bar{A}B + AB$$

$$Y = (A + B) \cdot (\bar{A} + B)$$

$$Y = B$$

- Come si fa con l'aritmetica, possiamo utilizzare un'algebra per semplificare le espressioni

$$\frac{1}{x} (x + xy) \Rightarrow \frac{x}{x} (1 + y) \Rightarrow (1 + y)$$

# Assiomi dell'algebra di Boole

| Axiom                           | Dual                             | Name         |
|---------------------------------|----------------------------------|--------------|
| A1 $B = 0 \text{ if } B \neq 1$ | A1' $B = 1 \text{ if } B \neq 0$ | Binary field |
| A2 $\bar{0} = 1$                | A2' $\bar{1} = 0$                | NOT          |
| A3 $0 \cdot 0 = 0$              | A3' $1 + 1 = 1$                  | AND/OR       |
| A4 $1 \cdot 1 = 1$              | A4' $0 + 0 = 0$                  | AND/OR       |
| A5 $0 \cdot 1 = 1 \cdot 0 = 0$  | A5' $1 + 0 = 0 + 1 = 1$          | AND/OR       |

# Assiomi dell'algebra di Boole

| Axiom                              | Dual                             | Name         |
|------------------------------------|----------------------------------|--------------|
| A1 $B = 0 \text{ if } B \neq 1$    | A1' $B = 1 \text{ if } B \neq 0$ | Binary field |
| A2 $\bar{0} = 1$                   | A2' $\bar{1} = 0$                | NOT          |
| A3 $0 \bullet 0 = 0$               | A3' $1 + 1 = 1$                  | AND/OR       |
| A4 $1 \bullet 1 = 1$               | A4' $0 + 0 = 0$                  | AND/OR       |
| A5 $0 \bullet 1 = 1 \bullet 0 = 0$ | A5' $1 + 0 = 0 + 1 = 1$          | AND/OR       |

- A1 e A1' ci dicono che il valore di una variabile booleana può essere 0 oppure 1

# Assiomi dell'algebra di Boole

| Axiom |                                 | Dual |                              | Name         |
|-------|---------------------------------|------|------------------------------|--------------|
| A1    | $B = 0 \text{ if } B \neq 1$    | A1'  | $B = 1 \text{ if } B \neq 0$ | Binary field |
| A2    | $\overline{0} = 1$              | A2'  | $\overline{1} = 0$           | NOT          |
| A3    | $0 \bullet 0 = 0$               | A3'  | $1 + 1 = 1$                  | AND/OR       |
| A4    | $1 \bullet 1 = 1$               | A4'  | $0 + 0 = 0$                  | AND/OR       |
| A5    | $0 \bullet 1 = 1 \bullet 0 = 0$ | A5'  | $1 + 0 = 0 + 1 = 1$          | AND/OR       |

- A1 e A1' ci dicono che il valore di una variabile booleana può essere 0 oppure 1
- A2 e A2' definiscono l'operatore NOT (di fatto questi assiomi ripropongono la tabella di verità dell'operatore)

# Assiomi dell'algebra di Boole

|    | Axiom                           |     | Dual                         | Name         |
|----|---------------------------------|-----|------------------------------|--------------|
| A1 | $B = 0 \text{ if } B \neq 1$    | A1' | $B = 1 \text{ if } B \neq 0$ | Binary field |
| A2 | $\bar{0} = 1$                   | A2' | $\bar{1} = 0$                | NOT          |
| A3 | $0 \bullet 0 = 0$               | A3' | $1 + 1 = 1$                  | AND/OR       |
| A4 | $1 \bullet 1 = 1$               | A4' | $0 + 0 = 0$                  | AND/OR       |
| A5 | $0 \bullet 1 = 1 \bullet 0 = 0$ | A5' | $1 + 0 = 0 + 1 = 1$          | AND/OR       |

- A3, A4 e A5 definiscono l'operatore AND
- A3', A4' e A5' definiscono l'operatore OR
- Notate che ogni assioma «primato» si ottiene dal corrispondente non primato invertendo, da un lato, OR e AND e dall'altro gli 0 e 1. Questo è un principio generale detto principio di dualità.

# Teoremi ad una variabile

| Theorem |                            | Dual                          | Name                   |
|---------|----------------------------|-------------------------------|------------------------|
| T1      | $B \cdot 1 = B$            | $T1'$                         | $B + 0 = B$            |
| T2      | $B \cdot 0 = 0$            | $T2'$                         | $B + 1 = 1$            |
| T3      | $B \cdot B = B$            | $T3'$                         | $B + B = B$            |
| T4      |                            | $\overline{\overline{B}} = B$ | Involution             |
| T5      | $B \cdot \overline{B} = 0$ | $T5'$                         | $B + \overline{B} = 1$ |

Teorema  $B \cdot 1 = B$

Dimostrazione:

- Supponiamo  $B=1$ , allora  $B \cdot 1 = 1 \cdot 1 = 1 = B$
- Supponiamo  $B=0$ , allora  $B \cdot 1 = 0 \cdot 1 = 0 = B$

Teorema  $B + \overline{B} = 1$

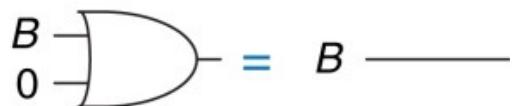
Dimostrazione:

- Supponiamo  $B=1$ , allora  $B + \overline{B} = 1 + 0 = 1$
- Supponiamo  $B=0$ , allora  $B + \overline{B} = 0 + 1 = 1$

# Corrispondenza hardware



(a)



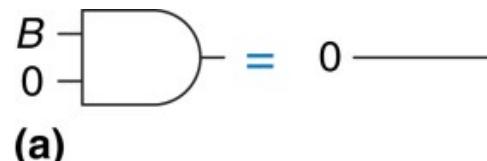
(b)



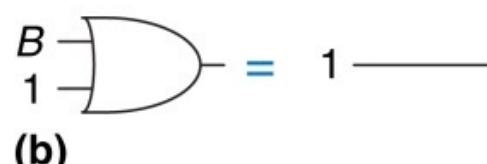
(a)



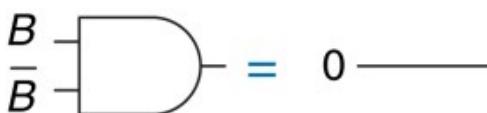
(b)



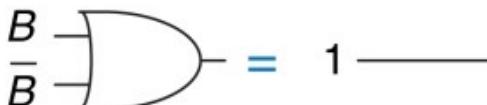
(a)



(b)



(a)



(b)

# Teoremi più variabili

| Number | Theorem                                                                                     | Name           |
|--------|---------------------------------------------------------------------------------------------|----------------|
| T6     | $B \bullet C = C \bullet B$                                                                 | Commutativity  |
| T7     | $(B \bullet C) \bullet D = B \bullet (C \bullet D)$                                         | Associativity  |
| T8     | $B \bullet (C + D) = (B \bullet C) + (B \bullet D)$                                         | Distributivity |
| T9     | $B \bullet (B+C) = B$                                                                       | Covering       |
| T10    | $(B \bullet C) + (B \bullet \bar{C}) = B$                                                   | Combining      |
| T11    | $(B \bullet C) + (\bar{B} \bullet D) + (C \bullet D) = (B \bullet C) + (\bar{B} \bullet D)$ | Consensus      |

# Teoremi più variabili: dualità

| #   | Theorem                                                                                          | Dual                                                                                 | Name           |
|-----|--------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------|----------------|
| T6  | $B \bullet C = C \bullet B$                                                                      | $B + C = C + B$                                                                      | Commutativity  |
| T7  | $(B \bullet C) \bullet D = B \bullet (C \bullet D)$                                              | $(B + C) + D = B + (C + D)$                                                          | Associativity  |
| T8  | $B \bullet (C + D) = (B \bullet C) + (B \bullet D)$                                              | $B + (C \bullet D) = (B + C)(B + D)$                                                 | Distributivity |
| T9  | $B \bullet (B + C) = B$                                                                          | $B + (B \bullet C) = B$                                                              | Covering       |
| T10 | $(B \bullet C) + (B \bullet \bar{C}) = B$                                                        | $(B + C) \bullet (B + \bar{C}) = B$                                                  | Combining      |
| T11 | $(B \bullet C) + (\bar{B} \bullet D) + (C \bullet D) =$<br>$(B \bullet C) + (\bar{B} \bullet D)$ | $(B + C) \bullet (\bar{B} + D) \bullet (C + D) =$<br>$(B + C) \bullet (\bar{B} + D)$ | Consensus      |

Principio di dualità:  $+ \leftrightarrow \cdot \quad 1 \leftrightarrow 0$

# Tecniche di dimostrazione

- Vi sono diversi metodi per dimostrare l'equivalenza di due espressioni.
  - Perfect induction: se le tabelle di verità di due espressioni coincidono allora le due espressioni sono equivalenti
  - usare assiomi e teoremi precedentemente provati per manipolare le espressioni fino ad ottenere espressioni uguali
  - Sfruttare la struttura delle espressioni e verificare che le tabelle di verità coincidono solo nei casi rilevanti

# Perfect induction: commutativity

| Number | Theorem                     | Name          |
|--------|-----------------------------|---------------|
| T6     | $B \bullet C = C \bullet B$ | Commutativity |

| $B$ | $C$ | $BC$ | $CB$ |
|-----|-----|------|------|
| 0   | 0   | 0    | 0    |
| 0   | 1   | 0    | 0    |
| 1   | 0   | 0    | 0    |
| 1   | 1   | 1    | 1    |

# Perfect induction: covering

| Number | Theorem               | Name     |
|--------|-----------------------|----------|
| T9     | $B \bullet (B+C) = B$ | Covering |

| $B$ | $C$ | $(B+C)$ | $B(B+C)$ |
|-----|-----|---------|----------|
| 0   | 0   | 0       | 0        |
| 0   | 1   | 1       | 0        |
| 1   | 0   | 1       | 1        |
| 1   | 1   | 1       | 1        |

# Perfect induction: consensus

| Number | Theorem                                                                                          | Name      |
|--------|--------------------------------------------------------------------------------------------------|-----------|
| T11    | $(B \bullet C) + (\bar{B} \bullet D) + (C \bullet D) =$<br>$(B \bullet C) + (\bar{B} \bullet D)$ | Consensus |

| B | C | D | $BC + \bar{B}D + CD$ | $BC + \bar{B}D$ |
|---|---|---|----------------------|-----------------|
| 0 | 0 | 0 | 0                    | 0               |
| 0 | 0 | 1 | 1                    | 1               |
| 0 | 1 | 0 | 0                    | 0               |
| 0 | 1 | 1 | 1                    | 1               |
| 1 | 0 | 0 | 0                    | 0               |
| 1 | 0 | 1 | 0                    | 0               |
| 1 | 1 | 0 | 1                    | 1               |
| 1 | 1 | 1 | 1                    | 1               |

# Limiti del perfect induction

- La tecnica del perfect induction è semplice ma «priva di intelligenza»
- Al crescere della lunghezza delle espressioni diventa sempre più laboriosa
- Al crescere delle variabili che occorrono nelle espressioni diventa estremamente più laboriosa
  - $4 \rightarrow 16$  checks
  - $5 \rightarrow 32$  checks
  - $6 \rightarrow 64$  checks
  - ...

# Proof teoretico: covering

| Number | Theorem               | Name     |
|--------|-----------------------|----------|
| T9     | $B \bullet (B+C) = B$ | Covering |

$$\begin{aligned} B \bullet (B+C) &= B \bullet B + B \bullet C && T8: \text{Distributivity} \\ &= B + B \bullet C && T3: \text{Idempotency} \\ &= B \bullet (1 + C) && T8: \text{Distributivity} \\ &= B \bullet (1) && T2: \text{Null element} \\ &= B && T1: \text{Identity} \end{aligned}$$

# Proof teoretico: combining

| Number | Theorem                                   | Name      |
|--------|-------------------------------------------|-----------|
| T10    | $(B \bullet C) + (B \bullet \bar{C}) = B$ | Combining |

$$\begin{aligned} B \bullet C + B \bullet \bar{C} &= B \bullet (C + \bar{C}) && \text{T8: Distributivity} \\ &= B \bullet (1) && \text{T5': Complements} \\ &= B && \text{T1: Identity} \end{aligned}$$

# Proof teoretico: consensus

| Number | Theorem                                                                                       | Name      |
|--------|-----------------------------------------------------------------------------------------------|-----------|
| T11    | $(B \bullet C) + (\bar{B} \bullet D) + (C \bullet D) =$ $(B \bullet C) + (\bar{B} \bullet D)$ | Consensus |

$$B \bullet C + \bar{B} \bullet D + C \bullet D$$

$$= BC + \bar{B}D + (CDB + CDB)$$

T10: Combining

$$= BC + \bar{B}D + BCD + \bar{B}CD$$

T6: Commutativity

$$= BC + BCD + \bar{B}D + \bar{B}CD$$

T6: Commutativity

$$= (BC + BCD) + (\bar{B}D + \bar{B}CD)$$

T7: Associativity

$$= BC + \bar{B}D$$

T9': Covering

# Per casi rilevanti

- Consideriamo T11

$$E: B \cdot C + \bar{B} \cdot D + C \cdot D$$

$$F: B \cdot C + \bar{B} \cdot D$$

Poiché  $E = F + CD$ , se  $F$  ritorna 1, allora  $E$  ritorna 1.

Occorre quindi dimostrare il contrario: se  $E$  ritorna 1, allora  $F$  ritorna 1.

Notate che  $E$  ritorna 1 sse (almeno) una fra  $B\bar{C}$ ,  $BD$ , o  $CD$  ritornano 1.

Mostreremo allora che l'alternativa  $CD$  è superflua, ovvero che quando  $CD$  ritorna 1 almeno una  $\bar{B}D$  e  $BC$  ritorna 1.

Supponiamo  $CD = 1$ , per definizione questo vuol dire che  $C=1$  e  $D=1$ . Quindi se  $B=0$ , allora  $\bar{B}D = 1$ , altrimenti  $BC=1$ .

Per casi rilevanti

| $B$ | $C$ | $D$ | $BC + \bar{B}D + CD$ | $BC + \bar{B}D$ |
|-----|-----|-----|----------------------|-----------------|
| 0   | 0   | 0   | 0                    | 0               |
| 0   | 0   | 1   | 1                    | 1               |
| 0   | 1   | 0   | 0                    | 0               |
| 0   | 1   | 1   | 1                    | 1               |
| 1   | 0   | 0   | 0                    | 0               |
| 1   | 0   | 1   | 0                    | 0               |
| 1   | 1   | 0   | 1                    | 1               |
| 1   | 1   | 1   | 1                    | 1               |

# Teoremi di De Morgan

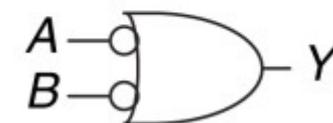
| #   | Theorem                                                                 | Dual                                                                    | Name               |
|-----|-------------------------------------------------------------------------|-------------------------------------------------------------------------|--------------------|
| T12 | $B_0 \cdot B_1 \cdot B_2 \dots =$<br>$\overline{B_0 + B_1 + B_2 \dots}$ | $\overline{B_0 + B_1 + B_2 \dots} =$<br>$B_0 \cdot B_1 \cdot B_2 \dots$ | DeMorgan's Theorem |

- La negata di un prodotto è uguale alla somma delle negate
- La negata di una somma è uguale al prodotto delle negate

# Teoremi di De Morgan

- I teoremi di De Morgan consentono di convertire un porta AND in una OR e viceversa

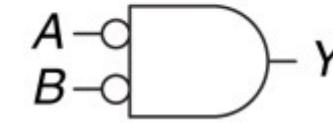
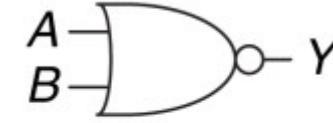
**NAND**



$$Y = \overline{AB} = \overline{A} + \overline{B}$$

| A | B | Y |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

**NOR**



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

| A | B | Y |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

# Esempio

$$\begin{aligned}Y &= \overline{(A+\overline{BD})\overline{C}} \\&= \overline{(A+\overline{BD})} + \overline{\overline{C}} \\&= (\overline{A} \bullet (\overline{BD})) + C \\&= (\overline{A} \bullet (BD)) + C \\&= \overline{A}BD + C\end{aligned}$$

# Altri tre teoremi fondamentali

- Sia  $E$  una espressione booleana che contiene una sotto espressione  $E_1$ . Se  $E_1 = E_2$ , allora  $E = E'$  dove  $E'$  è ottenuto sostituendo  $E_1$  con  $E_2$ .
  - Esempio:  $BC + \bar{B}D + CD = BC + \bar{B}D$   
 $(BC + \bar{B}D + CD)(FD + A) = (BC + \bar{B}D)(FD + A)$
- Se  $E_1 = E_2$  allora  $E'_1 = E'_2$ . Dove  $E'_1$  e  $E'_2$  sono ottenuti sostituendo uniformemente una variabile booleana  $A$  con una qualsiasi espressione  $E$ .
  - Esempio:  $A\bar{B} + AB = A$   
 $AD\bar{C}\bar{B} + AD\bar{C}B = AD\bar{C}$   
 $(CD + A)\bar{B} + (CD + A)B = (CD + A)$   
 $(CD + A)\bar{E}\bar{F} + (CD + A)EF = (CD + A)$
- Se  $E = E'$  allora  $\underline{\underline{E}} = \underline{\underline{E'}}$

# Leggi di de Morgan e dualità

Alla luce del teorema precedente risulta chiaro che il principio di dualità è una conseguenza dei teoremi di De Morgan

- $B \cdot C = C \cdot B \rightarrow \overline{B} \cdot \overline{C} = \overline{C} \cdot \overline{B} \rightarrow \overline{\overline{B} \cdot \overline{C}} = \overline{\overline{C} \cdot \overline{B}} \rightarrow \overline{\overline{B}} + \overline{\overline{C}} = \overline{\overline{C}} + \overline{\overline{B}} \rightarrow B + C = C + B$
- $B \cdot 0 = 0 \rightarrow \overline{B} \cdot 0 = 0 \rightarrow \overline{\overline{B} \cdot 0} = \overline{0} \rightarrow \overline{\overline{B} + 0} = \overline{0} \rightarrow B + 1 = 1$

# Teoremi di De Morgan e forme SOP/POS

I teoremi di De Morgan possono essere usati per ridurre una generica espressione E in forma SOP/POS senza «passare» per la tabella di verità

- Applica esaustivamente «De Morgan» per spingere la negazione nella struttura della formula
- Applica esaustivamente la proprietà distributiva dell'AND sull'OR (SOP)
- Applica esaustivamente la proprietà distributiva dell'OR sull'AND (POS)

# Esempio

$$\begin{aligned} Y &= \overline{\overline{(ACE + \overline{D})}} + B \\ &= (\overline{ACE} + \overline{\overline{D}}) \bullet \overline{B} \\ &= (\overline{\overline{ACE}} \bullet \overline{\overline{D}}) \bullet \overline{B} \\ &= ((\overline{AC} + \overline{E}) \bullet D) \bullet \overline{B} \\ &= ((AC + \overline{E}) \bullet D) \bullet \overline{B} \\ &= (ACD + D\overline{E}) \bullet \overline{B} \\ &= A\overline{B}CD + \overline{B}D\overline{E} \end{aligned}$$

# Da SOP a POS

I teoremi di De Morgan consentono anche di ricavare la forma POS a partire dalla SOP (e viceversa)

| A | B | Y | $\bar{Y}$ | minterm           |
|---|---|---|-----------|-------------------|
| 0 | 0 | 0 | <b>1</b>  | $\bar{A} \bar{B}$ |
| 0 | 1 | 0 | <b>1</b>  | $\bar{A} B$       |
| 1 | 0 | 1 | <b>0</b>  | $A \bar{B}$       |
| 1 | 1 | 1 | <b>0</b>  | $A B$             |

$$\bar{Y} = \bar{A} \bar{B} + \bar{A} B$$

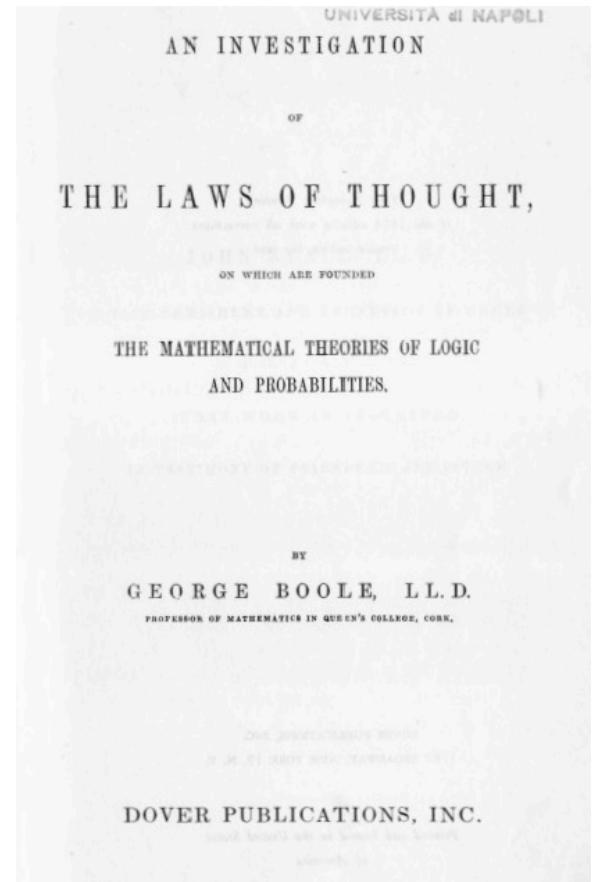
$$\bar{\bar{Y}} = \overline{\bar{A} \bar{B} + \bar{A} B} = \overline{\bar{A} \bar{B}} \cdot \overline{\bar{A} B} = (A + B) \cdot (A + \bar{B})$$

# Completezza degli operatori

- Abbiamo visto che una qualsiasi funzione booleana  $f:\{0,1\}^N \rightarrow \{0,1\}$  può essere espressa in forma SOP
- La forma SOP usa solo gli operatori NOT, AND, OR
- Questo vuol dire che questo insieme di operatori costituisce un insieme **completo**
- Per le leggi di de Morgan
  - $A \cdot B = \overline{\overline{A} + \overline{B}}$  (AND può essere espresso in termini di NOT e OR)
  - $A + B = \overline{\overline{A} \cdot \overline{B}}$  (OR può essere espresso in termini di NOT e AND)
- Quindi  $\{\text{NOT, AND, OR}\}$  è completo ma non minimale essendo  $\{\text{AND, NOT}\}$  e  $\{\text{OR, NOT}\}$  anche loro completi

# Un po' di storia...

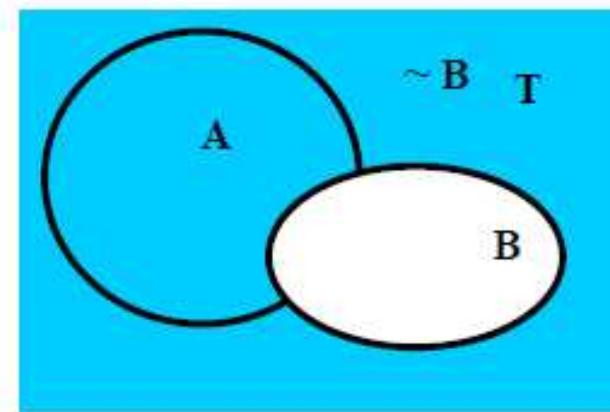
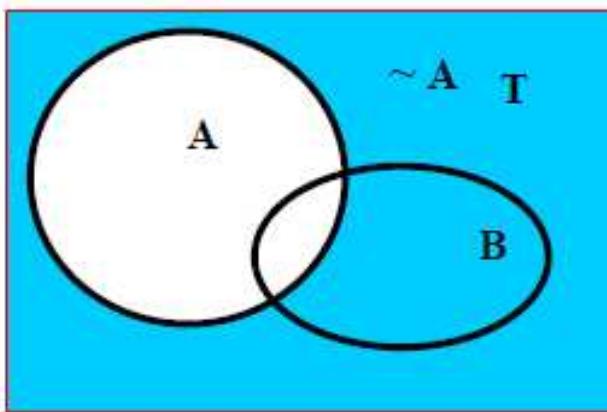
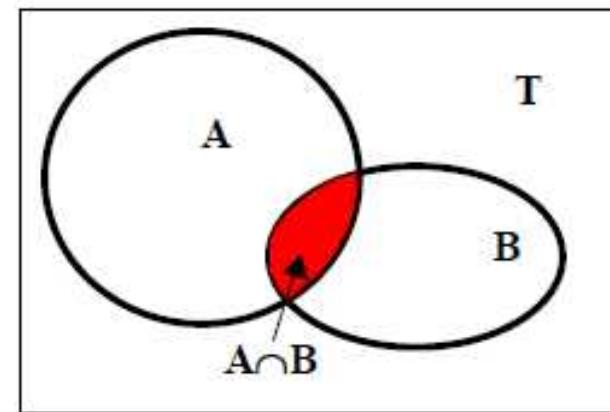
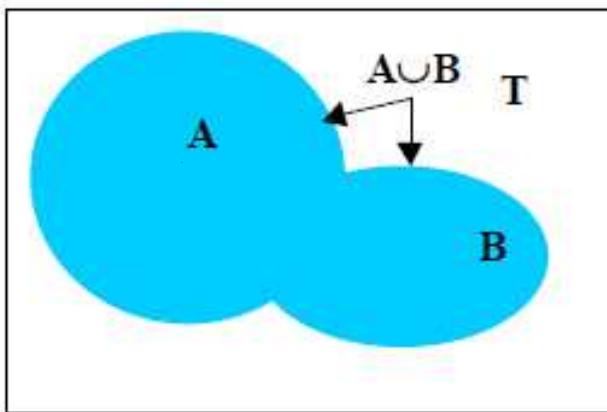
- L'algebra booleana prende nome dal matematico inglese [George Boole](#)
- L'oggetto di questa algebra non erano le porte logiche (che ancora non esistevano) ma *le leggi del pensiero e della deduzione*
- Infatti si può dare una interpretazione puramente logica delle espressioni booleane in cui
  - Le variabili corrispondono a delle proposizioni che possono essere [vere](#) (1) o [false](#) (0)
  - Gli operatori corrispondono alle relative preposizioni coordinate (e, o, non...)
  - «piove [e](#) ho l'ombrelllo» è vera sse «piove» e «ho l'ombrelllo» sono entrambe vere



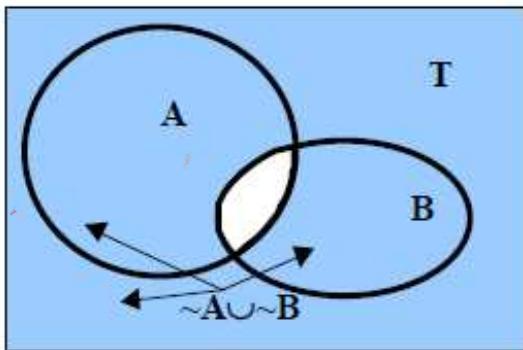
# Operatori insiemistici

- Anche gli usuali operatori insiemistici di unione, intersezione e complemento formano un algebra di Boole
- T insieme universo
- $2^T = \{A \mid A \subseteq T\}$  insieme dei sottoinsiemi di T
- $\cap \rightarrow$  AND
- $\cup \rightarrow$  OR
- $T^c \rightarrow$  NOT
- 1  $\rightarrow$  T
- 0  $\rightarrow$   $\emptyset$

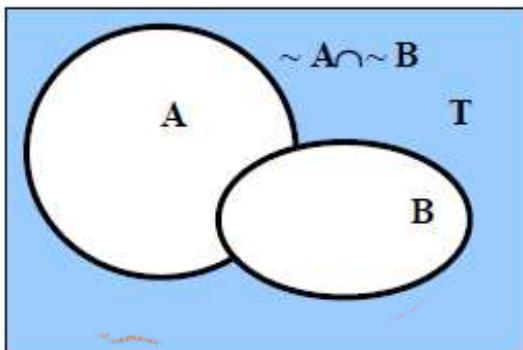
# Operatori insiemistici



# Operatori insiemistici



dal confronto con la figura  
relativa a  $A \cap B$  si ricava  
 $\sim A \cup \sim B = \sim (A \cap B)$



dal confronto con la  
figura relativa a  $A \cup B$  si  
ricava  
 $\sim A \cap \sim B = \sim (A \cup B)$

# Esercizi

- Sfruttando i teoremi precedenti verificare le seguenti proprietà della porta XOR

$$a \oplus a = 0$$

$$a \oplus 0 = a$$

$a \oplus 1 = \sim a$ , where  $\sim$  is bit complement.

$$a \oplus \sim a = 1$$

$$a \oplus b = b \oplus a \text{ (commutativity)}$$

$$a \oplus (b \oplus c) = (a \oplus b) \oplus c \text{ (associativity)}$$

Ricordarsi che, date due espressioni booleane E e F,  $E \oplus F = EF + \bar{E}\bar{F}$

# Esercizi

- Esercizi del libro “Digital Design and Computer Architecture – ARM edition”
  - 2.1
  - 2.2
  - 2.3
  - 2.4
- Verificare in maniera proof-teoretica che  
 $a \oplus b = (\bar{a} + \bar{b}) (a + b)$
- Dimostrare che la porta NAND è da sola un insieme completo (e minimale) di operatori booleani

$$a \oplus a = 0$$

$$a \oplus 0 = a$$

$a \oplus 1 = \sim a$ , where  $\sim$  is bit complement.

$$a \oplus \sim a = 1$$

$a \oplus b = b \oplus a$  (commutativity)

$a \oplus (b \oplus c) = (a \oplus b) \oplus c$  (associativity)

$$a \oplus a \oplus a = a$$

if  $a \oplus b = c$ , then  $c \oplus b = a$  and  $c \oplus a = b$ .

$$\rightarrow a\bar{a} + \bar{a}a = a\bar{a} + a\bar{a} = a\bar{a} = 0$$

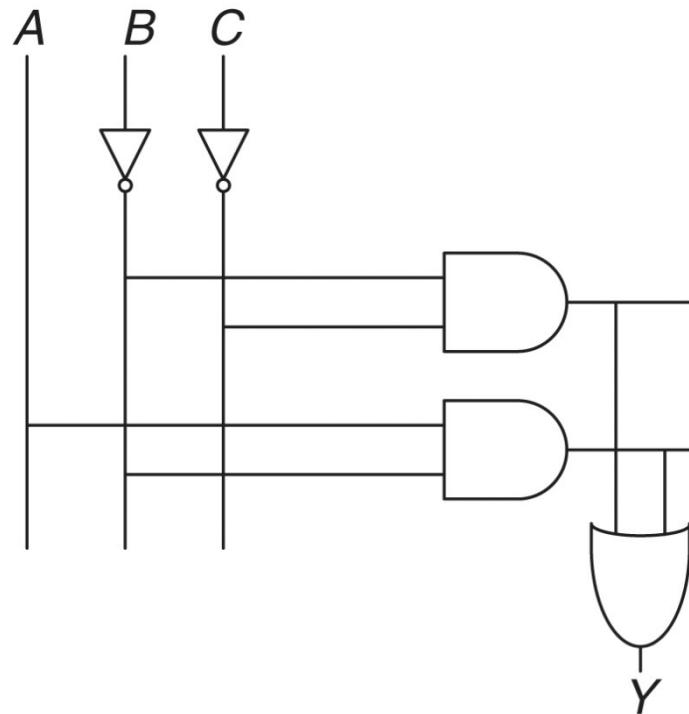
$$\rightarrow a(\overline{b\bar{c}} + \overline{\bar{b}c}) + \bar{a}(\bar{b}\bar{c} + \bar{\bar{b}}c) = a(\overline{b\bar{c}} \cdot \overline{\bar{b}c}) + \bar{a}b\bar{c} + \bar{a}\bar{b}c =$$

$$\rightarrow a[(\bar{b} + c) \cdot (\bar{b} + \bar{c})] + \bar{a}b\bar{c} + \bar{a}\bar{b}c = a[\bar{b} \cdot (\bar{b} + \bar{c}) + c(\bar{b} + \bar{c})] + \bar{a}b\bar{c} + \bar{a}\bar{b}c$$

$$\begin{aligned} &= (\bar{a}\bar{b} + \bar{a}c)\bar{c} + (\bar{a}b + \bar{a}\bar{b})c = \\ &= (\bar{a}\bar{b} + \bar{a}c)\bar{c} + (\bar{a}\bar{b} + \bar{a}\bar{b})c = \end{aligned}$$

# Perché semplificare una espressione?

- Importanza della minimizzazione: utilizzare meno porte logiche
- Esercizio: mostrare che  $\bar{A}\bar{B}\bar{C} + A\bar{B}\bar{C} + A\bar{B}C = \bar{B}\bar{C} + A\bar{B}$



$$Y = \bar{B}\bar{C} + AB$$

# Esempio 1

$$Y = A(\underline{AB} + \underline{ABC})$$

$$= A(AB(1 + C))$$

T8: Distributivity

$$= A(AB(1))$$

T2': Null Element

$$= A(AB)$$

T1: Identity

$$= (AA)B$$

T7: Associativity

$$= AB$$

T3: Idempotency

## Esempio 2

$$Y = \overline{ABC} + \overline{A}$$

$$= \overline{A}$$

oppure

$$= \overline{A}(BC + 1)$$

$$= \overline{A}(1)$$

$$= \overline{A}$$

T9' Covering:  $X + XY = X$

T8: Distributivity

T2': Null Element

T1: Identity

# Alcuni teoremi per semplificare espressioni

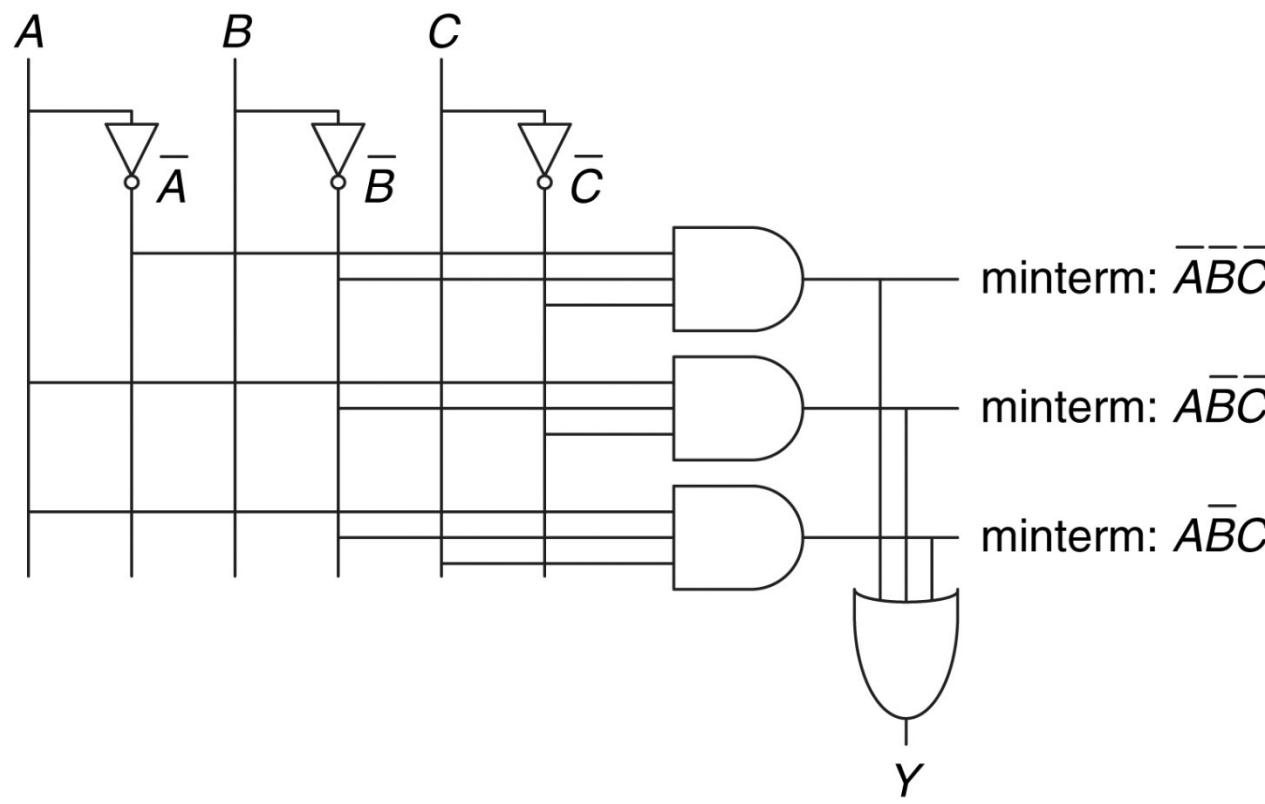
- Distributivity (T8, T8')
$$B(C+D) = BC + BD$$
$$B + CD = (B+C)(B+D)$$
- Covering (T9')
$$A + AP = A$$
- Combining (T10)
$$\overline{PA} + PA = P$$
- Expansion
$$P = \overline{PA} + PA$$
$$A = A + AP$$
- Duplication
$$A = A + A$$
- “Simplification” theorem
$$\overline{PA} + A = P + A$$
$$PA + \overline{A} = P + \overline{A}$$

# «Simplification theorem»

$$\begin{aligned} PA + \bar{A} &= PA + (\bar{A} + \bar{A}P) && T9' \text{ Covering} \\ &= PA + P\bar{A} + \bar{A} && T6 \text{ Commutativity} \\ &= P(A + \bar{A}) + \bar{A} && T8 \text{ Distributivity} \\ &= P(\mathbf{1}) + \bar{A} && T5' \text{ Complements} \\ &= P + \bar{A} && T1 \text{ Identity} \end{aligned}$$

# Schemi circuituali

- Ad ogni espressione booleana corrisponde un circuito combinatorio



$$Y = \bar{A}\bar{B}\bar{C} + \bar{A}\bar{B}C + A\bar{B}C$$

# Schemi circuitali SOP

- Le formule in forma SOP hanno degli schemi circuitali molto regolari:
  - Disegna una linea di input per ogni variabile che occorre positiva
  - Aggiungi delle linee con un NOT per ogni variabile che occorre negata
  - Per ogni mintermine disegna una porta AND e aggiungi in ingresso le linee che corrispondono ai relativi littorali
  - Collega tutte le uscite delle porte AND in un unico OR
- Per questo la forma SOP viene detta una logica a due livelli AND-OR

# Semplificare formule SOP

- Per T10  $P\bar{B} + PB = P$  per ogni implicante  $P$ .
- Un implicante è detto *implicante primo* se non può essere combinato con altri implicanti della formula per ottenere un nuovo implicante con meno litterali.
- Una espressione SOP è minimale se tutti i suoi implicanti sono primi
- Minimizzazione: *ridurre il numero di implicanti e per ogni implicante ridurre il numero di litterali*

# Semplificare formule SOP

- Nel minimizzare una SOP, può essere necessario «sdoppiare» un implicante allorché questo può essere ridotto in modi differenti.

| Step | Equation                                              | Justification      |
|------|-------------------------------------------------------|--------------------|
|      | $\bar{A}\bar{B}\bar{C} + A\bar{B}\bar{C} + A\bar{B}C$ |                    |
| 1    | $\bar{B}\bar{C}(\bar{A} + A) + A\bar{B}C$             | T8: Distributivity |
| 2    | $\bar{B}\bar{C}(1) + A\bar{B}C$                       | T5: Complements    |
| 3    | $\bar{B}\bar{C} + A\bar{B}C$                          | T1: Identity       |

Non è minimale  $A\bar{B}C$  e  $A\bar{B}\bar{C}$  possono ridursi in  $A\bar{B}$

# Semplificare formule SOP

| Step | Equation                                                                                                                | Justification      |
|------|-------------------------------------------------------------------------------------------------------------------------|--------------------|
|      | $\overline{A} \overline{B} \overline{C} + A \overline{B} \overline{C} + A \overline{B} C$                               |                    |
| 1    | $\overline{A} \overline{B} \overline{C} + A \overline{B} \overline{C} + A \overline{B} \overline{C} + A \overline{B} C$ | T3: Idempotency    |
| 2    | $\overline{B} \overline{C}(\overline{A} + A) + A \overline{B}(\overline{C} + C)$                                        | T8: Distributivity |
| 3    | $\overline{B} \overline{C}(1) + A \overline{B}(1)$                                                                      | T5: Complements    |
| 4    | $\overline{B} \overline{C} + A \overline{B}$                                                                            | T1: Identity       |



Il numero di implicanti è lo stesso ma  
 $A \overline{B}$  ha meno letterali di  $A \overline{B} C$

# Semplificare le forme POS

- Come si semplifica una forma POS?
- La proprietà principale che si usa è il combining  $(B+C) \cdot (B+\bar{C}) = B$
- Esempio:

$$(A + \bar{B} + \bar{C})(\bar{A} + B + \bar{C})(\bar{A} + \bar{B} + C)(\bar{A} + \bar{B} + \bar{C}) =$$

$$(A + \bar{B} + \bar{C})(\bar{A} + B + \bar{C})(\bar{A} + \bar{B} + C)(\bar{A} + \bar{B} + \bar{C})(\bar{A} + \bar{B} + \bar{C}) =$$

$$(A + \bar{B} + \bar{C})(\bar{A} + \bar{B} + \bar{C})(\bar{A} + B + \bar{C})(\bar{A} + \bar{B} + C)(\bar{A} + \bar{B} + \bar{C}) =$$

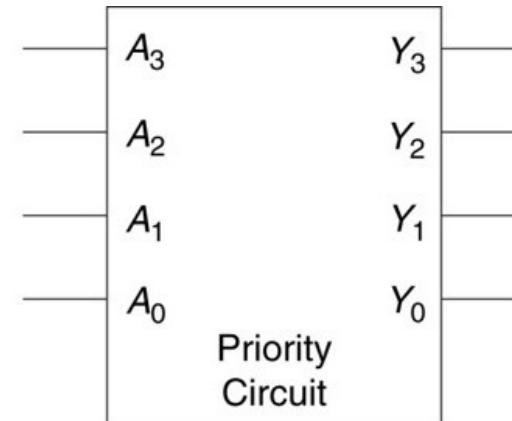
$$(B + \bar{C})(\bar{A} + B + \bar{C})(\bar{A} + \bar{B} + \bar{C})(\bar{A} + \bar{B} + C)(\bar{A} + \bar{B} + \bar{C}) =$$

$$(B + \bar{C})(\bar{A} + \bar{C})(\bar{A} + \bar{B} + C)(\bar{A} + \bar{B} + \bar{C}) =$$

$$(B + \bar{C})(\bar{A} + \bar{C})(\bar{A} + \bar{B})$$

# Circuito a priorità

- I circuiti a priorità vengono utilizzati per assegnare una risorsa condivisa secondo un ordine di priorità fra chi ne fa richiesta
- A esempio posso avere 4 possibili richiedenti con priorità  $3>2>1>0$
- Gli input  $A_0, \dots, A_3$  rappresentano le richieste della risorsa
- Gli output  $Y_0, \dots, Y_3$  rappresentano a chi assegnata la risorsa, di volta in volta uno solo di essi sarà uguale a 1

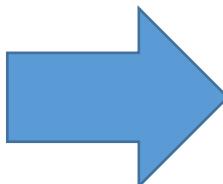


# Circuito a priorità

| $A_3$ | $A_2$ | $A_1$ | $A_0$ | $Y_3$ | $Y_2$ | $Y_1$ | $Y_0$ |
|-------|-------|-------|-------|-------|-------|-------|-------|
| 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     |
| 0     | 0     | 0     | 1     | 0     | 0     | 0     | 1     |
| 0     | 0     | 1     | 0     | 0     | 0     | 1     | 0     |
| 0     | 0     | 1     | 1     | 0     | 0     | 1     | 0     |
| 0     | 1     | 0     | 0     | 0     | 1     | 0     | 0     |
| 0     | 1     | 0     | 1     | 0     | 1     | 0     | 0     |
| 0     | 1     | 1     | 0     | 0     | 1     | 0     | 0     |
| 0     | 1     | 1     | 1     | 0     | 1     | 0     | 0     |
| 1     | 0     | 0     | 0     | 1     | 0     | 0     | 0     |
| 1     | 0     | 0     | 1     | 1     | 0     | 0     | 0     |
| 1     | 0     | 1     | 0     | 1     | 0     | 0     | 0     |
| 1     | 0     | 1     | 1     | 1     | 0     | 0     | 0     |
| 1     | 1     | 0     | 0     | 1     | 0     | 0     | 0     |
| 1     | 1     | 0     | 1     | 1     | 0     | 0     | 0     |
| 1     | 1     | 1     | 0     | 1     | 0     | 0     | 0     |
| 1     | 1     | 1     | 1     | 1     | 0     | 0     | 0     |

# Don't cares

| $A_3$ | $A_2$ | $A_1$ | $A_0$ | $Y_3$ | $Y_2$ | $Y_1$ | $Y_0$ |
|-------|-------|-------|-------|-------|-------|-------|-------|
| 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     |
| 0     | 0     | 0     | 1     | 0     | 0     | 0     | 1     |
| 0     | 0     | 1     | 0     | 0     | 0     | 1     | 0     |
| 0     | 0     | 1     | 1     | 0     | 0     | 1     | 0     |
| 0     | 1     | 0     | 0     | 0     | 1     | 0     | 0     |
| 0     | 1     | 0     | 1     | 0     | 1     | 0     | 0     |
| 0     | 1     | 1     | 0     | 0     | 1     | 0     | 0     |
| 0     | 1     | 1     | 1     | 0     | 1     | 0     | 0     |
| 1     | 0     | 0     | 0     | 1     | 0     | 0     | 0     |
| 1     | 0     | 0     | 1     | 1     | 0     | 0     | 0     |
| 1     | 0     | 1     | 0     | 1     | 0     | 0     | 0     |
| 1     | 0     | 1     | 1     | 1     | 0     | 0     | 0     |
| 1     | 1     | 0     | 0     | 1     | 0     | 0     | 0     |
| 1     | 1     | 0     | 1     | 1     | 0     | 0     | 0     |
| 1     | 1     | 1     | 0     | 1     | 0     | 0     | 0     |
| 1     | 1     | 1     | 1     | 1     | 0     | 0     | 0     |

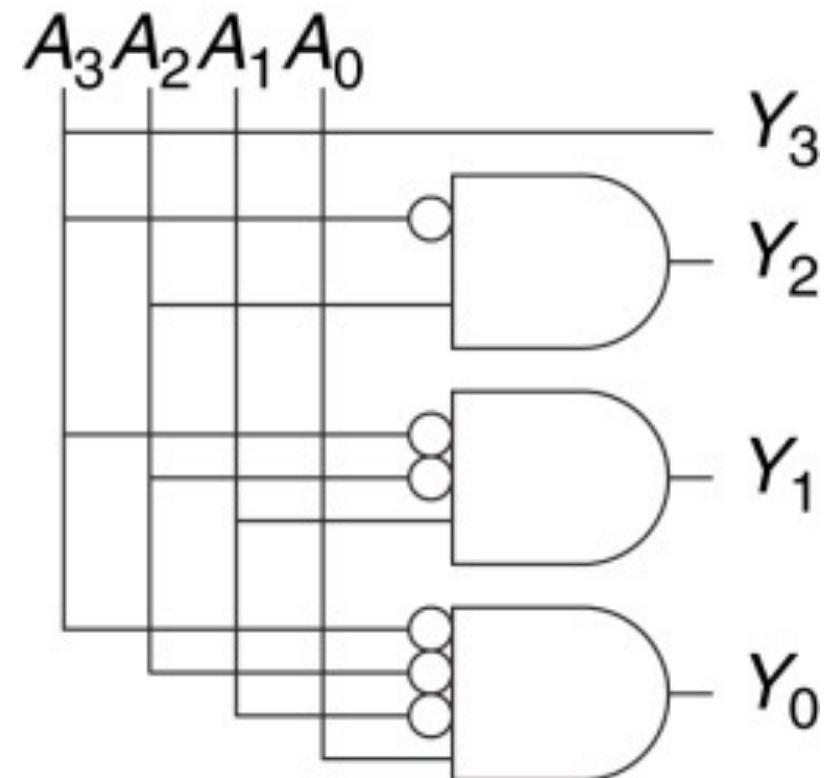


| $A_3$ | $A_2$ | $A_1$ | $A_0$ | $Y_3$ | $Y_2$ | $Y_1$ | $Y_0$ |
|-------|-------|-------|-------|-------|-------|-------|-------|
| 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     |
| 0     | 0     | 0     | 1     | 0     | 0     | 0     | 1     |
| 0     | 0     | 1     | X     | 0     | 0     | 1     | 0     |
| 0     | 1     | X     | X     | 0     | 1     | 0     | 0     |
| 1     | X     | X     | X     | 1     | 0     | 0     | 0     |

don't cares

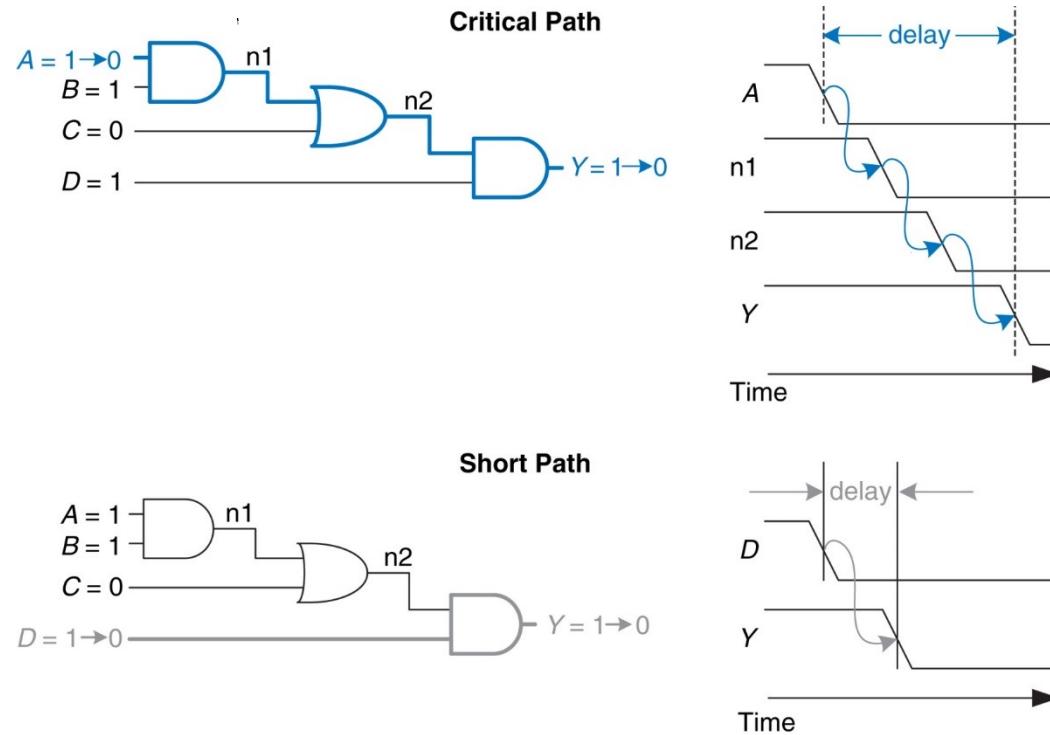
# Don't cares

| $A_3$ | $A_2$ | $A_1$ | $A_0$ | $Y_3$ | $Y_2$ | $Y_1$ | $Y_0$ |
|-------|-------|-------|-------|-------|-------|-------|-------|
| 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     |
| 0     | 0     | 0     | 1     | 0     | 0     | 0     | 1     |
| 0     | 0     | 1     | X     | 0     | 0     | 1     | 0     |
| 0     | 1     | X     | X     | 0     | 1     | 0     | 0     |
| 1     | X     | X     | X     | 1     | 0     | 0     | 0     |



# Vantaggi delle SOP/POS

- La logica a 2 livelli della forma SOP presenta dei vantaggi, ad esempio, nei tempi di propagazione

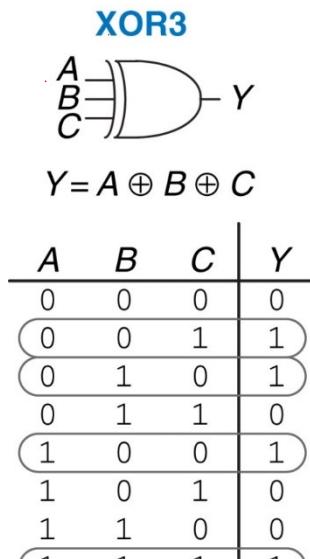


# Limiti delle forme SOP/POS

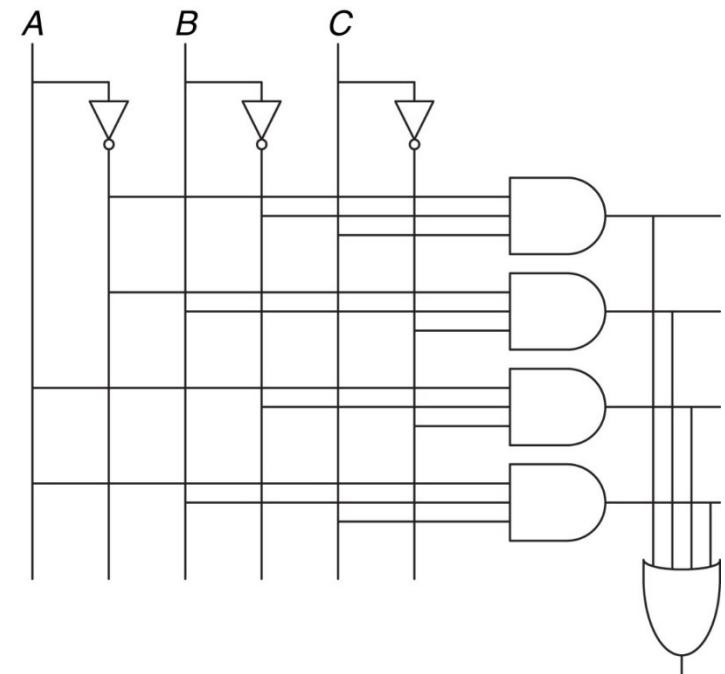
- Tuttavia alcune funzioni booleane, poste in forma SOP, sono estremamente poco succinte e quindi richiedono un numero considerevole di porte
- Presa la tabella di verità di funzione booleana di n variabili:
  - Se il numero di 1 nella colonna di output è piccolo allora la forma SOP è succinta
  - Se il numero di 0 nella colonna di output è piccolo allora la forma POS è succinta
  - Se il numero di 0 e 1 è più o meno lo stesso? Problema: avrò circa  $2^{n-1}$  mintermini/maxtermini

# Limiti delle forme SOP/POS

- Consideriamo ad esempio uno XOR a più variabili
- $Y=1$  sse il numero di input uguali a 1 è dispari
- XOR3 in forma SOP  
$$Y = \overline{A}\overline{B}C + \overline{A}B\overline{C} + A\overline{B}\overline{C} + ABC$$
- XOR8 richiede 128 AND8 e un OR128

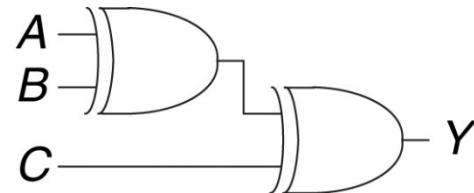


(a)

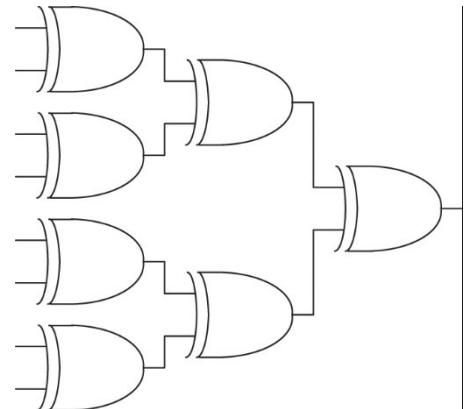


# Logiche multilivello

- Per ridurre il numero di porte logiche a volte è necessario ricorrere ad una logica multilivello
- Ad esempio è facile verificare che  $A \oplus B \oplus C = (A \oplus B) \oplus C$



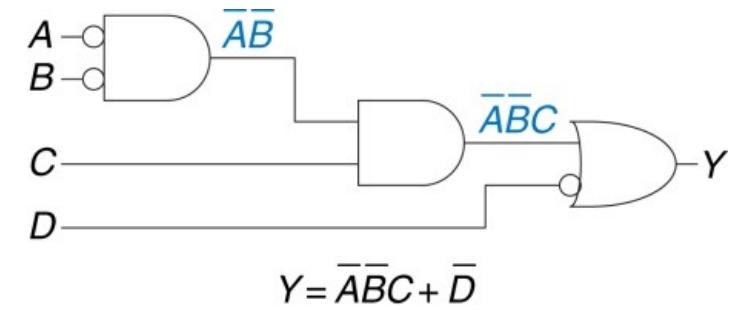
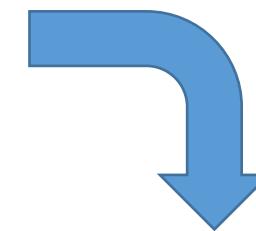
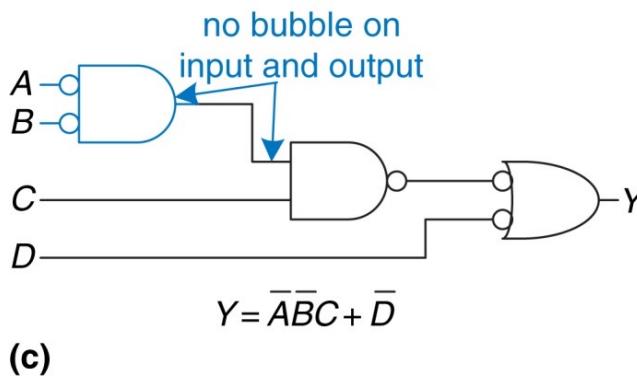
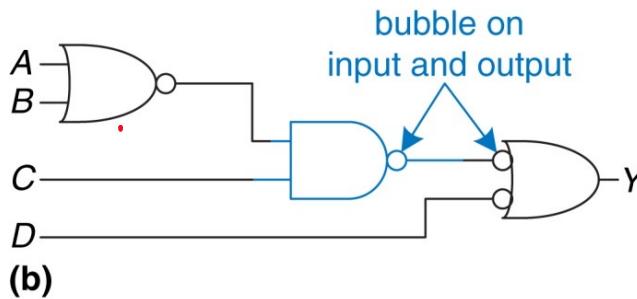
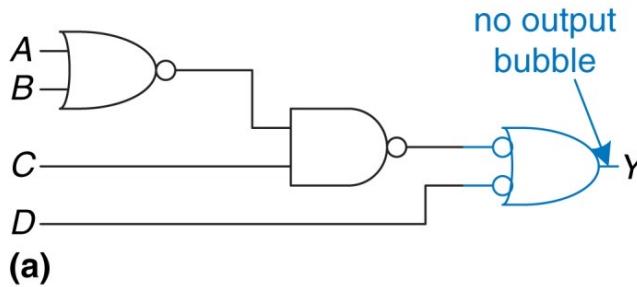
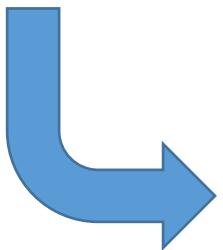
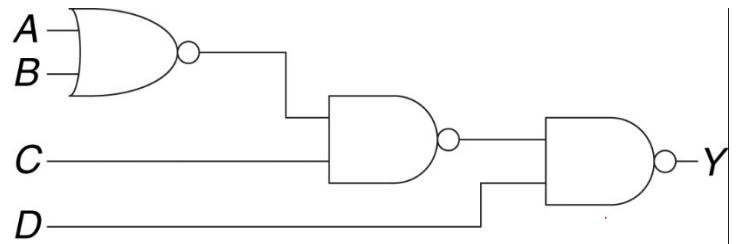
- Analogamente per XOR8:



# Bubble pushing

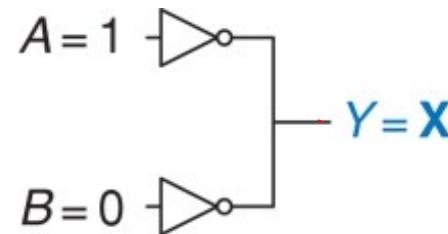
- Usando logiche multilivello e porte NAND/NOR a volte rende difficile capire quale funzione booleana un circuito realizza a causa delle molte negazioni annidate
- Per avere una espressione un po' più leggibile si possono applicare le leggi di De Morgan
- Il bubble pushing è una tecnica che applica sistematicamente le leggi di De Morgan e la legge della doppia negazione per eliminare le negazioni annidate
- Partendo dall'output Y si applicano a ritroso le leggi di De Morgan in modo che l'input e l'output di ogni nodo siano entrambi positivi o negativi

# Bubble pushing



# Valori illegali

- Considerate il circuito in figura. Quale sarà il valore di Y?



- Il valore del potenziale è portato ad essere contemporaneamente sia 0 che  $V_{DD}$
- Questo tipo di configurazioni vengono dette *illegali*. Il valore di Y dipenderà dalla natura fisica delle porte.
  - Stabile ma indeterminato
  - Oscillare fra i valori 0 e  $V_{DD}$
  - Avere un valore intermedio
- Configurazioni del genere possono indurre ad una forte dissipazione che riscalda e danneggia il circuito (corto circuito)

# Stato di alta impedenza

- Un nodo, oltre ad essere in uno stato 0/1, può anche essere in un stato Z *floating* o di alta impedenza (alta impedenza=no passaggio di corrente)
- Ad esempio un ingresso di una porta logica non collegato è nello stato Z
- Gli stati di alta impedenza vengono utilizzati per disconnettere una parte di un circuito dal resto. Per questo si utilizzano i tristate buffer

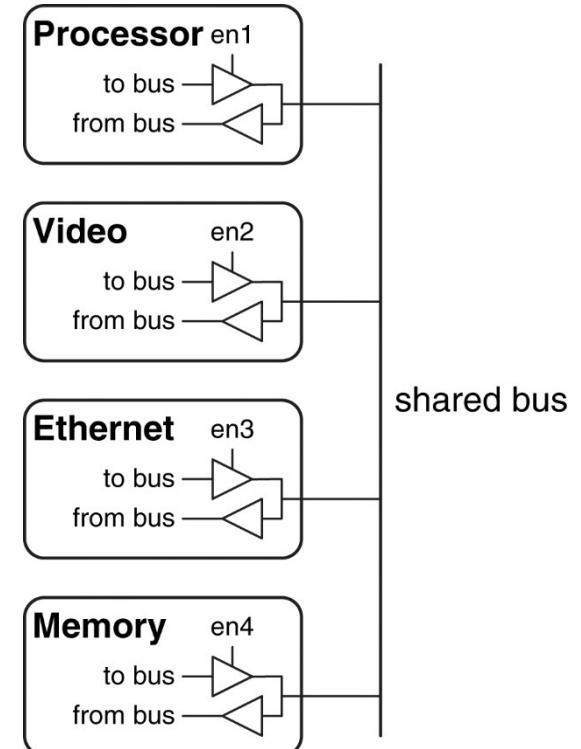
**Tristate  
Buffer**



| E | A | Y | E | A | Y |
|---|---|---|---|---|---|
| 0 | 0 | Z | 0 | 0 | 0 |
| 0 | 1 | Z | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 | Z |
| 1 | 1 | 1 | 1 | 1 | Z |

# Bus condiviso

- I tristate buffers sono usati in bus che connettono diversi chip
- Nell'esempio processore, scheda video e controller ethernet devono poter comunicare con la memoria centrale.
- Tuttavia, per evitare stati illegali, solo un componente alla volta può «immettere» segnali sul bus
- Gli altri componenti quindi devono essere temporaneamente disconnessi tramite un tristate buffer



# Esercizi

- Esercizi 2.13 – 2.15 – 2.16 – 2.17 – 2.25 – 2.26 – 2.27

# Mappe di Karnaugh

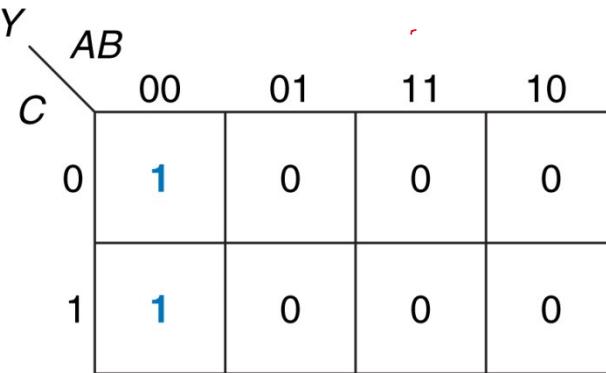
- Le mappe di Karnaugh sono un metodo per semplificare espressioni booleane in forma SOP
- In realtà non introducono tecniche di semplificazione nuove, sono semplicemente un **espediente grafico** che consente di rilevare più facilmente implicati che possono essere semplificati
- Quindi alla base delle mappe di Karnaugh c'è il solito principio:

$$PA + \bar{P}\bar{A} = P$$

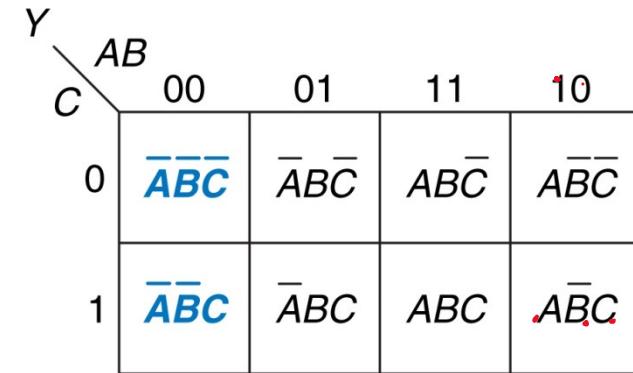
# Mappe di Karnaugh

| A | B | C | Y |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 |

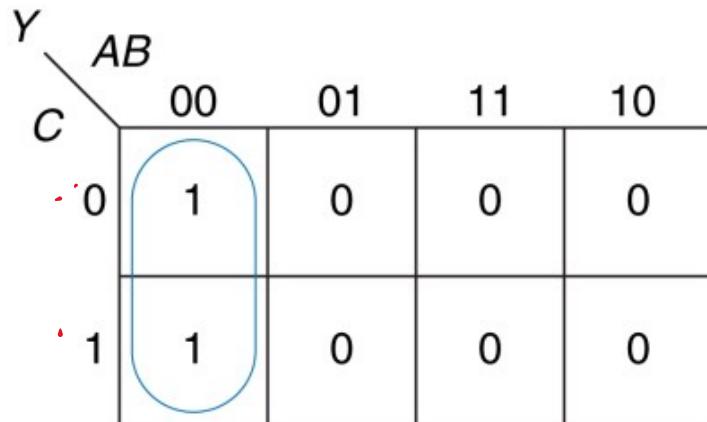
(a)



(b)



(c)

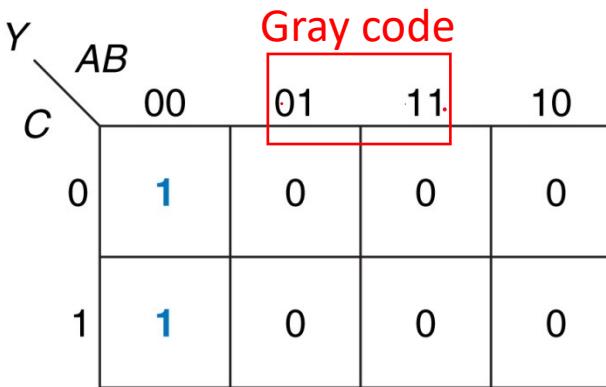


$$Y = \overline{A} \overline{B} \overline{C} + \overline{A} \overline{B} C = \overline{A} \overline{B} (\overline{C} + C) = \overline{A} \overline{B}$$

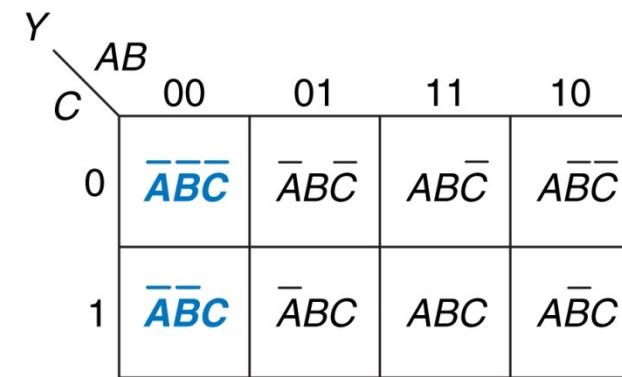
# Mappe di Karnaugh

| A | B | C | Y |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 |

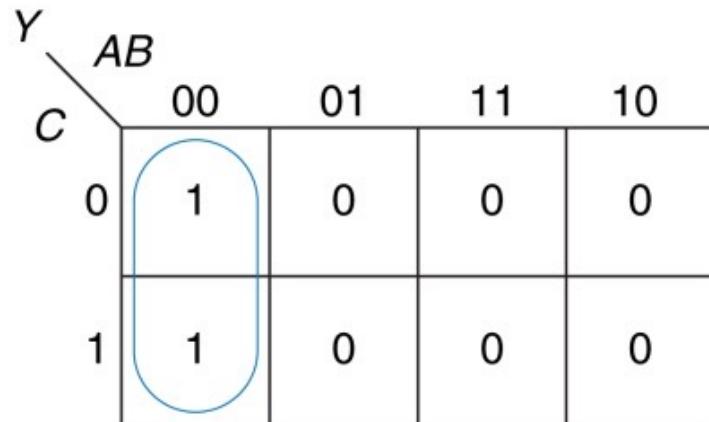
(a)



(b)



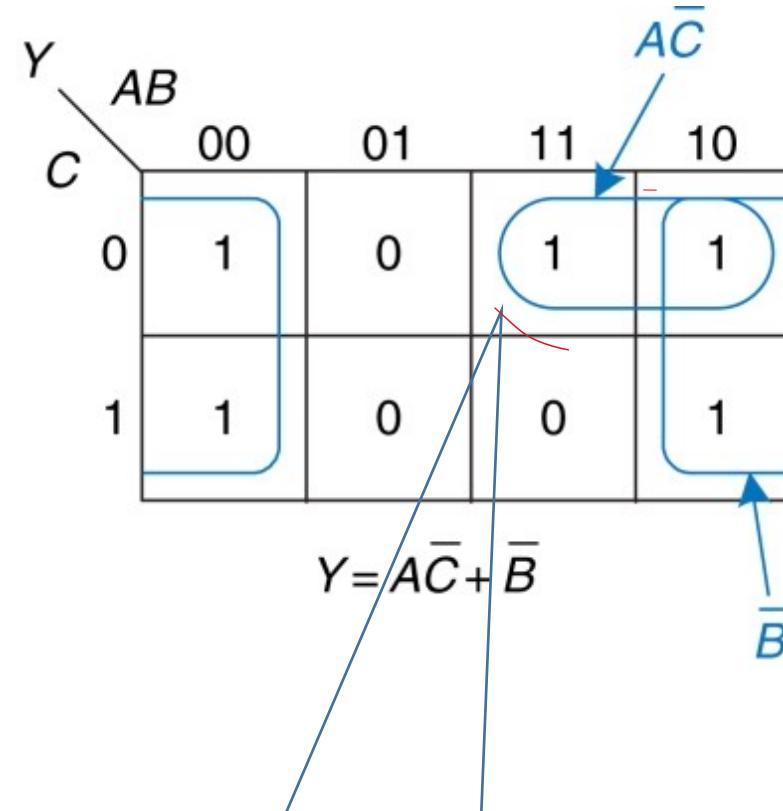
(c)



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

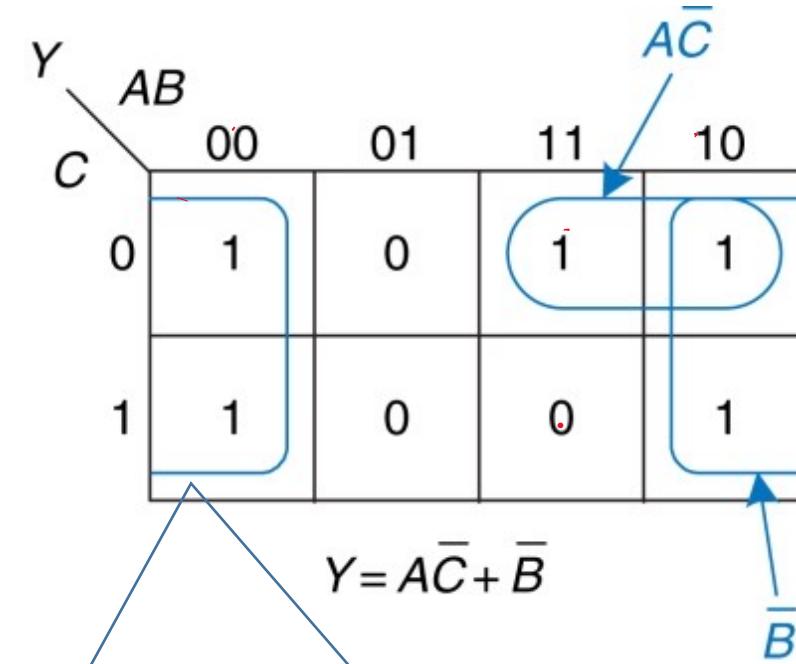
|  | $Y$ | $AB$ | $C$ | 00 | 01 | 11 | 10 |
|--|-----|------|-----|----|----|----|----|
|  |     |      | 0   | 1  | 0  | 1  | 1  |
|  |     |      | 1   | 1  | 0  | 0  | 1  |

| A | B | C | Y |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |



$$AB\bar{C} + A\bar{B}\bar{C} = A\bar{C}(B + \bar{B}) = A\bar{C}$$

|     |     | $A$ | $B$ | $C$ | $Y$ |
|-----|-----|-----|-----|-----|-----|
|     |     | 0   | 1   | 0   | 1   |
| $A$ | $B$ | 0   |     | 00  | 01  |
|     |     | 1   |     | 11  | 10  |
| 0   | 0   | 1   | 0   | 1   | 1   |
|     | 1   | 1   | 0   | 0   | 1   |

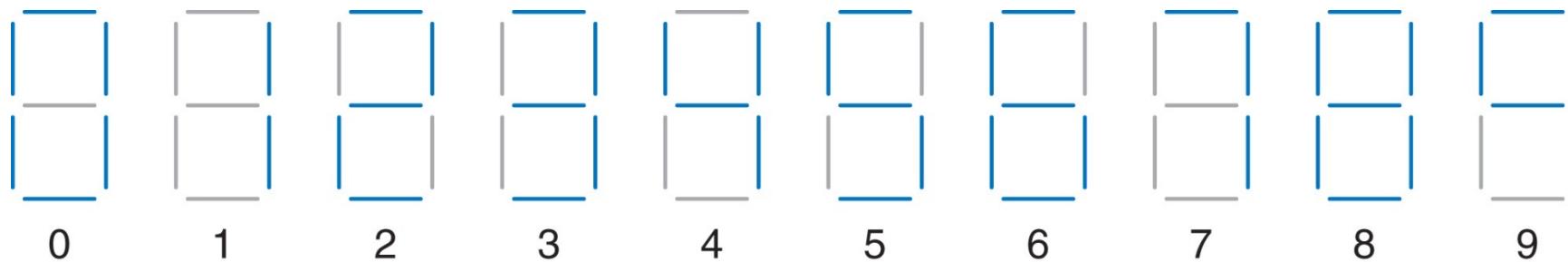
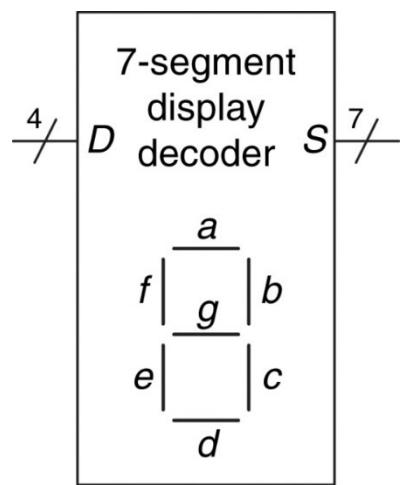


$$\bar{A}\bar{B}\bar{C} + \bar{A}\bar{B}C + A\bar{B}\bar{C} + A\bar{B}C = \bar{A}\bar{B}(\bar{C} + C) + A\bar{B} (\bar{C} + C) = \bar{A}\bar{B} + A\bar{B} = (\bar{A} + A)\bar{B} = \bar{B}$$

# Regole di minimizzazione con K-maps

- Usare il minimo numero di *bolle* necessari per ricoprire tutti gli 1
- Tutte le caselle sottese ad una bolla devono contenere un 1
- Ogni bolla ricopre un blocco di caselle che è una potenza di 2: 1,2,4,...
  - In una bolla le variabili che non assumono valore costante devono assumere entrambi i valori 0 e 1
- Ogni bolla deve essere il più largo possibile
- Una bolla può «trasbordare» i bordi della K-map
  - La K-map ha una struttura *toroidale*
- Una casella può essere ricoperta da più bolle se questo permette un numero inferiore di bolle

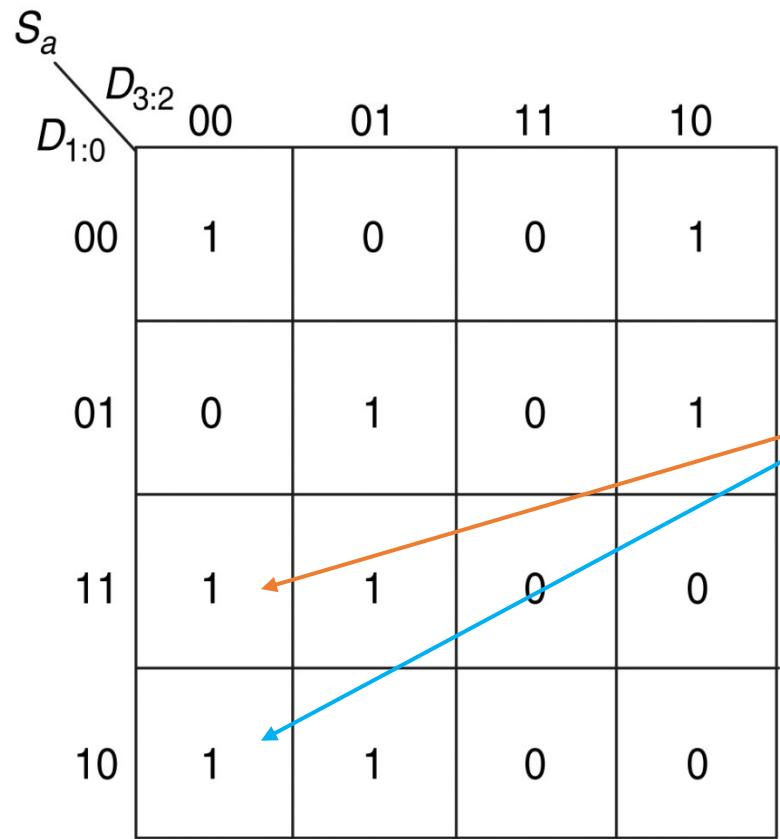
# Display a 7 segmenti



# Display a 7 segmenti

| $D_{3:0}$ | $S_a$ | $S_b$ | $S_c$ | $S_d$ | $S_e$ | $S_f$ | $S_g$ |
|-----------|-------|-------|-------|-------|-------|-------|-------|
| 0000      | 1     | 1     | 1     | 1     | 1     | 1     | 0     |
| 0001      | 0     | 1     | 1     | 0     | 0     | 0     | 0     |
| 0010      | 1     | 1     | 0     | 1     | 1     | 0     | 1     |
| 0011      | 1     | 1     | 1     | 1     | 0     | 0     | 1     |
| 0100      | 0     | 1     | 1     | 0     | 0     | 1     | 1     |
| 0101      | 1     | 0     | 1     | 1     | 0     | 1     | 1     |
| 0110      | 1     | 0     | 1     | 1     | 1     | 1     | 1     |
| 0111      | 1     | 1     | 1     | 0     | 0     | 0     | 0     |
| 1000      | 1     | 1     | 1     | 1     | 1     | 1     | 1     |
| 1001      | 1     | 1     | 1     | 0     | 0     | 1     | 1     |
| others    | 0     | /0    | 0     | 0     | 0     | 0     | 0     |

# Display a 7 segmenti



| $D_{3:0}$ | $S_a$ | $S_b$ | $S_c$ | $S_d$ | $S_e$ | $S_f$ | $S_g$ |
|-----------|-------|-------|-------|-------|-------|-------|-------|
| 0000      | 1     | 1     | 1     | 1     | 1     | 1     | 0     |
| 0001      | 0     | 1     | 1     | 0     | 0     | 0     | 0     |
| 0010      | 1     | 1     | 0     | 1     | 1     | 0     | 1     |
| 0011      | 1     | 1     | 1     | 1     | 0     | 0     | 1     |
| 0100      | 0     | 1     | 1     | 0     | 0     | 1     | 1     |
| 0101      | 1     | 0     | 1     | 1     | 0     | 1     | 1     |
| 0110      | 1     | 0     | 1     | 1     | 1     | 1     | 1     |
| 0111      | 1     | 1     | 1     | 0     | 0     | 0     | 0     |
| 1000      | 1     | 1     | 1     | 1     | 1     | 1     | 1     |
| 1001      | 1     | 1     | 1     | 0     | 0     | 1     | 1     |
| others    | 0     | 0     | 0     | 0     | 0     | 0     | 0     |

# Display a 7 segmenti

Diagram illustrating the logic for driving a 7-segment display using a 4-to-16 decoder ( $D_{1:0}$ ) and a 3-to-8 decoder ( $D_{3:2}$ ). The output  $S_a$  is connected to the enable input of the 4-to-16 decoder.

|  |  | 00 | 01 | 11 | 10 |   |
|--|--|----|----|----|----|---|
|  |  | 00 | 1  | 0  | 0  | 1 |
|  |  | 01 | 0  | 1  | 0  | 1 |
|  |  | 11 | 1  | 1  | 0  | 0 |
|  |  | 10 | 1  | 1  | 0  | 0 |

Diagram illustrating the logic for driving a 7-segment display using a 4-to-16 decoder ( $D_{1:0}$ ) and a 3-to-8 decoder ( $D_{3:2}$ ). The output  $S_b$  is connected to the enable input of the 4-to-16 decoder.

|  |  | 00 | 01 | 11 | 10 |   |
|--|--|----|----|----|----|---|
|  |  | 00 | 1  | 1  | 0  | 1 |
|  |  | 01 | 1  | 0  | 0  | 1 |
|  |  | 11 | 1  | 1  | 0  | 0 |
|  |  | 10 | 1  | 0  | 0  | 0 |

# Display a 7 segmenti

| $S_a$                           | $D_{3:2}$ | 00 | 01 | 11 | 10 |
|---------------------------------|-----------|----|----|----|----|
| $D_{1:0}$                       | 00        | 1  | 0  | 0  | 1  |
| $\bar{D}_2 \bar{D}_1 \bar{D}_0$ | 01        | 0  | 1  | 0  | 1  |
| $\bar{D}_3 D_2 D_0$             | 11        | 1  | 1  | 0  | 0  |
| $\bar{D}_3 D_1$                 | 10        | 1  | 1  | 0  | 0  |

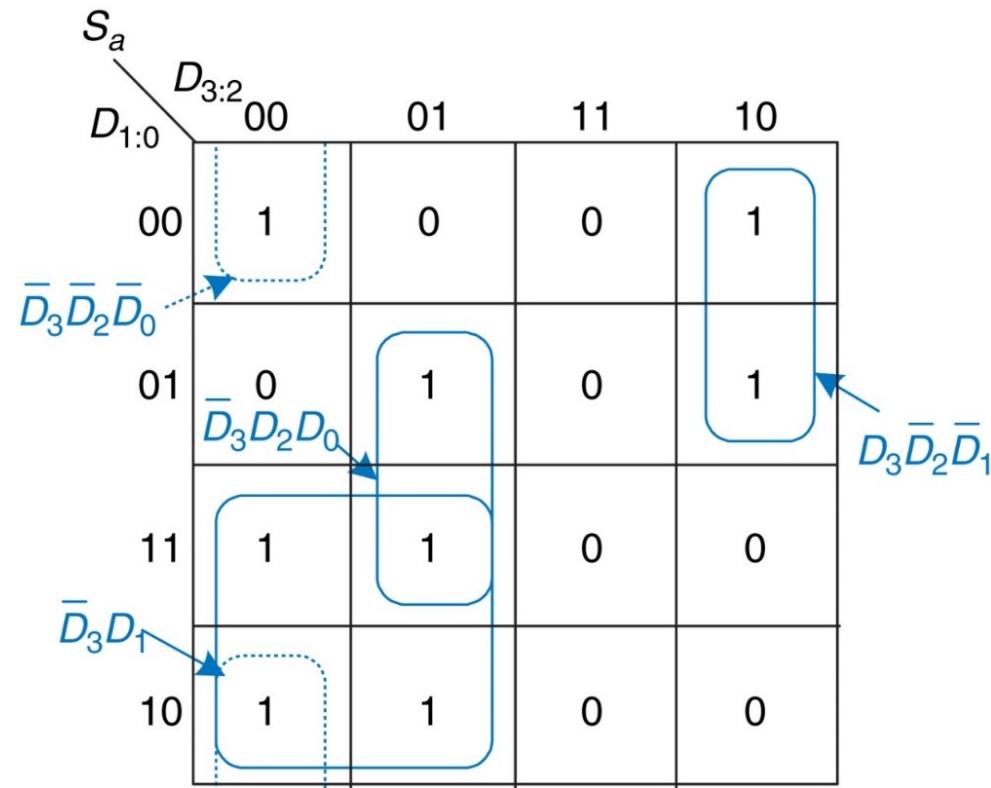
$$S_a = \bar{D}_3 D_1 + \bar{D}_3 D_2 D_0 + D_3 \bar{D}_2 \bar{D}_1 + \bar{D}_2 \bar{D}_1 \bar{D}_0$$

| $S_b$                           | $D_{3:2}$ | 00 | 01 | 11 | 10 |
|---------------------------------|-----------|----|----|----|----|
| $D_{1:0}$                       | 00        | 1  | 1  | 0  | 1  |
| $\bar{D}_3 \bar{D}_1 \bar{D}_0$ | 01        | 1  | 0  | 0  | 1  |
| $\bar{D}_3 D_1 D_0$             | 11        | 1  | 1  | 0  | 0  |
| $\bar{D}_3 \bar{D}_2$           | 10        | 1  | 0  | 0  | 0  |

$$S_b = \bar{D}_3 \bar{D}_2 + \bar{D}_2 \bar{D}_1 + \bar{D}_3 D_1 D_0 + \bar{D}_3 \bar{D}_1 \bar{D}_0$$

# Display a 7 segmenti

In generale, vi possono essere diverse forme minime:

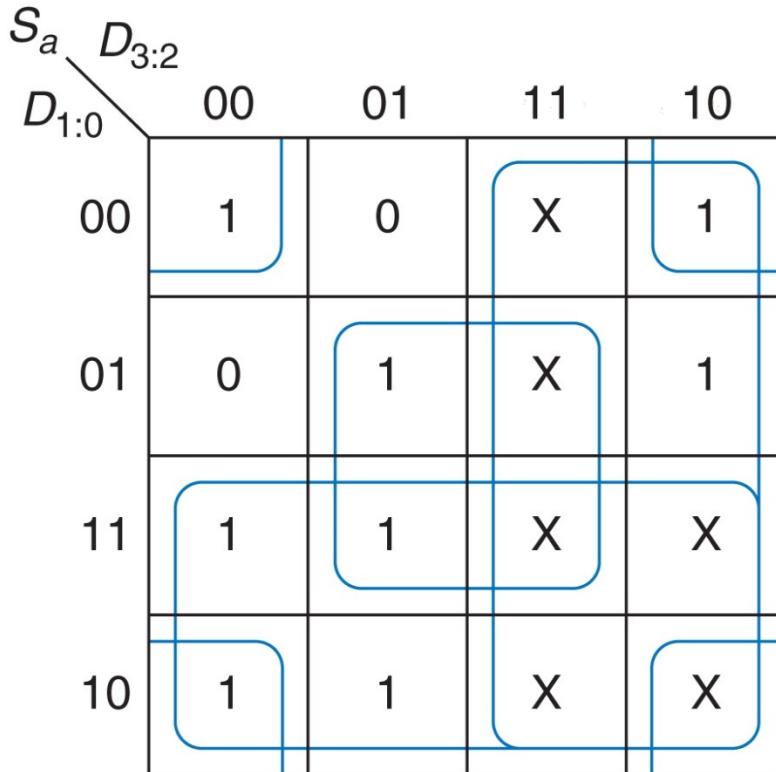


$$S_a = \bar{D}_3 D_1 + \bar{D}_3 D_2 D_0 + D_3 \bar{D}_2 \bar{D}_1 + \bar{D}_3 \bar{D}_2 \bar{D}_0$$

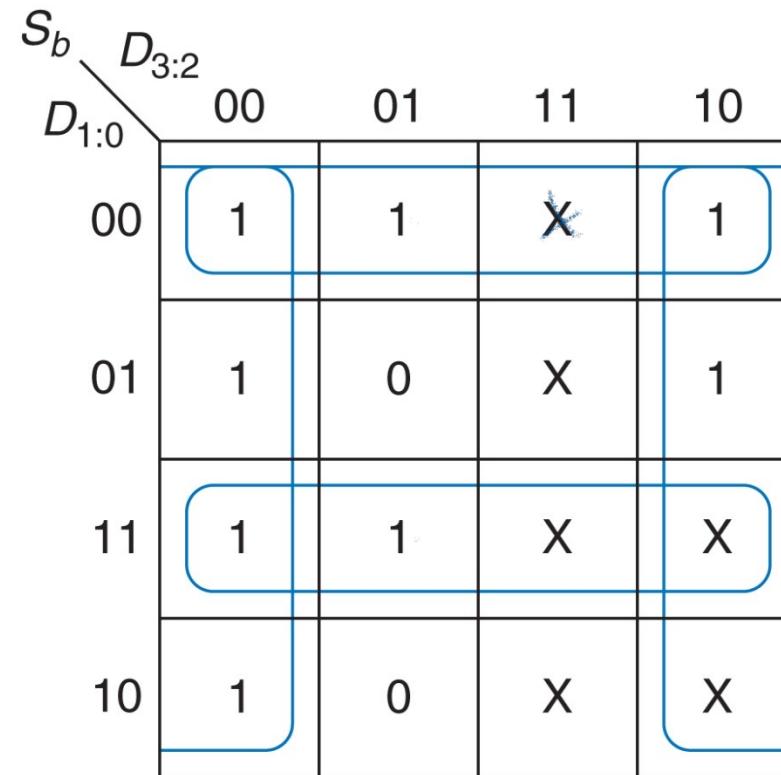
# Don't care in output

- Abbiamo visto nei circuiti a priorità come si possano utilizzare degli input «don't care» per avere una rappresentazione della tabella di verità più succinta
- Valori «don't care» (X) si possono avere anche in output allorché per una data configurazione degli input il valore dell'output è irrilevante (non ci interessa se sia 0 o 1)
- Ad esempio nel display a sette segmenti per gli input «illegali» 10-15 l'output può essere di tipo X
- Poiché non ci interessa se un valore X corrisponde in realtà ad uno 0 o un 1 allora nella minimizzazione di una espressione possiamo scegliere che valore dargli come più opportuno.
- In particolare in una K-map sostituiremo i valori X con degli 1 se questo consente di avere un numero inferiore di cerchi o cerchi più larghi

# K-map con valori «don't care»



$$S_a = D_3 + D_2 D_0 + \bar{D}_2 \bar{D}_0 + D_1$$



$$S_b = \bar{D}_2 + D_1 D_0 + \bar{D}_1 \bar{D}_0$$

# Esercizi su K-maps

| AB<br>CD | 00 | 01 | 11 | 10 |
|----------|----|----|----|----|
| 00       | 1  | 0  | 0  | 1  |
| 01       | 0  | 1  | 1  | 1  |
| 11       | 0  | 1  | 1  | 1  |
| 10       | 1  | 0  | 0  | 1  |

| AB<br>CD | 00 | 01 | 11 | 10 |
|----------|----|----|----|----|
| 00       | 1  | 0  | 0  | 1  |
| 01       | 1  | 1  | 0  | 1  |
| 11       | 1  | 1  | 1  | 1  |
| 10       | 1  | 0  | 0  | 1  |

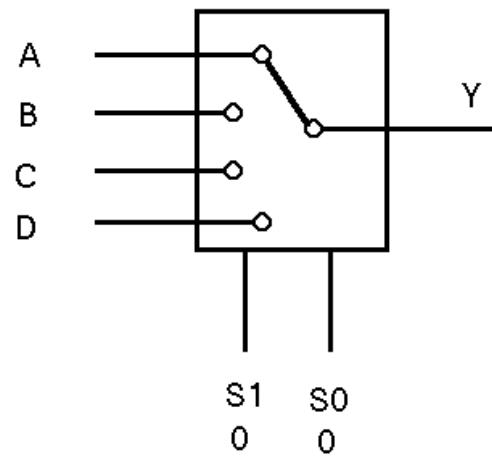
| AB<br>CD | 00 | 01 | 11 | 10 |
|----------|----|----|----|----|
| 00       | 1  | 0  | 0  | 0  |
| 01       | 0  | 1  | 1  | 1  |
| 11       | 1  | 1  | 1  | 1  |
| 10       | 1  | 0  | 0  | 0  |

| AB<br>CD | 00 | 01 | 11 | 10 |
|----------|----|----|----|----|
| 00       | 0  | 1  | 1  | 0  |
| 01       | 1  | 0  | 0  | 1  |
| 11       | 1  | 0  | 0  | 1  |
| 10       | 0  | 1  | 1  | 0  |

1

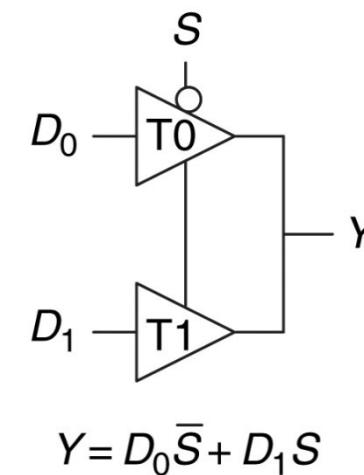
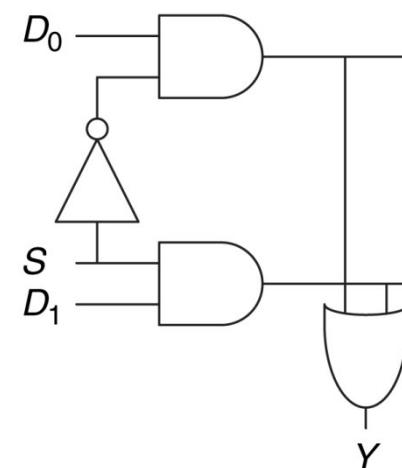
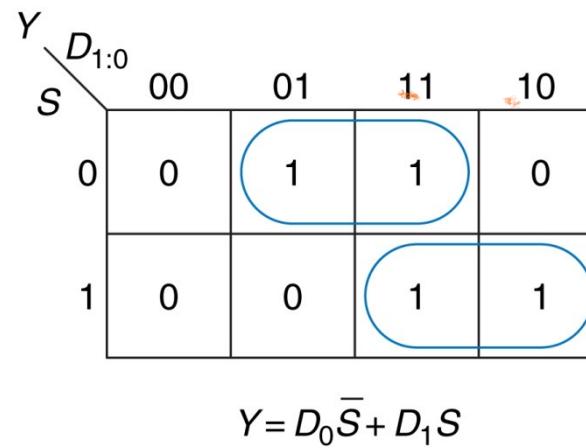
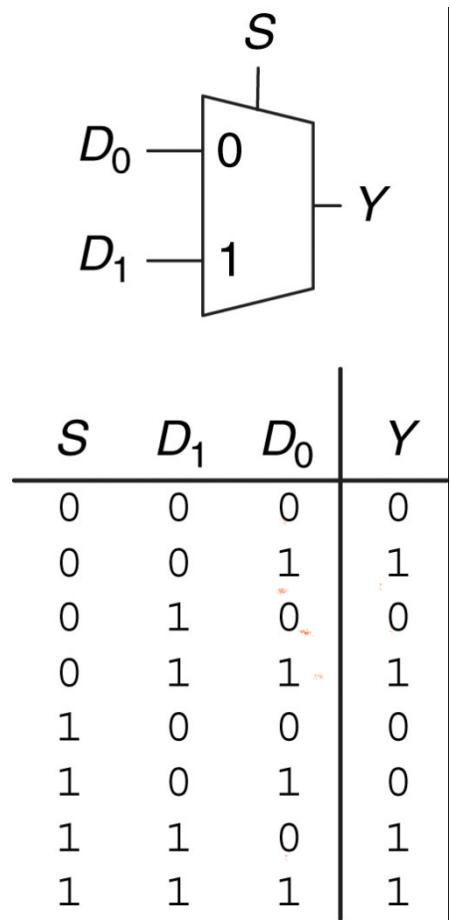
# Multiplexer

- Un multiplexer è sostanzialmente un selettore di linea

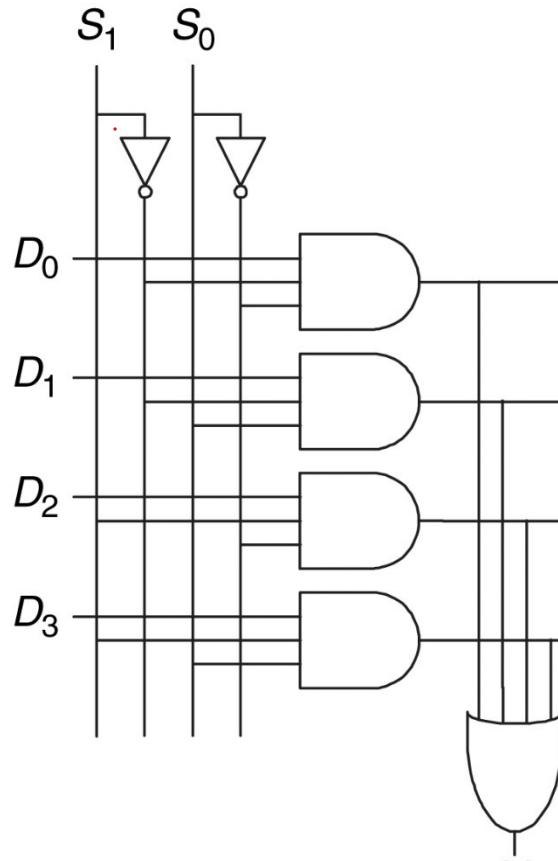
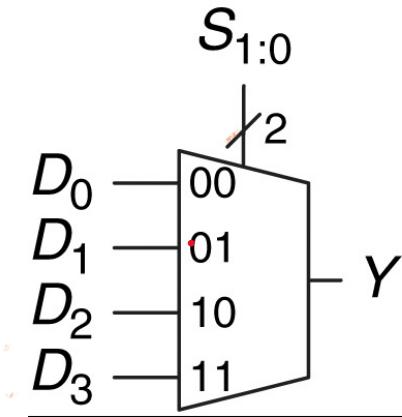


- In generale è costituito da  $N$  ingressi (dove  $N$  è una potenza di 2), 1 uscita, e  $\log_2 N$  linee di selezione che indicano a quale ingresso deve corrispondere l'output

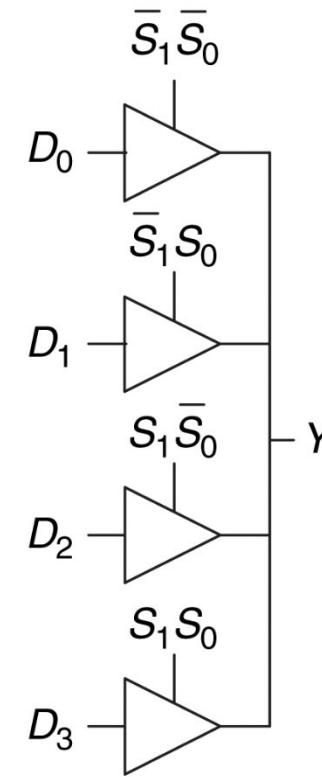
# Multiplexer 2:1



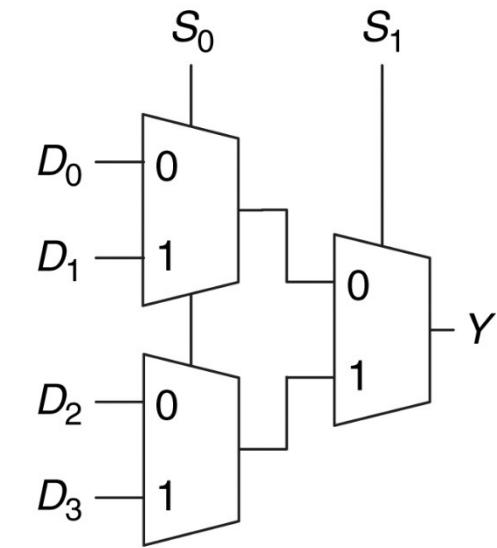
# Multiplexer 4:1



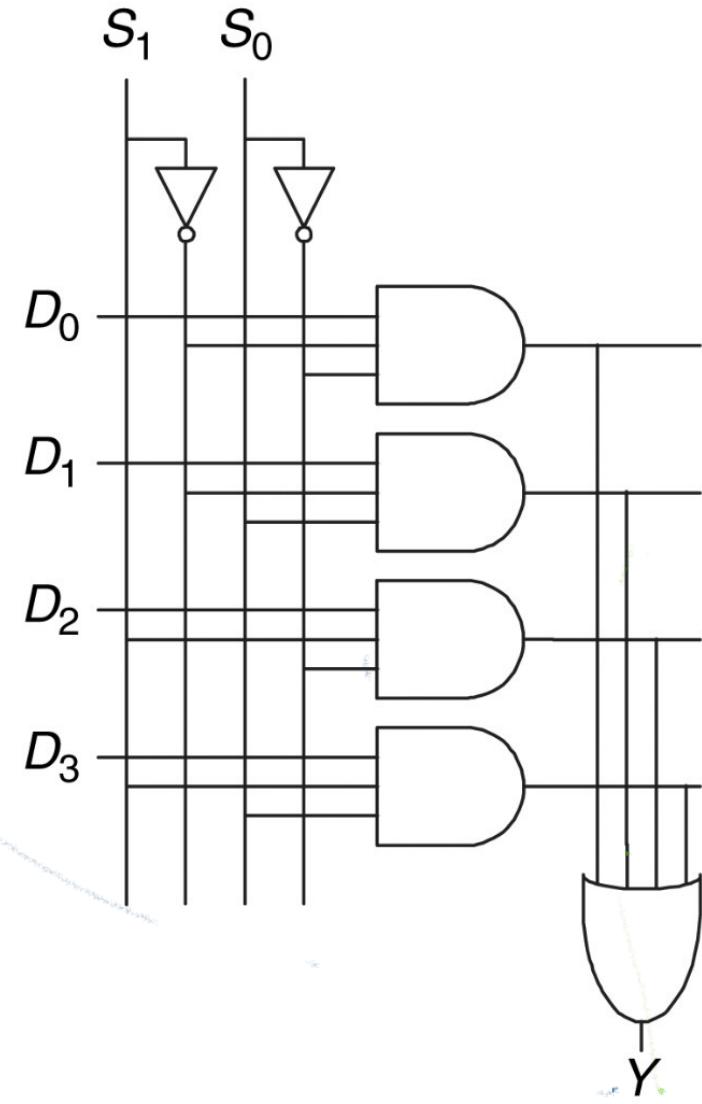
(a)



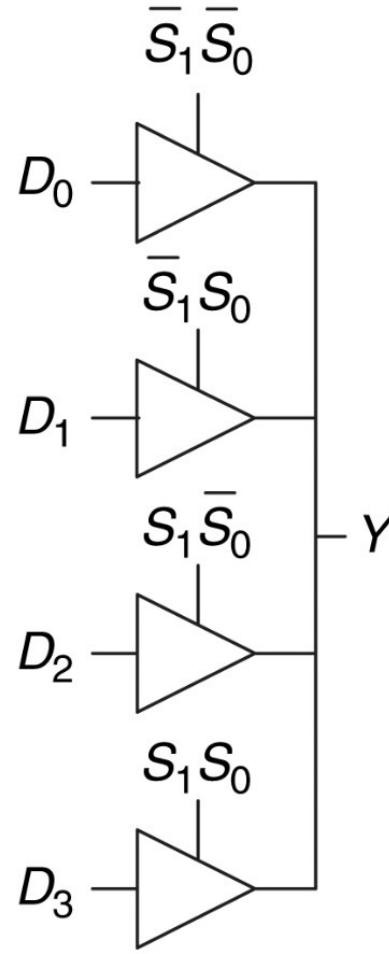
(b)



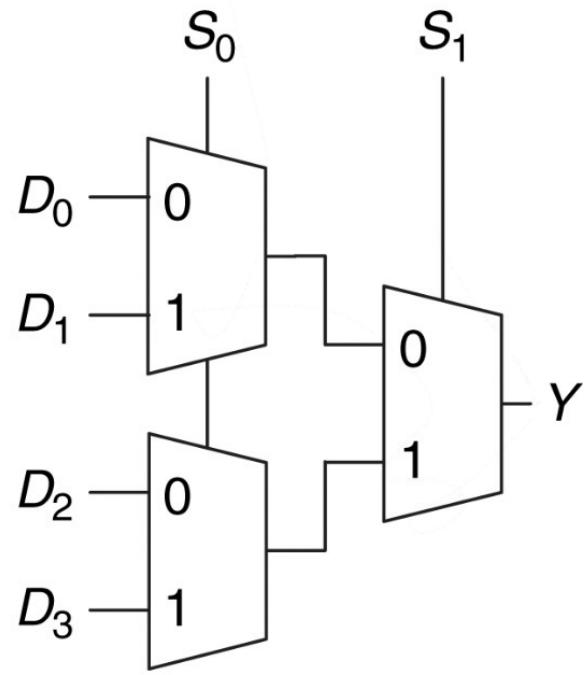
(c)



**(a)**



**(b)**



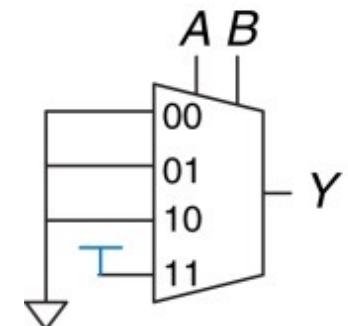
**(c)**

# Sintetizzare funzioni booleane con mux

- I multiplexer possono essere usati anche per sintetizzare delle funzioni booleane
- Sintetizzare una funzione di  $m$  variabili con un mux a  $2^m$  linee è molto semplice: le variabili saranno linee di selezione. Data una certa configurazione delle variabili, la linea di ingresso corrispondente sarà posta al valore della funzione in quella configurazione
- Di fatto le linee di ingresso riproducono la tabella di verità della funzione

| A | B | Y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

$Y = AB$

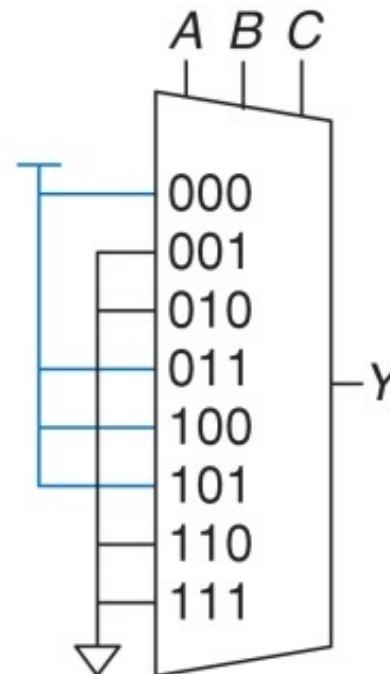


# Sintetizzare funzioni booleane con mux

| A | B | C | Y |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 |

$$Y = A\bar{B} + \bar{B}\bar{C} + \bar{A}BC$$

(a)

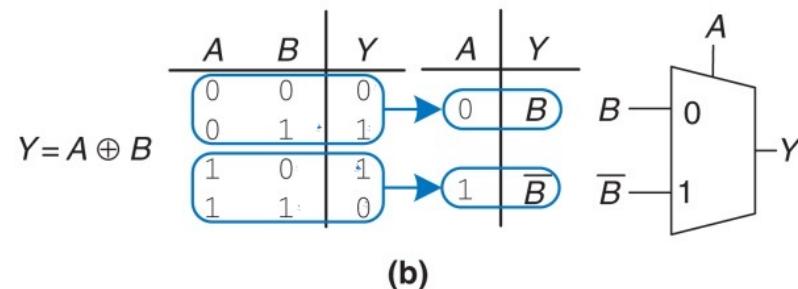
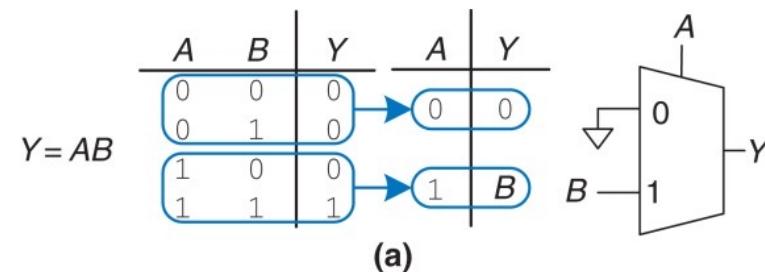


(b)

$$Y = \Sigma(0,3,4,5)$$

# Sintetizzare funzioni booleane con mux

- E' possibile utilizzare un mux con  $2^{m-1}$  ingressi per sintetizzare un funzione ad m variabili
- Le prime  $m-1$  variabili saranno linee di selezione, mentre le linee di ingresso possono essere poste a 0,1, oppure all'ultima variabile (positiva o negata)



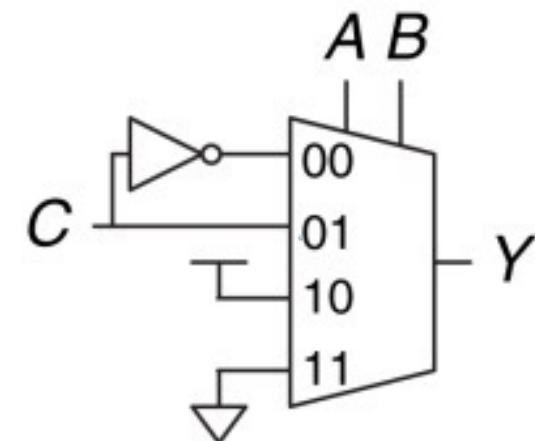
# Sintetizzare funzioni booleane con mux

| A | B | C | Y |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 |

(a)

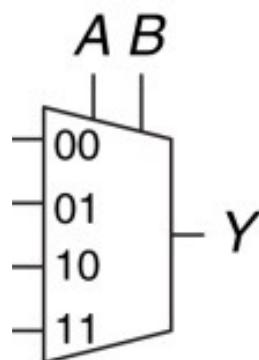
(b)

(c)



# Sintetizzare funzioni booleane con mux

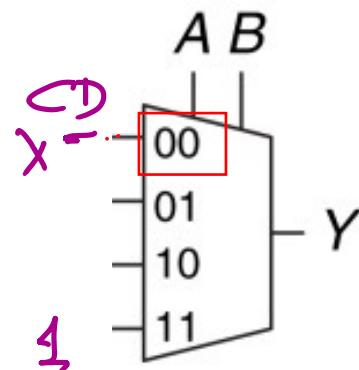
Supponiamo di avere mux 4:1 e di voler sintetizzare una funzione di 4 variabili



| <b>A B C D</b> | <b>Y</b> |
|----------------|----------|
| 0 0 0 0        | 0        |
| 0 0 0 1        | 0        |
| 0 0 1 0        | 0        |
| 0 0 1 1        | 1        |
| 0 1 0 0        | 1        |
| 0 1 0 1        | 1        |
| 0 1 1 0        | 0        |
| 0 1 1 1        | 0        |
| 1 0 0 0        | 1        |
| 1 0 0 1        | 1        |
| 1 0 1 0        | 0        |
| 1 0 1 1        | 0        |
| 1 1 0 0        | 1        |
| 1 1 0 1        | 1        |
| 1 1 1 0        | 1        |
| 1 1 1 1        | 1        |

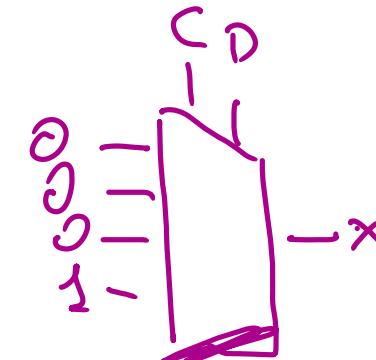
# Sintetizzare funzioni booleane con mux

Supponiamo di avere mux 4:1 e di voler sintetizzare una funzione di 4 variabili



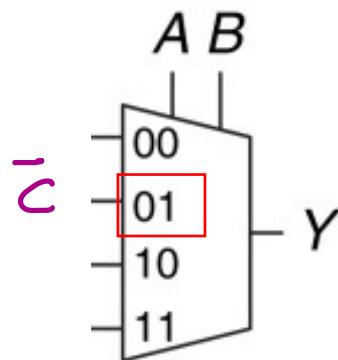
| A B C D | Y |
|---------|---|
| 0 0 0 0 | 0 |
| 0 0 0 1 | 0 |
| 0 0 1 0 | 0 |
| 0 0 1 1 | 1 |
| 0 1 0 0 | 1 |
| 0 1 0 1 | 1 |
| 0 1 1 0 | 0 |
| 0 1 1 1 | 0 |
| 1 0 0 0 | 1 |
| 1 0 0 1 | 1 |
| 1 0 1 0 | 0 |
| 1 0 1 1 | 0 |
| 1 1 0 0 | 1 |
| 1 1 0 1 | 1 |
| 1 1 1 0 | 1 |
| 1 1 1 1 | 1 |

CD



# Sintetizzare funzioni booleane con mux

Supponiamo di avere mux 4:1 e di voler sintetizzare una funzione di 4 variabili

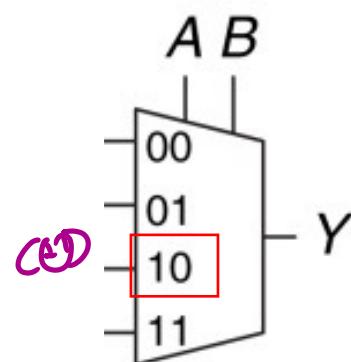


| A B C D | Y |
|---------|---|
| 0 0 0 0 | 0 |
| 0 0 0 1 | 0 |
| 0 0 1 0 | 0 |
| 0 0 1 1 | 1 |
| 0 1 0 0 | 1 |
| 0 1 0 1 | 1 |
| 0 1 1 0 | 0 |
| 0 1 1 1 | 0 |
| 1 0 0 0 | 1 |
| 1 0 0 1 | 1 |
| 1 0 1 0 | 0 |
| 1 0 1 1 | 0 |
| 1 1 0 0 | 1 |
| 1 1 0 1 | 1 |
| 1 1 1 0 | 1 |
| 1 1 1 1 | 1 |

$\bar{C}$

# Sintetizzare funzioni booleane con mux

Supponiamo di avere mux 4:1 e di voler sintetizzare una funzione di 4 variabili

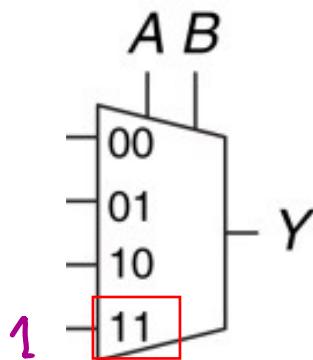


| A B C D | Y |
|---------|---|
| 0 0 0 0 | 0 |
| 0 0 0 1 | 0 |
| 0 0 1 0 | 0 |
| 0 0 1 1 | 1 |
| 0 1 0 0 | 1 |
| 0 1 0 1 | 1 |
| 0 1 1 0 | 0 |
| 0 1 1 1 | 0 |
| 1 0 0 0 | 0 |
| 1 0 0 1 | 1 |
| 1 0 1 0 | 1 |
| 1 0 1 1 | 0 |
| 1 1 0 0 | 1 |
| 1 1 0 1 | 1 |
| 1 1 1 0 | 1 |
| 1 1 1 1 | 1 |

$$C \oplus D$$

# Sintetizzare funzioni booleane con mux

Supponiamo di avere mux 4:1 e di voler sintetizzare una funzione di 4 variabili



| A B C D | Y |
|---------|---|
| 0000    | 0 |
| 0001    | 0 |
| 0010    | 0 |
| 0011    | 1 |
| 0100    | 1 |
| 0101    | 1 |
| 0110    | 0 |
| 0111    | 0 |
| 1000    | 1 |
| 1001    | 1 |
| 1010    | 0 |
| 1011    | 0 |
| 1100    | 1 |
| 1101    | 1 |
| 1110    | 1 |
| 1111    | 1 |

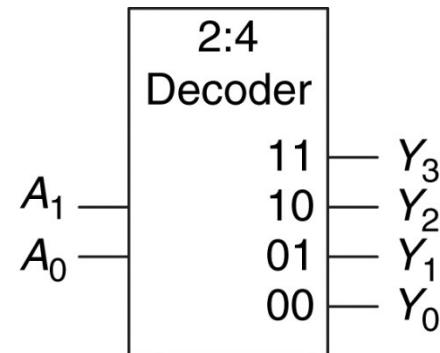
1

# Esercizi

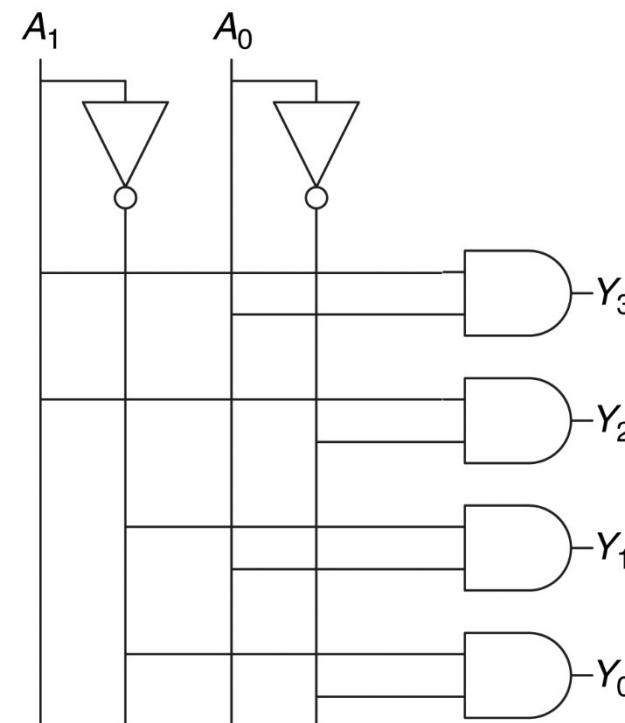
- Sintetizzare con multiplexer 4:1 la funzione di 4 variabili  $\Sigma(0,5,9,10,13)$
- Sintetizzare con multiplexer 4:1 la funzione di 4 variabili  $\Sigma(1,2,3,12,13)$
- Sintetizzare con multiplexer 4:1 la funzione di 4 variabili  $\Sigma(0,1,2,3,5,15)$
- Sintetizzare con multiplexer 4:1 la funzione di 4 variabili  $\Sigma(1,2,6,7,8,13)$

# Decoder

- Un decoder ha  $N$  linee di ingresso e  $2^N$  linee di uscita
- se  $m$  è numero rappresentato dagli input allora solo l' $m$ -esima linea di uscita è pari a 1 mentre tutte le altre sono a 0

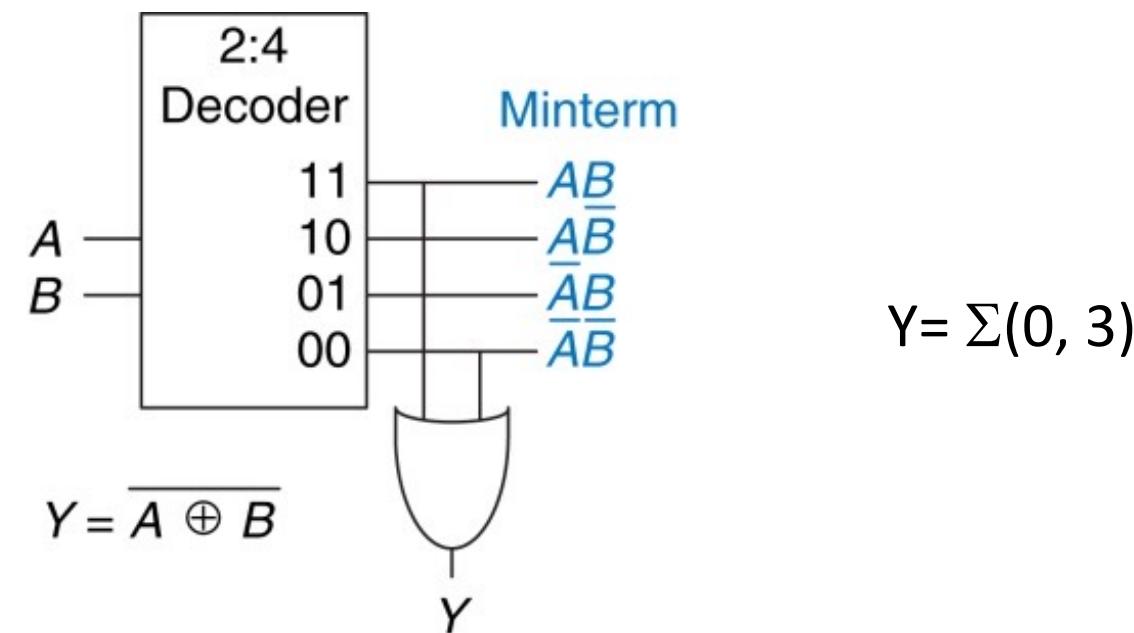


| $A_1$ | $A_0$ | $Y_3$ | $Y_2$ | $Y_1$ | $Y_0$ |
|-------|-------|-------|-------|-------|-------|
| 0     | 0     | 0     | 0     | 0     | 1     |
| 0     | 1     | 0     | 0     | 1     | 0     |
| 1     | 0     | 0     | 1     | 0     | 0     |
| 1     | 1     | 1     | 0     | 0     | 0     |



# Sintetizzare funzioni booleane con decoder

- Anche i decoder possono essere usati per sintetizzare funzioni booleane
- Basta mettere in OR tutte e solo le linee di uscita che occorrono nella sigma-expresione della funzione da sintetizzare



# Esercizi

| AB | 00 | 01 | 11 | 10 |
|----|----|----|----|----|
| CD |    |    |    |    |
| 00 | X  | 0  | 0  | X  |
| 01 | 0  | 1  | X  | 0  |
| 11 | 1  | 1  | 1  | 0  |
| 10 | 1  | X  | 0  | 1  |

| <b>AB</b> | <b>00</b> | <b>01</b> | <b>11</b> | <b>10</b> |
|-----------|-----------|-----------|-----------|-----------|
| <b>CD</b> |           |           |           |           |
| <b>00</b> | X         | 0         | 0         | X         |
| <b>01</b> | 0         | 1         | X         | 0         |
| <b>11</b> | 1         | 1         | 1         | 0         |
| <b>10</b> | 1         | X         | 0         | 1         |

$$A^*C + B^*D^* + BD$$

| AB | 00 | 01 | 11 | 10 |
|----|----|----|----|----|
| CD |    |    |    |    |
| 00 | X  | 0  | 0  | X  |
| 01 | 1  | 1  | X  | X  |
| 11 | 1  | 0  | 0  | 1  |
| 10 | 0  | X  | 0  | 1  |

| <b>AB</b> | <b>00</b> | <b>01</b> | <b>11</b> | <b>10</b> |
|-----------|-----------|-----------|-----------|-----------|
| <b>CD</b> |           |           |           |           |
| <b>00</b> | x         | 0         | 0         | x         |
| <b>01</b> | 1         | 1         | x         | x         |
| <b>11</b> | 1         | 0         | 0         | 1         |
| <b>10</b> | 0         | x         | 0         | 1         |

$$C^*D + AB^* + B^*D$$

| AB | 00 | 01 | 11 | 10 |
|----|----|----|----|----|
| CD |    |    |    |    |
| 00 | X  | 0  | 0  | X  |
| 01 | 1  | 0  | X  | X  |
| 11 | 1  | 0  | 0  | 1  |
| 10 | 1  | X  | 0  | 1  |

| <b>AB</b> | <b>00</b> | <b>01</b> | <b>11</b> | <b>10</b> |
|-----------|-----------|-----------|-----------|-----------|
| <b>CD</b> |           |           |           |           |
| <b>00</b> | X         | 0         | 0         | X         |
| <b>01</b> | 1         | 0         | X         | X         |
| <b>11</b> | 1         | 0         | 0         | 1         |
| <b>10</b> | 1         | X         | 0         | 1         |

B\*

A cosa è uguale la espressione  $(A+C)(A^*+B)(B+C)$

- (A).  $(A+B)(A+C)$
- (B).  $(A+B)(A^*+C)$
- (C).  $(A+C)(A+B^*)$
- (D).  $(A+C)(A^*+B)$
- (E).  $(A+C^*)(A+B^*)$

A cosa è uguale la espressione  $(A+C)(A^*+B)(B+C)$

- (A).  $(A+B)(A+C)$
- (B).  $(A+B)(A^*+C)$
- (C).  $(A+C)(A+B^*)$
- (D).  $(A+C)(A^*+B)$**
- (E).  $(A+C^*)(A+B^*)$

Indicare il numero di funzioni booleane (distinte) di quattro variabili che soddisfano la condizione:  $f(0,B,C,D)=0$ , per qualsiasi valore di B,C e D.

Indicare il numero di funzioni booleane (distinte) di quattro variabili che soddisfano la condizione:  $f(0,B,C,D)=0$ , per qualsiasi valore di B,C e D.

| A | B | C | D | Y |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | x |
| 1 | 0 | 0 | 1 | x |
| 1 | 0 | 1 | 0 | x |
| 1 | 0 | 1 | 1 | x |
| 1 | 1 | 0 | 0 | x |
| 1 | 1 | 0 | 1 | x |
| 1 | 1 | 1 | 0 | x |
| 1 | 1 | 1 | 1 | x |

$$2^8=256$$

Indicare il numero di funzioni booleane (distinte) di quattro variabili che soddisfano la condizione: se  $A > D$  allora  $f(A,B,C,D) = 1$ .

Indicare il numero di funzioni booleane (distinte) di quattro variabili che soddisfano la condizione: se  $A > D$  allora  $f(A,B,C,D)=1$ .

| A | B | C | D | Y |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | x |
| 0 | 0 | 0 | 1 | x |
| 0 | 0 | 1 | 0 | x |
| 0 | 0 | 1 | 1 | x |
| 0 | 1 | 0 | 0 | x |
| 0 | 1 | 0 | 1 | x |
| 0 | 1 | 1 | 0 | x |
| 0 | 1 | 1 | 1 | x |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | x |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 1 | x |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 | x |
| 1 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | x |

$$2^{12} = 4096$$

Ridurre la seguente espressione booleana in forma SOP minima:

$$\overline{(A \cdot \bar{C}) \oplus (B + D)} + A\bar{C}\bar{D}$$

Ridurre la seguente espressione booleana in forma SOP minima:

$$\begin{aligned}\overline{(A \cdot \bar{C}) \oplus (B + D)} + A\bar{C}\bar{D} &= (A \cdot \bar{C}) \cdot (B + D) + \overline{(A \cdot \bar{C})} \cdot \\ \overline{(B + D)} + A\bar{C}\bar{D} &= (A \cdot \bar{C}) \cdot (B + D) + (\bar{A} + C) \cdot \bar{B} \cdot \bar{D} + A\bar{C}\bar{D} = \\ A\bar{B}\bar{C} + A\bar{C}D + \bar{A}\bar{B}\bar{D} + \bar{B}C\bar{D} + A\bar{C}\bar{D} &= A\bar{C} + \bar{B}\bar{D}\end{aligned}$$

| AB | 00 | 01 | 11 | 10 |
|----|----|----|----|----|
| CD |    |    |    |    |
| 00 | 1  | 0  | 1  | 1  |
| 01 | 0  | 0  | 1  | 1  |
| 11 | 0  | 0  | 0  | 0  |
| 10 | 1  | 0  | 0  | 1  |

Usando solo dei multiplexer 2:1 e i segnali di ingresso  $A, B, C, \bar{A}, \bar{B}, \bar{C}, 0, 1$  disegnare un circuito equivalente alla espressione booleana

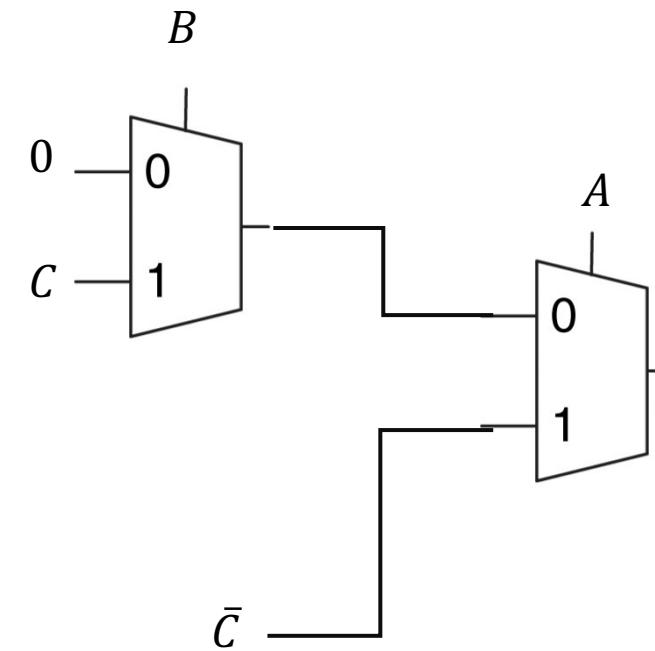
$$(A \oplus C) \cdot (A + B)$$

| A B C | Y |
|-------|---|
| 0 0 0 | 0 |
| 0 0 1 | 0 |
| 0 1 0 | 0 |
| 0 1 1 | 1 |
| 1 0 0 | 1 |
| 1 0 1 | 0 |
| 1 1 0 | 1 |
| 1 1 1 | 0 |

Usando solo dei multiplexer 2:1 e i segnali di ingresso  $A, B, C, \bar{A}, \bar{B}, \bar{C}, 0, 1$  disegnare un circuito equivalente alla espressione booleana

| A B C | Y |
|-------|---|
| 0 0 0 | 0 |
| 0 0 1 | 0 |
| 0 1 0 | 0 |
| 0 1 1 | 1 |
| 1 0 0 | 1 |
| 1 0 1 | 0 |
| 1 1 0 | 1 |
| 1 1 1 | 0 |

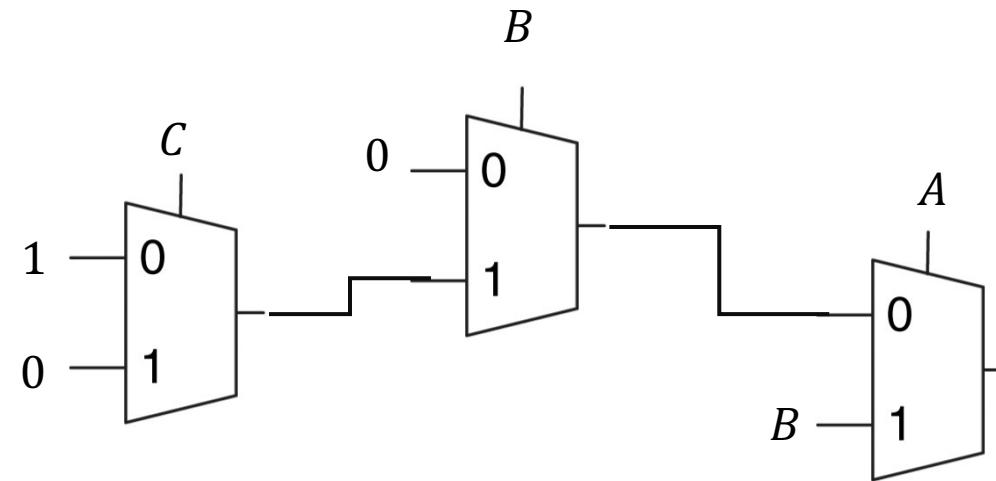
$$(A \oplus C) \cdot (A + B)$$



Usando solo 2 multiplexer 2:1 e i segnali di ingresso  $A, B, C, 0,1$  disegnare un circuito equivalente alla espressione booleana

| A B C | Y |
|-------|---|
| 0 0 0 | 0 |
| 0 0 1 | 0 |
| 0 1 0 | 1 |
| 0 1 1 | 0 |
| 1 0 0 | 0 |
| 1 0 1 | 0 |
| 1 1 0 | 1 |
| 1 1 1 | 1 |

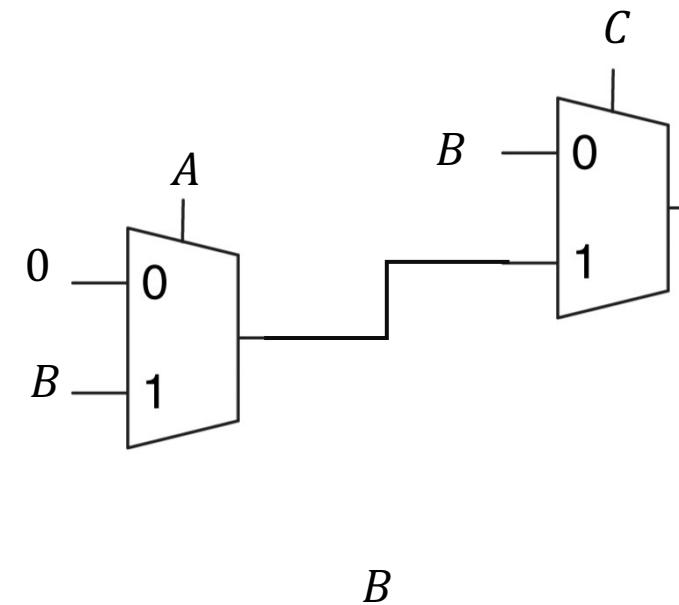
$$(B \cdot \bar{C}) + (A \cdot B)$$



Usando solo 2 multiplexer 2:1 e i segnali di ingresso  $A, B, C, 0,1$  disegnare un circuito equivalente alla espressione booleana

| A B C | Y |
|-------|---|
| 0 0 0 | 0 |
| 0 0 1 | 0 |
| 0 1 0 | 1 |
| 0 1 1 | 0 |
| 1 0 0 | 0 |
| 1 0 1 | 0 |
| 1 1 0 | 1 |
| 1 1 1 | 1 |

$$(B \cdot \bar{C}) + (A \cdot B)$$



# Circuiti sequenziali

# Logica sequenziale

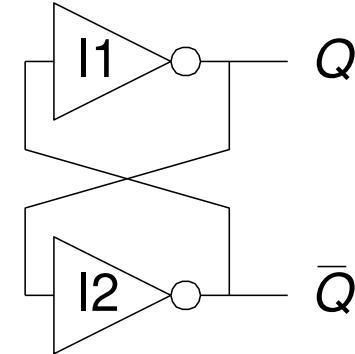
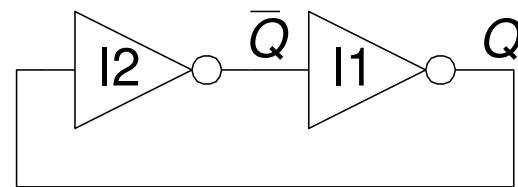
- Nei sistemi sequenziali l'output dipende sia dal valore corrente sia da valori precedenti dell'input. In tal senso si dice che il sistema ha *memoria*
- *Stato interno*: rappresenta l'informazione che mantiene la *storia* di un circuito sequenziale ed è necessaria per prevedere il suo comportamento futuro
- Come vedremo lo stato di un sistema sarà memorizzato in componenti come i latches e flip-flop
- Un circuito sequenziale (*sincrono*) avrà una topologia ben precisa:
  - logica combinatoria: definisce l'evoluzione del sistema
  - Banchi di flip-flop: servono a memorizzare gli stati del sistema
- Un aspetto peculiare dei sistemi sequenziali è quello della retroazione (*feedback*), ovvero il segnali di output vengono riportati in input

# State elements

- Lo stato di un circuito influenzerà l'evoluzione del sistema
- Gli *state elements* sono tutte quelle componenti circuitali che vengono adoperate per memorizzare lo stato di un circuito
  - Circuiti bistabili
  - SR Latch
  - D Latch
  - D Flip-flop

# Circuito bistabile

- *building block* per altri state elements
- Two outputs:  $Q$ ,  $\bar{Q}$
- No inputs

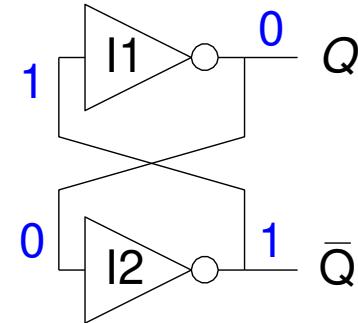


# Analisi del circuito bistabile

- Considera i due possibili casi:

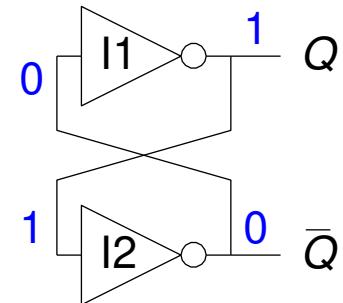
$Q = 0$ :

allora  $Q = 0, \bar{Q} = 1$  (consistente)



$Q = 1$ :

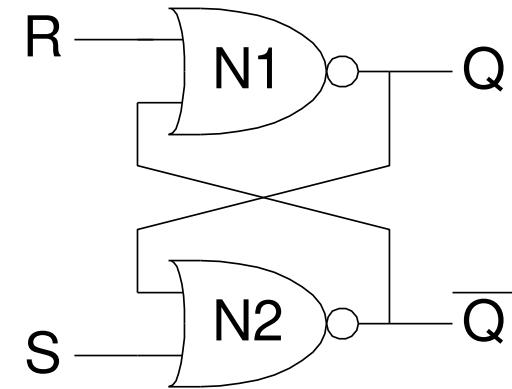
allora  $Q = 1, \bar{Q} = 0$  (consistente)



- Memorizza 1 bit nella variabile di stato  $Q$  (or  $\bar{Q}$ )
- Ma non ci sono input per controllare questo stato!

# SR (Set/Reset) Latch

- SR Latch
- Consideriamo i 4 possibili stati:
  - $S=1, R=0$
  - $S=0, R=1$
  - $S=0, R=0$
  - $S=1, R=1$



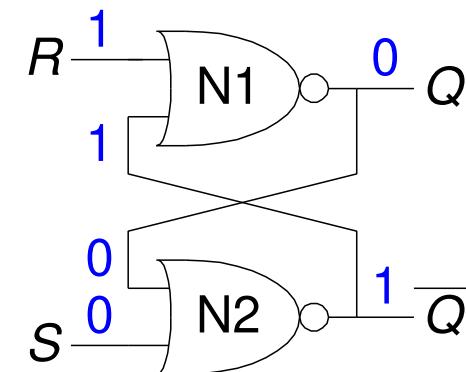
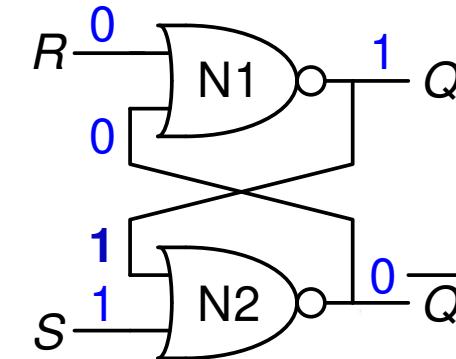
# Analisi di un SR Latch

$S = 1, R = 0:$

allora  $Q = 1$  e  $\bar{Q} = 0$

$S = 0, R = 1:$

allora  $Q = 0$  e  $\bar{Q} = 1$



# Analisi di un SR Latch

$S = 1, R = 0:$

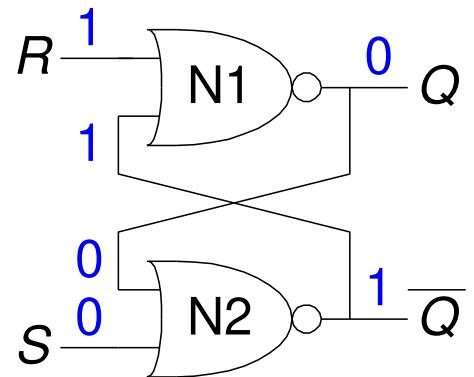
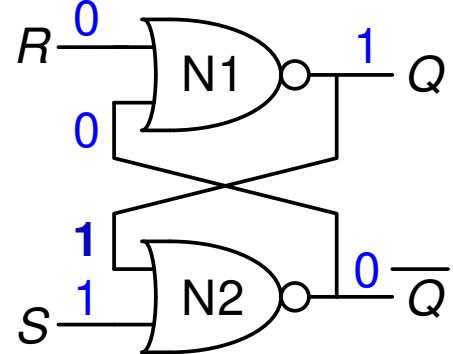
allora  $Q = 1$  e  $\bar{Q} = 0$

*operazione Set*

$S = 0, R = 1:$

allora  $Q = 0$  e  $\bar{Q} = 1$

*operazione Reset*

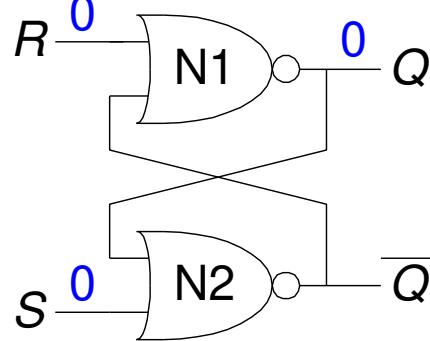


# Analisi di un SR Latch

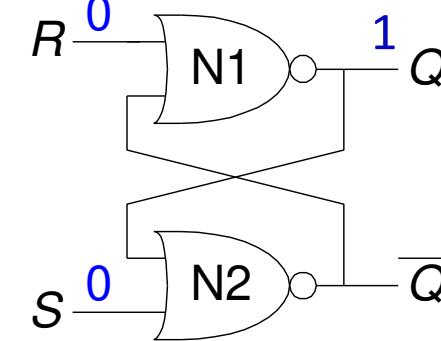
$S = 0, R = 0:$

allora  $Q = Q_{prev}$

$Q_{prev} = 0$

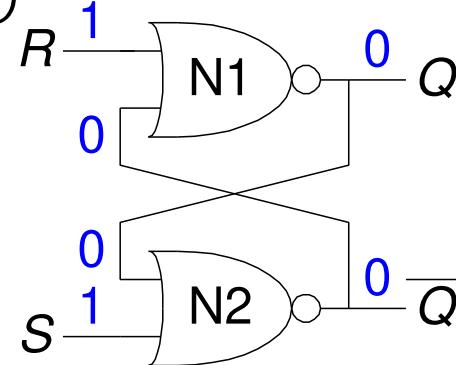


$Q_{prev} = 1$



$S = 1, R = 1:$

allora  $Q = 0, \bar{Q} = 0$

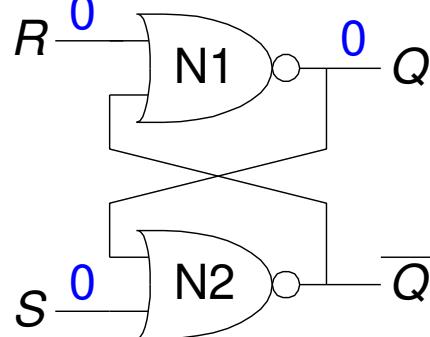


# Analisi di un SR Latch

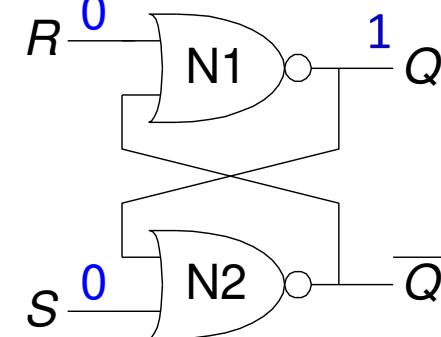
$S = 0, R = 0:$

allora  $Q = Q_{prev}$   
**Memoria**

$Q_{prev} = 0$



$Q_{prev} = 1$

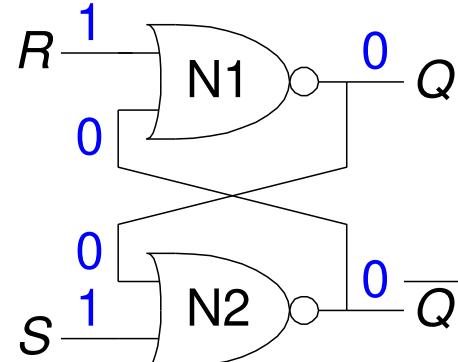


$S = 1, R = 1:$

allora  $Q = 0, \bar{Q} = 0$

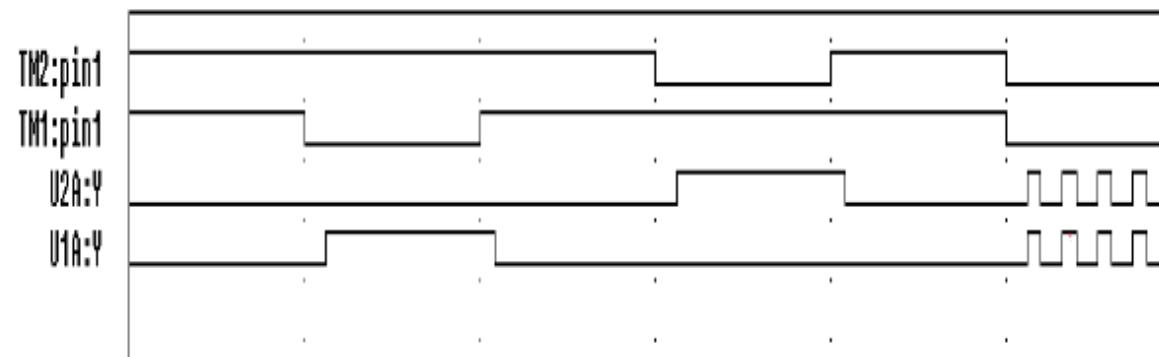
**Stato non valido**

$Q \neq \text{NOT } \bar{Q}$



# Analisi di un SR Latch

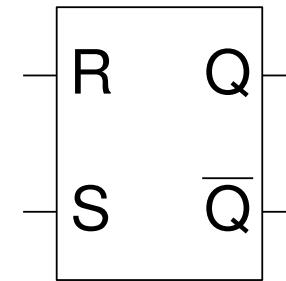
- Se dalla condizione  $S=R=1$  si passa alla condizione  $S=R=0$  allora
  - Se i tempi di propagazione sono uguali allora il circuito va in oscillazione
  - Nell'ipotesi, più realistica che le porte abbiano ritardi anche lievemente differenti, il circuito si mette in uno dei due stati possibili. Anche in questo caso, però, lo stato finale non è predicibile



# Simbolo per un SR Latch

- SR sta per Set/Reset
  - Memorizza un bit ( $Q$ )
- **Set:** Pone l'output a 1  
( $S = 1, R = 0, Q = 1$ )
- **Reset:** Pone l'output a 0  
( $S = 0, R = 1, Q = 0$ )
- **Memoria:** mantiene memoria dell'output  
( $S = 0, R = 0, Q = Q_{\text{prev}}$ )

SR Latch  
Symbol

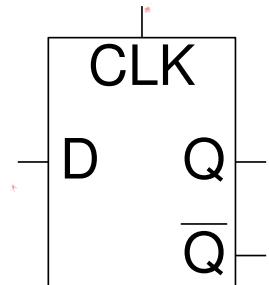


*Occorre evitare lo stato non valido  $S = R = 1$*

# D Latch

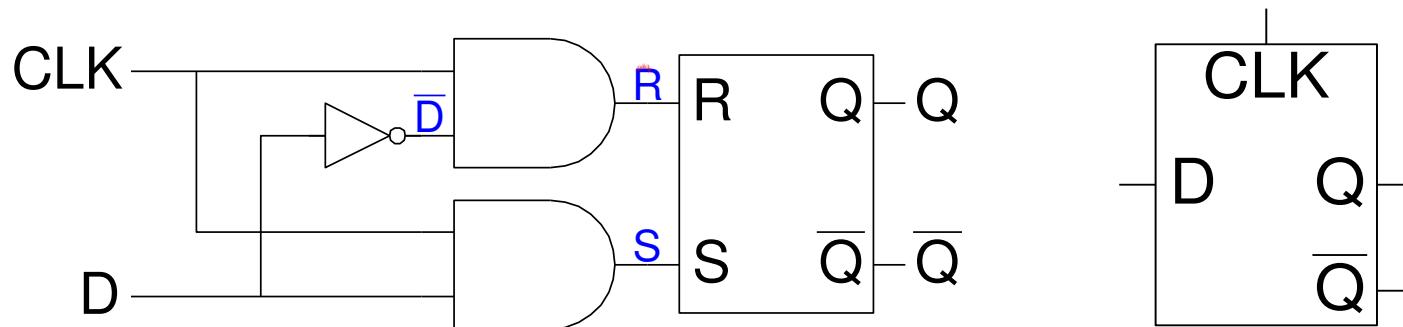
- 2 input:  $CLK$ ,  $D$
- $CLK$ : controlla *quando* l'output cambia
- $D$  (data input): controlla *in che cosa* l'output cambia
- Se  $CLK = 1$ ,  
 $D$  passa fino a  $Q$  (*transparente*)
- Se  $CLK = 0$ ,  
 $Q$  mantiene il suo valore precedente (*opaco*)

D Latch  
Symbol



Evita lo stato non valido in cui  $Q \neq \text{NOT } Q$

# D Latch

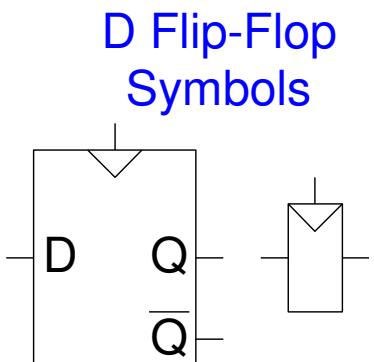


| $CLK$ | $D$ | $\bar{D}$ | $S$ | $R$ | $Q$        | $\bar{Q}$        |
|-------|-----|-----------|-----|-----|------------|------------------|
| 0     | X   | $\bar{X}$ | 0   | 0   | $Q_{prev}$ | $\bar{Q}_{prev}$ |
| 1     | 0   | 1         | 0   | 1   | 0          | 1                |
| 1     | 1   | 0         | 1   | 0   | 1          | 0                |

# D Flip-Flop

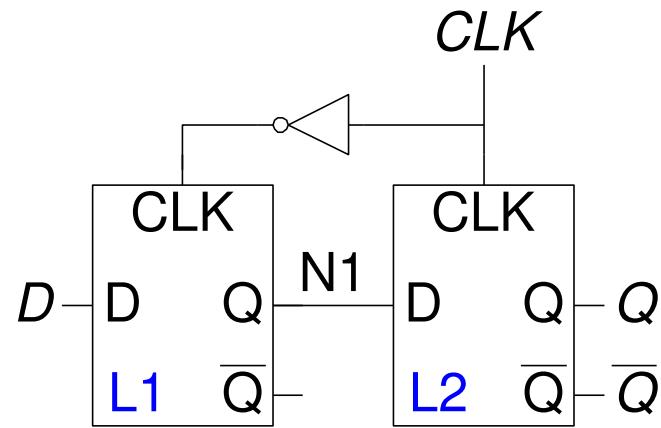
- Inputs:  $CLK, D$
- Funzione:
  - Quando  $CLK$  passa da 0 a 1,  $D$  passa fino a  $Q$
  - Altrimenti,  $Q$  mantiene il suo valore precedente
- $Q$  cambia solo durante la transizione di  $CLK$  da 0 a 1

*Queste tipologie di componenti sono dette edge-triggered perché sono pilotate non da un valore ma da una transizione (di  $CLK$ )*

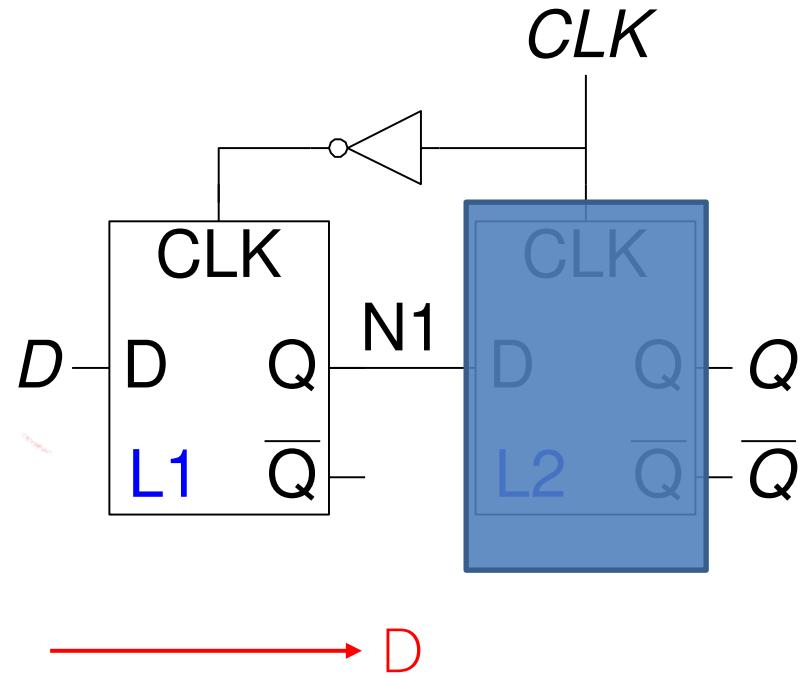


# D Flip-Flop

- 2 D latch (L1 e L2) controllati da clock complementari
- Quando  $CLK = 0$ 
  - L1 è trasparente
  - L2 è opaco
  - $D$  passa fino a N1
- Quando  $CLK = 1$ 
  - L2 è trasparente
  - L1 è opaco
  - N1 passa fino a  $Q$
- Quindi,  $D$  passa fino a  $Q$  sulla transizione di  $CLK$  da 0 a 1
- Ulteriori variazioni di  $D$  quando  $CLK=1$  (risp.  $CLK=0$ ) non passano a  $Q$  perché L1 (risp. L2) è opaco



# D Flip-Flop

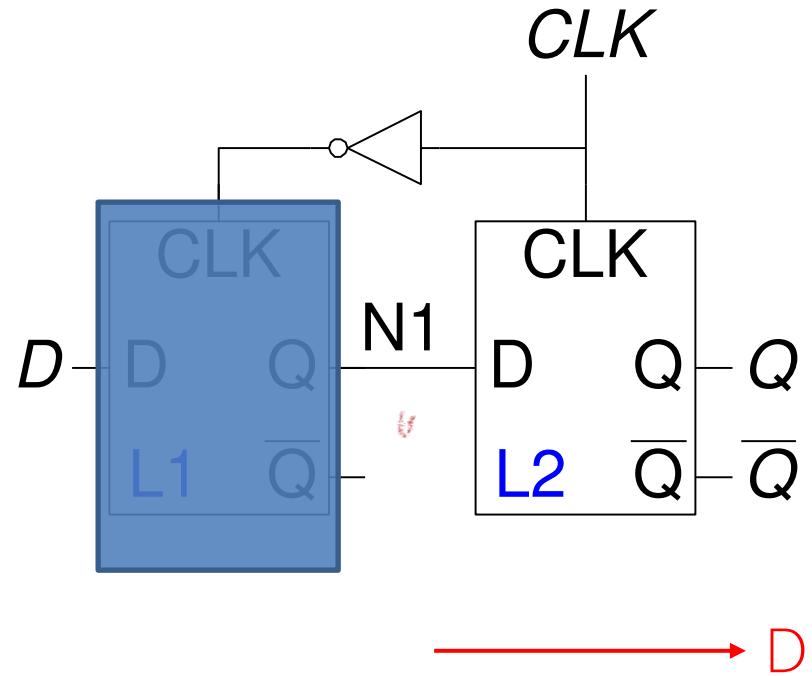


$$CLK = 0$$

$$N1 = D$$

$$Q = Q_{\text{prev}}$$

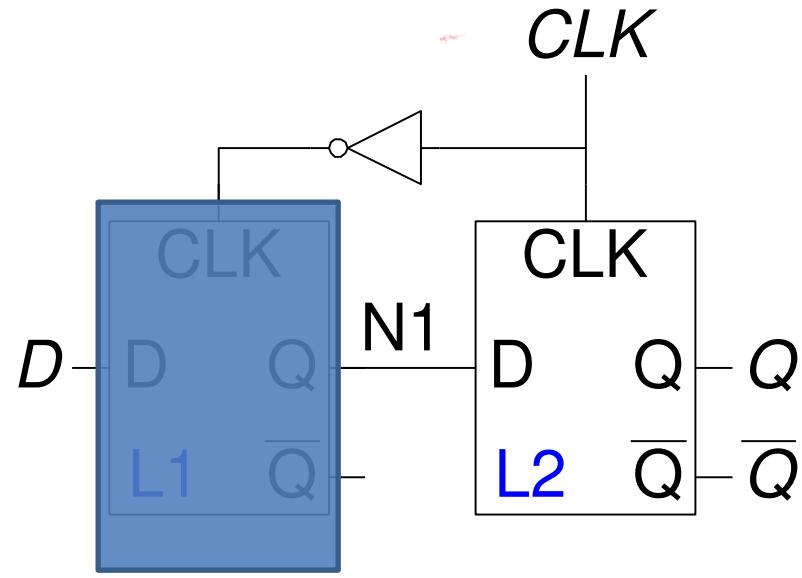
# D Flip-Flop



*CLK*  $0 \rightarrow 1$

$N1 = N1_{\text{prev}}$   
 $Q = N1$  (ovvero *D*)

# D Flip-Flop

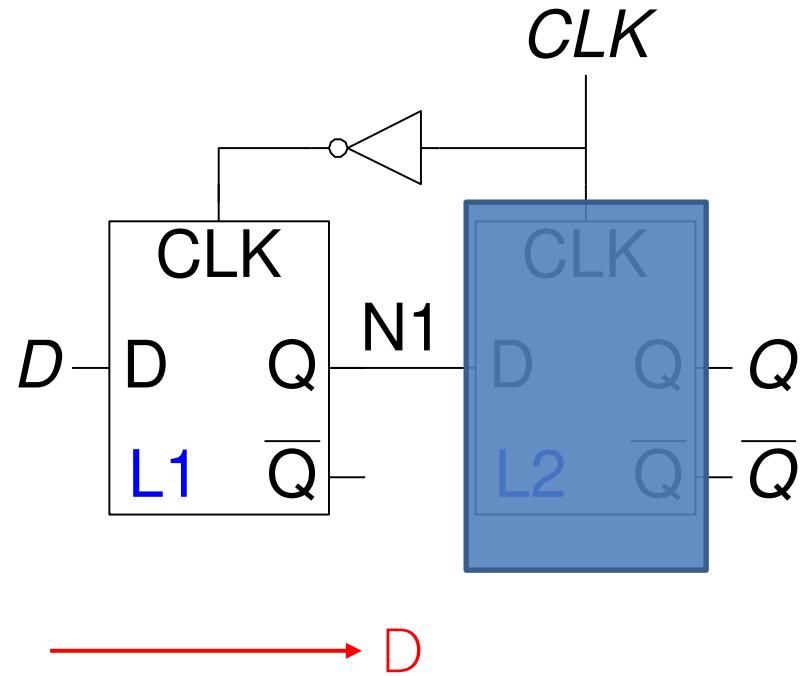


D

$CLK = 1$

$N1 = N1_{\text{prev}}$   
 $Q = N1$

# D Flip-Flop

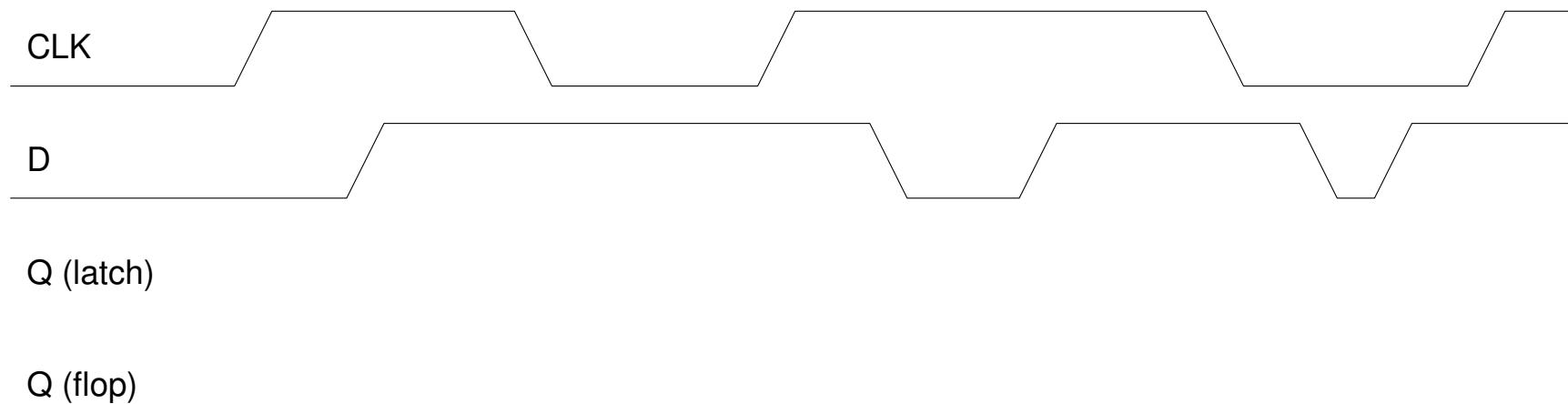
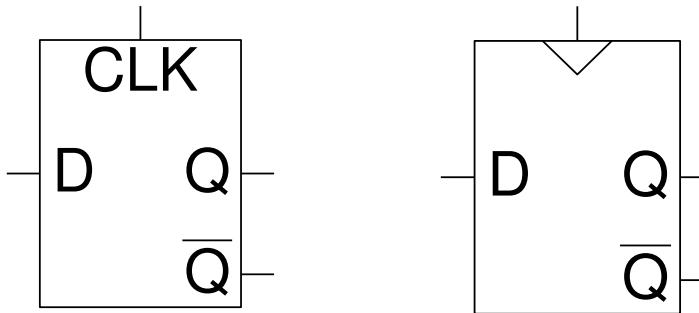


CLK 1 → 0

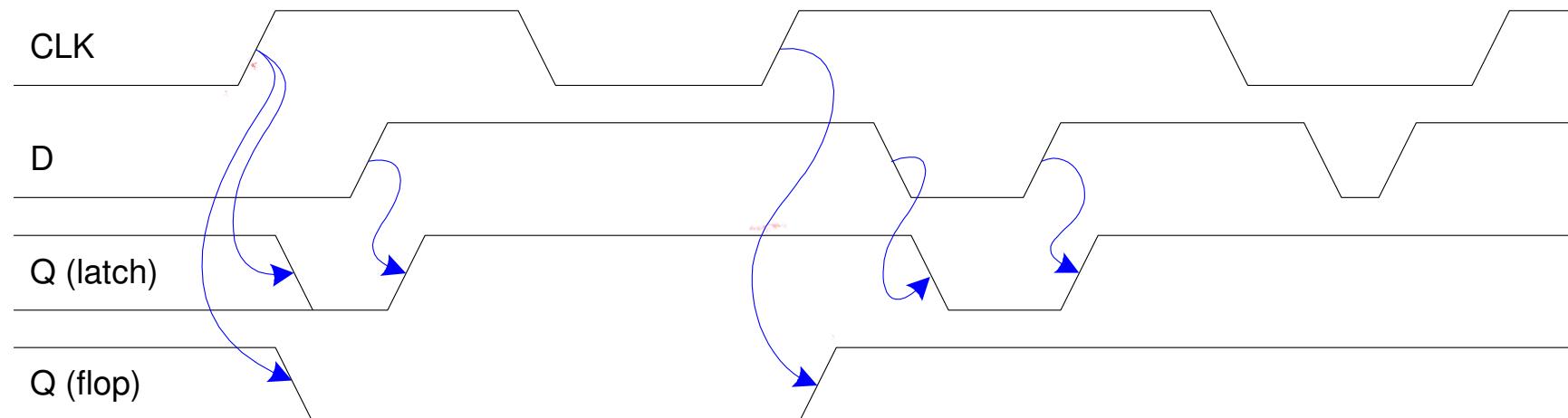
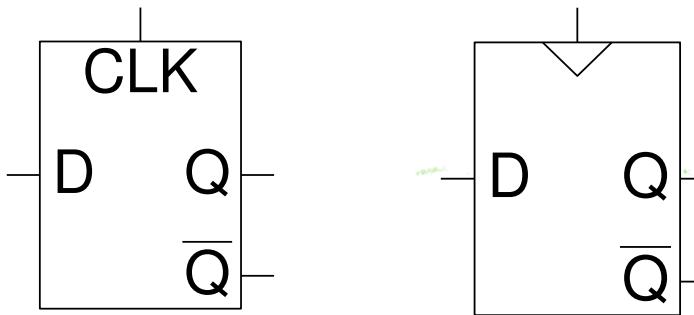
N1 = D

Q = Q<sub>prev</sub>

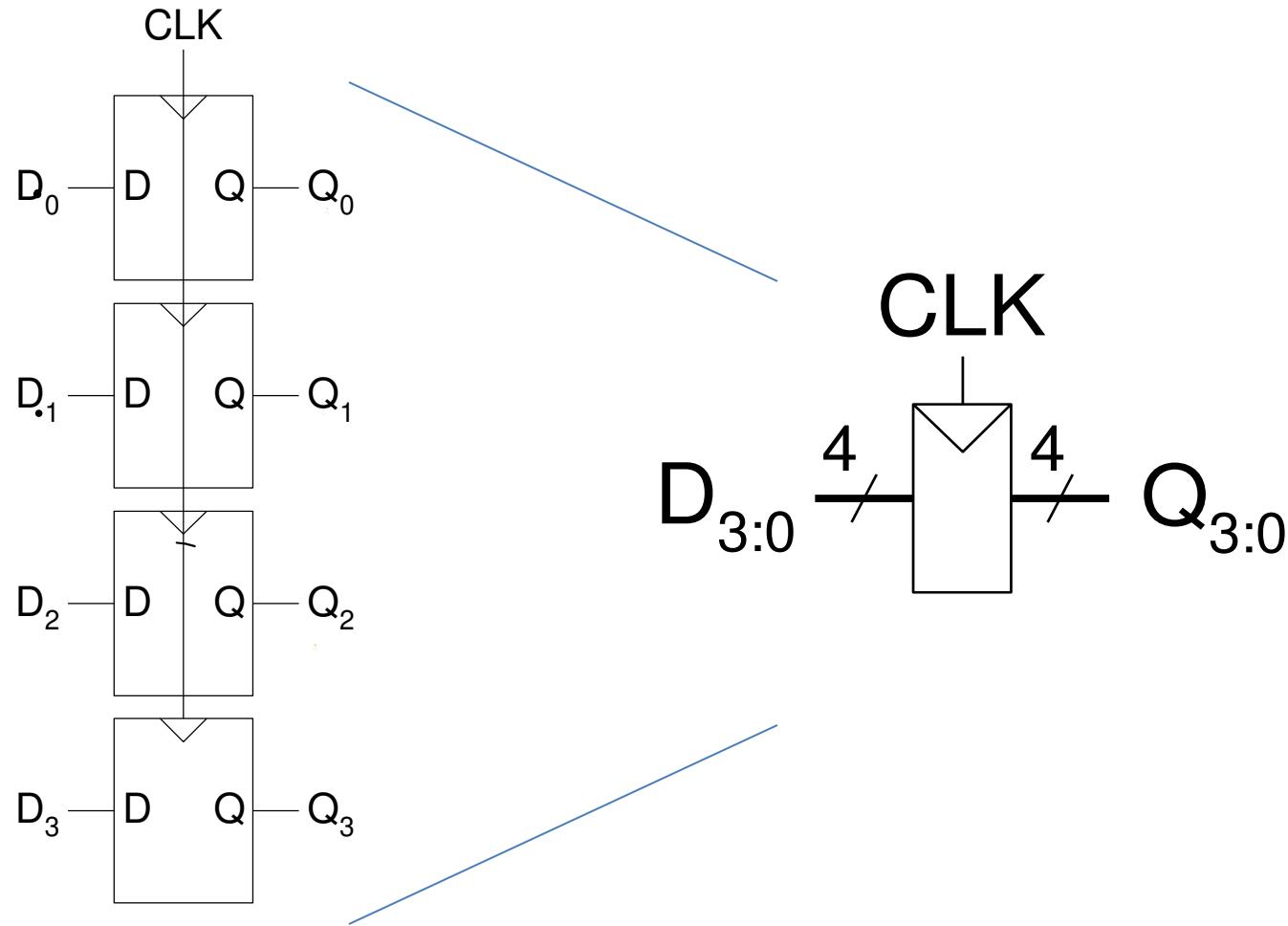
# D Latch vs. D Flip-Flop



# D Latch vs. D Flip-Flop

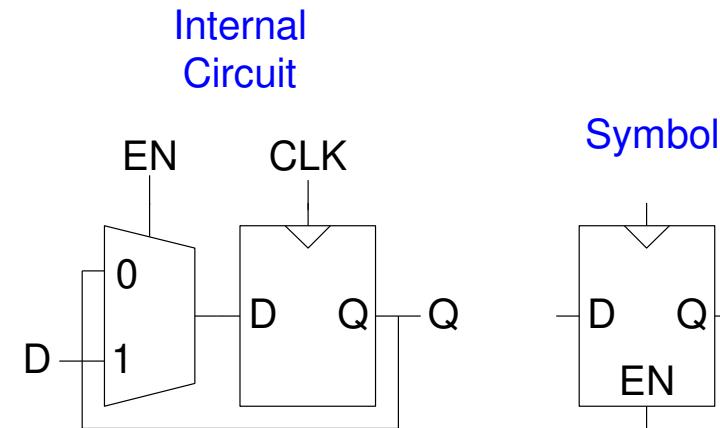


# Registri: Multi-bit Flip-Flop



# Flip-Flops “enabled”

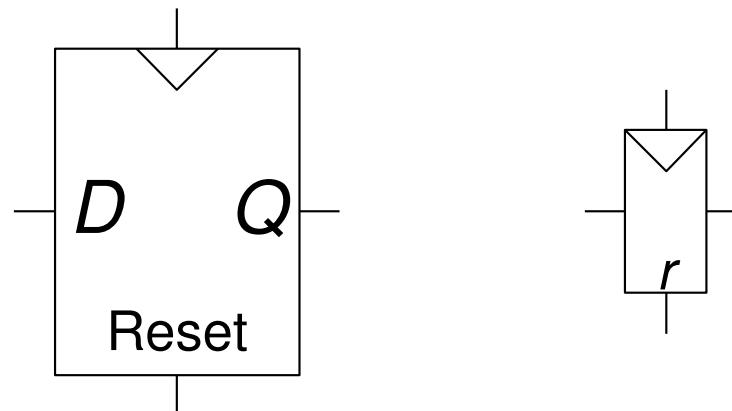
- *Inputs: CLK, D, EN*
- L'input enable (*EN*) stabilisce quando un nuovo valore di *D* è memorizzato
- *EN= 1*: *D* passa fino a *Q* (clock: 0 → 1)
- *EN= 0*: il flip-flop mantiene il suo stato precedente



# Flip-Flops “resetabili”

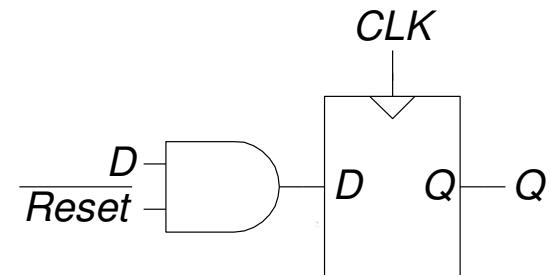
- *Inputs: CLK, D, Reset*
- *Reset = 1: Q = 0*
- *Reset = 0: il flip-flop si comporta “normalmente” come un D flip-flop*

## Symbols



# Flip-Flops “resettabili”

- Vi sono due tipi di flip-flop resettabili:
  - **Sincroni:** il reset è pilotato dal clock
  - **Asincroni:** il reset avviene non appena  $Reset = 1$
- Flip-flop sincroni:

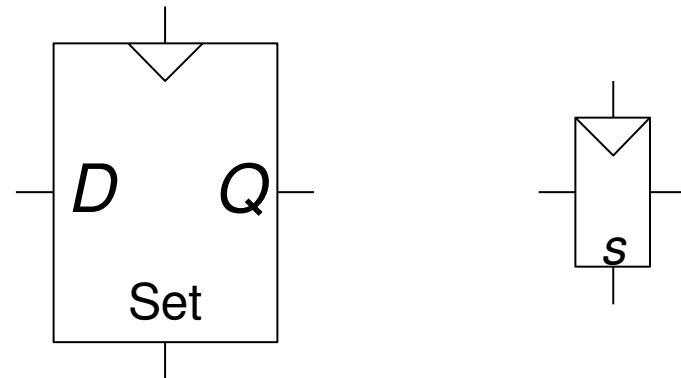


- Per i flip-flop asincroni occorre modificare il circuito interno del flip-flop

# Flip-Flops “settabili”

- *Inputs: CLK, D, Set*
- *Set = 1: Q=1*
- *Set = 0: il flip-flop si comporta “normalmente” come un D flip-flop*

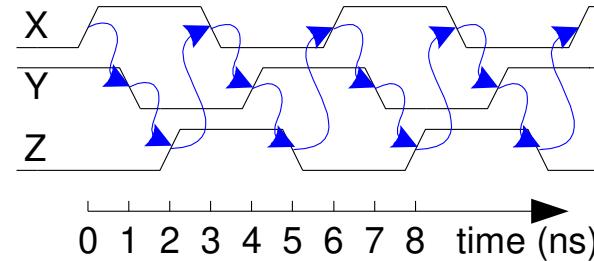
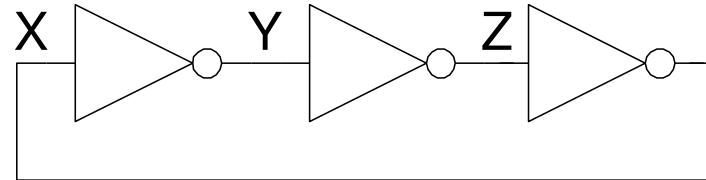
## Symbols



# Esercizi

- Esercizi 3.1, 3.3, 3.5, 3.7, 3.8, 3.13, 3.15

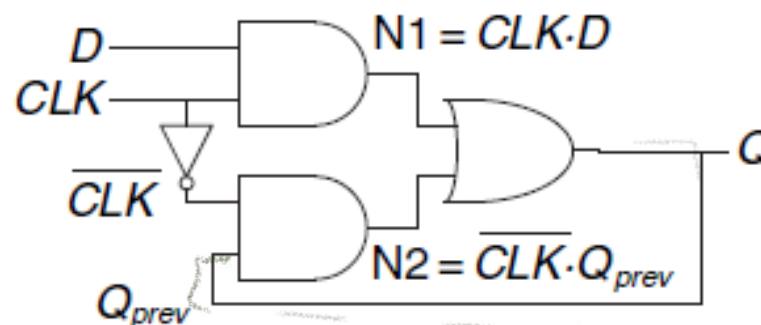
# Criticità nella logica sequenziale



- Questi circuiti vengono detti “astabili” poiché hanno un comportamento oscillante
- Il periodo di oscillazione dipende dai ritardi degli inverter
- Idealmente è di 6 ns tuttavia può variare a causa di diversi fattori
  - differenze nella manifattura
  - temperatura
- Circuito *asincrono*: l'output è retroazionato in maniera diretta

# Criticità nella logica sequenziale

- In casi più complessi che comprendono l'uso di più porte AND, NOT, OR il comportamento di una rete asincrona può dipendere fortemente dai ritardi accumulati sui singoli cammini

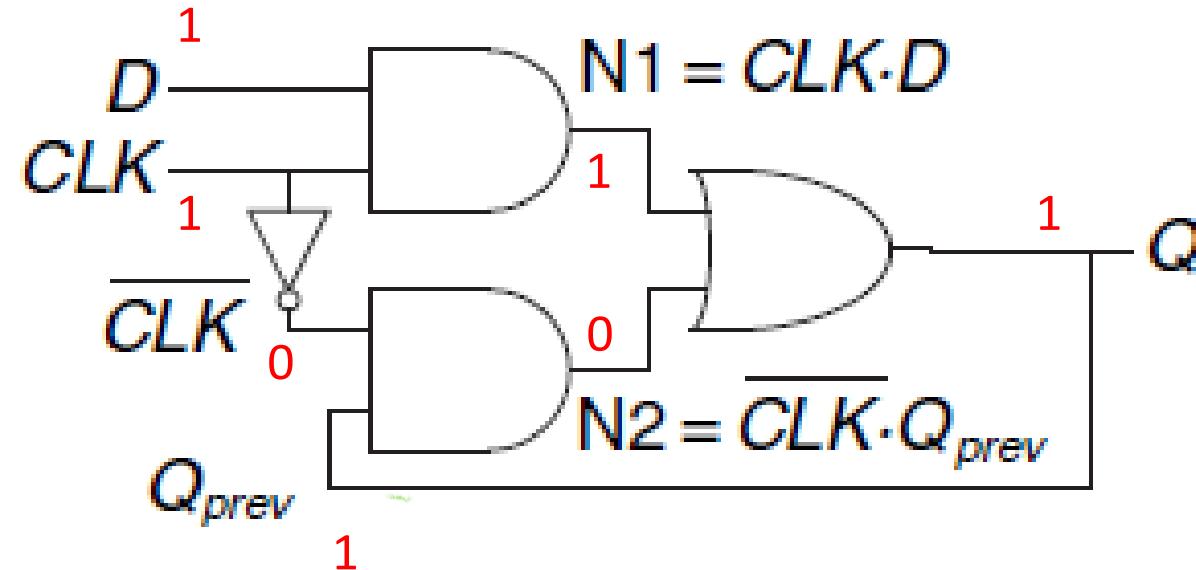


$$Q = CLK \cdot D + \overline{CLK} \cdot Q_{prev}$$

| $CLK$ | $D$ | $Q_{prev}$ | $Q$ |
|-------|-----|------------|-----|
| 0     | 0   | 0          | 0   |
| 0     | 0   | 1          | 1   |
| 0     | 1   | 0          | 0   |
| 0     | 1   | 1          | 1   |
| 1     | 0   | 0          | 0   |
| 1     | 0   | 1          | 0   |
| 1     | 1   | 0          | 1   |
| 1     | 1   | 1          | 1   |

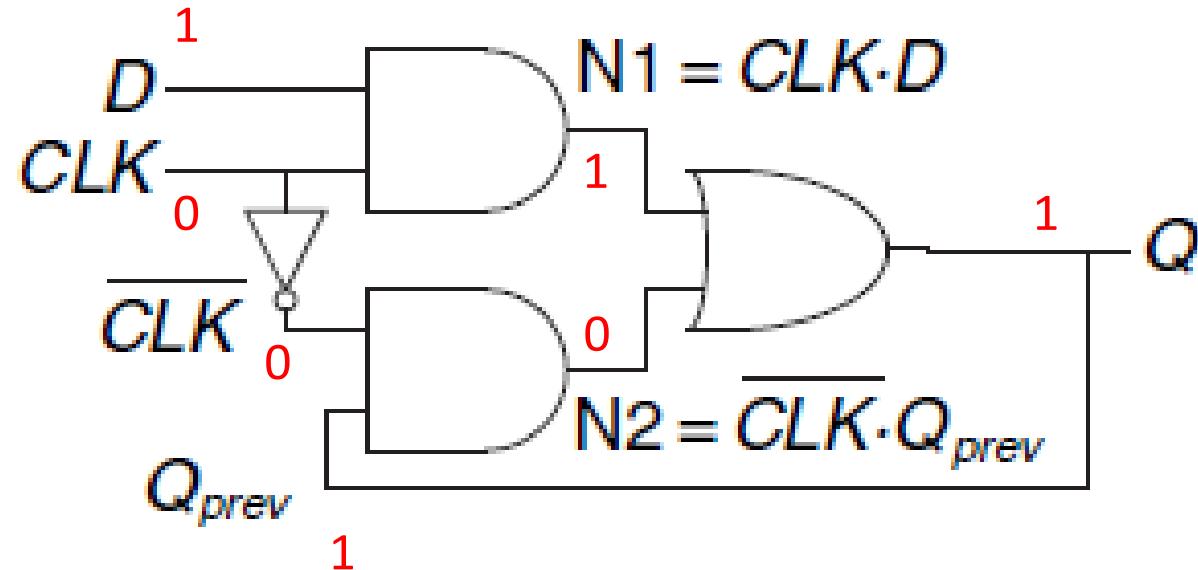
# Criticità nella logica sequenziale

$$D=1, \text{CLK}=1 \rightarrow Q=1$$



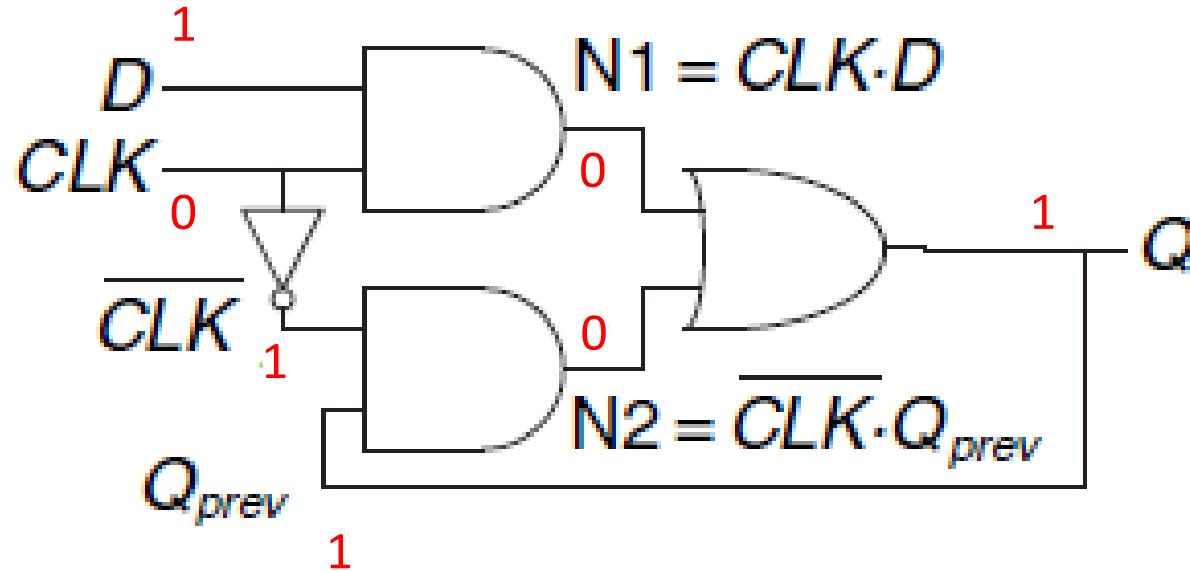
# Criticità nella logica sequenziale

$t_0$  CLK 1 → 0



# Criticità nella logica sequenziale

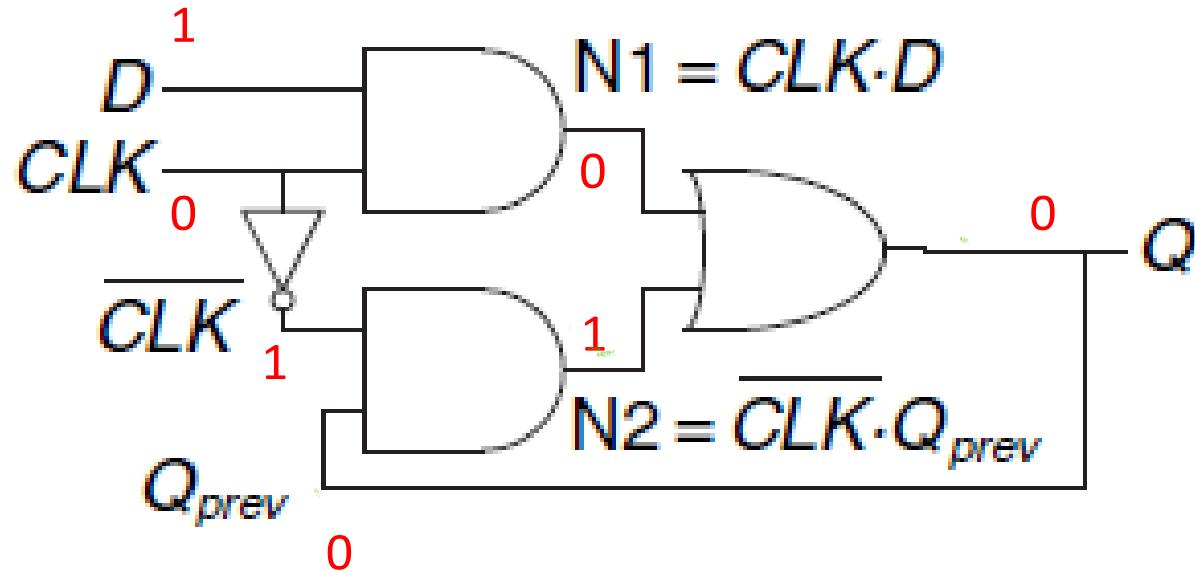
$t_1$



# Criticità nella logica sequenziale

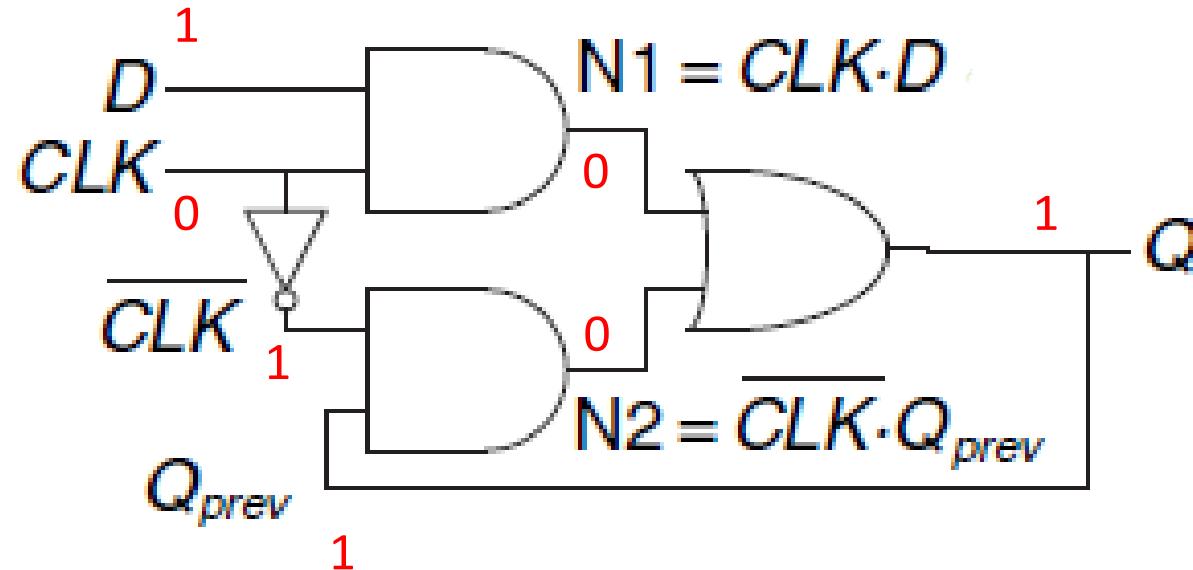
$t_2$

$$1 \cdot E = E$$



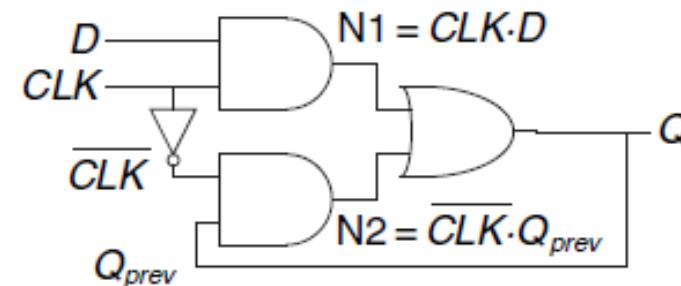
# Criticità nella logica sequenziale

$t_3 (= t_1)$



# Criticità nella logica sequenziale

- In casi più complessi che comprendono l'uso di più porte AND, NOT, OR il comportamento di una rete asincrona può dipendere fortemente dai ritardi accumulati sui singoli cammini



- $D=1, CLK=1 \rightarrow Q=1$
- $CLK 1 \rightarrow 0 \rightarrow Q$  oscilla  $(Q=Q_{prev}=1)$

# Logiche sequenziali sincrone

- I circuiti asincroni presentano delle criticità a volte difficilmente analizzabili
  - Dipendono dalla struttura fisica dei componenti
- Per questo si cerca di evitare di retroazionare l'output in maniera diretta e si interpone un registro nel ciclo di retroazione
- *Nell'ipotesi che il clock sia più lento del ritardo accumulato sul cammino, il registro consente al sistema di essere sincronizzato col clock: circuito sincrono*

# Logiche sequenziali sincrone

- In generale un circuito sequenziale sincrono ha un insieme finito di stati  $\{S_0, \dots, S_{k-1}\}$
- Logica combinatoria:

$$\text{out} = f(\text{in})$$

- Logica sequenziale sincrona:

$$\text{out} = f(\text{in}, s_c)$$

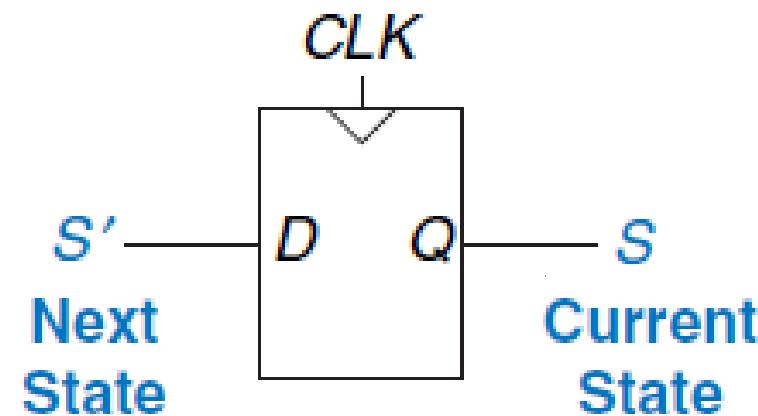
$$s_n = g(\text{in}, s_c)$$

# Design di logiche sequenziali sincrone

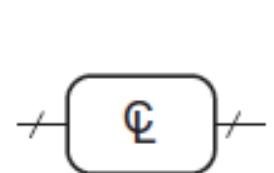
- Inserire registri nei cammini ciclici
- I registri determinano lo **stato**  $S_0, \dots, S_{k-1}$  del sistema
- I cambiamenti di stato sono determinati dalle transizioni del clock: il sistema è sincronizzato con il clock
- *Regole* di composizione:
  - Ogni componente è un registro o un circuito combinatorio
  - Almeno un componente è un registro
  - Tutti i registri sono sincronizzati con un unico clock
  - Ogni ciclo contiene almeno un registro
- Due tipici circuiti sequenziali sincroni
  - Finite State Machines (FSMs)
  - Pipelines

# Current state /Next state

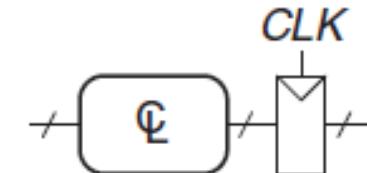
- Un flip-flop D è il più semplice circuito sequenziale sincrono
  - $Q = s_c$
  - $D = s_n$



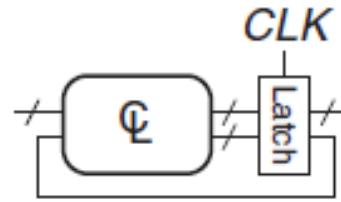
# Esempi



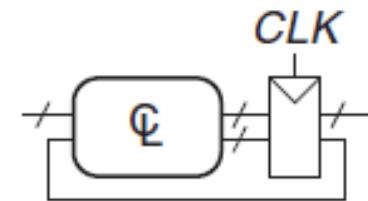
(a)



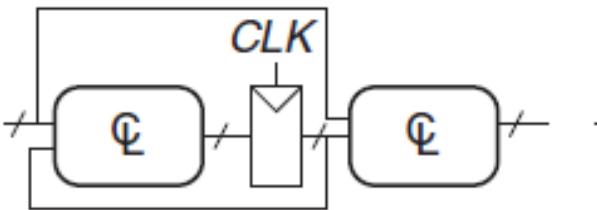
(b)



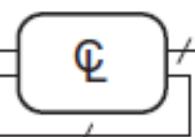
(c)



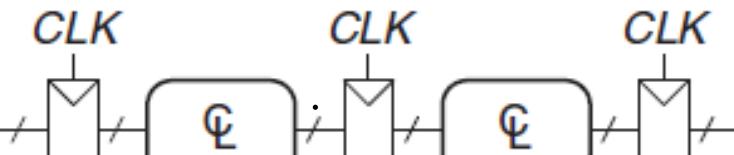
(d)



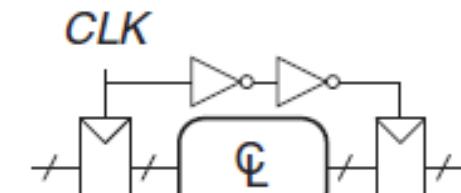
(e)



(f)



(g)



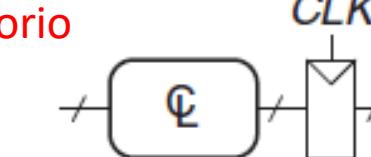
(h)

# Esempi

NO: circuito combinatorio



(a)

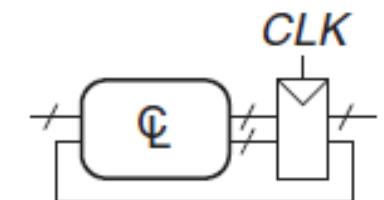


(b)

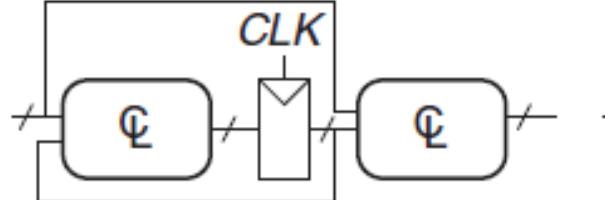


(c)

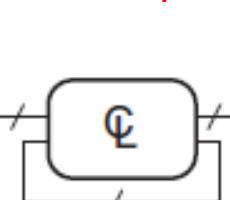
NO: latch e non flip-flop



(d)

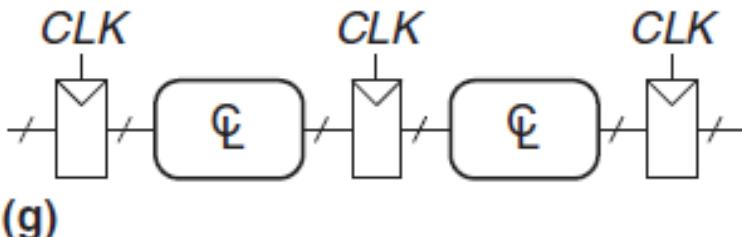


(e)

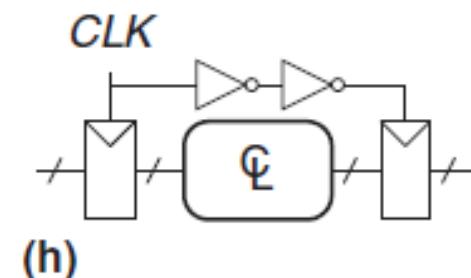


(f)

NO: circuito sequenziale asincrono



(g)

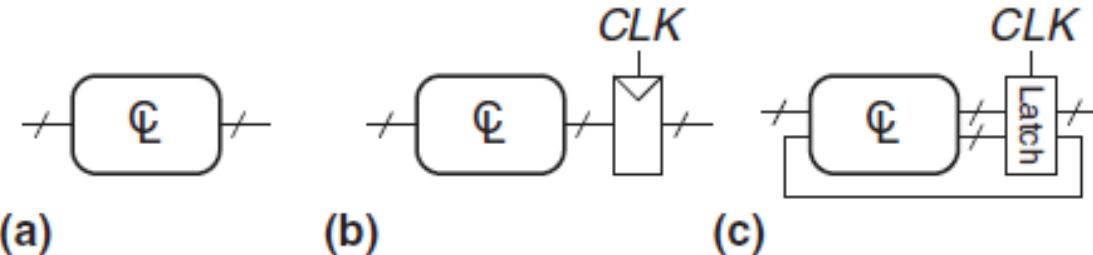


(h)

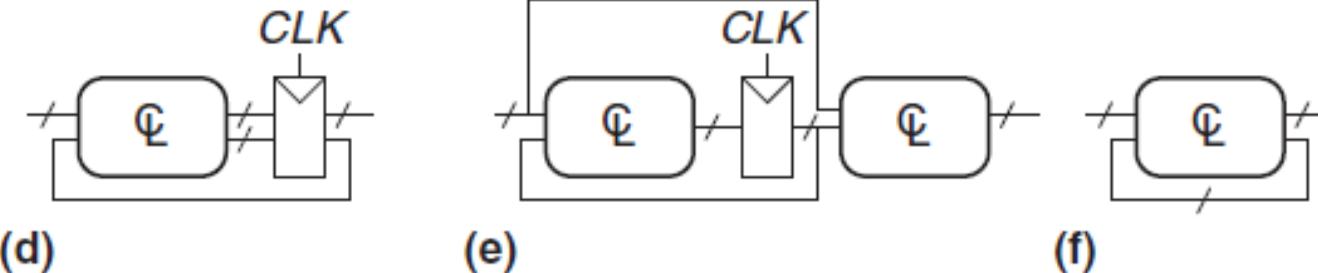
NO: i registri non hanno lo stesso clock

# Esempi

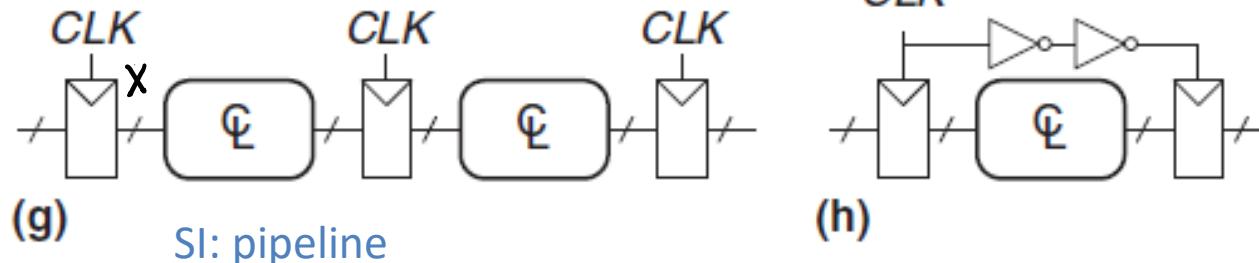
SI: ma senza feedback



SI: FSM

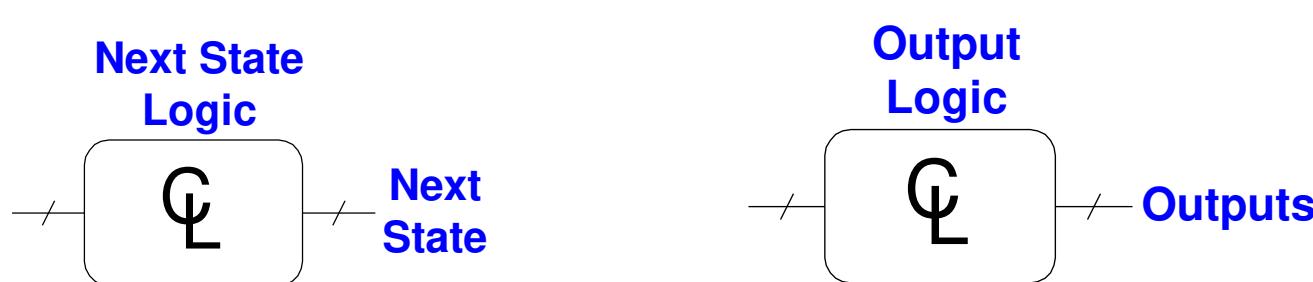
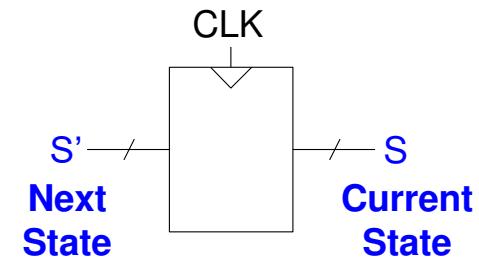


CLK



# Finite State Machines

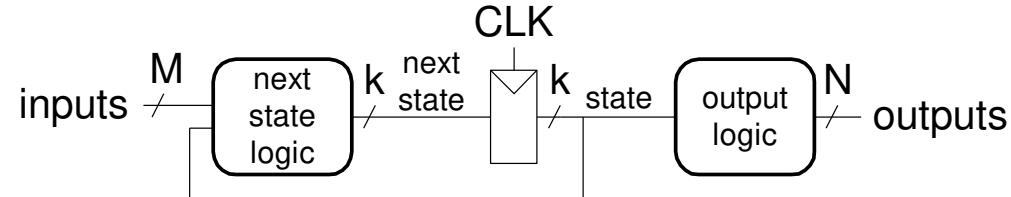
- **State register**
  - Memorizzano lo stato corrente
  - Caricano il prossimo stato (clock edge)
- **Logica combinatoria**
  - “Computa” il prossimo stato (**g**)
  - “Computa” gli output (**f**)



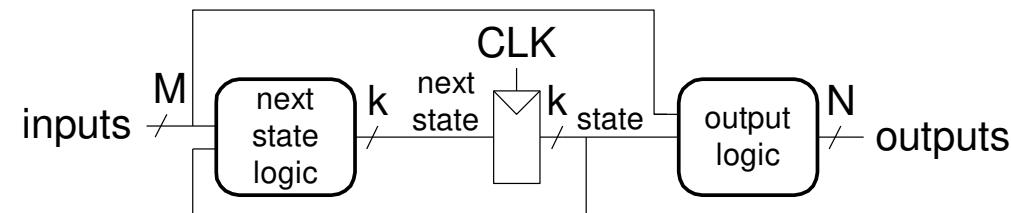
# Finite State Machines

- $s_n$  dipende sia dall'input che da  $s_c$   
$$s_n = g(in, s_c)$$
- 2 tipi di FSM a seconda della logica di output:
  - **Moore FSM:**  $out=f(s_c)$
  - **Mealy FSM:**  $out=f(in, s_c)$

Moore FSM

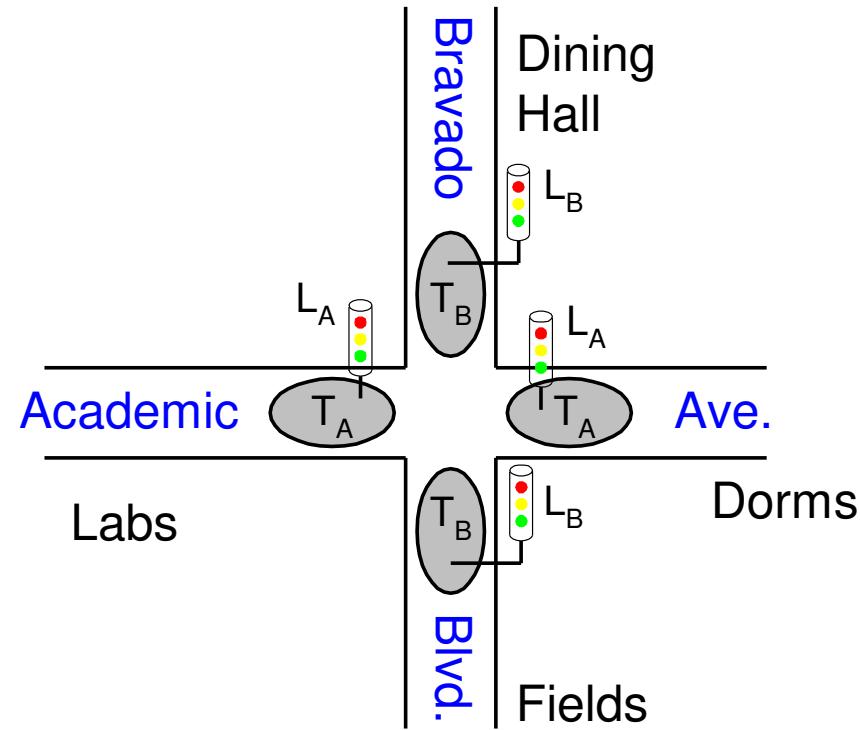


Mealy FSM



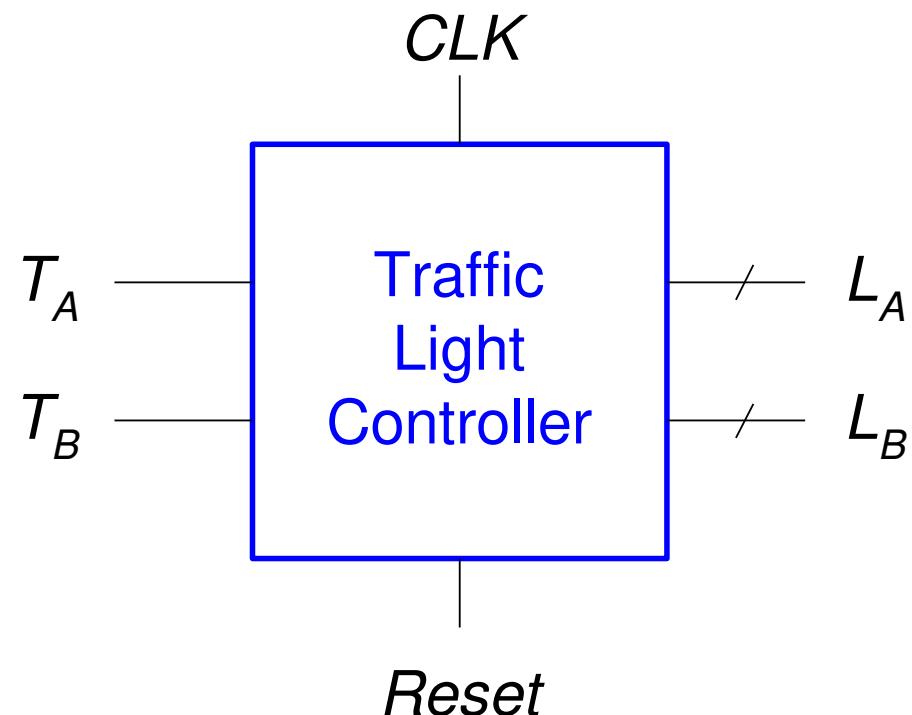
# Esempio: semaforo

- Sensori:  $T_A, T_B$  (TRUE quando c'è traffico)
- Luci:  $L_A, L_B$



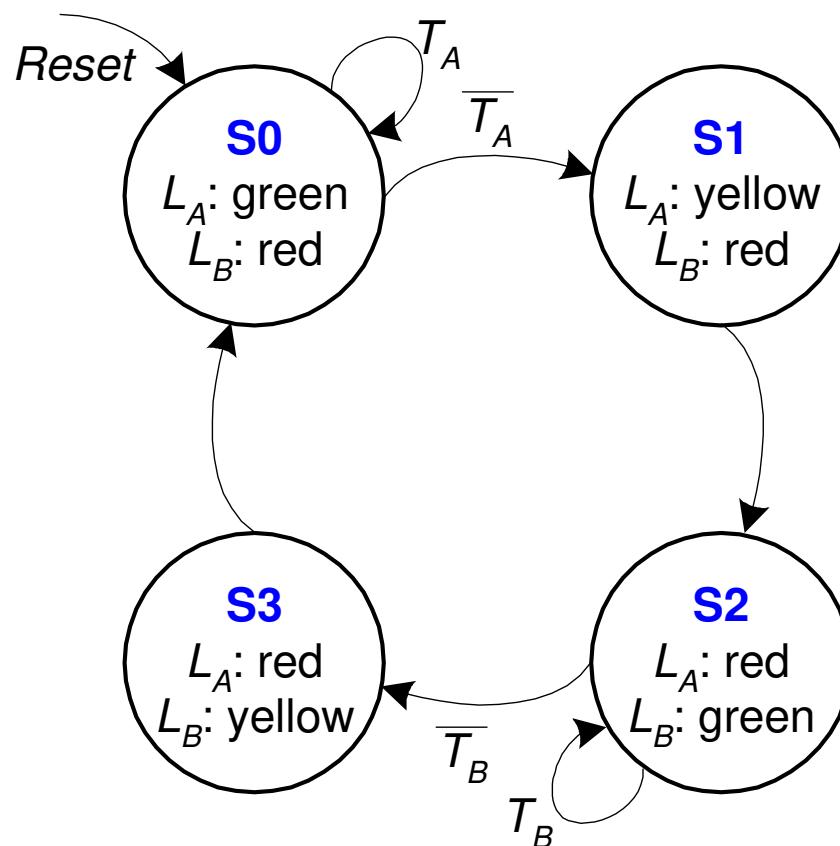
# Semaforo: *black box*

- Inputs:  $CLK$ ,  $Reset$ ,  $T_A$ ,  $T_B$
- Outputs:  $L_A$ ,  $L_B$



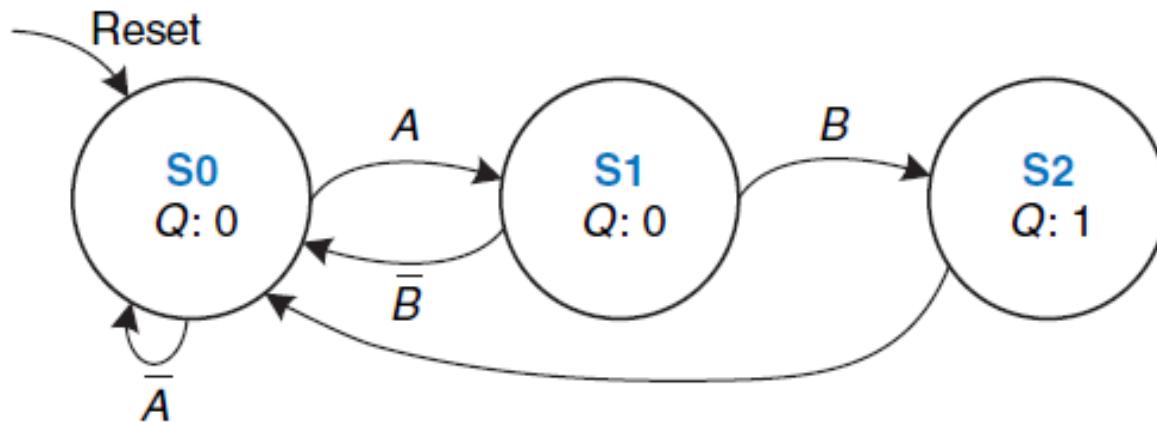
# Diagramma di transizione: Moore FSM

- **Stati:** labellati con gli outputs
- **Transizioni:** labellate con gli inputs



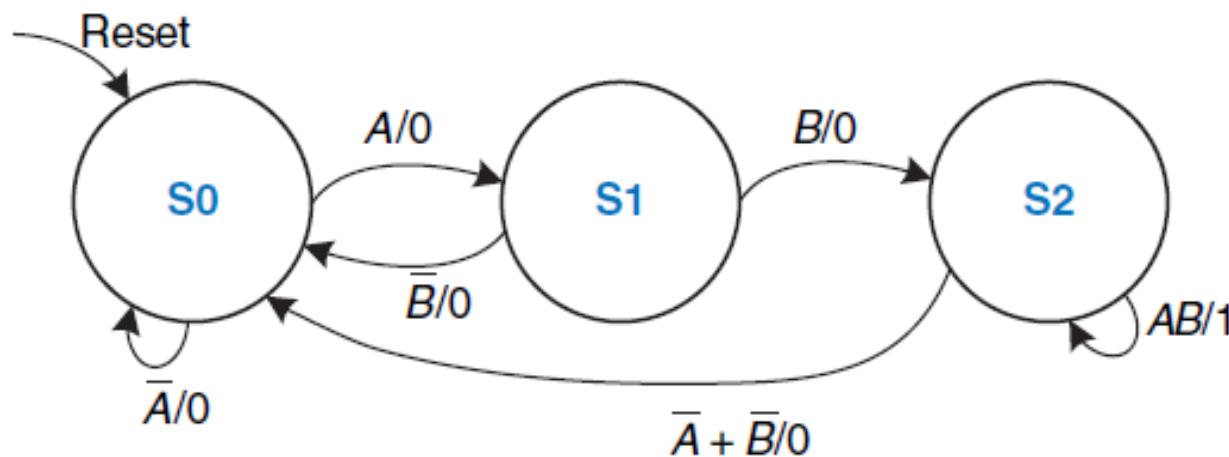
# Esempio Moore FSM

- Quale è il comportamento della FSM seguente?



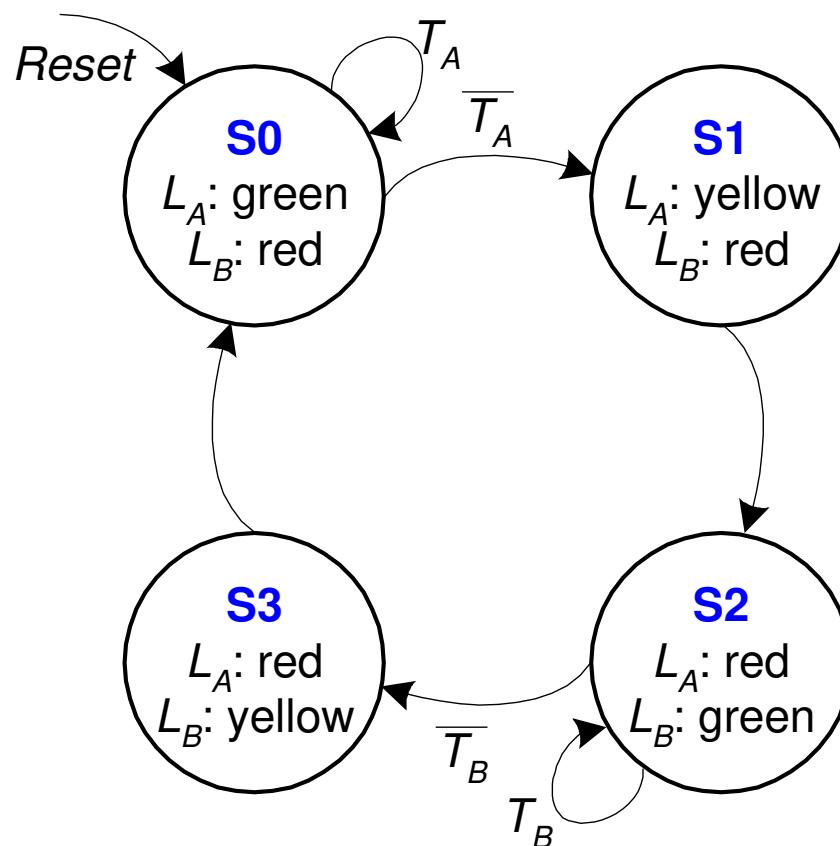
# Esempio Mealey FSM

- Quale è il comportamento della FSM seguente?



# Diagramma di transizione: Moore FSM

- **Stati:** labellati con gli outputs
- **Transizioni:** labellate con gli inputs



# Tabella di transizione

| Current State | Inputs |       | Next State |
|---------------|--------|-------|------------|
|               | $T_A$  | $T_B$ |            |
| S0            | 0      | X     | S1         |
| S0            | 1      | X     | S0         |
| S1            | X      | X     | S2         |
| S2            | X      | 0     | S3         |
| S2            | X      | 1     | S2         |
| S3            | X      | X     | S0         |

# Tabella di transizione

| Current State |       | Inputs |       | Next State |        |
|---------------|-------|--------|-------|------------|--------|
| $S_1$         | $S_0$ | $T_A$  | $T_B$ | $S'_1$     | $S'_0$ |
| 0             | 0     | 0      | X     | 0          | 1      |
| 0             | 0     | 1      | X     | 0          | 0      |
| 0             | 1     | X      | X     | 1          | 0      |
| 1             | 0     | X      | 0     | 1          | 1      |
| 1             | 0     | X      | 1     | 1          | 0      |
| 1             | 1     | X      | X     | 0          | 0      |

| State | Encoding |
|-------|----------|
| S0    | 00       |
| S1    | 01       |
| S2    | 10       |
| S3    | 11       |

$$S'_1 = S_1 \oplus S_0$$

$$S'_0 = \overline{S_1} \overline{S_0} T_A + S_1 \overline{S_0} T_B$$

————— —————

# Tabella dell'output

| Current State |       | Outputs  |          |          |          |
|---------------|-------|----------|----------|----------|----------|
| $S_1$         | $S_0$ | $L_{A1}$ | $L_{A0}$ | $L_{B1}$ | $L_{B0}$ |
| 0             | 0     | 0        | 0        | 1        | 0        |
| 0             | 1     | 0        | 1        | 1        | 0        |
| 1             | 0     | 1        | 0        | 0        | 0        |
| 1             | 1     | 1        | 0        | 0        | 1        |

| Output | Encoding |
|--------|----------|
| green  | 00       |
| yellow | 01       |
| red    | 10       |

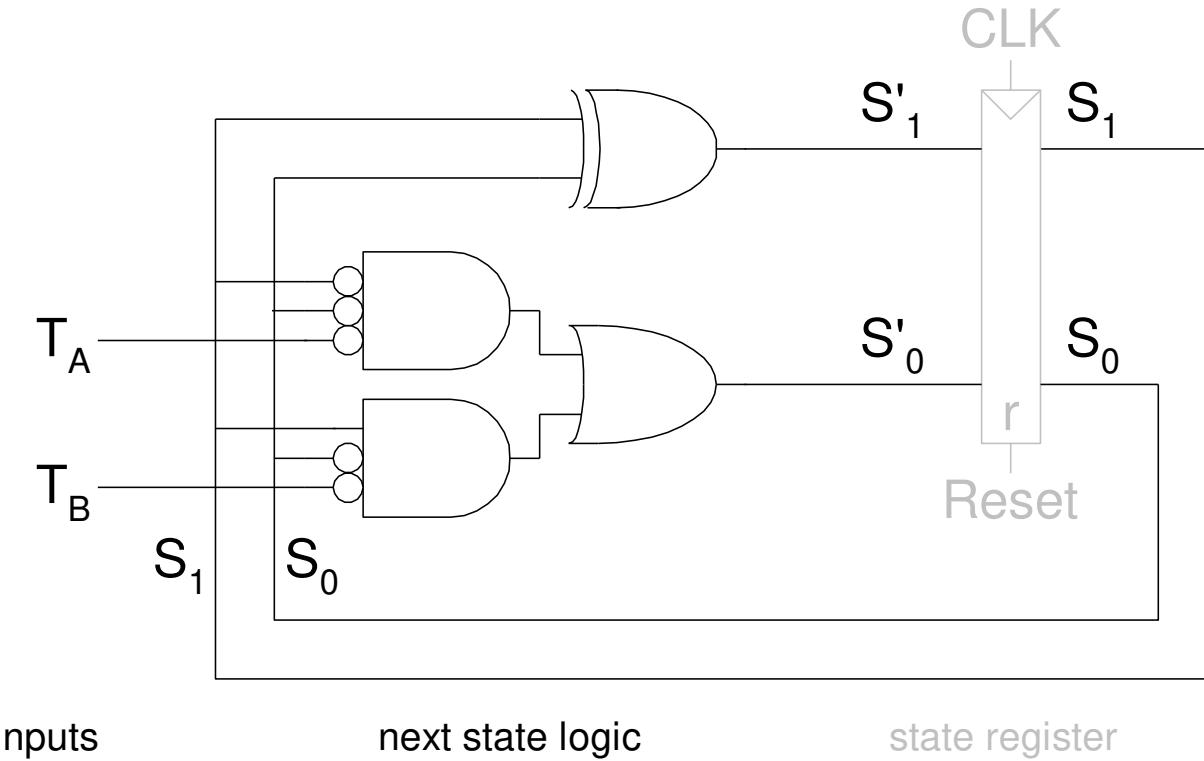
$$L_{A1} = S_1$$

$$L_{A0} = \overline{S}_1 S_0$$

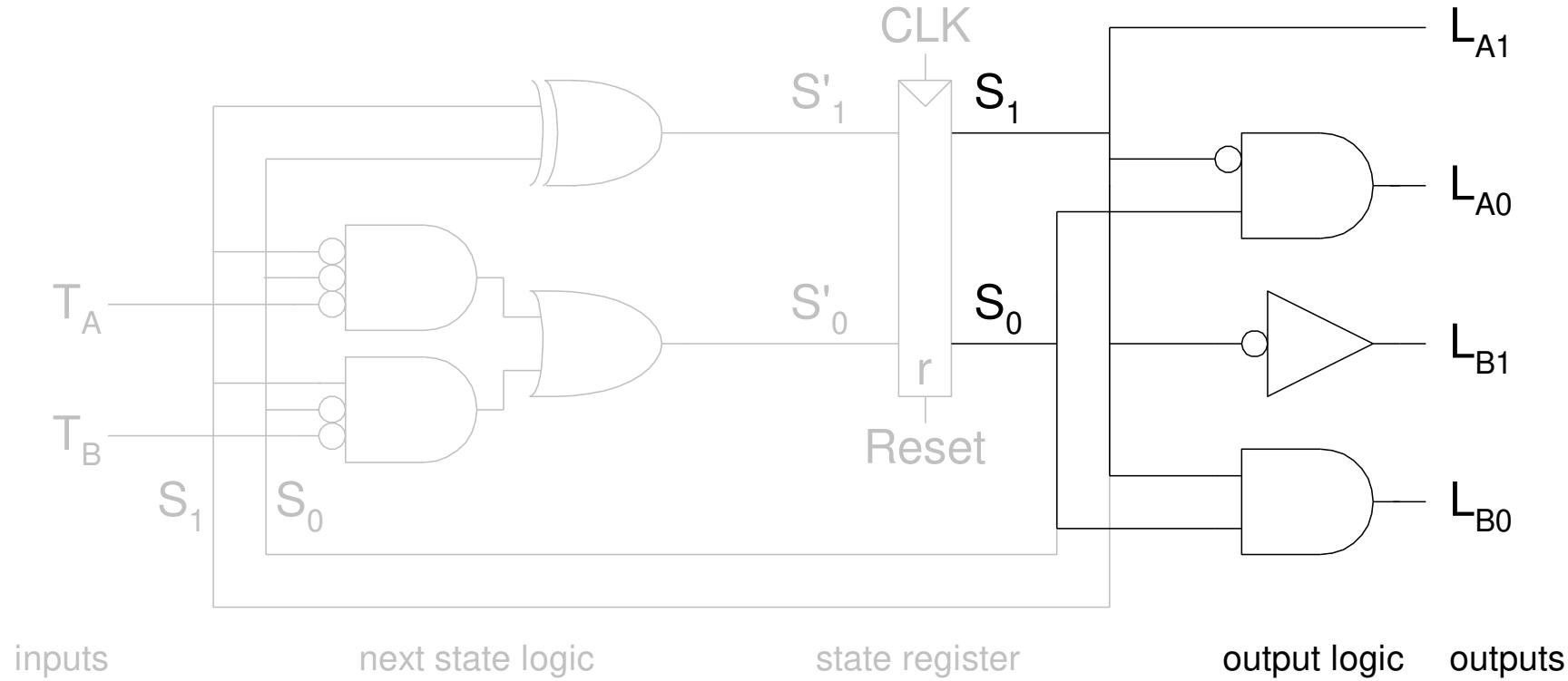
$$L_{B1} = \overline{S}_1$$

$$L_{B0} = S_1 S_0$$

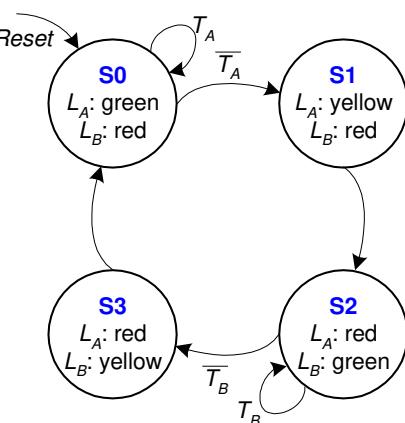
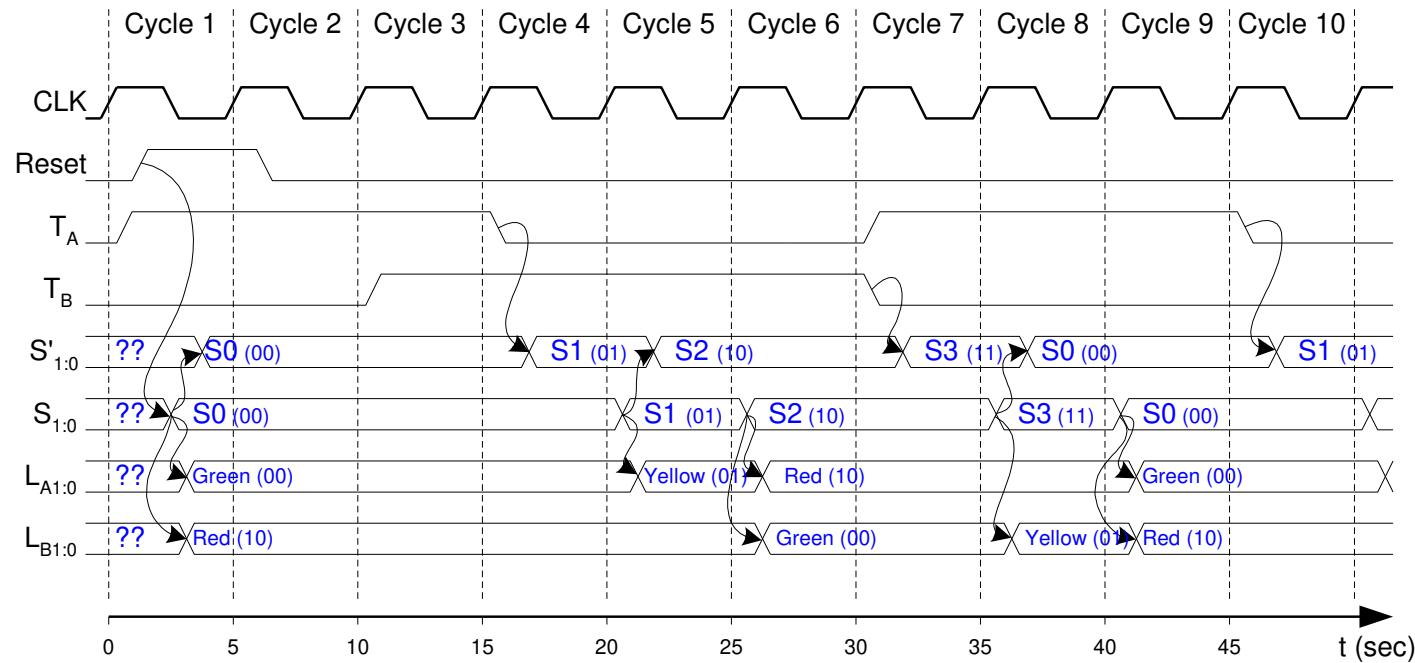
# Schema della logica di transizione



# Schema della logica di output



# FSM Timing Diagram

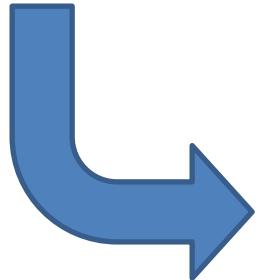


# Encoding degli stati

- Encoding binario:
  - i.e., per 4 stati, 00, 01, 10, 11
- Encoding *one-hot*
  - Un bit per stato
  - Solo un bit HIGH alla volta
  - i.e., per 4 stati, 0001, 0010, 0100, 1000
  - Richiede più flip-flops
  - Spesso la logica combinatoria associata è più semplice

# Moore vs Mealy FSM

Alyssa P. Hacker has a snail that crawls down a paper tape with 1's and 0's on it. The snail smiles whenever the last two digits it has crawled over are 01. Design Moore and Mealy FSMs of the snail's brain.



10011000101101

goal=1

dist\_prox\_goal  $\geq 2$

10011000101101

goal=0

dist\_prox\_goal  $\geq 2$

10011000101101

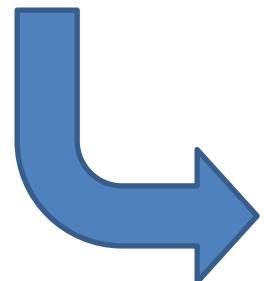
goal=0

dist\_prox\_goal  $\geq 1$

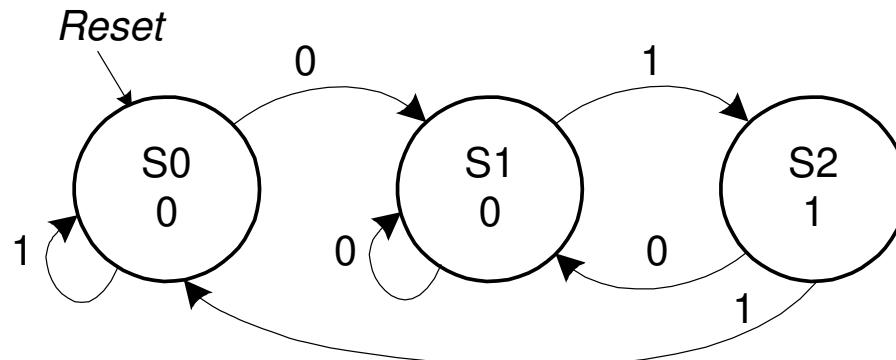
10011000101101

# Moore vs Mealy FSM

Alyssa P. Hacker has a snail that crawls down a paper tape with 1's and 0's on it. The snail smiles whenever the last two digits it has crawled over are 01. Design Moore and Mealy FSMs of the snail's brain.

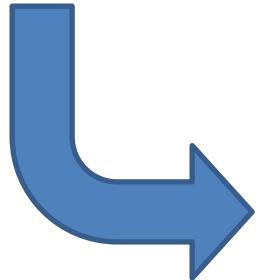


**Moore FSM**



# Moore vs Mealy FSM

Alyssa P. Hacker has a snail that crawls down a paper tape with 1's and 0's on it. The snail smiles whenever the last two digits it has crawled over are 01. Design Moore and Mealy FSMs of the snail's brain.



10011000101101

0→

goal=0; dist\_prox\_goal $\geq$ 1

1→goal=0; dist\_prox\_goal $\geq$ 2

10011000101101

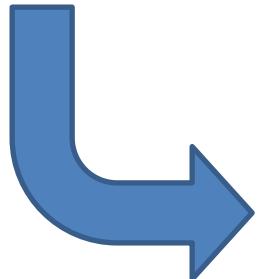
0→goal=0; dist\_prox\_goal $\geq$ 1

1→ goal=1; dist\_prox\_goal $\geq$ 2

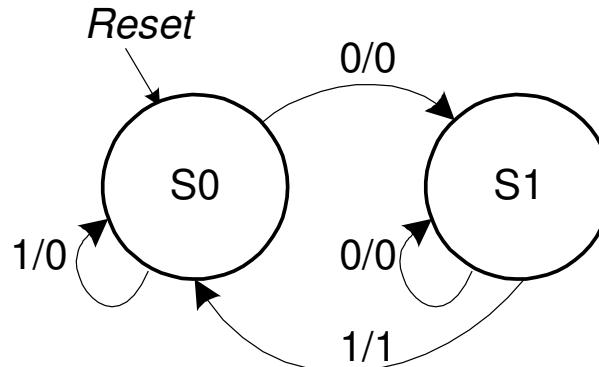
10011000101101

# Moore vs Mealy FSM

Alyssa P. Hacker has a snail that crawls down a paper tape with 1's and 0's on it. The snail smiles whenever the last two digits it has crawled over are 01. Design Moore and Mealy FSMs of the snail's brain.



**Mealy FSM**



# Tabella transizione Moore FSM

| Current State |       | Inputs<br>$A$ | Next State |        |
|---------------|-------|---------------|------------|--------|
| $S_1$         | $S_0$ |               | $S'_1$     | $S'_0$ |
| 0             | 0     | 0             | 0          | 1      |
| 0             | 0     | 1             | 0          | 0      |
| 0             | 1     | 0             | 0          | 1      |
| 0             | 1     | 1             | 1          | 0      |
| 1             | 0     | 0             | 0          | 1      |
| 1             | 0     | 1             | 0          | 0      |

| State | Encoding |
|-------|----------|
| S0    | 00       |
| S1    | 01       |
| S2    | 10       |

$$S_1' = S_0 A$$

$$S_0' = \overline{A}$$

# Tabella transizione Moore FSM

| Current State |       | Inputs<br>$A$ | Next State |        |
|---------------|-------|---------------|------------|--------|
| $S_1$         | $S_0$ |               | $S'_1$     | $S'_0$ |
| 0             | 0     | 0             | 0          | 1      |
| 0             | 0     | 1             | 0          | 0      |
| 0             | 1     | 0             | 0          | 1      |
| 0             | 1     | 1             | 1          | 0      |
| 1             | 0     | 0             | 0          | 1      |
| 1             | 0     | 1             | 0          | 0      |

| State | Encoding |
|-------|----------|
| S0    | 00       |
| S1    | 01       |
| S2    | 10       |

$$\begin{aligned}S_1' &= S_0 A \\S_0' &= \bar{A}\end{aligned}$$

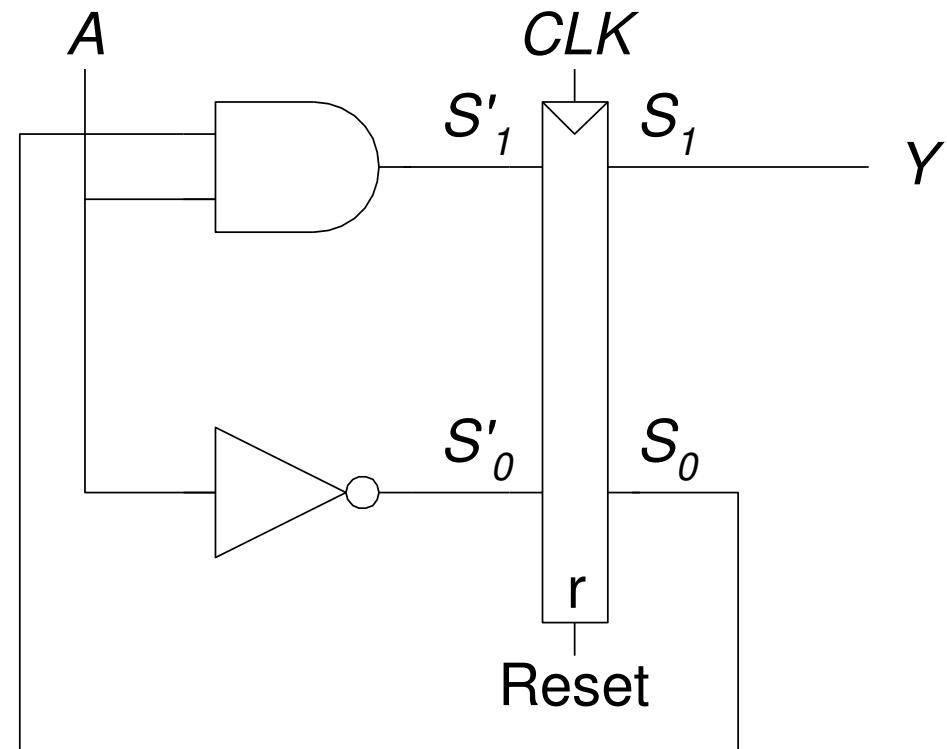
Manca  $S_1$  negato, perché?

# Tabella output Moore FSM

| Current State |       | Output |
|---------------|-------|--------|
| $S_1$         | $S_0$ | $Y$    |
| 0             | 0     | 0      |
| 0             | 1     | 0      |
| 1             | 0     | 1      |

$$Y = S_1$$

# Schema Moore FSM



# Tabella transizione/output Mealy FSM

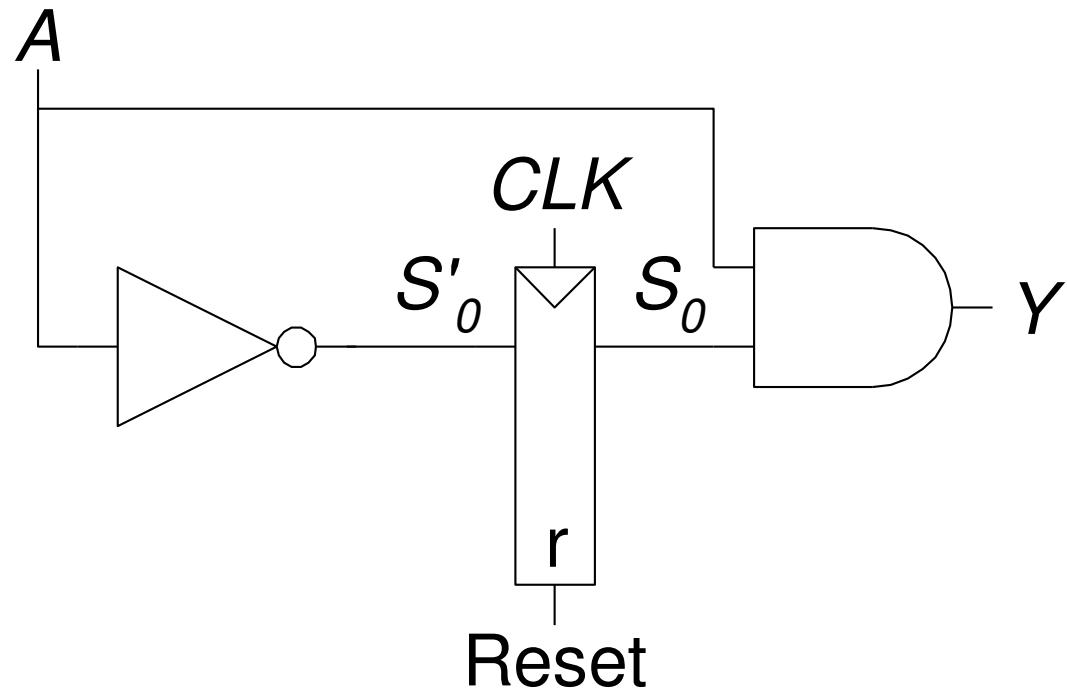
| Current State | Input | Next State | Output |
|---------------|-------|------------|--------|
| $S$           | $A$   | $S'$       | $Y$    |
| 0             | 0     | 1          | 0      |
| 0             | 1     | 0          | 0      |
| 1             | 0     | 1          | 0      |
| 1             | 1     | 0          | 1      |

| State | Encoding |
|-------|----------|
| $S_0$ | 0        |
| $S_1$ | 1        |

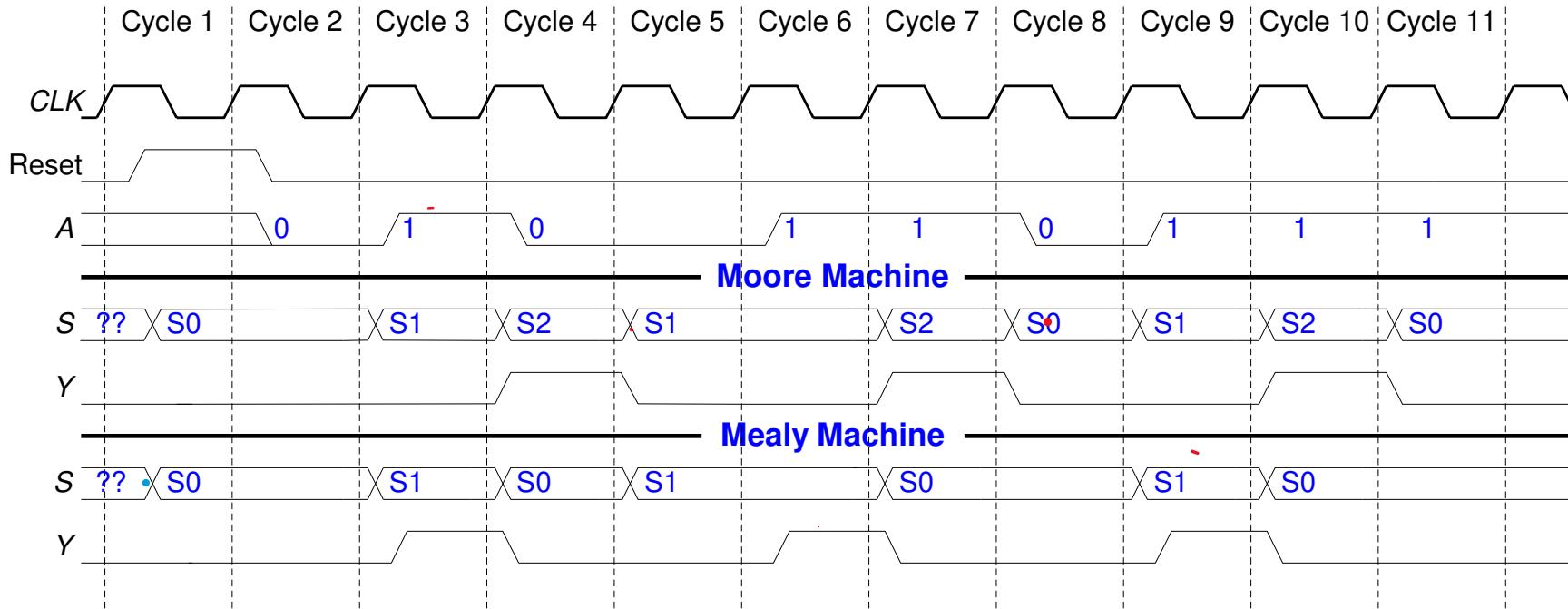
$$S' = \bar{A}$$

$$Y = SA$$

# Schema Mealy FSM



# Moore & Mealy Timing



# Fattorizzazione di FSM

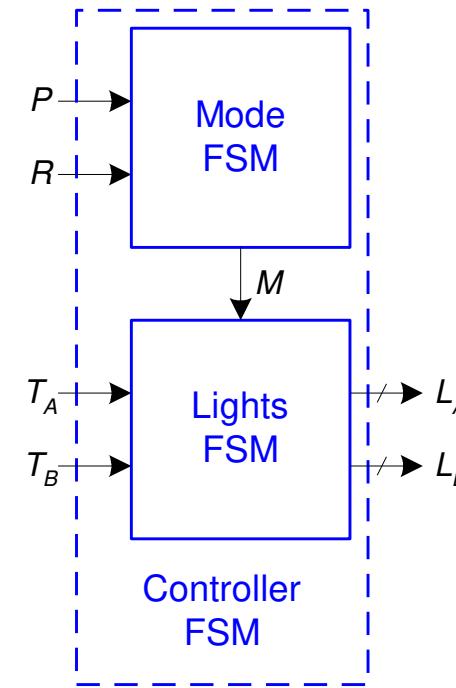
- Fattorizzare consiste nel suddividere una FSM complessa in FSM più piccole che interagiscono fra loro
- Esempio: Considerate di voler modificare il controller di semafori per tener conto di possibili parate
  - Altri due inputs:  $P$ ,  $R$
  - Se  $P = 1$ , entra in modalità *Parade* e il semaforo di Bravado Blvd rimane verde
  - Se  $R = 1$ , lascia la modalità *Parade*

# Parade FSM

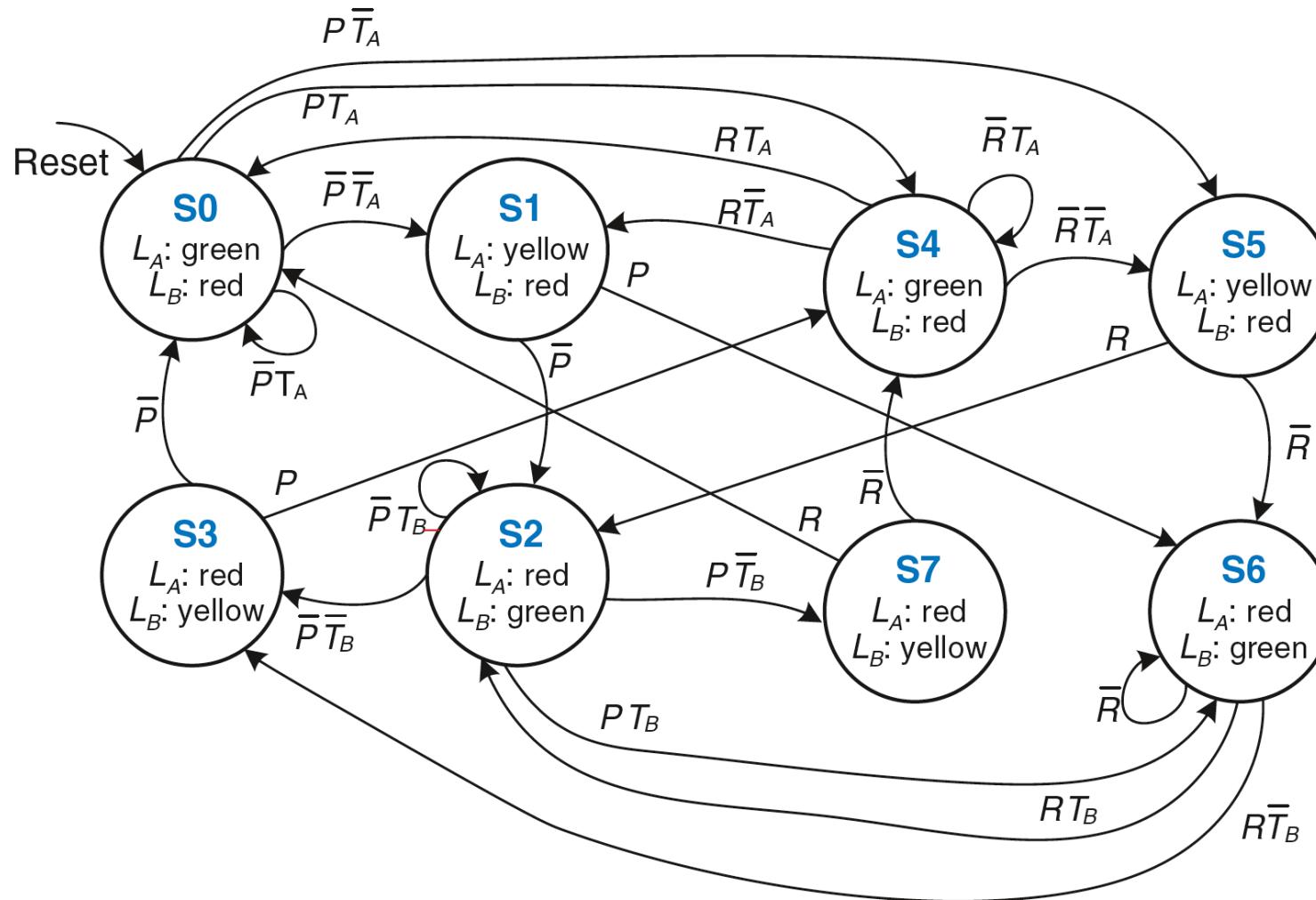
FSM non fattorizzato



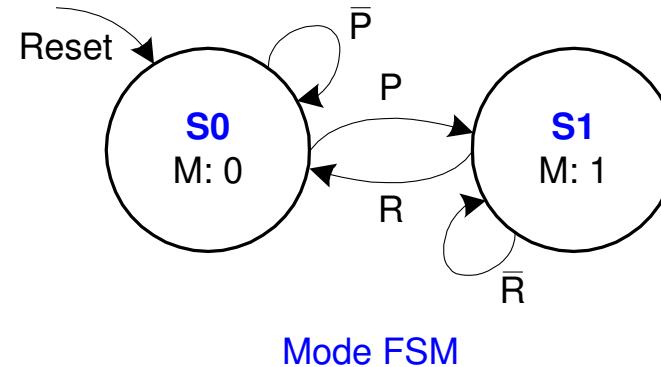
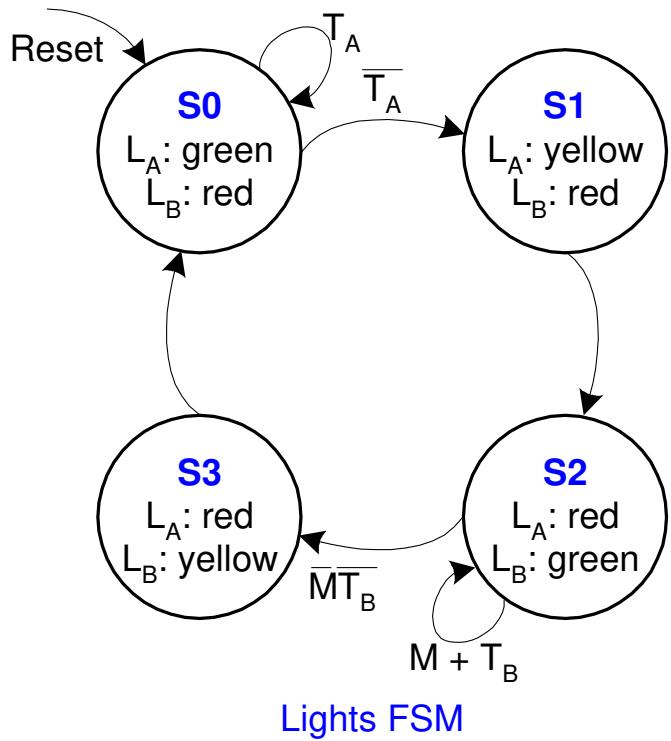
FSM fattorizzato



# FSM non fattorizzato



# FSM fattorizzato



# Progettare una FSM

1. Identificare gli input e output
2. Abbozzare uno state transition diagram
3. Scrivere la state transition table
4. Selezionare un encoding degli stati
5. Macchina di Moore/Mealy:
  - a. Riscrivere la state transition table con l'encoding degli stati
  - b. Scrivere la output table
6. Scrivere le equazioni booleane relative alla logica di prossimo stato e alla logica di output
6. Minimizzare le equazioni
7. Fare uno schema del circuito

# Esempio

- Progettare una Mealy FSM F con due input (A e B) e un output Q.
  - $Q=1$  sse A e B assumono rispettivamente il valore precedente
  - es:

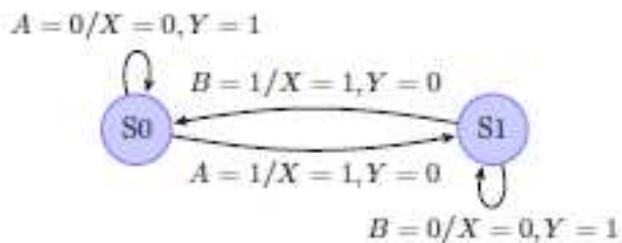
|   |   |          |   |   |          |   |   |          |   |
|---|---|----------|---|---|----------|---|---|----------|---|
| A | 0 | 0        | 1 | 0 | 0        | 0 | 1 | 1        | 0 |
| B | 1 | 1        | 1 | 1 | 1        | 0 | 1 | 1        | 1 |
| Q | 0 | <b>1</b> | 0 | 0 | <b>1</b> | 0 | 0 | <b>1</b> | 0 |

# Esercizi

## ■ Esercizi 3.23, 3.31

3. Il seguente diagramma di transizione per una macchina di Mealy ha due input  $A$  e  $B$  e due output  $X$  e  $Y$ . Indicare le formule SOP minime relative alla variabile di stato ( $S$ ) e alle due variabili di output.

Codifica dello stato:



| stato | $S$ |
|-------|-----|
| $S_0$ | 0   |
| $S_1$ | 1   |

Formule minime SOP:

- $S^t$ : \_\_\_\_\_
- $X$ : \_\_\_\_\_
- $Y$ : \_\_\_\_\_

# Parallelismo

- **2 tipi di parallelismo:**
  - **Spaziale**
    - duplicare l'hardware per eseguire più task contemporaneamente
  - **Temporale**
    - Il task è suddiviso in più fasi
    - Le diverse fasi sono eseguite in pipelining

# Latenza e throughput

- **Token:** Gruppo di input da processare per ottenere un output significativo
- **Latency:** Tempo che occorre ad un token per essere processato e produrre un output
- **Throughput:** Numero di output prodotti per unità di tempo

**il parallelismo incrementa il throughput**

# Esempio di parallelismo

- Ben vuole fare delle torte
- 5 minuti per preparare una torta
- 15 minuti per cucinarla

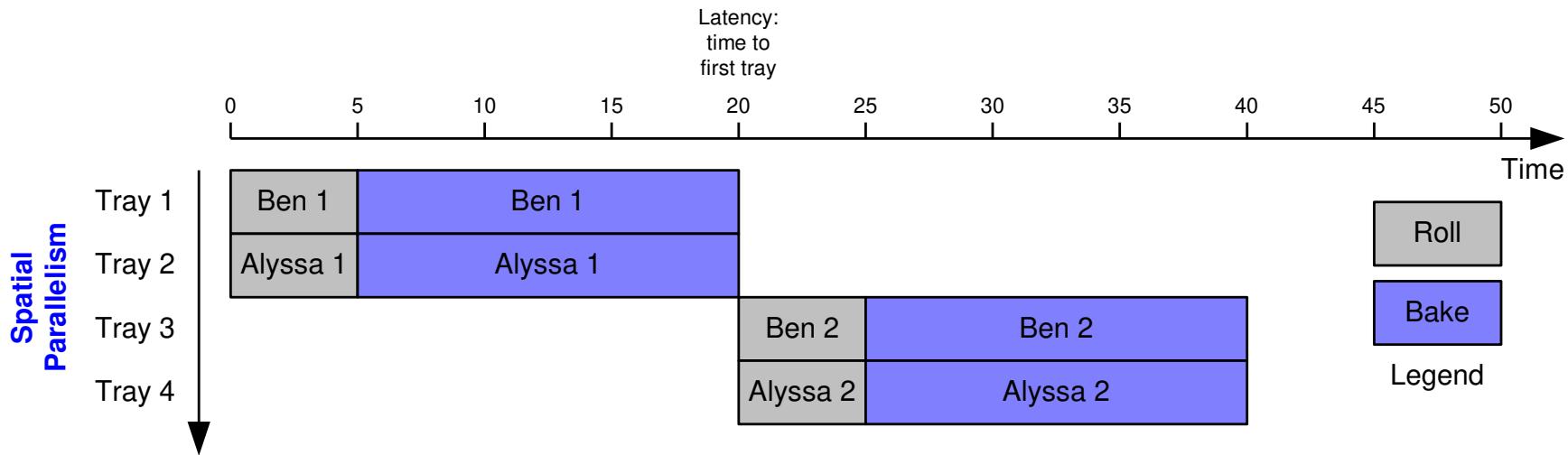
**Latency** =  $5 + 15 = 20$  minuti = **1/3 h**

**Throughput** =  $\frac{1}{latency}$  = **3 torte/h**

# Esempio di parallelismo

- **parallelismo spaziale:** Ben chiede a Allysa di aiutarlo, usando anche il suo forno
- **parallelismo temporale:**
  - 2 fasi: preparare e cucinare
  - Mentre una torta è in forno, Ben prepara ne prepara un'altra, etc.

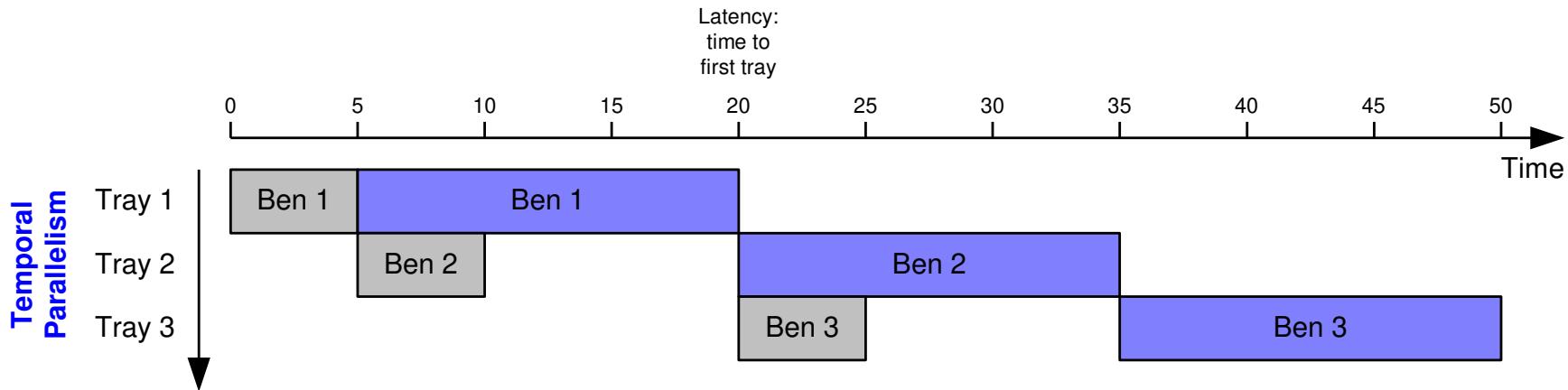
# Parallelismo spaziale



$$\text{Latency} = 5 + 15 = 20 \text{ minuti} = \mathbf{1/3 \text{ h}}$$

$$\text{Throughput} = 2 \frac{1}{latency} = \mathbf{6 \text{ torte/h}}$$

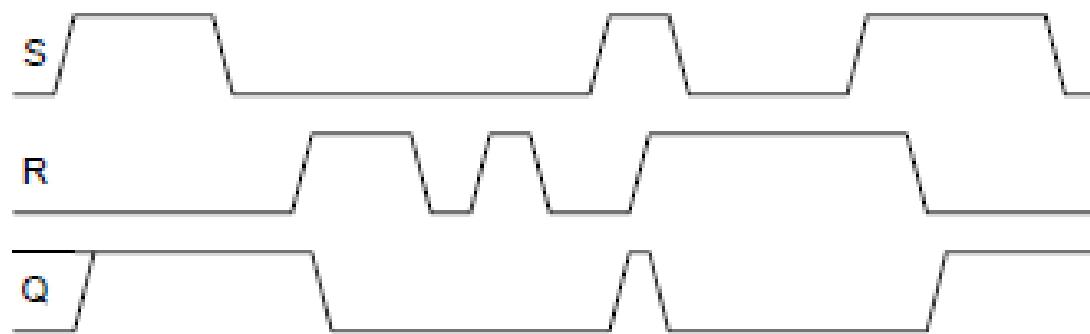
# Parallelismo temporale



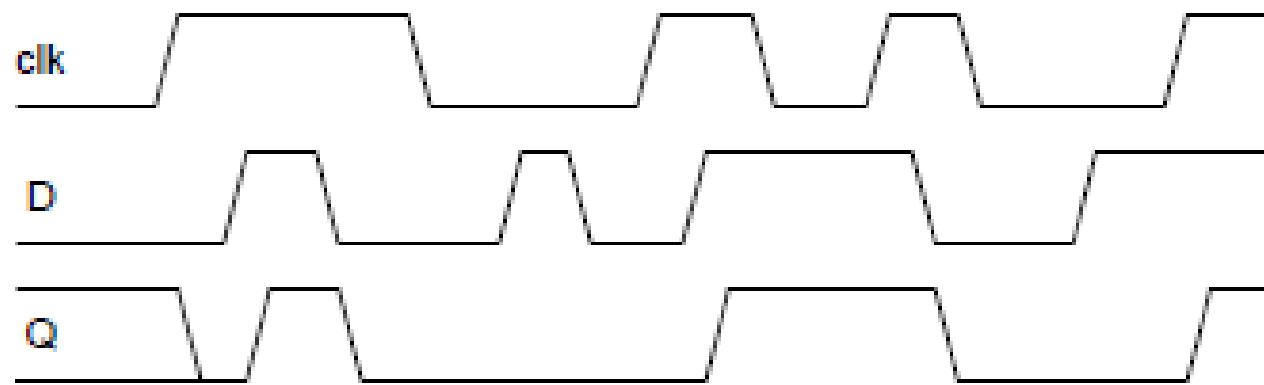
**Latency** =  $5 + 15 = 20$  minuti = **1/3 h**

**Throughput**  $\approx \frac{4}{65} 60 \approx$  **3,7 torte/h**

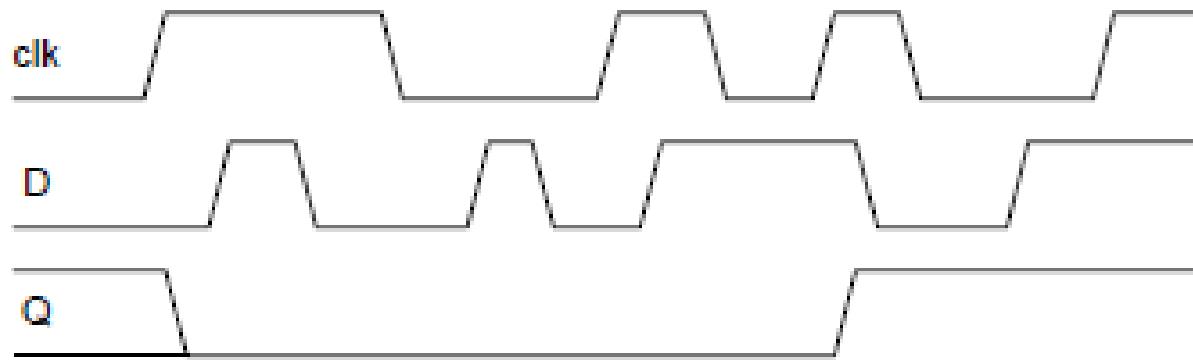
# Esercizio 3.1



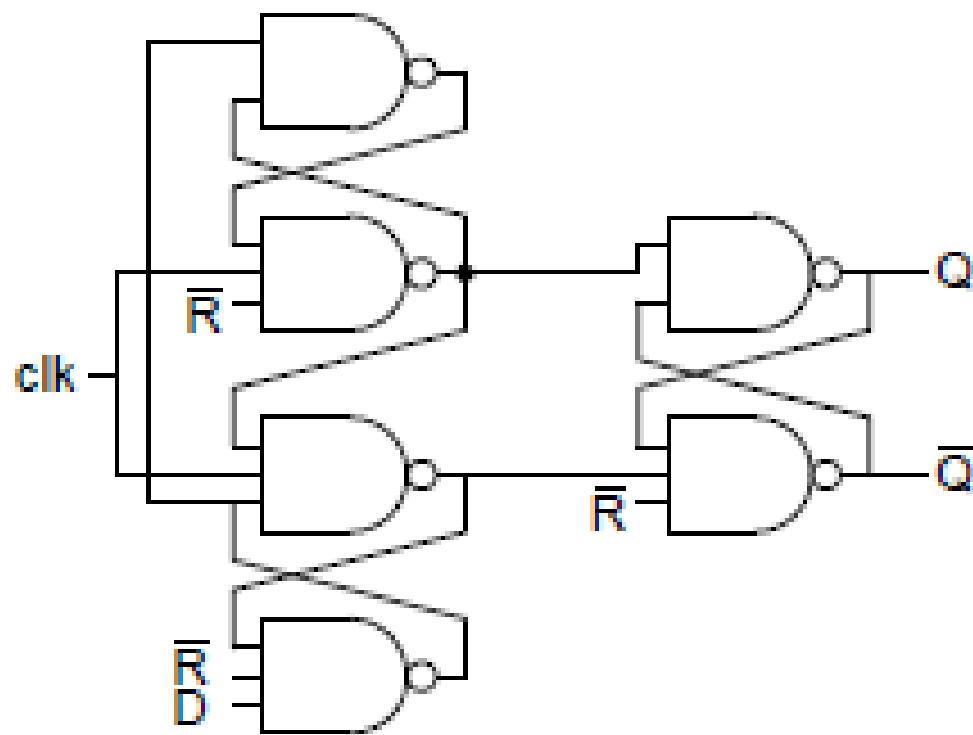
# Esercizio 3.3



# Esercizio 3.5

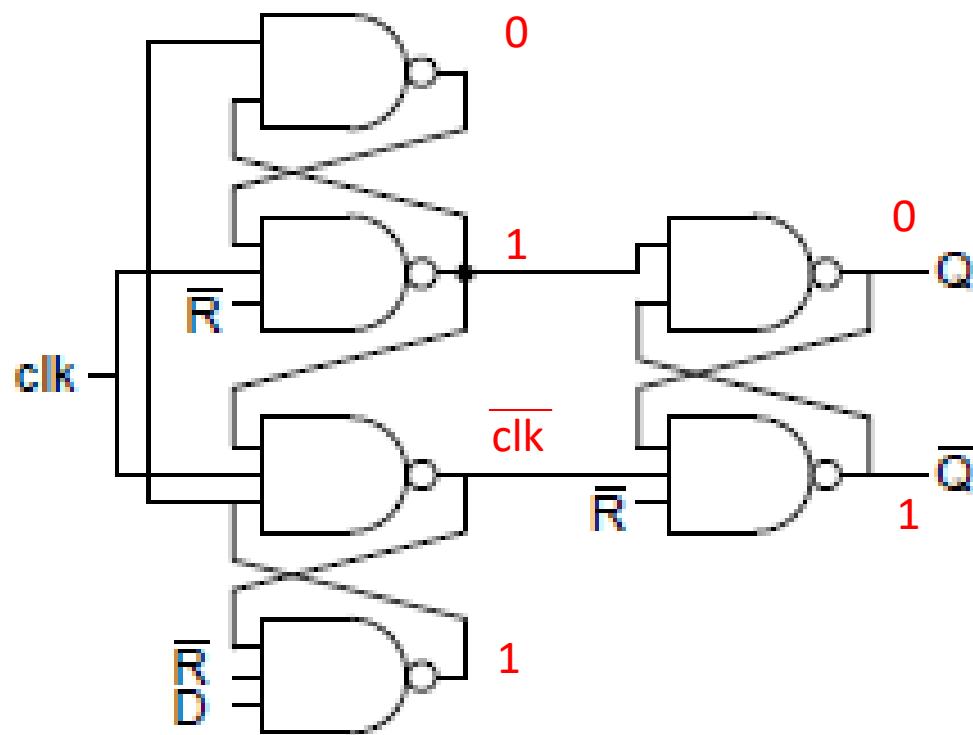


# Esercizio 3.13



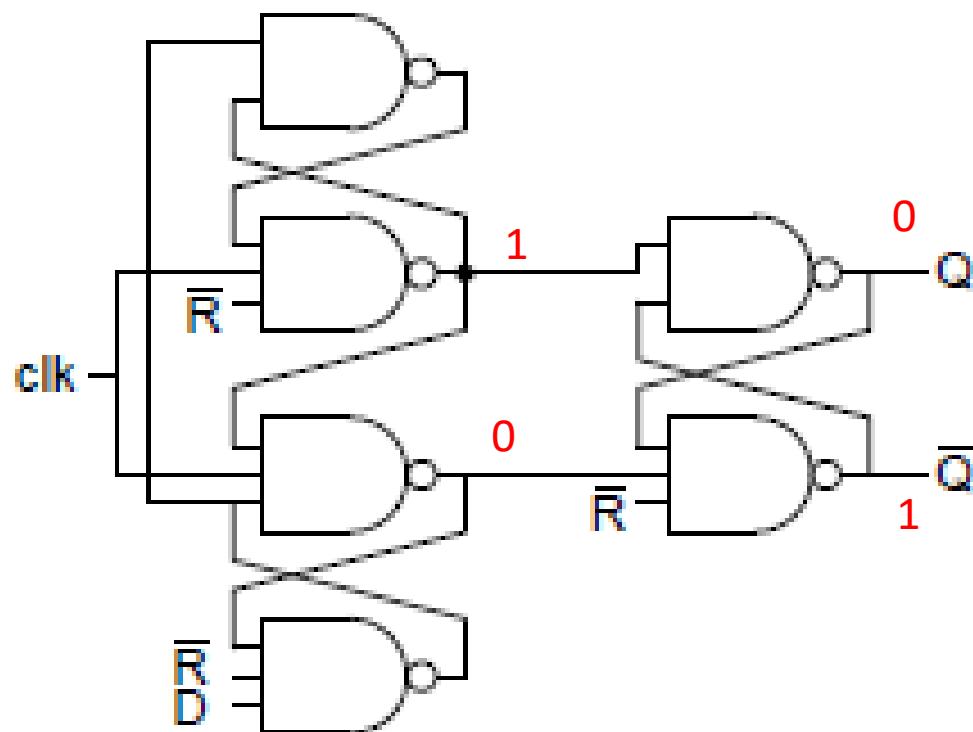
# Esercizio 3.13

R=1



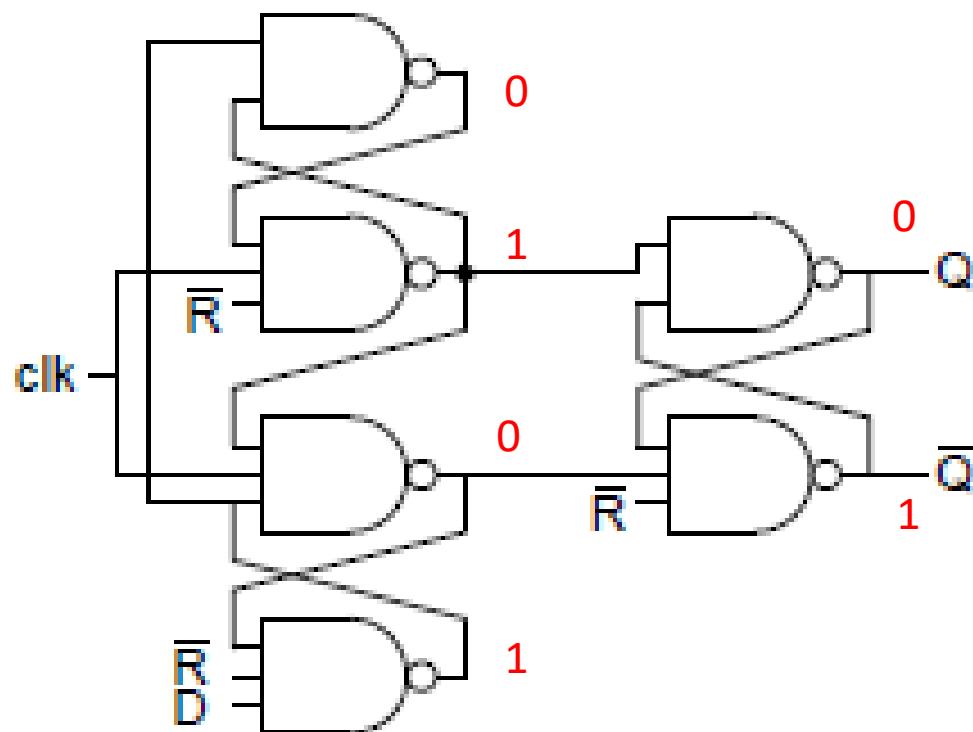
# Esercizio 3.13

$R \rightarrow 0$   $\text{clk}=1$



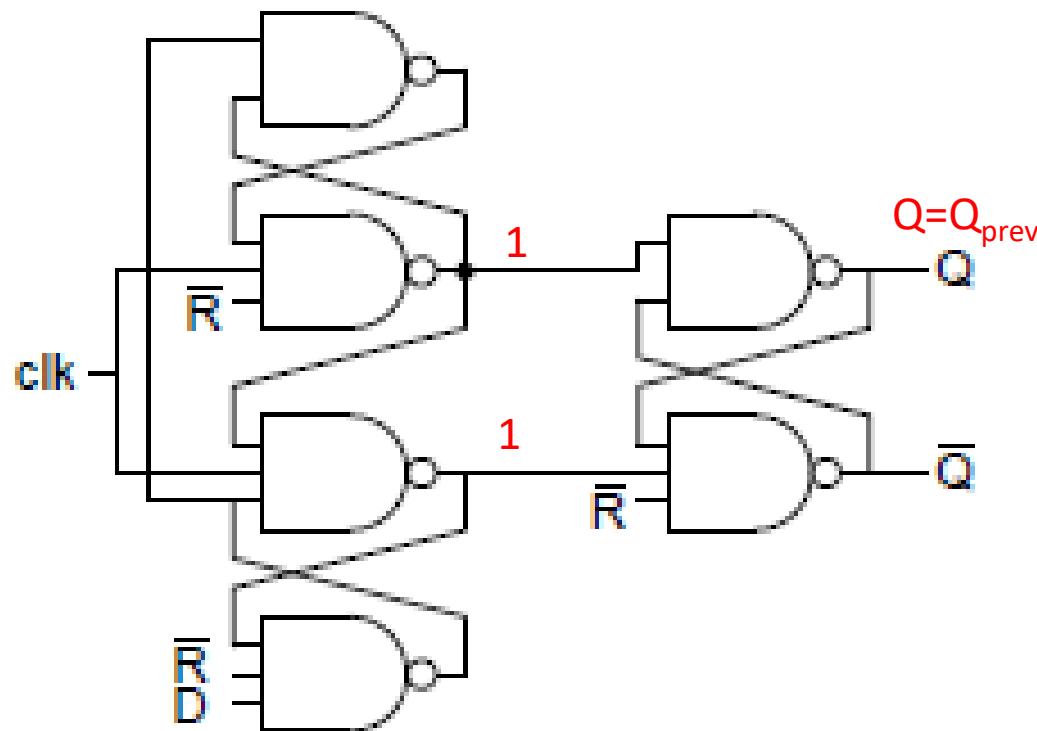
# Esercizio 3.13

$R \rightarrow 0$   $\text{clk}=1$



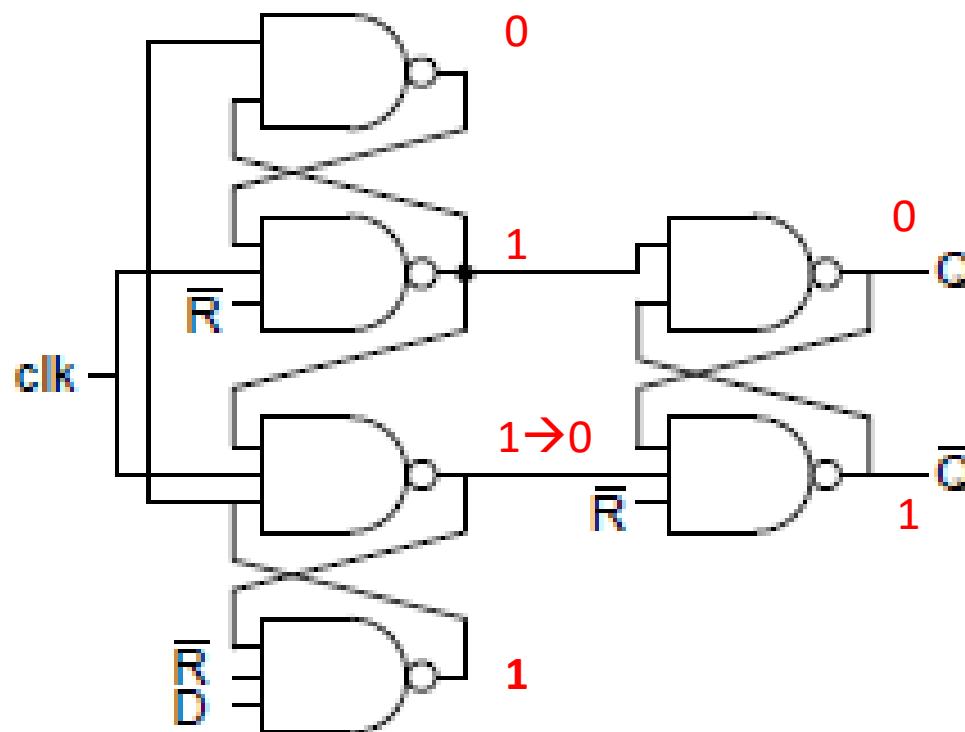
# Esercizio 3.13

$R=0, \text{clk}=0$



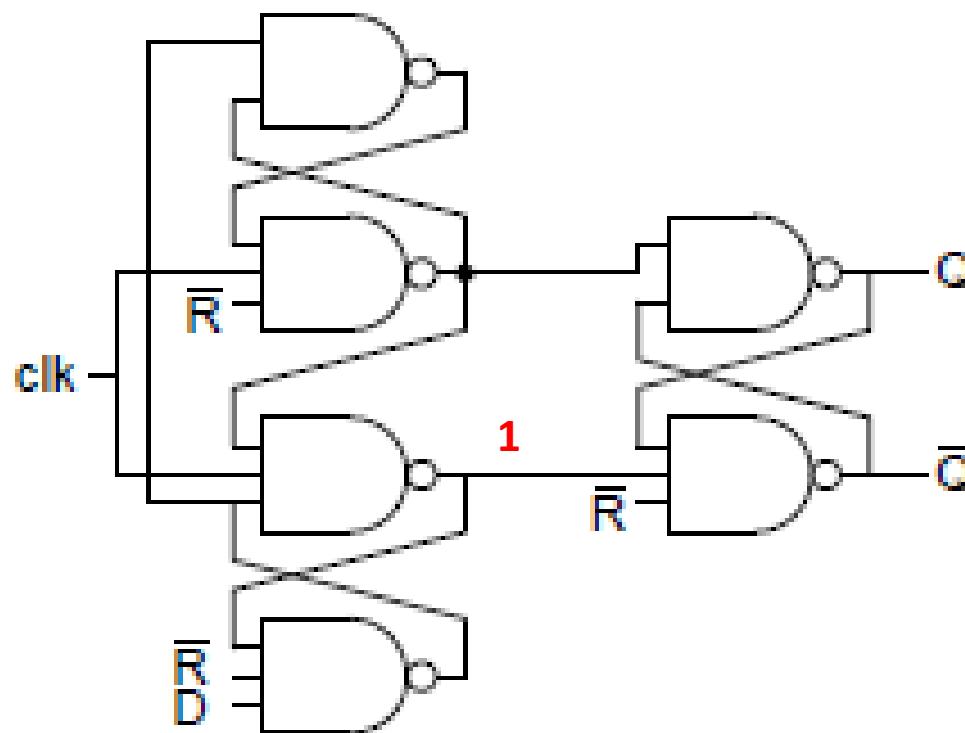
# Esercizio 3.13

R=0 clk $\rightarrow$ 1 D=0



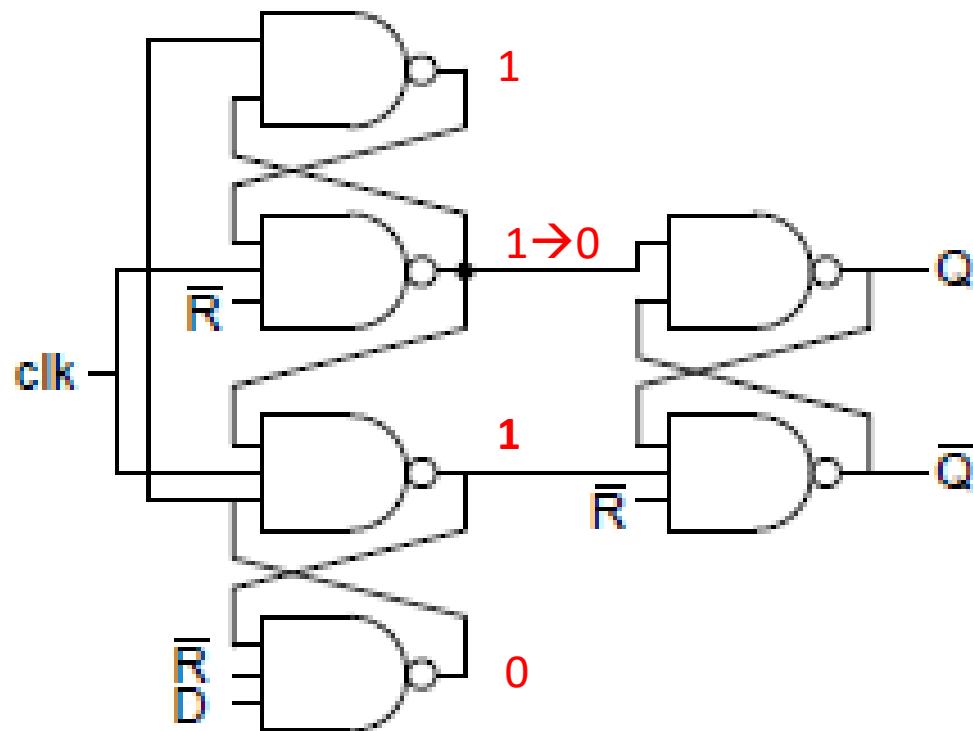
# Esercizio 3.13

R=0 clk $\rightarrow$ 1 D=1



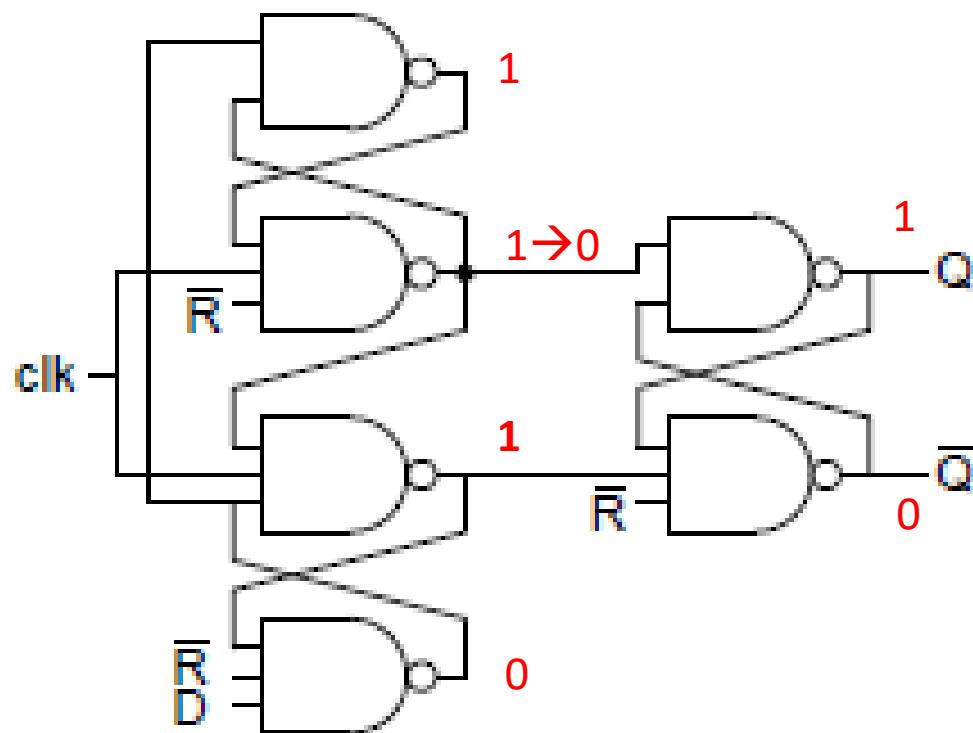
# Esercizio 3.13

R=0 clk $\rightarrow$ 1 D=1

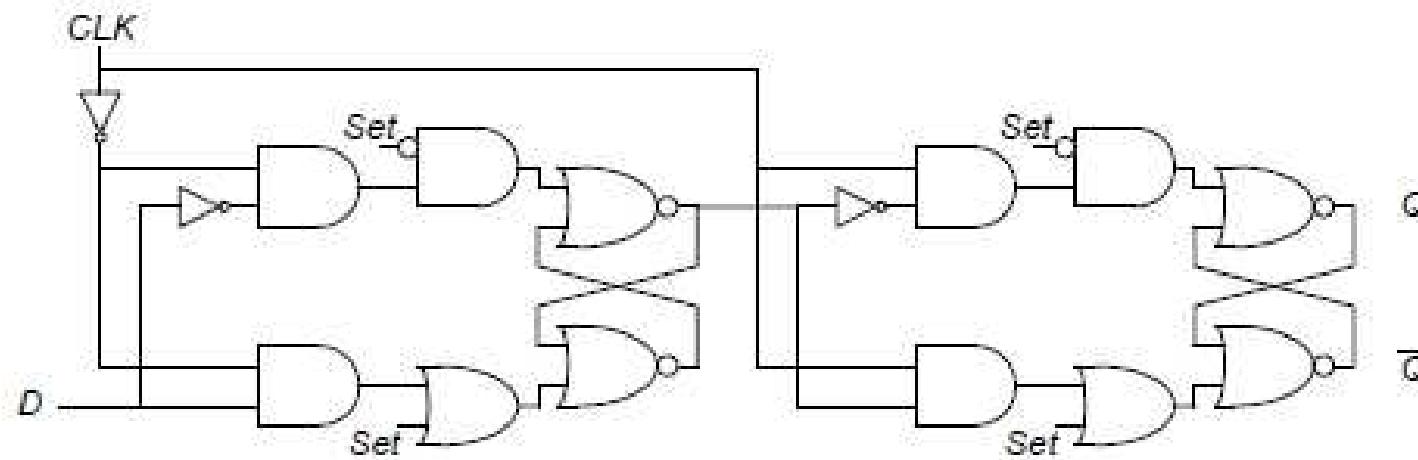


# Esercizio 3.13

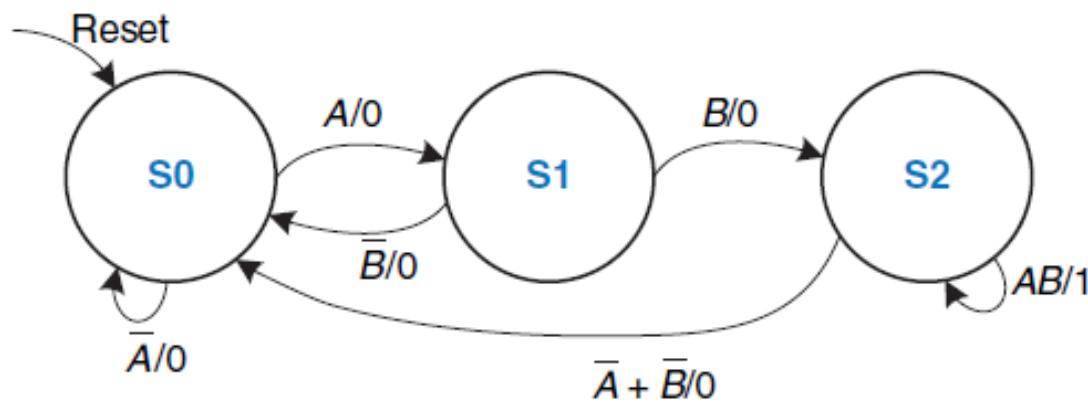
R=0 clk $\rightarrow$ 1 D=1



# Esercizio 3.15



# Esercizio 3.23



# Esercizio 3.23

| current state |       | inputs |     | next state |        | output |
|---------------|-------|--------|-----|------------|--------|--------|
| $s_1$         | $s_0$ | $a$    | $b$ | $s'_1$     | $s'_0$ | $q$    |
| 0             | 0     | 0      | X   | 0          | 0      | 0      |
| 0             | 0     | 1      | X   | 0          | 1      | 0      |
| 0             | 1     | X      | 0   | 0          | 0      | 0      |
| 0             | 1     | X      | 1   | 1          | 0      | 0      |
| 1             | 0     | 1      | 1   | 1          | 0      | 1      |
| 1             | 0     | 0      | 0   | 0          | 0      | 0      |
| 1             | 0     | 0      | 1   | 0          | 0      | 0      |
| 1             | 0     | 1      | 0   | 0          | 0      | 0      |

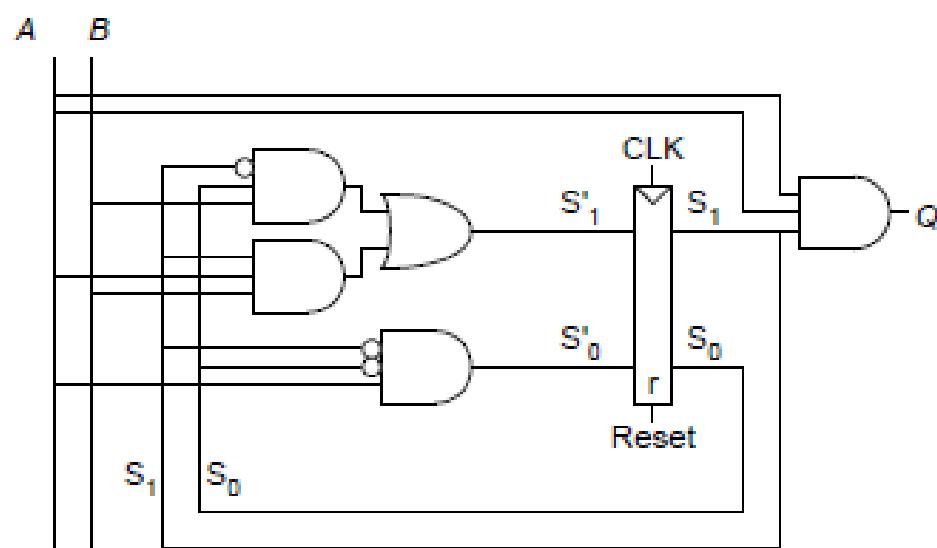
| state | encoding<br>$s_{1:0}$ |
|-------|-----------------------|
| S0    | 00                    |
| S1    | 01                    |
| S2    | 10                    |

# Esercizio 3.23

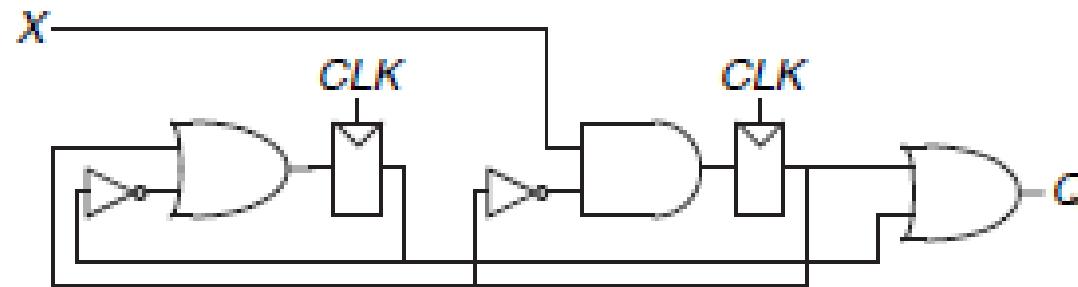
$$S_1 = \overline{S_1} S_0 B + S_1 A B$$

$$S_0 = \overline{S_1} \overline{S_0} A$$

$$Q' = S_1 A B$$



# Esercizio 3.31



# Esercizio 3.31

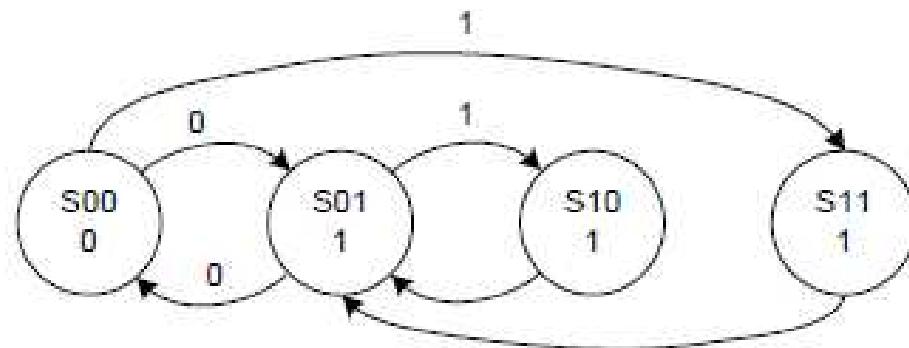
| current state |       | input | next state |        |
|---------------|-------|-------|------------|--------|
| $s_1$         | $s_0$ | x     | $s'_1$     | $s'_0$ |
| 0             | 0     | 0     | 0          | 1      |
| 0             | 0     | 1     | 1          | 1      |
| 0             | 1     | 0     | 0          | 0      |
| 0             | 1     | 1     | 1          | 0      |
| 1             | X     | X     | 0          | 1      |

TABLE 3.7 State transition table with binary encodings for Exercise 3.31

| current state |       | output |
|---------------|-------|--------|
| $s_1$         | $s_0$ | q      |
| 0             | 0     | 0      |
| 0             | 1     | 1      |
| 1             | X     | 1      |

TABLE 3.8 Output table for Exercise 3.31

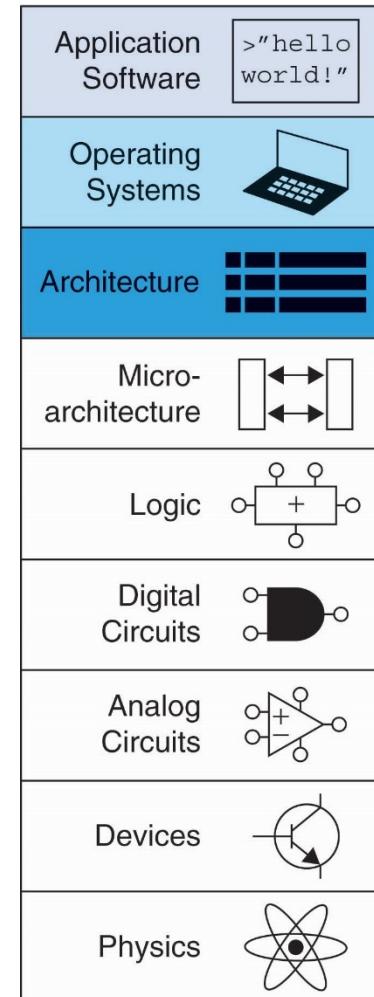
# Esercizio 3.31



# ARM assembly

**Architettura:** descrizione operazionale di computer

- Come un programmatore *vede* a basso livello un computer
- Definisce un insieme di istruzioni e di registri che fungono da operandi per tali istruzioni
- Istruzioni ARM
  - **Assembly language:** formato human-readable
  - **Machine language:** formato computer-readable
- Abbiamo già visto nelle lezioni precedenti come le diverse tipologie di istruzioni (memory, data-processing, e branch) sono tradotte in formato macchina



# Locazione degli Operandi

- Operandi di una istruzione ARM
  - Registri
  - Costanti (*immediates*)
- ARM ha 16 registri a 32 bit (R0... R15) che sono fisicamente equivalenti fra loro, ma dal punto di vista logico sono usati con scopi specifici

| Name            | Use                                          |
|-----------------|----------------------------------------------|
| <b>R0</b>       | Argument / return value / temporary variable |
| <b>R1-R3</b>    | Argument / temporary variables               |
| <b>R4-R11</b>   | Saved variables                              |
| <b>R12</b>      | Temporary variable                           |
| <b>R13 (SP)</b> | Stack Pointer                                |
| <b>R14 (LR)</b> | Link Register                                |
| <b>R15 (PC)</b> | Program Counter                              |

# Generazione di costanti

E' possibile definire costanti con l'istruzione MOV:

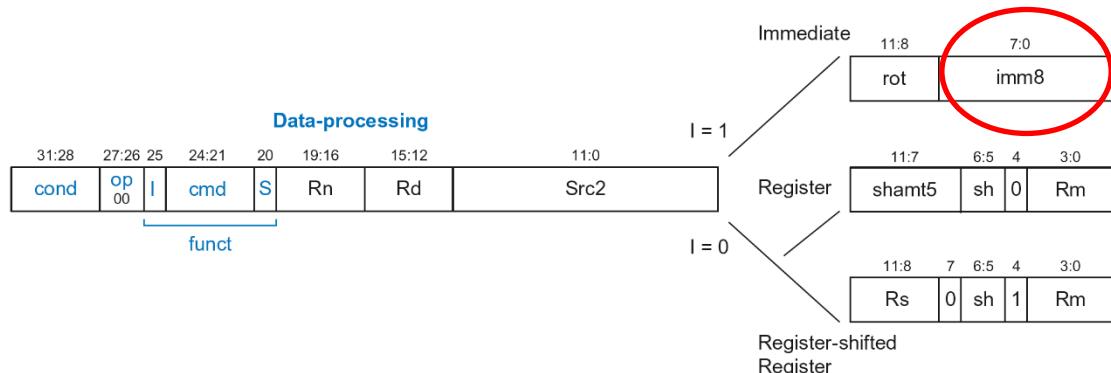
## C Code

```
//int: 32-bit signed word  
int a = 23;  
int b = 0x45;
```

## ARM Assembly Code

```
; R0 = a, R1 = b  
MOV R0, #23  
MOV R1, #0x45
```

Le costanti così generate hanno una precisione massima di 8 bits



# Generazione di costanti

E' possibile definire costanti con l'istruzione MOV:

## C Code

```
//int: 32-bit signed word  
int a = 23;  
int b = 0x45;
```

## ARM Assembly Code

```
; R0 = a, R1 = b  
MOV R0, #23  
MOV R1, #0x45
```

**Nota:** MOV può anche essere usato per spostare il contenuto di un registro in un altro registro:

```
MOV R7, R9
```

# Generazione di costanti

Generare costanti usando MOV e ORR:

## C Code

```
int a = 0x7EDC8765;
```

## ARM Assembly Code

```
# R0 = a
MOV R0, #0x7E000000
ORR R0, R0, #0xDC0000
ORR R0, R0, #0x8700
ORR R0, R0, #0x65
```

# Istruzioni condizionali

## C Code

## ARM Assembly Code

; R0=f, R1=g, R2=h, R3=i, R4=j

```
if (i == j)      CMP R3, R4          ; set flags with R3-R4
    f = g + h;   BNE L1           ; if i!=j, skip if block
                  ADD R0, R1, R2 ; f = g + h
```

L1

```
f = f - i;      SUB R0, R0, R3 ; f = f - i
```

**Nota:** il codice assembly effettua il test opposto ( $i \neq j$ ) rispetto a quello di alto livello ( $i == j$ )

# Istruzioni condizionali

## C Code

```
if (i == j)  
    f = g + h;  
f = f - i;
```

## ARM Assembly Code

;R0=f, R1=g, R2=h, R3=i, R4=j

|             |                                        |
|-------------|----------------------------------------|
| if (i == j) | CMP R3, R4 ; set flags with R3-R4      |
| f = g + h;  | ADDEQ R0, R1, R2 ; if (i==j) f = g + h |
| f = f - i;  | SUB R0, R0, R2 ; f = f - i             |

# Istruzioni condizionali

Codice alternativo per piccoli blocchi di codice:

## Original

```
CMP R3, R4  
BNE L1  
ADD R0, R1, R2  
L1  
SUB R0, R0, R3
```

## Alternate Assembly Code

```
;R0=f, R1=g, R2=h, R3=i, R4=j  
  
CMP R3, R4 ; set flags with R3-R4  
ADDEQ R0, R1, R2 ; if (i==j) f = g + h  
SUB R0, R0, R3 ; f = f - i
```

# Istruzioni condizionali

## C Code

```
if (i == j)  
    f = g + h;  
  
else  
    f = f - i;
```

## ARM Assembly Code

```
;R0=f, R1=g, R2=h, R3=i, R4=j  
  
CMP R3, R4          ; set flags with R3-R4  
BNE L1              ; if i!=j, skip if block  
ADD R0, R1, R2      ; f = g + h  
B L2                ; branch past else block  
  
L1  
SUB R0, R0, R3      ; f = f - i  
L2
```

# Istruzioni condizionali

Codice alternativo per piccoli blocchi di codice:

## Original

```
CMP R3, R4  
BNE L1  
ADD R0, R1, R2  
B L2  
L1  
SUB R0, R0, R3  
L2
```

## Alternate Assembly Code

```
;R0=f, R1=g, R2=h, R3=i, R4=j  
  
CMP R3, R4 ; set flags with R3-R4  
ADDEQ R0, R1, R2 ; if (i==j) f = g + h  
SUBNE R0, R0, R3 ; else f = f - i
```

# Istruzioni iterative

## C Code

```
// determines the power ; R0 = pow, R1 = x
// of x such that 2x = 128 MOV R0, #1 ; pow = 1
int pow = 1;           MOV R1, #0 ; x = 0
int x    = 0;
```

```
while (pow != 128) {
    pow = pow * 2;
    x  = x + 1;
}
```

## ARM Assembly Code

```
; R0 = pow, R1 = x
MOV R0, #1 ; pow = 1
MOV R1, #0 ; x = 0

WHILE
    CMP R0, #128 ; R0-128
    BEQ DONE ; if (pow==128)
               ; exit loop
    LSL R0, R0, #1 ; pow=pow*2
    ADD R1, R1, #1 ; x=x+1
    B WHILE ; repeat loop

DONE
```

Il codice assembly verifica la condizione opposta (`pow == 128`) a quella del C (`pow != 128`).

# Istruzioni iterative

```
for (initialization; condition; loop operation)  
    statement
```

- **initialization:** eseguita prima che il loop inizi
- **condition:** condizione di continuazione che è verificata all'inizio di ogni iterazione
- **loop operation:** eseguita alla fine di ogni iterazione
- **statement:** eseguito ad ogni iterazione, ovvero fintantoché la condizione di continuazione è verificata

# Istruzioni iterative

## C Code

```
// adds numbers from 1-9 ; R0 = i, R1 = sum
```

```
int sum = 0;
```

```
int i;
```

```
for (i=1; i!=10; i=i+1)
```

```
    sum = sum + i;
```

## ARM Assembly Code

```
MOV R1, #0 ; sum = 0
```

```
MOV R0, #1 ; i = 1
```

```
FOR
```

```
CMP R0, #10 ; R0-10
```

```
BEQ DONE ; if (i==10)  
          ; exit loop
```

```
ADD R1, R1, R0 ; sum=sum + i
```

```
ADD R0, R0, #1 ; i = i + 1
```

```
B FOR ; repeat loop
```

```
DONE
```

# Istruzioni iterative

In ARM, i loop decrescenti fino a 0 sono più efficienti

## C Code

```
// adds numbers from 1-9 ; R0 = i, R1 = sum
```

```
int sum = 0;
```

```
int i;
```

```
for (i=9; i!=0; i=i-1)
```

```
    sum = sum + i;
```

## ARM Assembly Code

```
MOV R1, #0 ; sum = 0
```

```
MOV R0, #9 ; i = 9
```

```
FOR
```

```
ADD R1, R1, R0 ; sum=sum + i
```

```
SUBS R0, R0, #1 ; i = i - 1
```

```
; and set flags
```

```
BNE FOR ; if (i!=0)
```

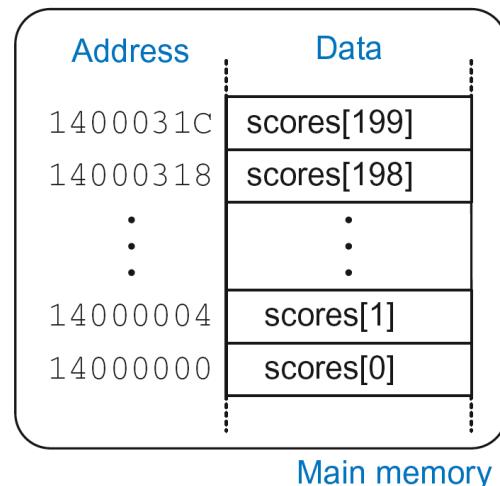
```
; repeat loop
```

Si risparmiano 2 istruzioni per ogni iterazione:

- Si accorpano decremento e comparazione: SUBS R0, R0, #1
- Solo un branch invece di due
- Nota l'uso di NE che corrisponde a  $\bar{Z}$ , ovvero R0 non è ancora 0

# Arrays

- Consente l'accesso ad una quantità di dati simili
  - **Index:** accesso ad un qualche elemento
  - **Size:** numero di elementi
- Esempio: array di parole (32bit)
  - **Base address** = 0x14000000 (indirizzo di scores[0])
  - Gli altri elementi sono raggiungibili a partire dal base address



# Arrays

## C Code

```
int array[5];  
array[0] = array[0] * 8;  
array[1] = array[1] * 8;
```

## ARM Assembly Code

```
; R0 = array base address  
MOV R0, #0x60000000 ; R0 = 0x60000000  
  
LDR R1, [R0] ; R1 = array[0]  
LSL R1, R1, 3 ; R1 = R1 << 3 = R1*8  
STR R1, [R0] ; array[0] = R1  
  
LDR R1, [R0, #4] ; R1 = array[1]  
LSL R1, R1, 3 ; R1 = R1 << 3 = R1*8  
STR R1, [R0, #4] ; array[1] = R1
```

# Arrays

## C Code

```
int array[200];
int i;
for (i=199; i >= 0; i = i - 1)
    array[i] = array[i] * 8;
```

LDR R2, [R0, R1, LSL #2]

Poiché gli elementi degli array sono  
degli interi, ovvero parole di 4 byte,  
gli indirizzi devono scorrere di 4 in 4

## ARM Assembly Code

```
; R0 = array base address, R1 = i
MOV R0, 0x60000000
MOV R1, #199
.
FOR
    LDR R2, [R0, R1, LSL #2]      ; R2 = array(i)
    LSL R2, R2, #3                ; R2 = R2<<3 = R2*8
    STR R2, [R0, R1, LSL #2]      ; array(i) = R2
    SUBS R1, R1, #1               ; i = i - 1
                                ; and set flags
    BPL FOR                      ; if (i>=0) repeat loop
```

# Chiamate di funzioni

- **Caller:** funzione chiamante, in questo caso `main`
- **Callee:** funzione chiamata, in questo caso `sum`

## C Code

```
void main()
{
    int y;
    y = sum(42, 7);
    ...
}

int sum(int a, int b)
{
    return (a + b);
}
```

# Contratto

- **Caller:**
  - Passa gli argomenti al callee
  - Esegue un jump al callee
- **Callee:**
  - Esegue la funzione chiamata
  - ritorna il risultato al caller
  - ritorna nel punto di chiamata del caller
  - non deve sovrascrivere i saved register e la memoria usata dal caller

# Convenzioni ARM per chiamata a funzione

- **Chiamata a funzione:** branch and link BL
- **Return** da funzione: ripristina in PC il valore del link register  
`MOV PC, LR`
- **Argomenti:** R0 – R3
- **Valore di ritorno:** R0

# Esempio di chiamata a funzione

## C Code

```
int main() {  
    simple();  
    a = b + c;  
}
```

```
void simple() {  
    return;  
}
```

## ARM Assembly Code

```
0x00000200 MAIN      BL  SIMPLE  
0x00000204           ADD R4, R5, R6
```

...

```
0x00401020 SIMPLE   MOV PC, LR
```

**void** significa che **simple** non ritorna un valore

# Esempio di chiamata a funzione

## C Code

```
int main() {  
    simple();  
    a = b + c;  
}
```

```
void simple() {  
    return;  
}
```

## ARM Assembly Code

|            |      |                  |
|------------|------|------------------|
| 0x00000200 | MAIN | <b>BL SIMPLE</b> |
| 0x00000204 |      | ADD R4, R5, R6   |
| ...        |      |                  |

|            |        |                   |
|------------|--------|-------------------|
| 0x00401020 | SIMPLE | <b>MOV PC, LR</b> |
|------------|--------|-------------------|

**BL**

branches to SIMPLE

LR = 0x00000204

**MOV PC, LR**

makes PC = LR

(the next instruction executed is at 0x00000204)

# Argomenti e valore di ritorno

## Convenzioni ARM:

- Argomenti: R0 - R3
- Valore di ritorno: R0

## C Code

```
int main() {
    int y;
    ...
    y = diffofsums(2, 3, 4, 5); // 4 arguments
    ...
}

int diffofsums(int f, int g, int h, int i) {
    int result;
    result = (f + g) - (h + i);
    return result;           // return value
}
```

# Argomenti e valore di ritorno

## ARM Assembly Code

```
; R4 = y
MAIN
...
    MOV R0, #2          ; argument 0 = 2
    MOV R1, #3          ; argument 1 = 3
    MOV R2, #4          ; argument 2 = 4
    MOV R3, #5          ; argument 3 = 5
    BL DIFFOFSUMS      ; call function
    MOV R4, R0          ; y = returned value
...
; R4 = result
DIFFOFSUMS
    ADD R8, R0, R1      ; R8 = f + g
    ADD R9, R2, R3      ; R9 = h + i
    SUB R4, R8, R9      ; result = (f + g) - (h + i)
    MOV R0, R4          ; put return value in R0
    MOV PC, LR          ; return to caller
```

# Argomenti e valore di ritorno

## ARM Assembly Code

```
; R4 = result
DIFFOFSUMS
    ADD R8, R0, R1      ; R8 = f + g
    ADD R9, R2, R3      ; R9 = h + i
    SUB R4, R8, R9      ; result = (f + g) - (h + i)
    MOV R0, R4          ; put return value in R0
    MOV PC, LR          ; return to caller
```

- `diffofsums` sovrascrive indebitamente i 3 registri `R4, R8, R9` dedicati alle saved variables della funzione chiamante `main`
- `diffofsums` invece dovrebbe usare lo *stack* in memoria per memorizzare temporaneamente i valori di questi registri prima di operare su di essi

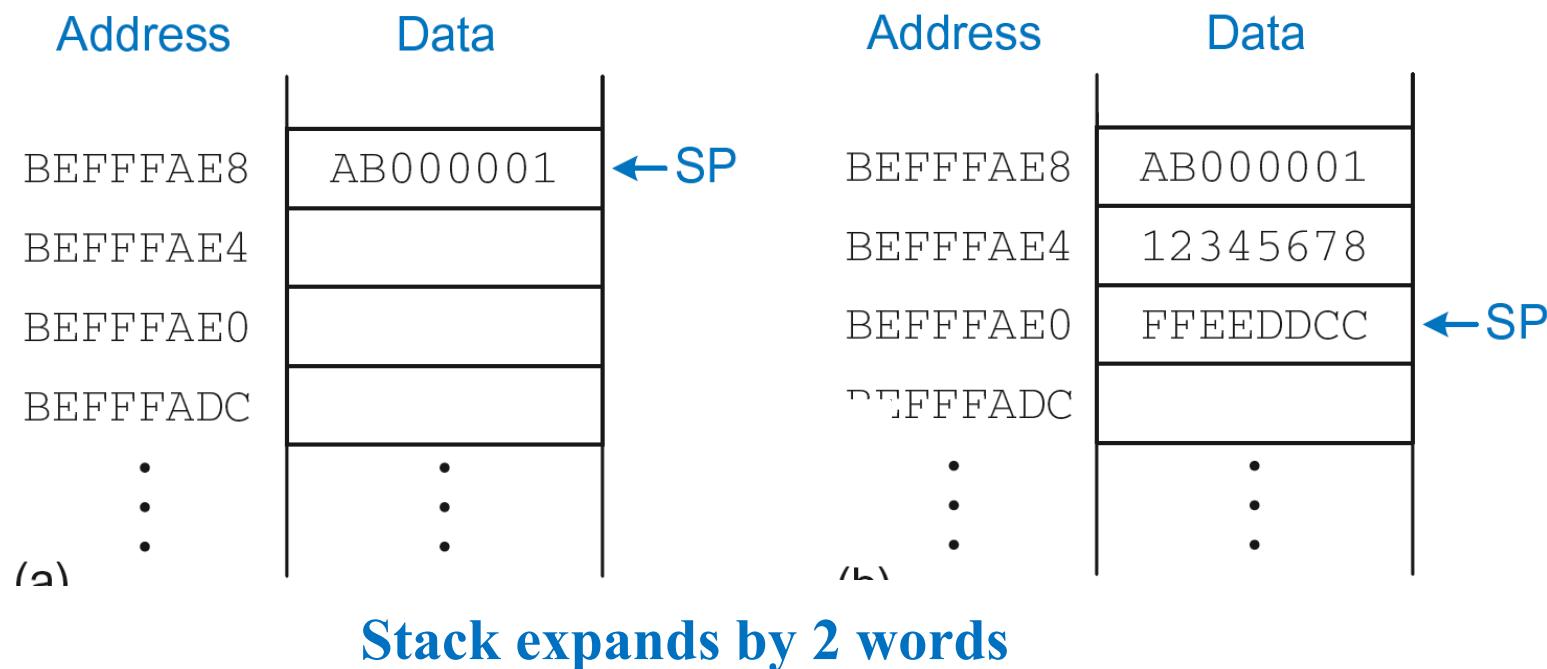
# Lo stack delle funzioni

- Lo stack costituisce la memoria usata per salvare temporaneamente il valore delle variabili
- È organizzato come una pila last-in-first-out (LIFO)
- Operazioni sullo stack
  - **Expands:** usa più memoria quando necessaria (push)
  - **Contracts:** usa meno memoria quando possibile (pop)



# Lo stack delle funzioni

- In termini di indirizzi di memoria, l'espansione dello stack avviene in senso decrescente (dagli indirizzi maggiori a quelli più piccoli)
- Stack pointer: SP *punta* al top dello stack



# Come le funzioni usano lo stack

- Le funzioni chiamate non devono avere side effect inattesi
- Invece, nell esempio precedente diffofsums sovrascrive indebitamente i 3 registri R4, R8, R9

## ARM Assembly Code

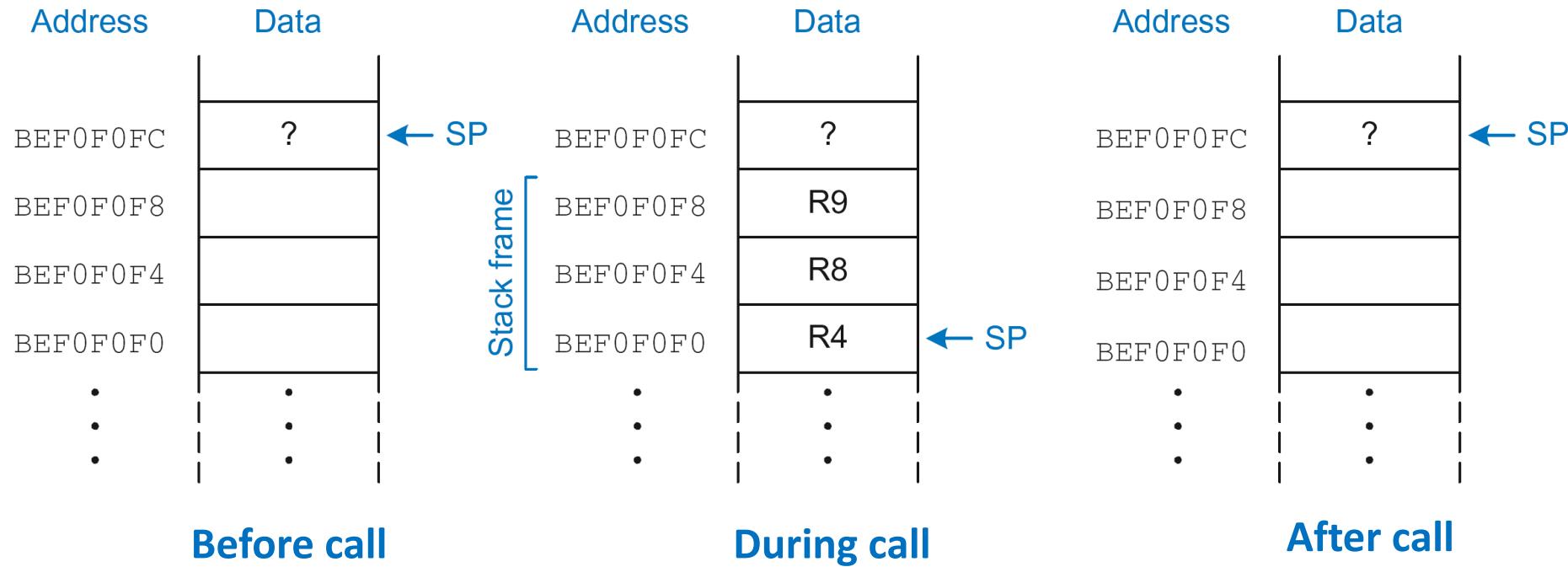
```
; R4 = result
DIFFOFSUMS
    ADD R8, R0, R1      ; R8 = f + g
    ADD R9, R2, R3      ; R9 = h + i
    SUB R4, R8, R9      ; result = (f + g) - (h + i)
    MOV R0, R4          ; put return value in R0
    MOV PC, LR          ; return to caller
```

# Memorizzare i registri sullo stack

## ARM Assembly Code

```
; R2 = result
DIFFOFSUMS
    SUB SP, SP, #12      ; make space on stack for 3 registers
    STR R9, [SP, #8]     ; save R9 on stack
    STR R8, [SP, #4]     ; save R8 on stack
    STR R4, [SP]          ; save R4 on stack
    ADD R8, R0, R1        ; R8 = f + g
    ADD R9, R2, R3        ; R9 = h + i
    SUB R4, R8, R9        ; result = (f + g) - (h + i)
    MOV R0, R4            ; put return value in R0
    LDR R4, [SP]          ; restore R4 from stack
    LDR R8, [SP, #4]       ; restore R8 from stack
    LDR R9, [SP, #8]       ; restore R9 from stack
    ADD SP, SP, #12        ; deallocate stack space
    MOV PC, LR             ; return to caller
```

# Memorizzare i registri sullo stack



# Callee/caller saved

| Preserved<br><i>Callee-Saved</i> | Nonpreserved<br><i>Caller-Saved</i> |
|----------------------------------|-------------------------------------|
| R4-R11                           | R12                                 |
| R14 (LR)                         | R0-R3                               |
| R13 (SP)                         | CPSR                                |
| stack above SP                   | stack below SP                      |

# Pop e push

- Salvare e ripristinare registri dallo stack è una operazione così frequente che ARM mette a disposizione delle istruzioni specifiche, Pop e Push, che consentono di avere un codice più succinto.
- Push consente di salvare in memoria più registri aggiornare consistentemente lo stack: PUSH { R4, R8, R9 }
- Pop ripristina uno o più registri e incrementa consistentemente il valore dello stack: POP { R4, R8, R9 }

# Chiamate a funzioni innestate

## C Code

```
int f1(int a, int b) {  
    int i, x;  
    x = (a + b) * (a - b);  
    for (i=0; i<a; i++)  
        x = x + f2(b+i);  
    return x;  
}  
  
int f2(int p) {  
    int r;  
    r = p + 5;  
    return r + p;  
}
```

## ARM Assembly Code

```
; R0=a, R1=b, R4=i, R5=x ; R0=p, R4=r  
F1  
    PUSH {R4, R5, LR}  
    ADD R5, R0, R1  
    SUB R12, R0, R1  
    MUL R5, R5, R12  
    MOV R4, #0  
FOR  
    CMP R4, R0  
    BGE RETURN  
    PUSH {R0, R1}  
    ADD R0, R1, R4  
    BL F2  
    ADD R5, R5, R0  
    POP {R0, R1}  
    ADD R4, R4, #1  
    B FOR  
RETURN  
    MOV R0, R5  
    POP {R4, R5, LR}  
    MOV PC, LR  
F2  
    PUSH {R4}  
    ADD R4, R0, 5  
    ADD R0, R4, R0  
    POP {R4}  
    MOV PC, LR
```

# Chiamate a funzioni innestate

## ARM Assembly Code

```
; R0=a, R1=b, R4=i, R5=x          ; R0=p, R4=r
F1                                         F2
    PUSH {R4, R5, LR} ; save regs      PUSH {R4}           ; save regs
    ADD   R5, R0, R1  ; x = (a+b)      ADD   R4, R0, 5    ; r = p+5
    SUB   R12, R0, R1 ; temp = (a-b)
    MUL   R5, R5, R12 ; x = x*temp
    MOV   R4, #0       ; i = 0        ADD   R0, R4, R0    ; return r+p
                                         POP   {R4}           ; restore regs
                                         MOV   PC, LR      ; return

FOR
    CMP   R4, R0       ; i < a?
    BGE   RETURN       ; no: exit loop
    PUSH  {R0, R1}     ; save regs
    ADD   R0, R1, R4  ; arg is b+i
    BL    F2          ; call f2(b+i)
    ADD   R5, R5, R0  ; x = x+f2(b+i)
    POP   {R0, R1}     ; restore regs
    ADD   R4, R4, #1  ; i++
    B    FOR          ; repeat loop

RETURN
    MOV   R0, R5       ; return x
    POP   {R4, R5, LR} ; restore regs
    MOV   PC, LR      ; return
```

| Address  | Data |
|----------|------|
| BEF7FF0C | ?    |
| BEF7FF08 |      |
| BEF7FF04 |      |
| BEF7FF00 |      |
| BEF7FEFC |      |
| BEF7FEF8 |      |
| BEF7FEF4 |      |
| ⋮        | ⋮    |
| ⋮        | ⋮    |
| ⋮        | ⋮    |

At beginning of f1

f1's stack frame

| Address  | Data |
|----------|------|
| BEF7FF0C | ?    |
| BEF7FF08 | LR   |
| BEF7FF04 | R5   |
| BEF7FF00 | R4   |
| BEF7FEFC | R1   |
| BEF7FEF8 | R0   |
| BEF7FEF4 |      |
| ⋮        | ⋮    |
| ⋮        | ⋮    |
| ⋮        | ⋮    |

Just before calling f2

f1's stack frame

f2's stack frame

| Address  | Data |
|----------|------|
| BEF7FF0C | ?    |
| BEF7FF08 | LR   |
| BEF7FF04 | R5   |
| BEF7FF00 | R4   |
| BEF7FEFC | R1   |
| BEF7FEF8 | R0   |
| BEF7FEF4 | R4   |
| ⋮        | ⋮    |
| ⋮        | ⋮    |
| ⋮        | ⋮    |

After calling f2

## C Code

```
int factorial(int n) {  
    if (n <= 1)  
        return 1;  
    else  
        return (n * factorial(n-1));  
}
```

# ARM Assembly Code

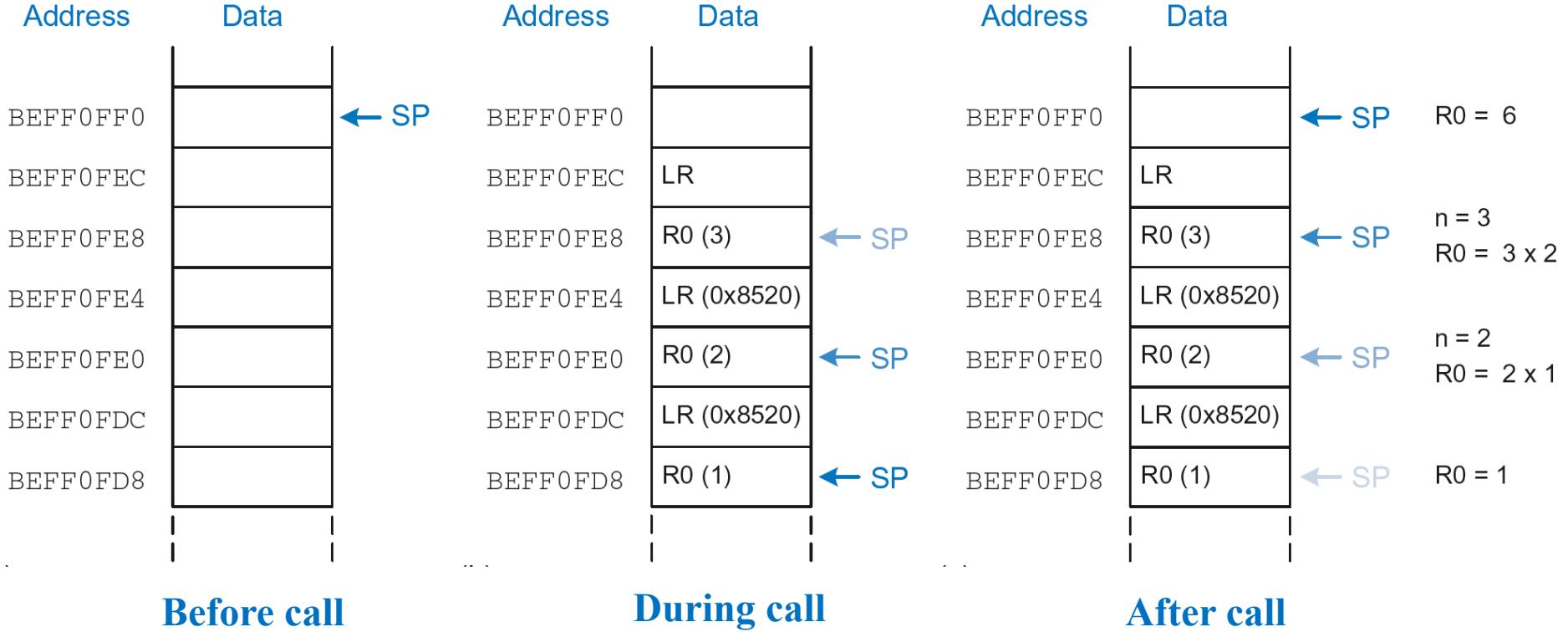
|             |           |                    |                             |
|-------------|-----------|--------------------|-----------------------------|
| <b>0x94</b> | FACTORIAL | STR R0, [SP, #-4]! | ; store R0 on stack         |
| <b>0x98</b> |           | STR LR, [SP, #-4]! | ; store LR on stack         |
| <b>0x9C</b> |           | CMP R0, #2         | ; set flags with R0-2       |
| <b>0xA0</b> |           | BHS ELSE           | ; if (r0>=2) branch to else |
| <b>0xA4</b> |           | MOV R0, #1         | ; otherwise return 1        |
| <b>0xA8</b> |           | ADD SP, SP, #8     | ; restore SP 1              |
| <b>0xAC</b> |           | MOV PC, LR         | ; return                    |
| <b>0xB0</b> | ELSE      | SUB R0, R0, #1     | ; n = n - 1                 |
| <b>0xB4</b> |           | BL FACTORIAL       | ; recursive call            |
| <b>0xB8</b> |           | LDR LR, [SP], #4   | ; restore LR                |
| <b>0xBC</b> |           | LDR R1, [SP], #4   | ; restore R0 (n) into R1    |
| <b>0xC0</b> |           | MUL R0, R1, R0     | ; R0 = n*factorial(n-1)     |
| <b>0xC4</b> |           | MOV PC, LR         | ; return                    |

## C Code

```
int factorial(int n) {  
    if (n <= 1)  
        return 1;  
  
    else  
        return (n * factorial(n-1));  
}
```

## ARM Assembly Code

```
0x94 FACTORIAL    STR R0, [SP, #-4]!  
0x98              STR LR, [SP, #-4]!  
0x9C              CMP R0, #2  
0xA0              BHS ELSE  
0xA4              MOV R0, #1  
0xA8              ADD SP, SP, #8  
0xAC              MOV PC, LR  
0xB0 ELSE         SUB R0, R0, #1  
0xB4              BL  FACTORIAL  
0xB8              LDR LR, [SP], #4  
0xBC              LDR R1, [SP], #4  
0xC0              MUL R0, R1, R0  
0xC4              MOV PC, LR
```



# Esercizi

| cond | Mnemonic     | Name                                | CondEx                                           |
|------|--------------|-------------------------------------|--------------------------------------------------|
| 0000 | EQ           | Equal                               | Z                                                |
| 0001 | NE           | Not equal                           | $\overline{Z}$                                   |
| 0010 | CS/HS        | Carry set / unsigned higher or same | C                                                |
| 0011 | CC/LO        | Carry clear / unsigned lower        | $\overline{C}$                                   |
| 0100 | MI           | Minus / negative                    | N                                                |
| 0101 | PL           | Plus / positive or zero             | $\overline{N}$                                   |
| 0110 | VS           | Overflow / overflow set             | V                                                |
| 0111 | VC           | No overflow / overflow clear        | $\overline{V}$                                   |
| 1000 | HI           | Unsigned higher                     | $\overline{Z}C$                                  |
| 1001 | LS           | Unsigned lower or same              | Z OR $\overline{C}$                              |
| 1010 | GE           | Signed greater than or equal        | $\overline{N} \oplus V$                          |
| 1011 | LT           | Signed less than                    | $N \oplus V$                                     |
| 1100 | GT           | Signed greater than                 | $\overline{Z}(\overline{N} \oplus \overline{V})$ |
| 1101 | LE           | Signed less than or equal           | Z OR ( $N \oplus V$ )                            |
| 1110 | AL (or none) | Always / unconditional              | Ignored                                          |

# Esercizi

Si supponga che lo stato dei registri R0, R1, e R2 sia rispettivamente 0x00000100, 0x0000002A, e 0x00000133. Calcolare lo stato dei flag NZCV del Current Program Status Register al termine delle seguenti istruzioni:

CMP R0 R1

SUBGES R0 R0 R2

ADDLTS R0 R1 R2

# Esercizi

Si supponga che lo stato dei registri R0, R1, e R2 sia rispettivamente 0x00000100, 0x0000002A, e 0x00000133. Calcolare lo stato dei flag NZCV del Current Program Status Register al termine delle seguenti istruzioni:

CMP R0 R1

SUBGES R0 R0 R2

ADDLTS R0 R1 R2

R0 è maggiore di R1 quindi SUBGES viene eseguita

R0= 0x FFFF FFCD      N: 1 Z:0 C:0 V:0

La condizione LT (N XOR V) è soddisfatta quindi ADDLTS viene eseguita

R0= 0x 0000 015D

N: 0 Z:0 C:0 V:0

# Esercizi

Si supponga che lo stato dei registri R0, R1, e R2 sia rispettivamente #00000100, #00000100, e #0000010A. Calcolare lo stato dei flag NZCV del Current Program Status Register al termine delle seguenti istruzioni:

CMP R0 R1  
SUBGES R0 R0 R2  
ADDLTS R0 R1 R2

# Esercizi

Il seguente codice assembly rappresenta la funzione fattoriale, ma ci sono due errori. Scrivere nell'apposito riquadro il codice corretto. [Si ricordi che SP sta per stack pointer, LR per link register e PC per program counter.]

```
FACTORIAL
    PUSH R0, LR
    CMP R0, #1
    BLT ELSE
    MOV R0, #1
    ADD SP, SP, #8
    MOV PC, LR
ELSE
    SUB R0, R0, #1
    BL FACTORIAL
    POP R1, LR
    MUL R0, R1, R0
    MOV PC, SP
```

# Esercizi

Il seguente codice assembly rappresenta la funzione fattoriale, ma ci sono due errori. Scrivere nell'apposito riquadro il codice corretto. [Si ricordi che SP sta per stack pointer, LR per link register e PC per program counter.]

```
FACTORIAL
    PUSH R0, LR
    CMP R0, #1
    BLT ELSE
    MOV R0, #1
    ADD SP, SP, #8
    MOV PC, LR
ELSE
    SUB R0, R0, #1
    BL FACTORIAL
    POP R1, LR
    MUL R0, R1, R0
    MOV PC, SP
```

```
FACTORIAL
    PUSH R0, LR
    CMP R0, #1
    BGT ELSE
    MOV R0, #1
    ADD SP, SP, #8
    MOV PC, LR
ELSE
    SUB R0, R0, #1
    BL FACTORIAL
    POP R1, LR
    MUL R0, R1, R0
    MOV PC, LR
```

# Esercizi

Il seguente codice assembly rappresenta la funzione fattoriale, ma ci sono due errori. Scrivere nell'apposito riquadro il codice corretto. [Si ricordi che SP sta per stack pointer, LR per link register e PC per program counter.]

```
FACTORIAL
    PUSH R0, LR
    CMP R0, #1
    BGT ELSE
    MOV R0, #1
    ADD SP, SP, #8
    MOV PC, LR
ELSE
    SUB R0, R0, #2
    BL FACTORIAL
    POP R1, LR
    MUL R0, R1, R0
    MOV PC, SP
```

# Esercizi

Il seguente codice assembly rappresenta la funzione fattoriale, ma ci sono due errori. Scrivere nell'apposito riquadro il codice corretto. [Si ricordi che SP sta per stack pointer, LR per link register e PC per program counter.]

```
FACTORIAL
    PUSH R0, LR
    CMP R0, #1
    BGT ELSE
    MOV R0, #1
    ADD SP, SP, #8
    MOV PC, LR
ELSE
    SUB R0, R0, #2
    BL FACTORIAL
    POP R1, LR
    MUL R0, R1, R0
    MOV PC, SP
```

```
FACTORIAL
    PUSH R0, LR
    CMP R0, #1
    BGT ELSE
    MOV R0, #1
    ADD SP, SP, #8
    MOV PC, LR
ELSE
    SUB R0, R0, #1
    BL FACTORIAL
    POP R1, LR
    MUL R0, R1, R0
    MOV PC, LR
```

# Esercizi

Si consideri il seguente programma assembly:

```
MOVE R0, #5
MOVE R1, #0x23
LOOP
SUBS R1, R1, R0
ADD R0, R0, #10
BPL LOOP
ADD R1, R1, #20
```

Indicare esadecimale il valore di R1 al termine dell'esecuzione.

# Esercizi

Si consideri il seguente programma assembly:

|                                                                  |        |
|------------------------------------------------------------------|--------|
| <b>MOVE R0, #5</b>                                               | R1 35  |
| <b>MOVE R1, #0x23</b>                                            | R0 5   |
| <b>LOOP</b>                                                      |        |
| <b>SUBS R1, R1, R0</b>                                           | R1 30  |
| <b>ADD R0, R0, #10</b>                                           | R0 15  |
| <b>BPL LOOP</b>                                                  | 30>=0? |
| <b>ADD R1, R1, #20</b>                                           | R1 15  |
| Indicare esadecimale il valore di R1 al termine dell'esecuzione. | R0 25  |
|                                                                  | 15>=0? |

R1 -10  
R0 35

-10>=0?

# Esercizi

Si consideri il seguente programma assembly:

```
MOVE R0, #5
MOVE R1, #0x23
LOOP
CMP R1, R0
BLT DONE
SUB R0, R1, R0
SUB R1, R1, #3
B LOOP
DONE
ADD R1, R1, #10
```

Indicare esadecimale il valore di R1 al termine dell'esecuzione.

# Esercizi

R1 35

R0 5

35<5?

R1 32

R0 30

32<30?

R1 29

R0 2

29<2?

R1 26

R0 27

26<27?

R1 36

R1 0x24

Si consideri il seguente programma assembly:

**MOVE R0, #5**

**MOVE R1, #0x23**

**LOOP**

**CMP R1, R0**

**BLT DONE**

**SUB R0, R1, R0**

**SUB R1, R1, #3**

**B LOOP**

**DONE**

**ADD R1, R1, #10**

Indicare esadecimale il valore di R1 al termine dell'esecuzione.

FP, SP

-8  
-12  
-16  
-20  
-24

i  
j  
f  
g  
h

FP

1

1

5

3

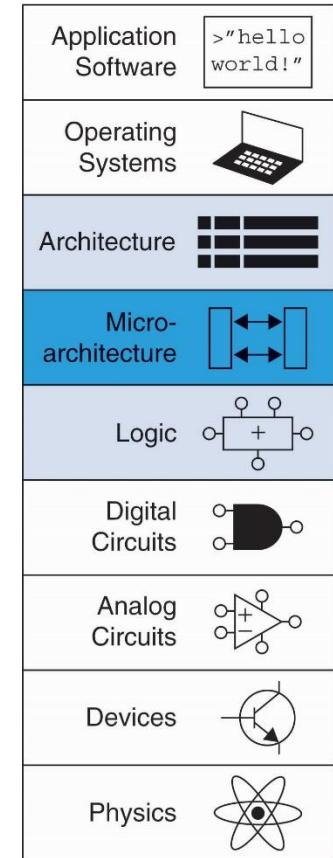
2

SP

# Micro-architettura

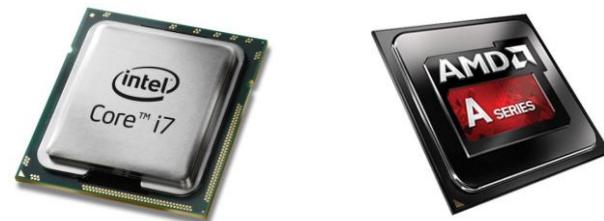
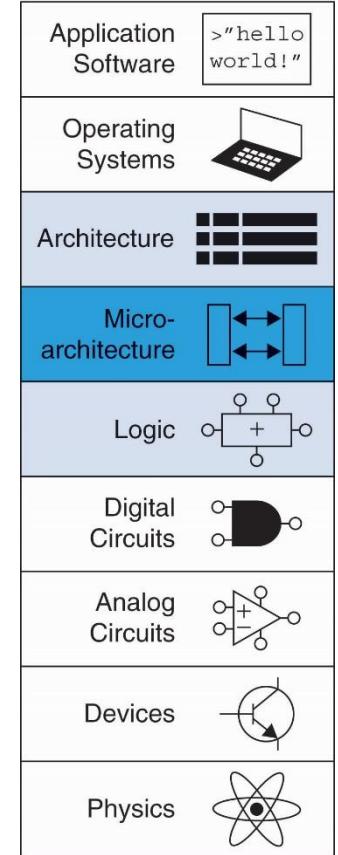
**Architettura:** rappresenta la struttura di un calcolatore dal punto di vista di un programmatore

- È definita dal set di istruzioni e operandi che costituisce il *linguaggio macchina*
- Al linguaggio macchina corrisponde un linguaggio *assembly*
  - **Assembly language:** formato human-readable delle istruzioni
  - **Machine language:** formato computer-readable (1 e 0)
- Esistono molte architetture differenti
  - **ARM**
  - **X86**
  - **MIPS**
  - **SPARC**
  - **PowerPC**



**Microarchitettura:** definisce la disposizione specifica di registri, ALU, macchine a stati finiti, le memorie e altri blocchi logici necessari per implementare una architettura

- Come due algoritmi di ordinamento (bubble e quick sort) realizzano la stessa funzione con procedure differenti, così due microarchitetture possono realizzare la medesima architettura ma soluzioni tecnologiche e prestazioni molto differenti
- Ad esempio Intel e AMD realizzano diversi modelli (microarchitetture) della medesima architettura x86.



# Architettura ARM

- Sviluppata negli anni 80 dalla Advanced RISC Machines – ora chiamata ARM Holdings
- Circa 10 miliardi di processor ARM venduti ogni anno
- Impiegata soprattutto nel settore degli smartphone e tablet, ma ormai il segmento laptop si sta orientando verso questa architettura
- Circa il 75% della popolazione umana usa prodotti equipaggiati da un processore ARM

# Design principles

L'architettura ARM segue quattro principi di progettazione fondamentali:

1. Regularity supports design simplicity
2. Make the common case fast
3. Smaller is faster
4. Good design demands good compromises

# Regularity supports simplicity

## C Code

a = b + c;

## ARM Assembly Code

ADD a, b, c

## C Code

a = b - c;

## ARM Assembly Code

SUB a, b, c

ADD: mnemonico – indica l'operazione da eseguire

b, c: operandi sorgente (source operands)

a: operando destinazione (destination operand)

## Regularity supports design simplicity

- Ogni istruzione è rappresentata da una parola di 32 bit (anche se alcune istruzioni richiederebbero meno di 32 bit di codifica, gestire istruzioni di lunghezza variabile aumenterebbe la complessità)
- I formati sono consistenti, nell'esempio precedente stesso numero di operandi (due sorgenti e una destinazione)
- *Questo facilita l'encoding in hardware*

# Make the common case fast

- L'architettura ARM comprende solo istruzioni semplici e di uso frequente
- L'hardware necessario per decodificare ed eseguire le istruzioni è così *semplice* e veloce
- Istruzioni complesse (che sono meno comuni) corrispondono a più istruzioni ARM

## C Code

```
a = b + c - d;
```

## ARM assembly code

```
ADD t, b, c ; t = b + c  
SUB a, t, d ; a = t - d
```

- ARM è un **Reduced Instruction Set Computer (RISC)**: poche istruzioni e semplici
- X86 è un **Complex Instruction Set Computers (CISC)**: molte istruzioni complesse

# Smaller is faster

I dati sono memorizzati:

- Direttamente all'interno di una istruzione.
  - Costanti (anche dette *immediates*)
- In registri
  - ARM ha solo 16 registri
  - I registri sono più veloci delle memorie di ordine superiore
  - Ci focalizzeremo su di una architettura in cui ogni registro è costituito da 32 bits
- In memoria
  - Più lenta dei registri ma più capiente
  - A sua volta suddivisa in diversi livelli: cache, centrale, di massa
  - le istruzioni ARM operano esclusivamente sui registri, quindi i dati residenti nella memoria devono essere spostati in un registro prima di poter essere elaborati.

# Good design demands good compromises

Al fine di massimizzare la semplicità sarebbe preferibile anche avere un unico formato per le istruzioni. Tuttavia, questa scelta risulterebbe troppo restrittiva.

Il compromesso scelto da ARM (esempio del principio 4) è di avere **tre** possibili **formati di istruzione**:

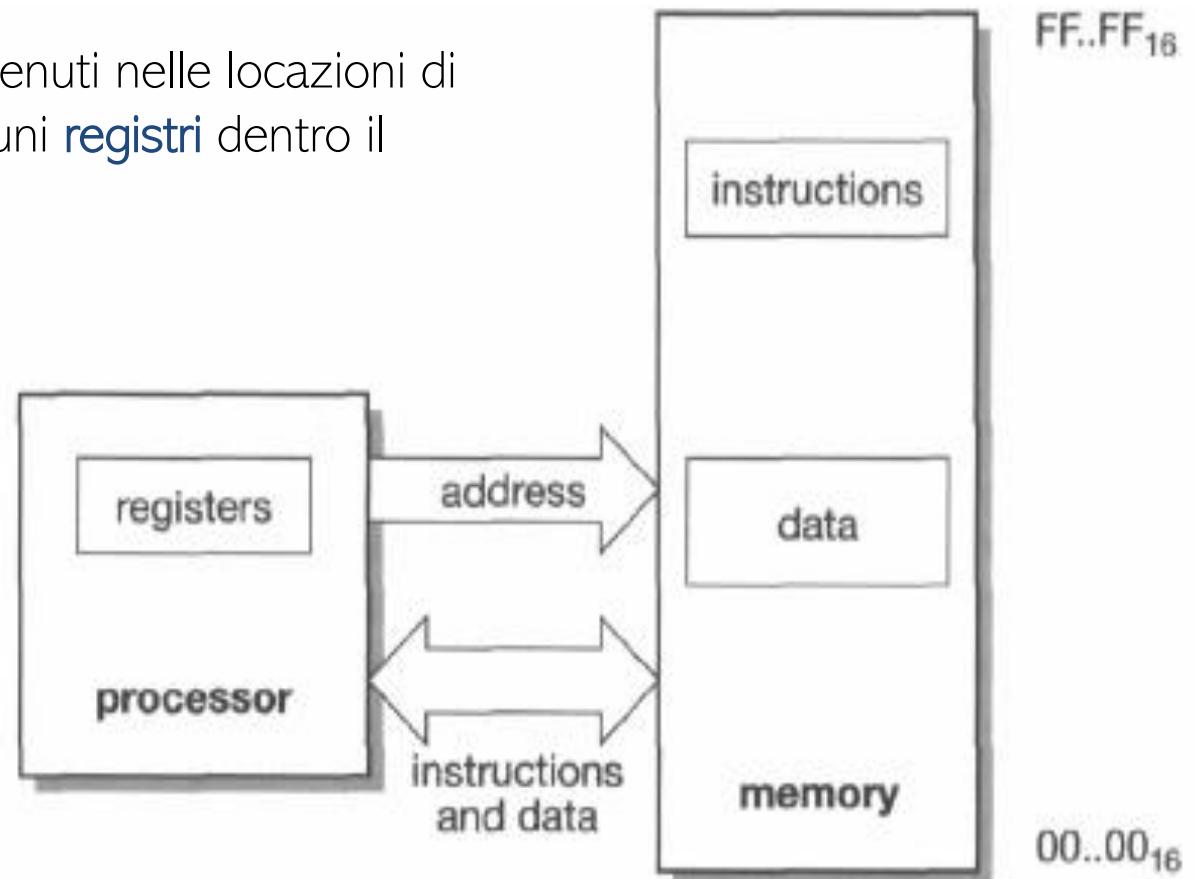
- ▶ Data-processing;
- ▶ Memory;
- ▶ Branch.

# Cos'è un processore

Un processore general-purpose è un **automa** a stati finiti, che esegue **istruzioni** rilocate in una memoria.

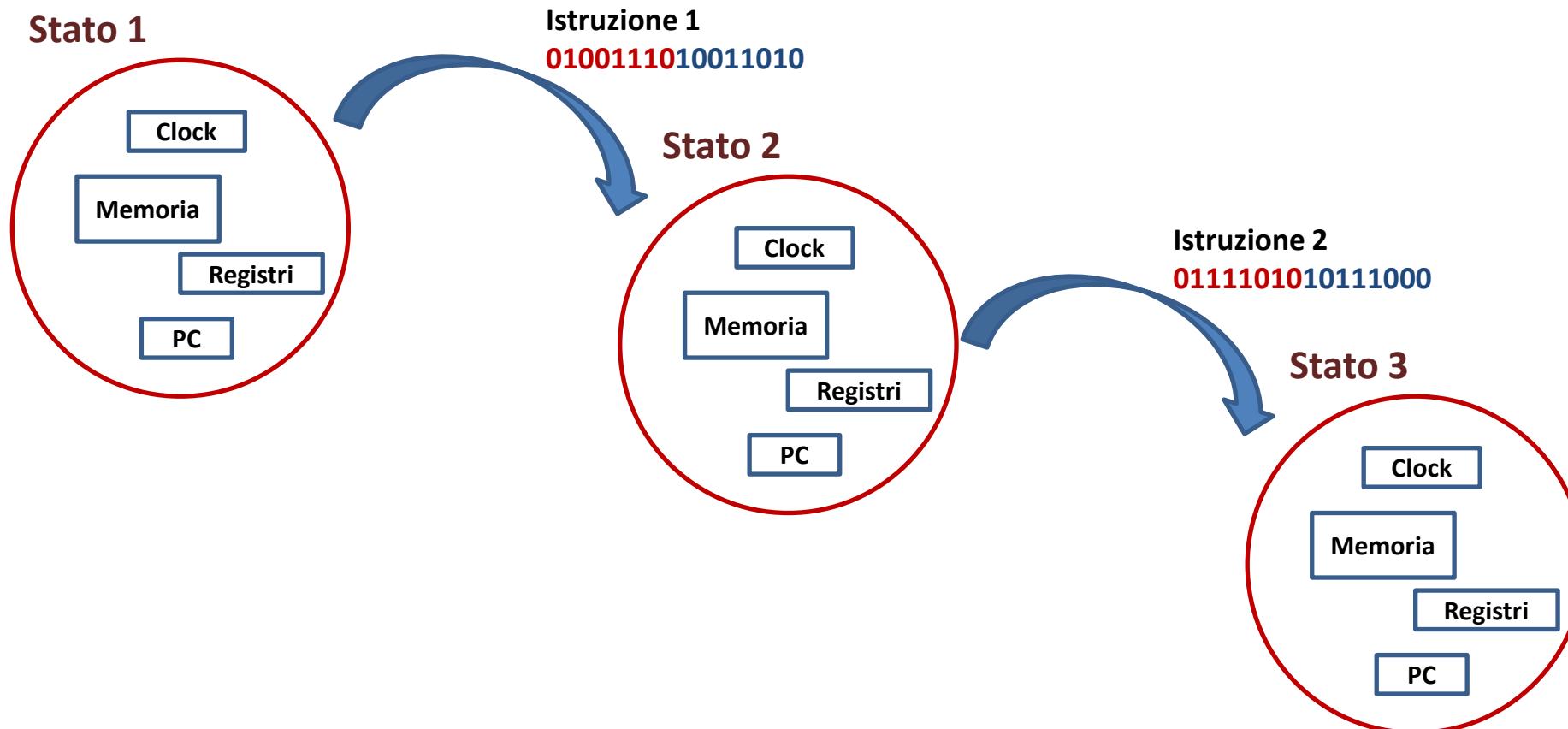
Lo **stato** del sistema è definito dai valori contenuti nelle locazioni di **memoria** insieme con i valori contenuti in alcuni **registri** dentro il processore stesso.

Ogni **istruzione** definisce in che modo lo stato deve cambiare e quale istruzione deve essere eseguita successivamente.



# Microarchitettture come automi

Una microarchitettura può essere vista come un automa.



# Variazioni di stato

- I registri, la memoria dati e la memoria istruzioni operano mediante logica combinatoria.
- Tutti i componenti effettuano la scrittura sul fronte alto del clock, cosicché lo stato del sistema cambia solo su un fronte del clock.
- Gli indirizzi, i dati ed il segnale di write enable devono essere impostati prima del fronte del clock e mantenuti immutati per un tempo superiore al ritardo di propagazione.
- Sia gli elementi di stato, che il microprocessore sono costituiti da logica combinatoria e da componenti sincronizzate dal clock. L'intero sistema è, quindi, sincrono e può essere visto come una complessa macchina a stati finiti o come l'insieme di macchine a stati finiti semplici, che interagiscono fra loro.

# Microarchitetture a ciclo singolo

Saranno prese in considerazione tre diverse microarchitetture ARM:

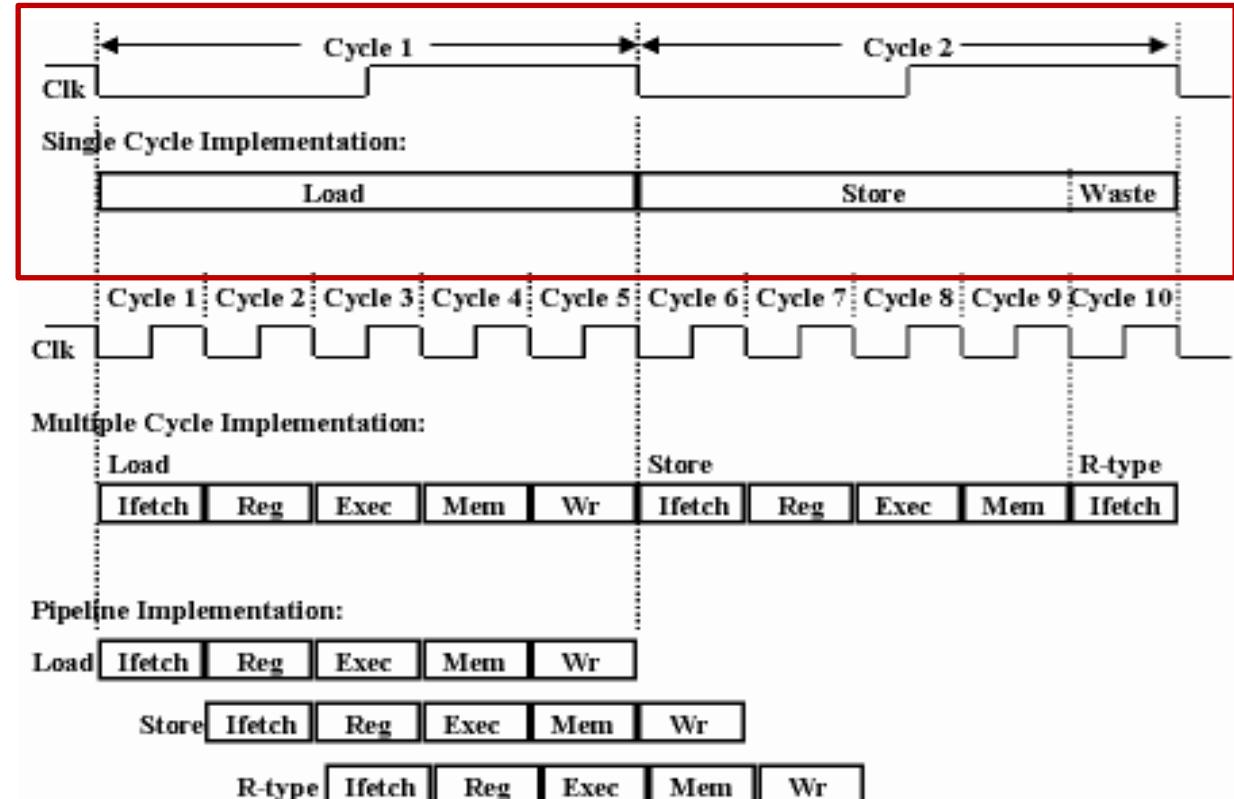
**Ciclo singolo:** l'intera istruzione è eseguita in un singolo ciclo.

Vantaggi:

- ▶ semplice da comprendere;
- ▶ unità di controllo molto semplice;
- ▶ non richiede stati non architetturali.

Svantaggi:

- ▶ tempo pari a quello dell'istruzione più lenta;
- ▶ memoria dati e memoria istruzioni separate;



# Microarchitettture a ciclo multiplo

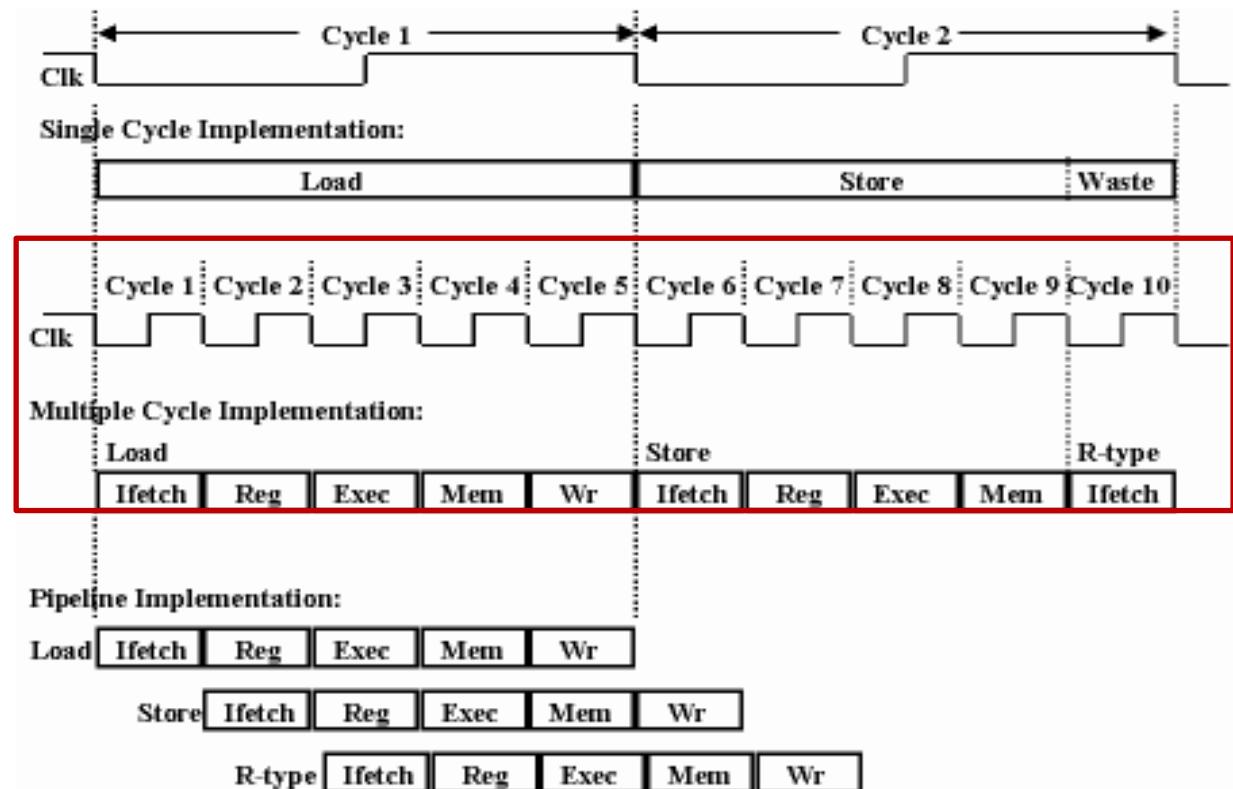
**Ciclo multiplo:** esegue una istruzione in più cicli brevi.

Vantaggi:

- riuso dei componenti;
- durata variabile delle istruzioni;
- non richiede la separazione delle memorie.

Svantaggi:

- richiede stati non architetturali;
- esegue una istruzione per volta.



# Microarchitture con pipeline

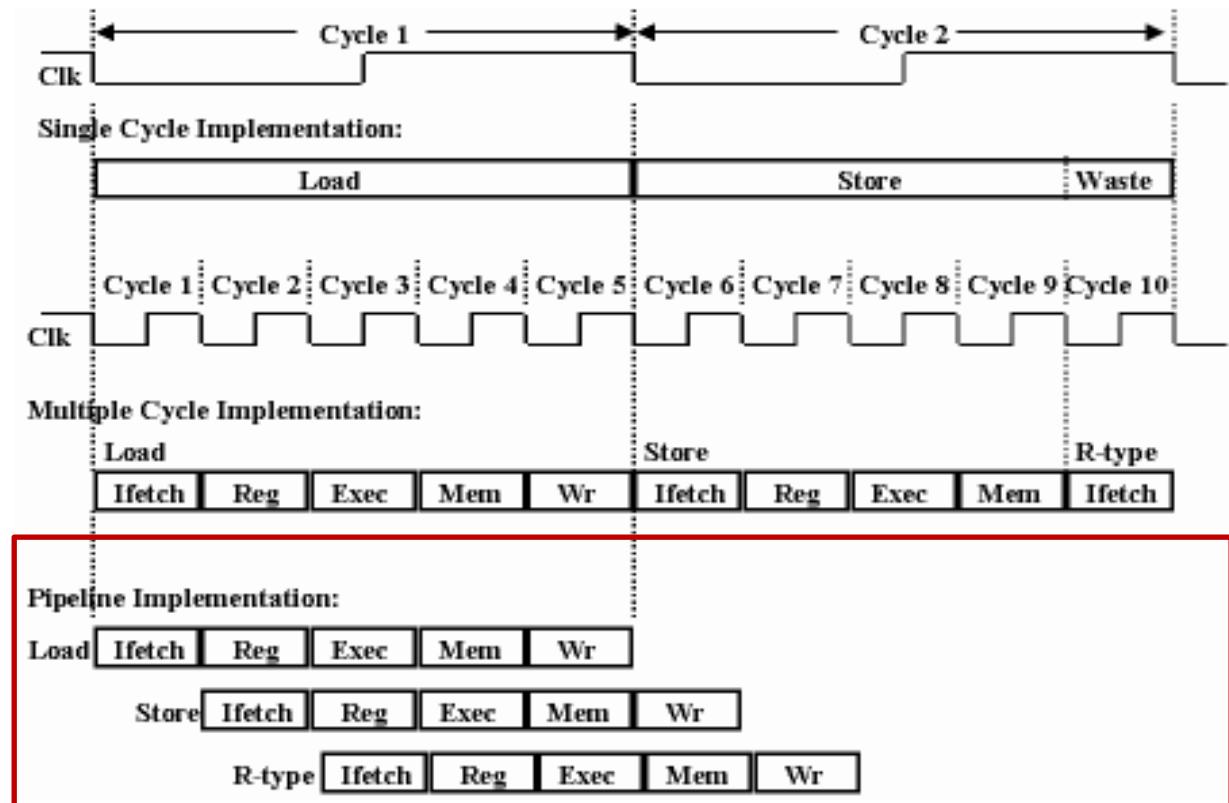
**Pipelining:** si applica al processore a ciclo singolo e ne migliora le performance.

Vantaggi:

- ▶ esegue più istruzioni contemporaneamente (parallelismo temporale);
- ▶ si può accedere a dati e registri contemporaneamente.

Svantaggi:

- ▶ logica di controllo più complessa.
- ▶ richiede registri di pipeline.



# Misura delle prestazioni

Ci sono molti modi per misurare le performance di un processore.

Una misura affidabile consiste nel valutare le prestazioni di tempo rispetto all'esecuzione di un insieme fissato di programmi, che prende il nome di benchmark.

## CINT2006 (Integer Component of SPEC CPU2006):

| Benchmark                      | Language | Application Area               | Brief Description                                                                                                                                   |
|--------------------------------|----------|--------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------|
| <a href="#">400.perlbench</a>  | C        | Programming Language           | Derived from Perl V5.8.7. The workload includes SpamAssassin, MHonArc (an email indexer), and specdiff (SPEC's tool that checks benchmark outputs). |
| <a href="#">401.bzip2</a>      | C        | Compression                    | Julian Seward's bzip2 version 1.0.3, modified to do most work in memory, rather than doing I/O.                                                     |
| <a href="#">403.gcc</a>        | C        | C Compiler                     | Based on gcc Version 3.2, generates code for Opteron.                                                                                               |
| <a href="#">429.mcf</a>        | C        | Combinatorial Optimization     | Vehicle scheduling. Uses a network simplex algorithm (which is also used in commercial products) to schedule public transport.                      |
| <a href="#">445.gobmk</a>      | C        | Artificial Intelligence: Go    | Plays the game of Go, a simply described but deeply complex game.                                                                                   |
| <a href="#">456.hmmer</a>      | C        | Search Gene Sequence           | Protein sequence analysis using profile hidden Markov models (profile HMMs)                                                                         |
| <a href="#">458.sjeng</a>      | C        | Artificial Intelligence: chess | A highly-ranked chess program that also plays several chess variants.                                                                               |
| <a href="#">462.libquantum</a> | C        | Physics / Quantum Computing    | Simulates a quantum computer, running Shor's polynomial-time factorization algorithm.                                                               |
| <a href="#">464.h264ref</a>    | C        | Video Compression              | A reference implementation of H.264/AVC, encodes a videotream using 2 parameter sets. The H.264/AVC standard is expected to replace MPEG2           |
| <a href="#">471.omnetpp</a>    | C++      | Discrete Event Simulation      | Uses the OMNet++ discrete event simulator to model a large Ethernet campus network.                                                                 |
| <a href="#">473.astar</a>      | C++      | Path-finding Algorithms        | Pathfinding library for 2D maps, including the well known A* algorithm.                                                                             |
| <a href="#">483.xalancbmk</a>  | C++      | XML Processing                 | A modified version of Xalan-C++, which transforms XML documents to other document types.                                                            |

# Misura delle prestazioni

Il tempo di esecuzione è calcolato come:

$$\text{Tempo di esecuzione} = (\# \text{istruzioni}) \left( \frac{\text{cicli}}{\text{istruzione}} \right) \left( \frac{\text{secondi}}{\text{ciclo}} \right)$$

CPI: Cycles/instruction

clock period: seconds/cycle

IPC: instructions/cycle = IPC

# Misura delle prestazioni

Il tempo di esecuzione è calcolato come:

$$\text{Tempo di esecuzione} = (\# \text{istruzioni}) \left( \frac{\text{cicli}}{\text{istruzione}} \right) \left( \frac{\text{secondi}}{\text{ciclo}} \right)$$

CPI: Cycles/instruction

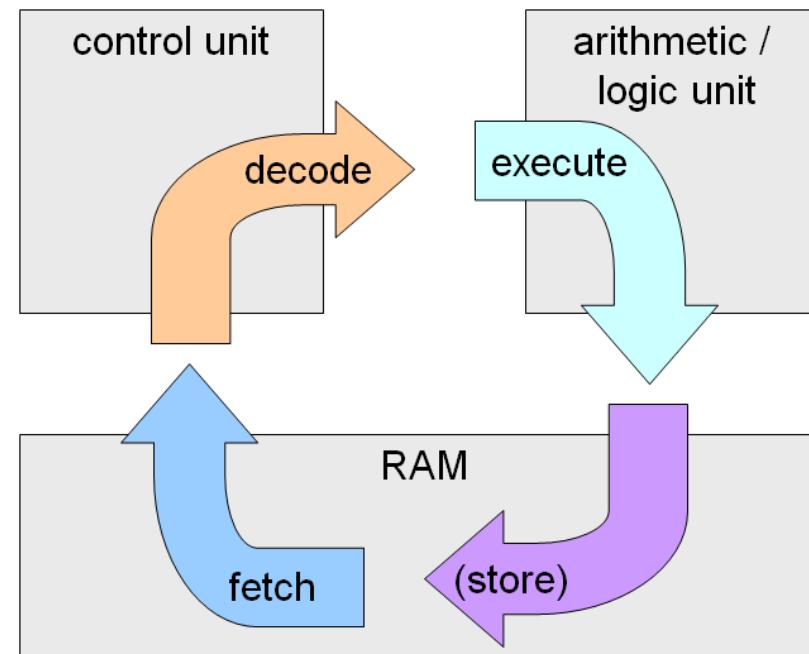
clock period: seconds/cycle

IPC: instructions/cycle = IPC

La frequenza di clock corrisponde  
all'inverso del clock period

# Microarchitettura: schema generale

Una istruzione ha un ciclo di vita che consta di quattro fasi principali



# Progettazione

Una microarchitettura consta di due componenti che interagiscono fra loro:

► il **datapath**:

- determina il flusso dei dati durante l'esecuzione di una istruzione
- opera su dati e contiene strutture quali le memorie, i registri, l'ALU, e i multiplexer
- considereremo un datapath a 32 bit.

► l'**unità di controllo**:

- riceve l'istruzione corrente dal datapath ed "istruisce" il datapath su come eseguire tale istruzione.
- Produce i valori di selezione dei multiplexer, i segnali di abilitazione alla scrittura dei registri e della memoria.

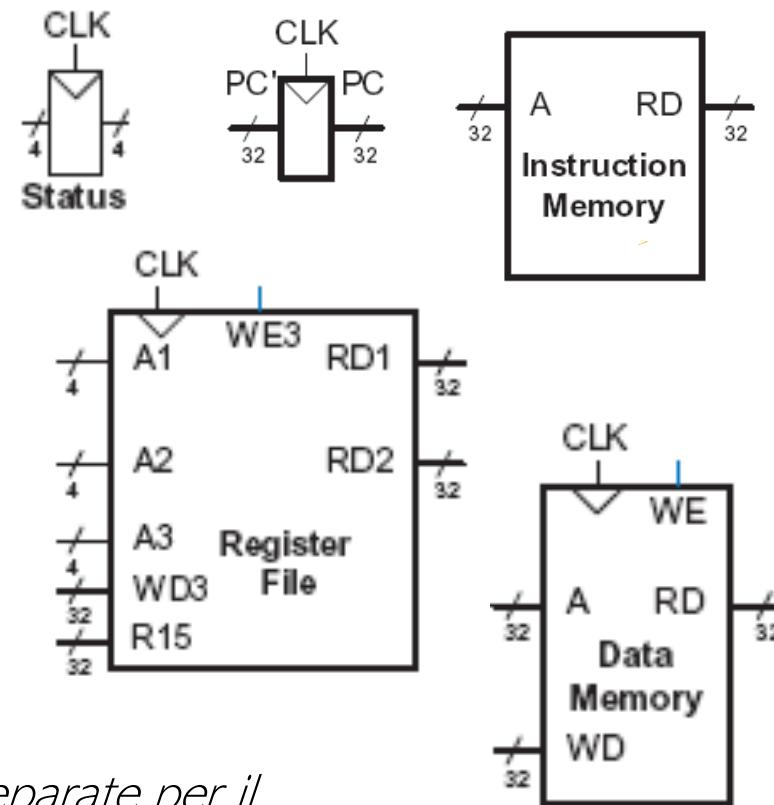
Procederemo in modo incrementale: Elementi di stato (program counter, registri, e registro di stato). Logica combinatoria fra gli elementi di stato. Gestione della memoria.

# Progettazione

Il miglior modo di procedere nel processo di progettazione consiste nel considerare prima gli elementi di stato (program counter, registri, registro di stato) e aggiungere successivamente la logica combinatoria.

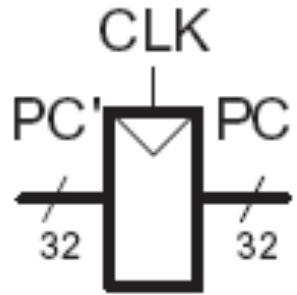
Suddivideremo la memoria in cinque elementi di stato:

- ▶ il program counter
- ▶ i registri (register file)
- ▶ il registro di stato (status register)
- ▶ la memoria istruzioni (instruction memory)
- ▶ la memoria dati (data memory).

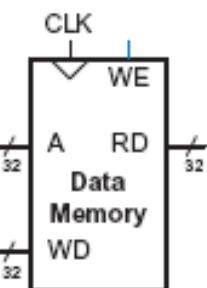
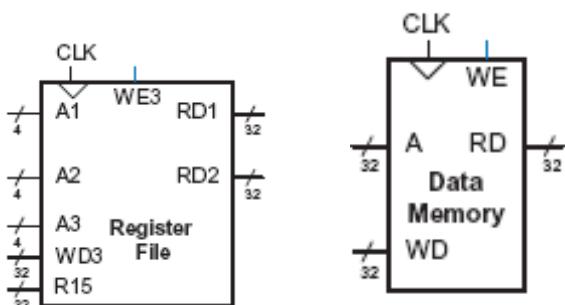
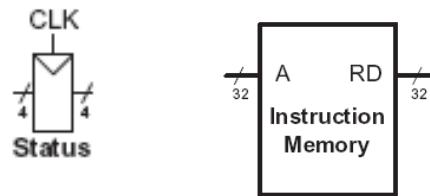


*La memoria per le istruzioni e quella per i dati sono separate per il solo scopo di facilitare la comprensione.*

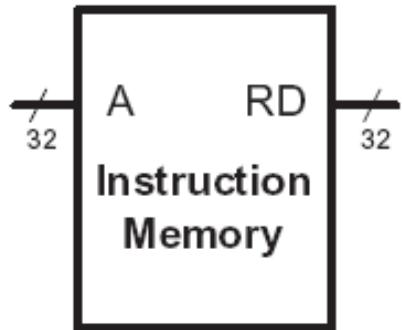
# Program counter



Sebbene il PC sia logicamente parte del file registro, esso viene letto e scritto ad ogni ciclo indipendentemente dagli altri registri ed è quindi implementato come un registro autonomo 32-bit. La sua uscita, PC, indica l'indirizzo dell'istruzione corrente. Il suo ingresso, PC', indica l'indirizzo della successiva istruzione.



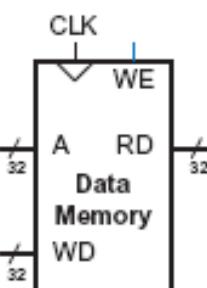
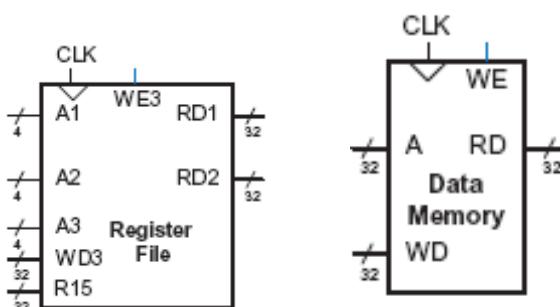
# Instruction Memory



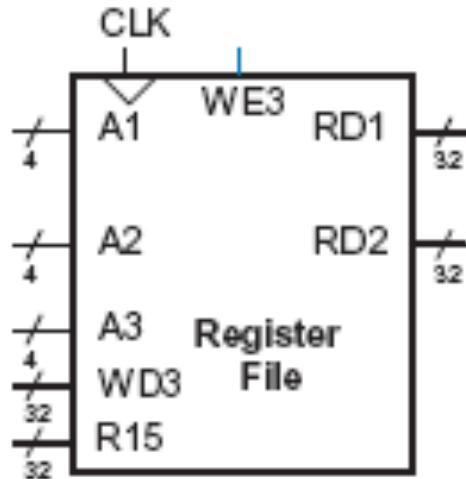
essa ha una singola porta di sola lettura (*semplificazione*).

**A** indica l'indirizzo di una istruzione che verrà riportata (*letta*) nel registro **RD**.

*Considerando che in genere le memorie sono byte addressable ed essendo sia A che RD a 32 bit, qual'è il numero massimo di istruzioni che una architettura 32 bit supporta?*

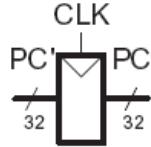
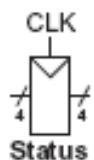


# File register



consiste di 16 registri (R0,...,R15) a 32 bit

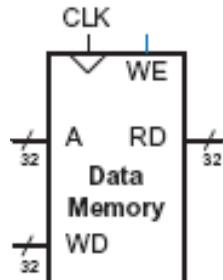
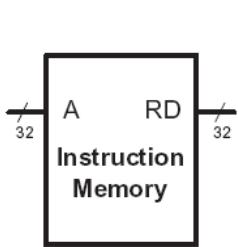
*Dal punto di vista logico i registri non sono equivalenti, svolgono al livello architetturale funzioni diverse. In particolare:*



R13: stack pointer

R14: link register

R15: proveniente dal program counter



# File register

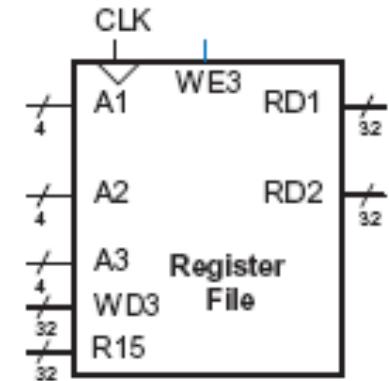
Esso ha due porte di indirizzamento per la lettura **A1** e **A2** ed una per la scrittura **A3**.

Ciascuna porta di lettura ha un input di 4 bit ( $2^4=16$ ), che specificano uno dei registri come operando, che viene letto nei registri **RD1** o **RD2**, rispettivamente.

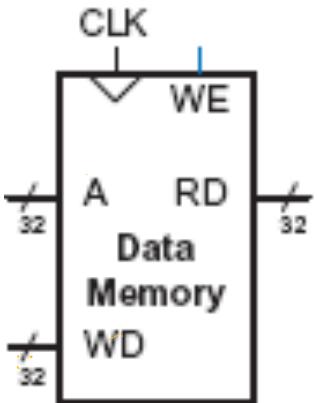
La porta di scrittura ha:

- ▶ un indirizzo di 4 bit **A3** (dove scrivere);
- ▶ un elemento di 32 bit **WD3** (cosa scrivere);
- ▶ un segnale di Write Enable (**WE3**).

Se **WE** è 1, il dato **WD3** è scritto nel registro specificato da **A3**.



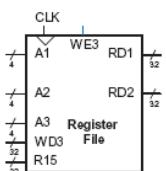
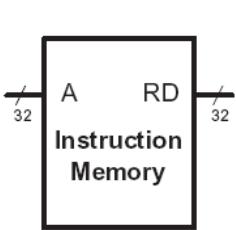
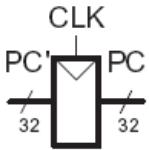
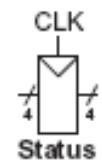
# Data Memory



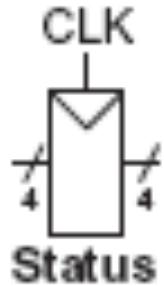
Il **Data Memory**: ha una singola porta di lettura/scrittura.

Quando il segnale **WE** (Write Enable) è attivo, essa scrive il dato **WD** nella cella puntata dall'indirizzo **A** durante il fronte alto del clock.

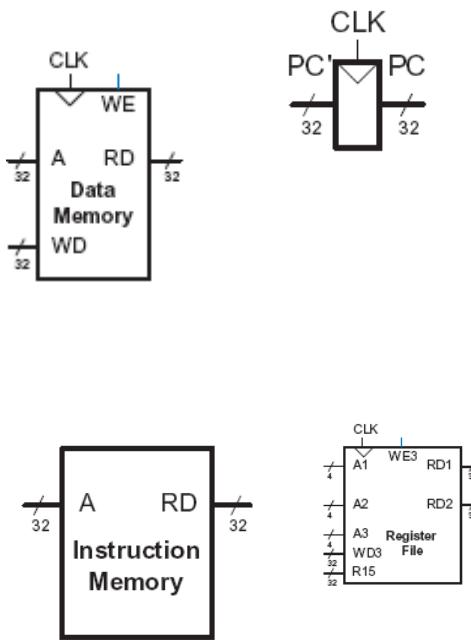
Quando il segnale **WE** (Write Enable) è 0, essa legge i dati nella cella puntata dall'indirizzo **A** durante il fronte alto del clock e li pone nel registro **RD**.



# Status



Memorizza i flags **N,Z,C,V** fornite dall'ALU al fine di eseguire istruzioni condizionate



| Flag     | Name            | Description                              |
|----------|-----------------|------------------------------------------|
| <i>N</i> | <b>Negative</b> | Instruction result is negative           |
| <i>Z</i> | <b>Zero</b>     | Instruction results in zero              |
| <i>C</i> | <b>Carry</b>    | Instruction causes an unsigned carry out |
| <i>V</i> | <b>oVerflow</b> | Instruction causes an overflow           |

# Istruzioni di base

Al fine di facilitare la comprensione dell'architettura di un processore ARM, considereremo un limitato **set** di istruzioni:

- Istruzioni di Data-processing:
  - **ADD, SUB, AND, ORR**
  - Con registri e modalità di indirizzamento diretto, ma senza shifts
- Istruzioni di memoria:
  - **LDR, STR**
  - con positive immediate offset
- Istruzioni di salto:
  - **B**

# Indirizzi di memoria

- Ad ogni **byte** corrisponde un unico indirizzo
  - Una parola è di 32-bit = 4 bytes, quindi gli indirizzi delle parole incrementano di 4 in 4

| Byte address |    |    |    | Word address |
|--------------|----|----|----|--------------|
|              |    |    |    | •<br>•<br>•  |
| 13           | 12 | 11 | 10 | 00000010     |
| F            | E  | D  | C  | 0000000C     |
| B            | A  | 9  | 8  | 00000008     |
| 7            | 6  | 5  | 4  | 00000004     |
| 3            | 2  | 1  | 0  | 00000000     |
| MSB          |    |    |    | LSB          |

- Indirizzo delle seconda parola =  $2 \times 4 = 8$
  - Indirizzo della decima parola =  $10 \times 4 = 40 = 0x00000028$

# Istruzioni di memoria: load

Una istruzione che legge in memoria è detta *load*

- Mnemonico: LDR (*load register*)
- Vi sono differenti modi per eseguire un load, ad esempio:

LDR R0, [R1, #12]

indirizzo:

- *base address + offset*
- indirizzo = (R1 + 12)

risultato:

- Il dato memorizzato all'indirizzo (R1 + 12) viene *caricato* nel registro R0

# Istruzioni di memoria: load

Esempio: Carica nel registro R3 la parola all'indirizzo 8

```
MOV R2, #0  
LDR R3, [R2, #8]
```

$$\text{indirizzo} = (R2 + 8) = 8$$

R3 = 0x01EE2842 (dopo il load)

| Word address | Data |     |     |     | Word number |
|--------------|------|-----|-----|-----|-------------|
| ...          | ...  | ... | ... | ... | ...         |
| 00000010     | C D  | 1 9 | A 6 | 5 B | Word 4      |
| 0000000C     | 4 0  | F 3 | 0 7 | 8 8 | Word 3      |
| 00000008     | 0 1  | E E | 2 8 | 4 2 | Word 2      |
| 00000004     | F 2  | F 1 | A C | 0 7 | Word 1      |
| 00000000     | A B  | C D | E F | 7 8 | Word 0      |

(b) Width = 4 bytes

# Istruzioni di memoria: load

Vi sono anche altre segnature di LDR, ad esempio:

LDR R0, [R1, R2]      LDR R0, [R1, R2, LSL #2]

$R0 \leftarrow \text{mem32}[R1 + R2]$        $R0 \leftarrow \text{mem32}[R1 + (R2 * 4)]$

# Istruzioni di memoria: store

Un'istruzione che scrive in memoria è detta *store*

- Mnemonico: STR (*store register*)
- Semantica speculare a LDR:

STR R0, [R1, #12]

mem32[R1 + 12]  $\leftarrow$  R0

- Segniture analoghe

STR R1, [R0, R2]

STR R0, [R1, R2, LSL #1]

mem32[R0 + R2]  $\leftarrow$  R1

mem32[R1 + (R2\*2)]  $\leftarrow$  R0

# Istruzioni di memoria: store

Esempio: Memorizza il valore di R7 nella 21esima parola di memoria.

Indirizzo di memoria =  $4 \times 21 = 84 = 0x54$

MOV R5, #0

STR R7, [R5, #0x54]

L'offset può essere scritto in decimale o esadecimale

| Word address | Data            | Word number |
|--------------|-----------------|-------------|
| 00000010     | C D 1 9 A 6 5 B | Word 4      |
| 0000000C     | 4 0 F 3 0 7 8 8 | Word 3      |
| 00000008     | 0 1 E E 2 8 4 2 | Word 2      |
| 00000004     | F 2 F 1 A C 0 7 | Word 1      |
| 00000000     | A B C D E F 7 8 | Word 0      |

Width = 4 bytes

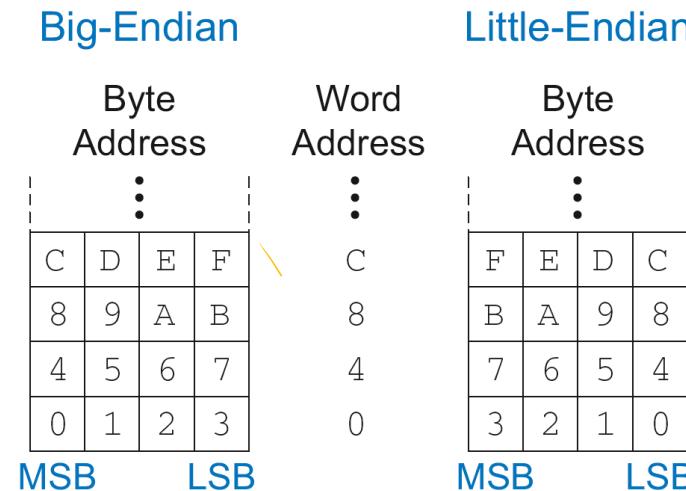
# Varianti: LDRB, STRB

LDRB/STRB eseguono rispettivamente il load e lo store di un singolo byte

Come sono indirizzati i byte all'interno di una parola?

**Little-endian:** la numerazione dei byte inizia dal byte meno significativo

**Big-endian:** la numerazione dei byte inizia dal byte più significativo



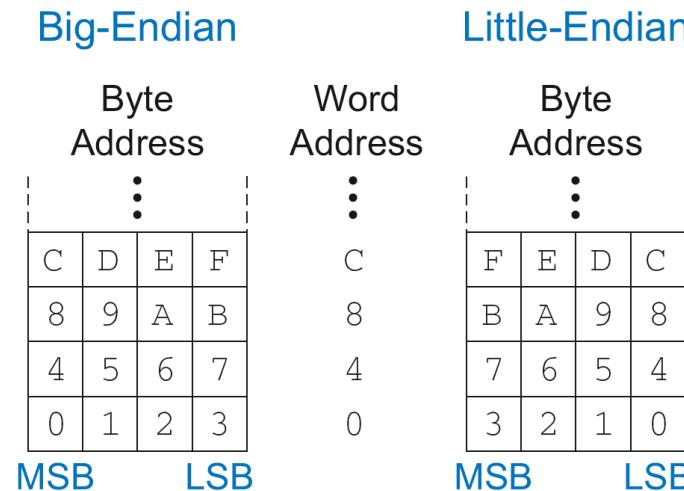
# Varianti: LDRB, STRB

LDRB/STRB eseguono rispettivamente il load e lo store di un singolo byte

Come sono indirizzati i byte all'interno di una parola?

**Little-endian:** la numerazione dei byte inizia dal byte meno significativo

**Big-endian:** la numerazione dei byte inizia dal byte più significativo



**IBM PowerPC    Intel x86**

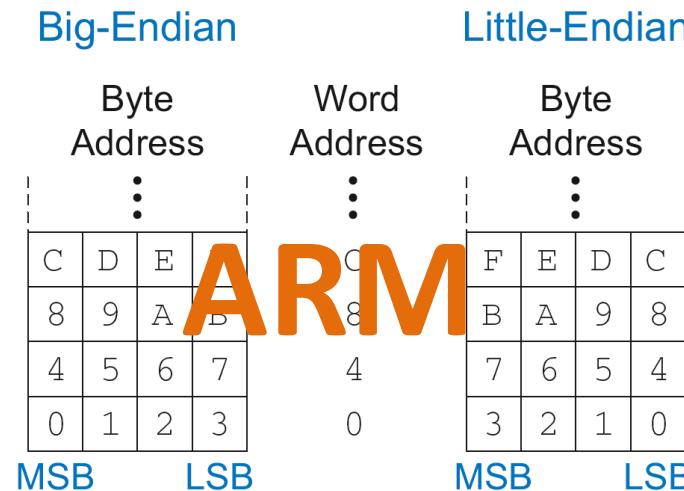
# Varianti: LDRB, STRB

LDRB/STRB eseguono rispettivamente il load e lo store di un singolo byte

Come sono indirizzati i byte all'interno di una parola?

**Little-endian:** la numerazione dei byte inizia dal byte meno significativo

**Big-endian:** la numerazione dei byte inizia dal byte più significativo



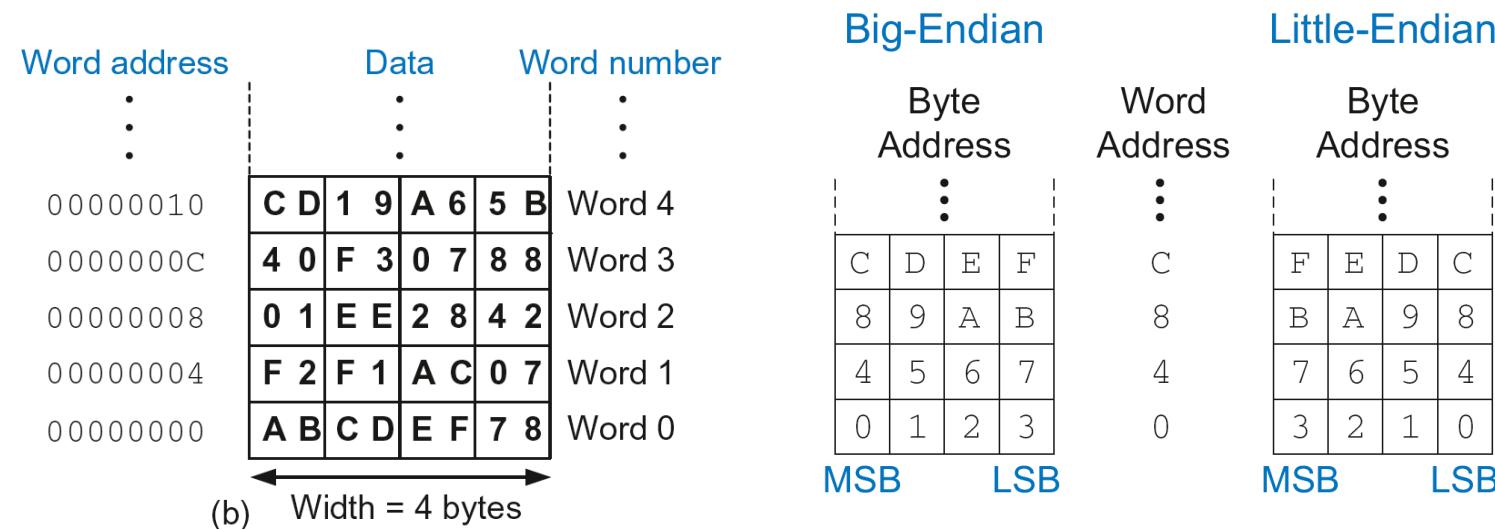
**IBM PowerPC    Intel x86**

# Varianti: LDRB, STRB

Esempio: si assuma che R2 vale 0 ( 0x00000000) e che si esegua l'istruzione

LDRB R7, [R2, #9]

Quale sarà il valore di R7 in un sistema little-endian/ big-endian?



**Big-endian:**

0x000000EE

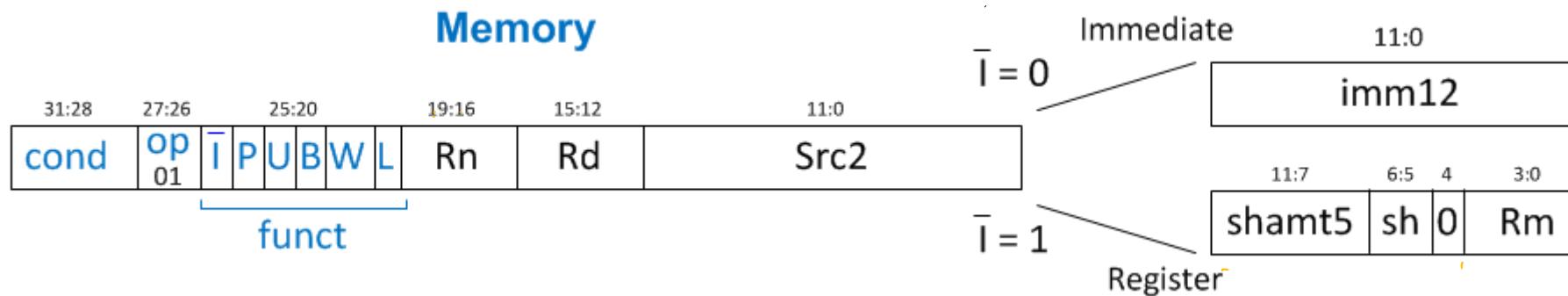
**Little-endian:**

0x00000028

# Codifiche istruzioni di memoria

## LDR, STR, LDRB, STRB

- $op = 01_2$
- $Rn$  = registro *base address*
- $Rd$  = destinazione (load), sorgente (store)
- $Src2$  = offset: registro (*shiftato* opzionalmente) o immediate
- $funct$  = 6 bit di controllo



# Indicizzazione

ARM fornisce tre tipi di indicizzazione:

## Offset

L'indirizzo è calcolato sommando o sottraendo un offset al contenuto del registro di base.

## Preindex

L'indirizzo viene calcolato come nel caso dell'offset e viene scritto nel registro di base;

## Postindex

L'indirizzo corrisponde al contenuto del registro di base;

L'offset viene aggiunto o sottratto al contenuto del registro di base e riscritto nel registro di base

**Table 6.4 ARM indexing modes**

| Mode       | ARM Assembly      | Address | Base Register |
|------------|-------------------|---------|---------------|
| Offset     | LDR R0, [R1, R2]  | R1 + R2 | Unchanged     |
| Pre-index  | LDR R0, [R1, R2]! | R1 + R2 | R1 = R1 + R2  |
| Post-index | LDR R0, [R1], R2  | R1      | R1 = R1 + R2  |

# Indicizzazione

ARM fornisce tre tipi di indicizzazione:

## Offset

L'indirizzo è calcolato sommando o sottraendo un offset al contenuto del registro di base.

## Preindex

L'indirizzo viene calcolato come nel caso dell'offset e viene scritto nel registro di base;

## Postindex

L'indirizzo corrisponde al contenuto del registro di base;

L'offset viene aggiunto o sottratto al contenuto del registro di base e riscritto nel registro di base

## Esempi

Offset: LDR R1, [R2, #4] ; R1 = mem32[R2+4]

Preindex: LDR R3, [R5, #16] ! ; R3 = mem32[R5+16]  
; R5 = R5 + 16

Postindex: LDR R8, [R1], #8 ; R8 = mem32[R1]  
; R1 = R1 + 8

# Campi di funzione

Type of Operation

| L | B | Instruction |
|---|---|-------------|
| 0 | 0 | STR         |
| 0 | 1 | STRB        |
| 1 | 0 | LDR         |
| 1 | 1 | LDRB        |

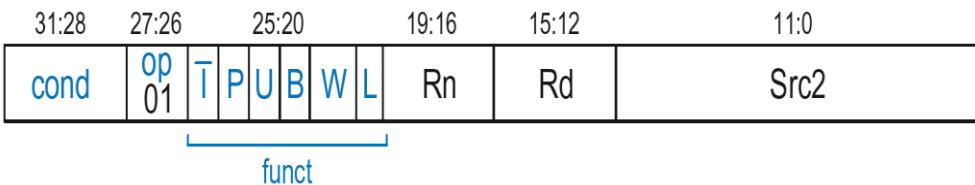
Indexing Mode

| P | W | Indexing Mode |
|---|---|---------------|
| 0 | 1 | Not supported |
| 0 | 0 | Postindex     |
| 1 | 0 | Offset        |
| 1 | 1 | Preindex      |

Add/Subtract Immediate/Register Offset

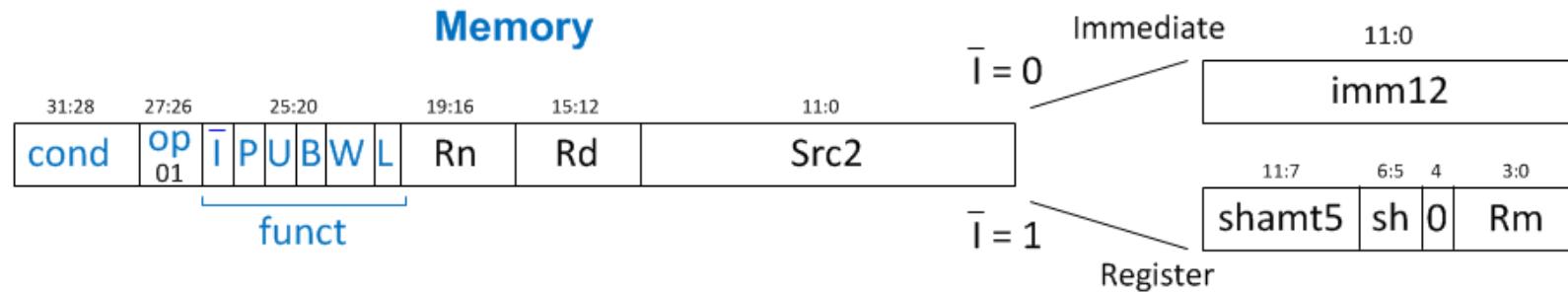
| Value | T                        | U                         |
|-------|--------------------------|---------------------------|
| 0     | Immediate offset in Src2 | Subtract offset from base |
| 1     | Register offset in Src2  | Add offset to base        |

Memory



# Campo sh

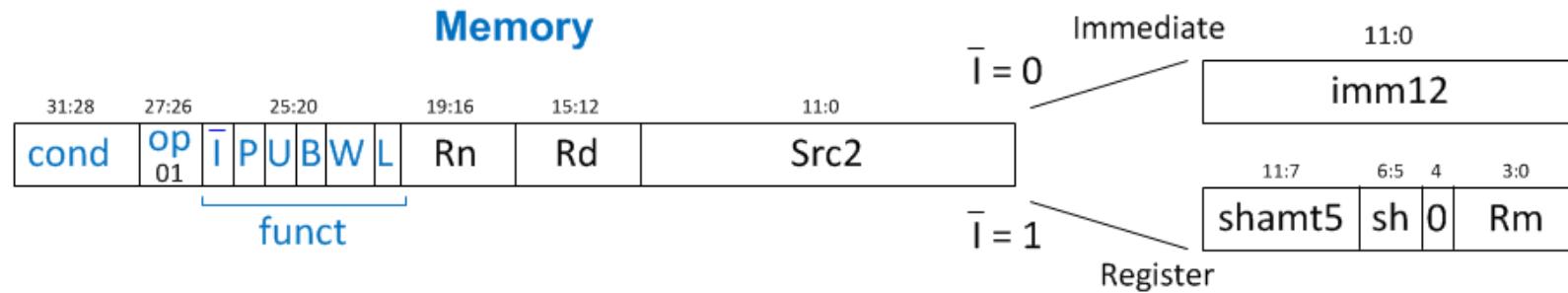
Sh codifica il tipo di shift (i.e., >>, <<, >>>, ROR)



| Shift Type | sh     |
|------------|--------|
| LSL        | $00_2$ |
| LSR        | $01_2$ |
| ASR        | $10_2$ |
| ROR        | $11_2$ |

# Campo sh

Sh codifica il tipo di shift (i.e., >>, <<, >>>, ROR)



| Shift Type | sh     |
|------------|--------|
| LSL        | $00_2$ |
| LSR        | $01_2$ |
| ASR        | $10_2$ |
| ROR        | $11_2$ |

Perché non ci sono ASL e ROL?

# Esempi

STR R11, [R5], #-26

- Operation:  $\text{mem}[R5] \leq R11; R5 = R5 - 26$
- $\text{cond} = 1110_2$  (14) for unconditional execution
- $\text{op} = 01_2$  (1) for memory instruction
- $\text{funct} = 0000000_2$  (0)
- $I = 0$  (immediate offset),  $P = 0$  (postindex),
- $U = 0$  (subtract),  $B = 0$  (store word),  $W = 0$  (postindex),
- $L = 0$  (store)
- $Rd = 11$ ,  $Rn = 5$ ,  $imm12 = 26$

Field Values

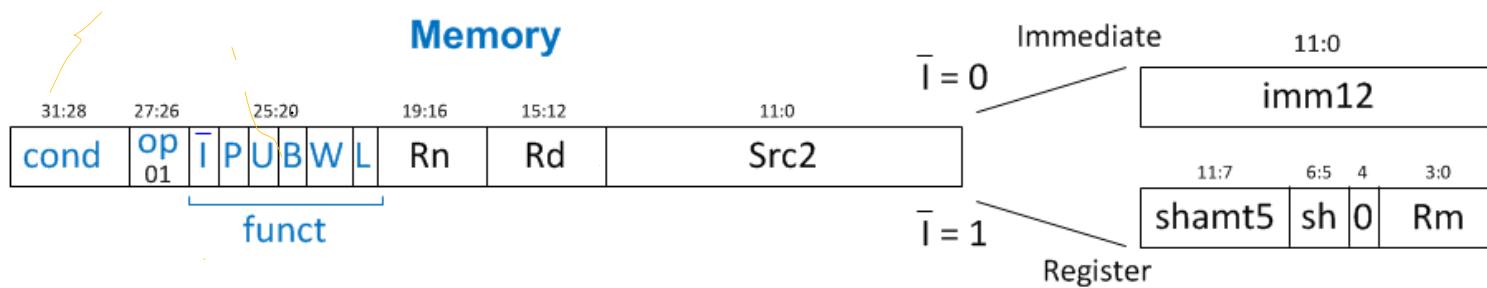
|      | 31:28    | 27:26  | 25:20       | 19:16 | 15:12 | 11:0           |
|------|----------|--------|-------------|-------|-------|----------------|
| cond | $1110_2$ | $01_2$ | $0000000_2$ | 5     | 11    | 26             |
| op   |          |        |             | Rn    | Rd    | imm12          |
| E    | 1110     | 01     | 000000      | 0101  | 1011  | 0000 0001 1010 |

# Esempi

LDR R3, [R4, R5]

- Operation:  $R3 \leftarrow \text{mem}[R4 + R5]$
- $\text{cond} = 1110_2$  (14) for unconditional execution
- $\text{op} = 01_2$  (1) for memory instruction
- $\text{funct} = 111001_2$  (57)
- $I = 1$  (register offset),  $P = 1$  (offset indexing),
- $U = 1$  (add),  $B = 0$  (load word),  $W = 0$  (offset indexing),
- $L = 1$  (load)
- $Rd = 3$ ,  $Rn = 4$ ,  $Rm = 5$  ( $\text{shamt5} = 0$ ,  $sh = 00$ )

1110 01 111001 0100 0011 00000 00 0 0101 = 0xE7943005

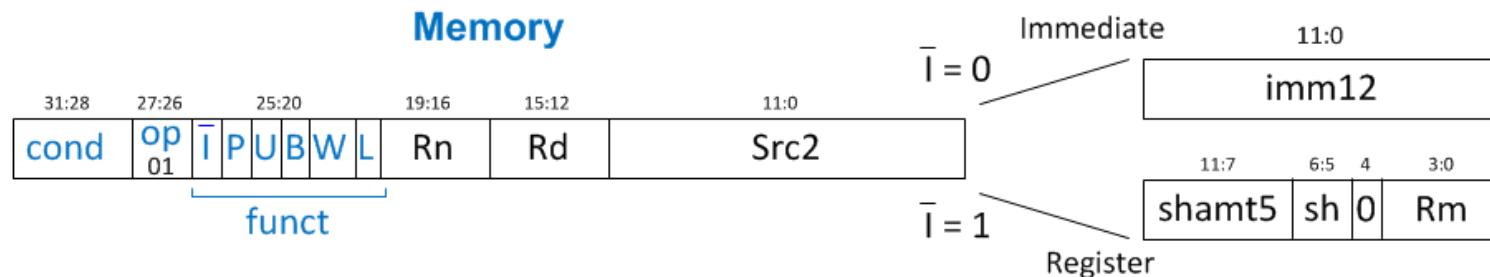


# Esempi

STR R9, [R1, R3, LSL #2]

- Operation:  $\text{mem}[\text{R1} + (\text{R3} \ll 2)] \leq \text{R9}$
- $\text{cond} = 1110_2$  (14) for unconditional execution
- $\text{op} = 01_2$  (1) for memory instruction
- $\text{funct} = 111000_2$  (0)
- $I = 1$  (register offset),  $P = 1$  (offset indexing),
- $U = 1$  (add),  $B = 0$  (store word),  $W = 0$  (offset indexing),
- $L = 0$  (store)
- $Rd = 9$ ,  $Rn = 1$ ,  $Rm = 3$ ,  $shamt = 2$ ,  $sh = 00_2$  (LSL)

1110 01 111000 0001 1001 00010 00 0 0011 = 0xE7819103



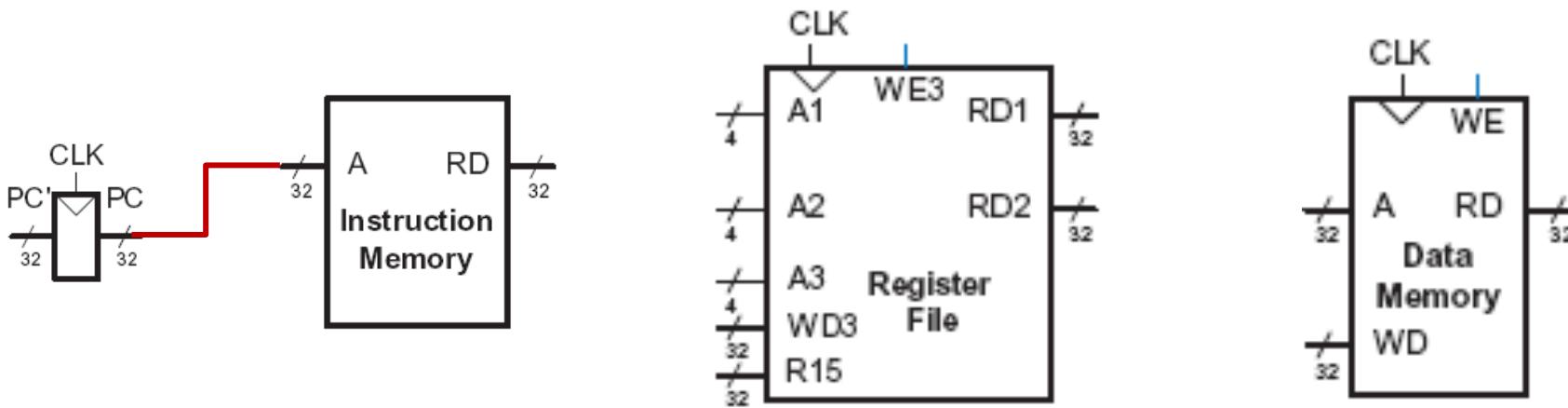
# Datapath LDR

**LDR Rd, [Rn, imm12]**

Il **PC** contiene l'indirizzo dell'istruzione da eseguire.

Il primo passo è quello di leggere questa istruzione dalla memoria istruzioni, per cui il PC viene collegato all'indirizzo di ingresso della memoria di istruzioni.

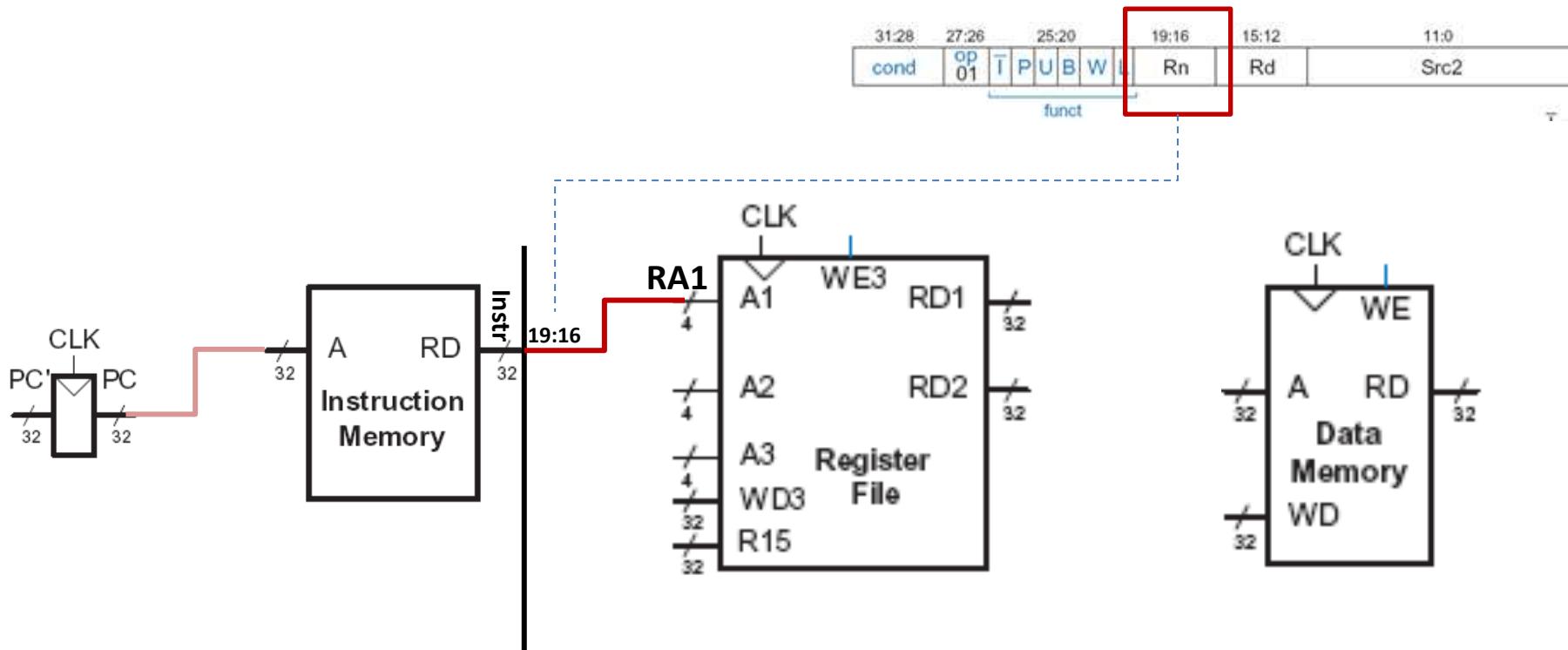
L'istruzione a 32 bit viene letta ed è rappresentata dall'etichetta **Instr**.



# Datapath LDR

Il passo successivo è quello di leggere il registro contenente l'indirizzo di base.  
(L'indirizzo di) questo registro è specificato nel campo **Rn** dell'istruzione,  $\text{Instr}_{19:16}$

Questi bit vengono collegati all'ingresso indirizzo di una delle porte del file register (**A1**). Il register file *legge* il valore di registro in **RD1**.

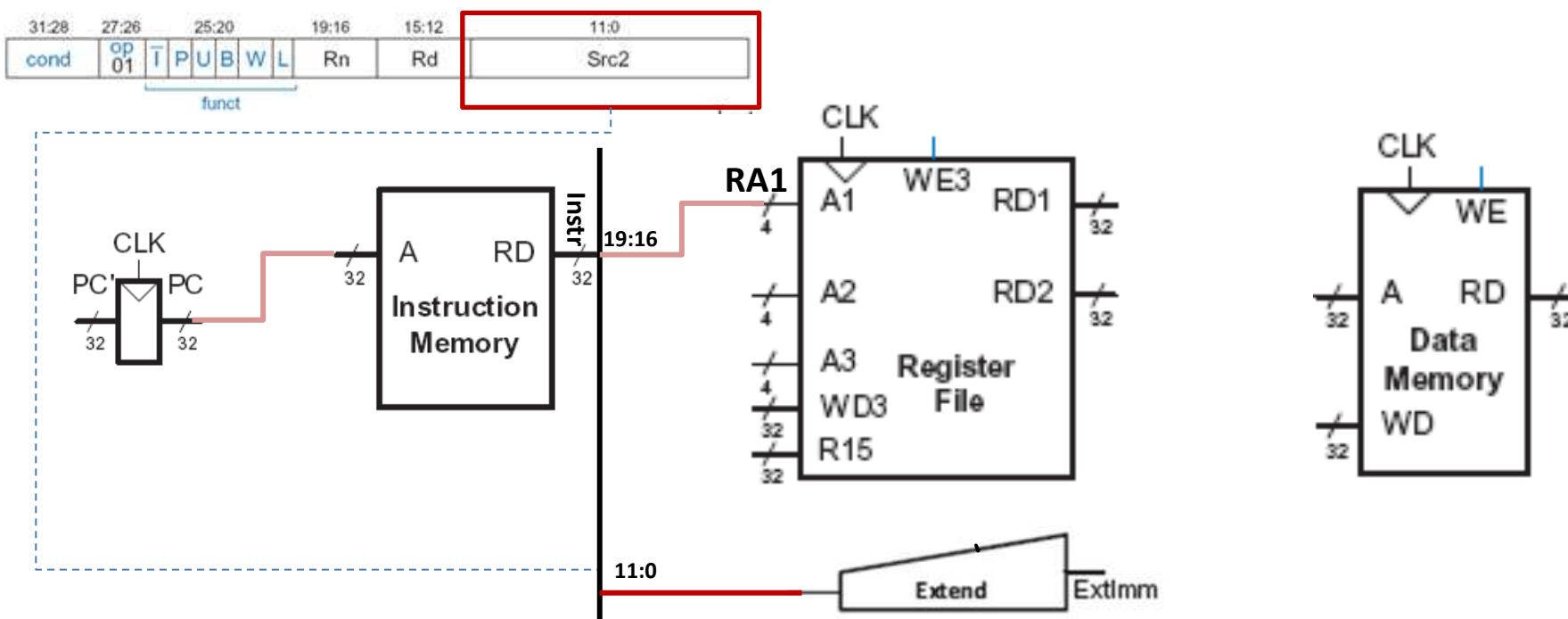


# Datapath LDR

L'istruzione **LDR** richiede anche un **offset**, il quale è memorizzato nell'istruzione stessa e corrisponde ai bit  $\text{Instr}_{11:0}$ .

L'offset è un valore senza segno, quindi deve essere esteso a 32 bit.

Il valore a 32 bit (**ExtImm**) è tale che  $\text{ExtImm}_{31:12} = 0$  e  $\text{ExtImm}_{11:0} = \text{Instr}_{11:0}$ .

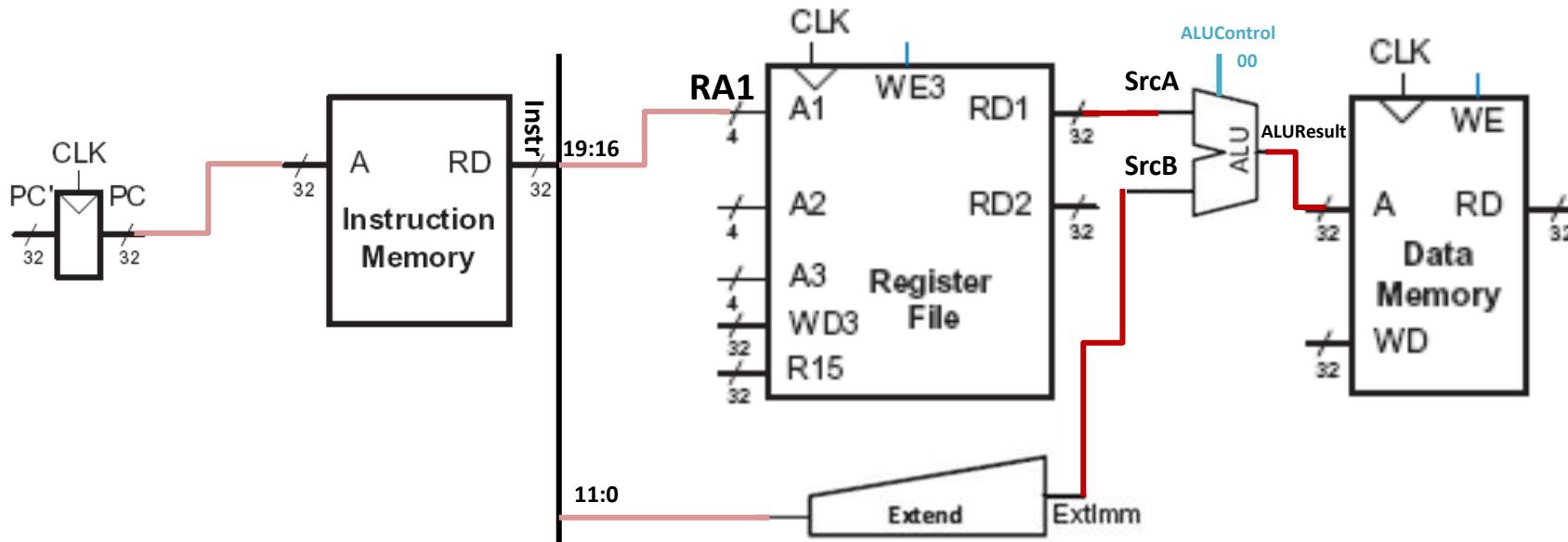


# Datapath LDR

Il processore deve aggiungere **l'indirizzo di base** all'offset per trovare l'indirizzo di memoria a cui leggere. La somma è effettuata per mezzo di una **ALU**.

La **ALU** riceve due operandi (**srcA** e **srcB**). **srcA** proviene dal register file, mentre **srcB** in questo esempio proviene da **ExtImm**. Inoltre, il segnale a 2-bit **ALUControl** specifica l'operazione: una somma (indicata con 00), se il valore del bit U è uguale a 1, una sottrazione (indicata 01), viceversa.

La **ALU** genera un valore a 32 bit **ALUResult**, che viene inviato alla memoria dati come indirizzo di lettura.

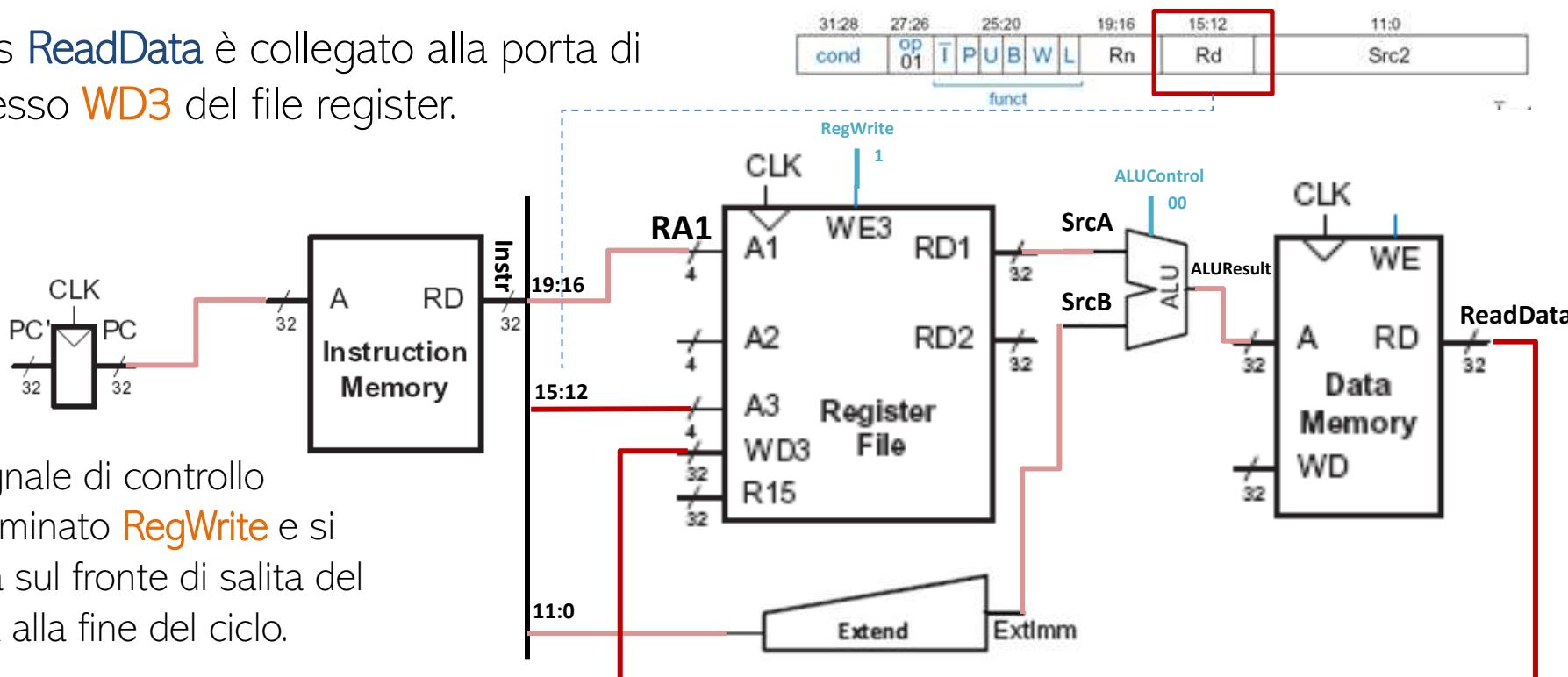


# Datapath LDR

I dati vengono letti dalla memoria dati sul bus **ReadData** e poi sono scritti nel registro destinazione alla fine del ciclo.

Il registro di destinazione per l'istruzione **LDR** è specificato nel campo **Rd**, **Instr<sub>15:12</sub>**, che è collegato all'indirizzo di ingresso **A3**.

Il bus **ReadData** è collegato alla porta di ingresso **WD3** del file register.



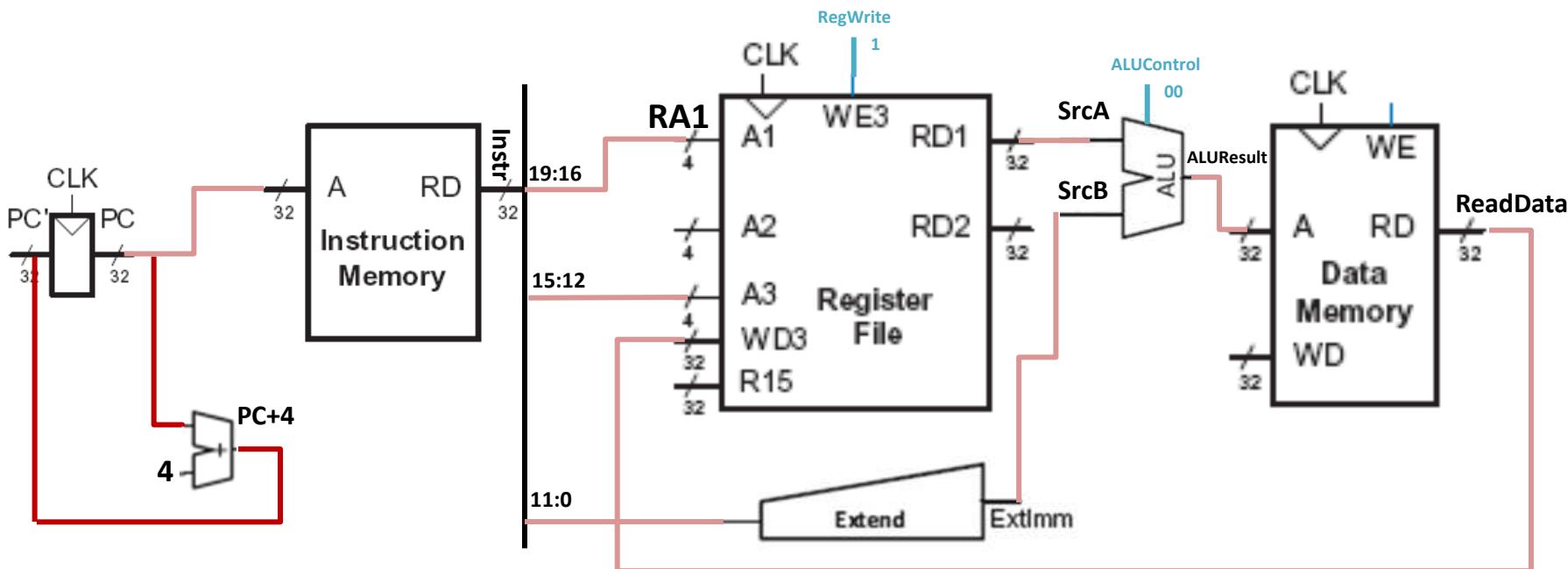
Il segnale di controllo denominato **RegWrite** e si attiva sul fronte di salita del clock alla fine del ciclo.

# Datapath LDR

Contemporaneamente il processore deve calcolare l'indirizzo della successiva istruzione **PC'**.

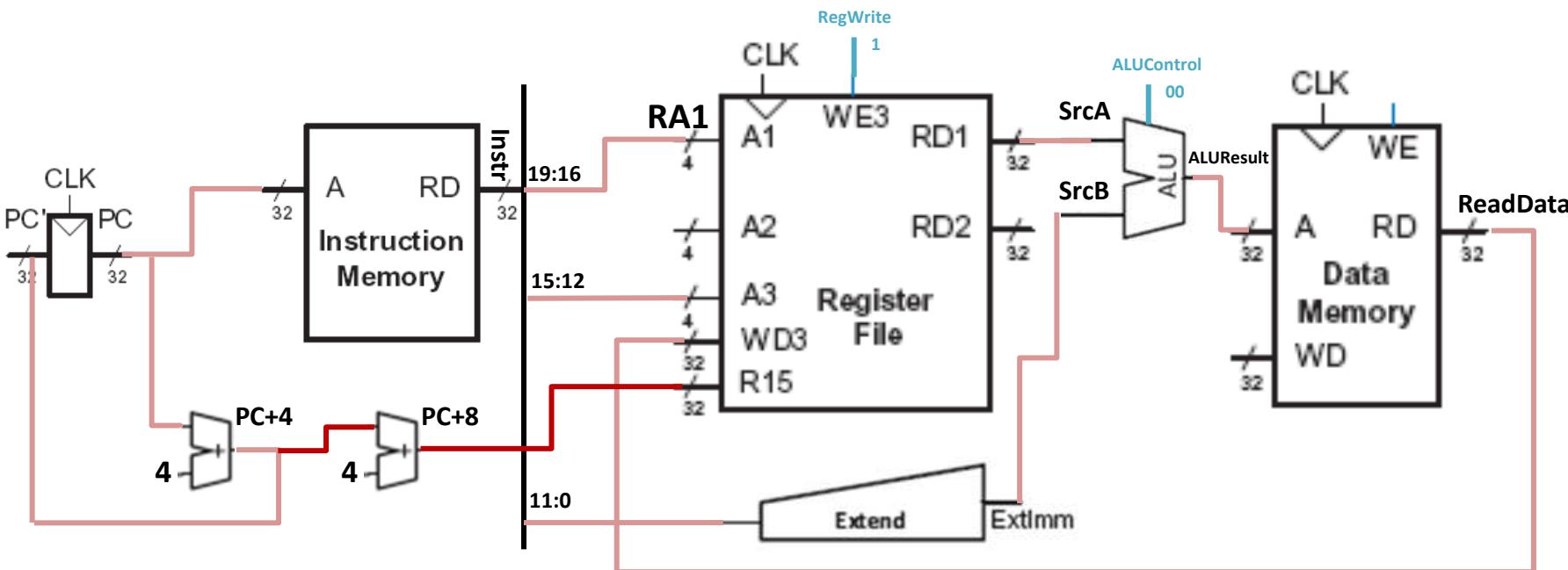
Le istruzioni sono a 32 bit (4 byte), quindi l'istruzione successiva è a **PC + 4**.

Si utilizza un **sommatore** per incrementare il **PC** di 4. Il nuovo indirizzo viene scritto nel contatore di programma sul successivo fronte di salita del clock.



# Datapath LDR

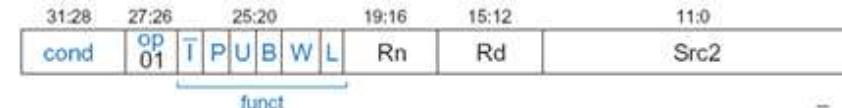
Nelle architetture ARM il registro **R15** contiene il valore **PC+8**, per cui è necessario un ulteriore **sommatore** (+4), la cui uscita sia collegata all'ingresso di **R15**.



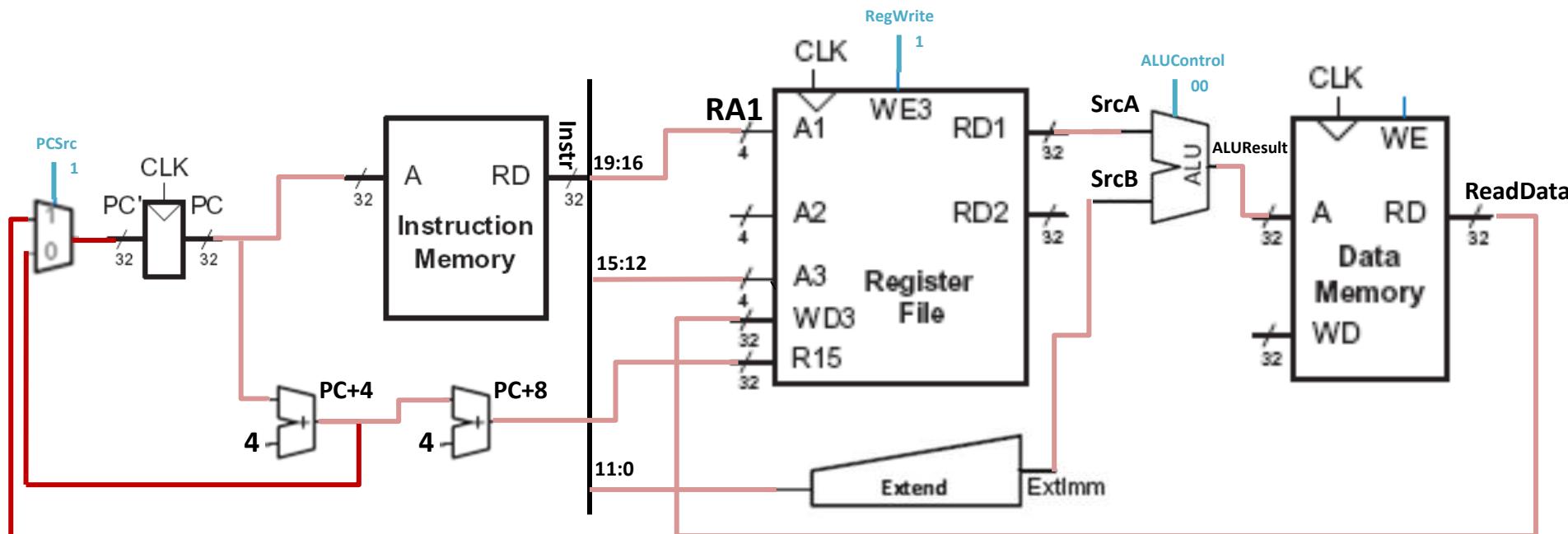
# Datapath LDR

Infine, trattandosi di uno stored program paradigm, l'istruzione successiva potrebbe essere letta anche dalla memoria e quindi corrispondere al contenuto di **ReadData**. Un **multiplexer**, permette di selezionare fra:

- 0 – PCPlus4
- 1 – ReadData.

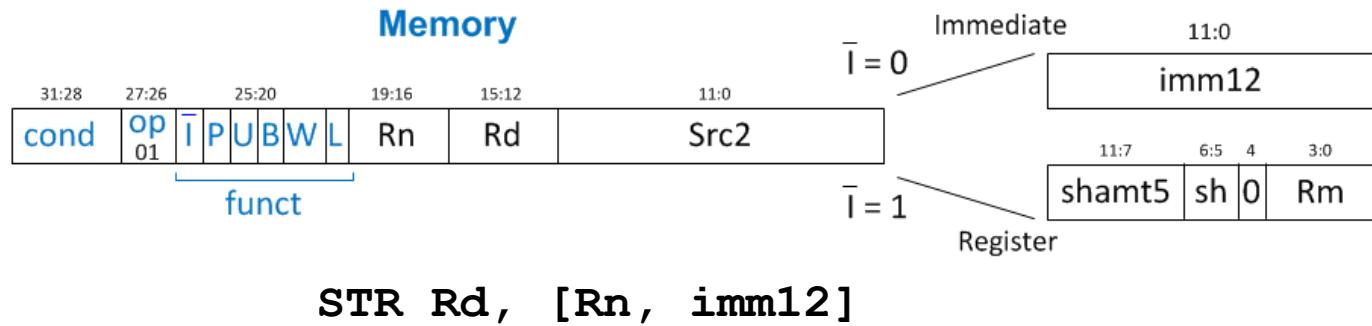


Il segnale di controllo associato al multiplexer è **PCSrc**.



# Datapath STR

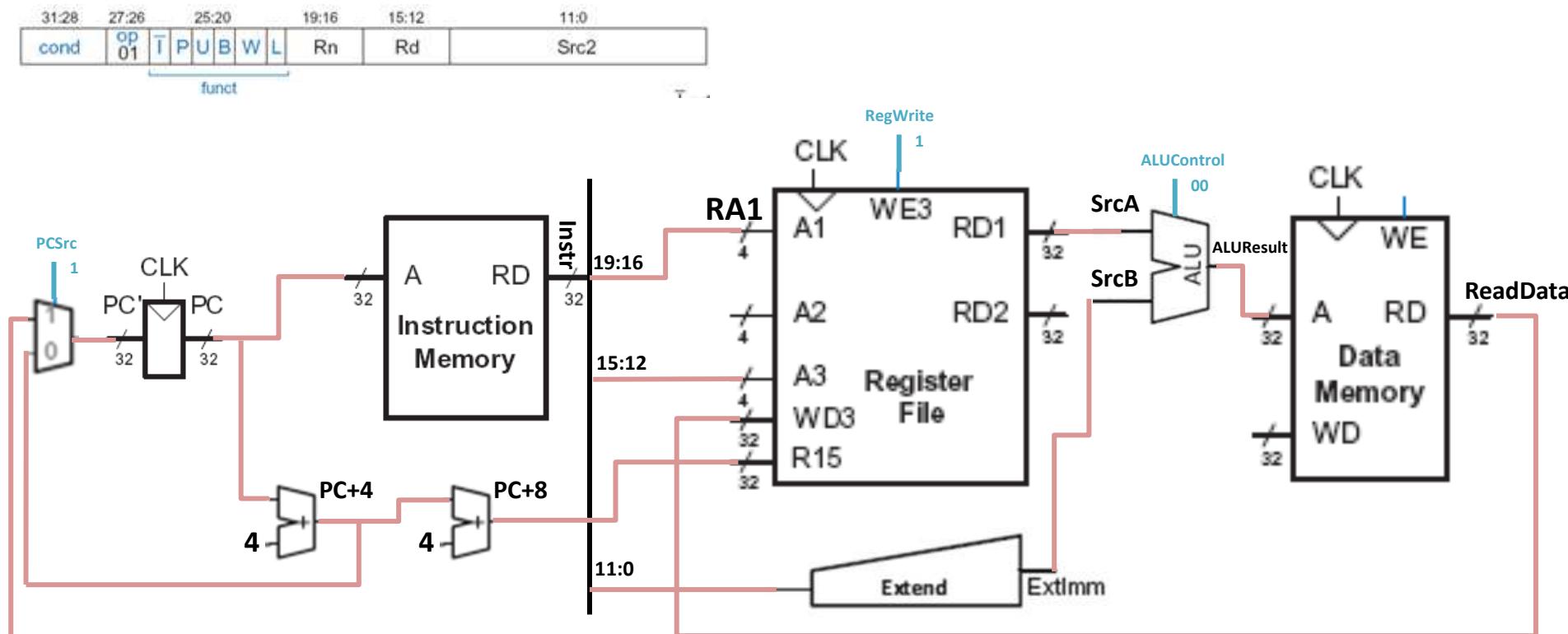
L'istruzione **STR** scrive una parola di 32 bit contenuta in un registro nella memoria centrale. Il modo in cui questa operazione viene effettuata dipende dalla politica di indirizzamento specificata.



# Datapath STR

Estendiamo il **datapath** in modo da poter gestire anche l'istruzioni **STR**.

Come LDR, **STR** legge un indirizzo di base dalla porta **A1** del register file e completa l'immagine. L'**ALU** aggiunge l'indirizzo di base alla costante per trovare l'indirizzo di memoria. Tutte queste funzioni sono già supportate nel datapath.

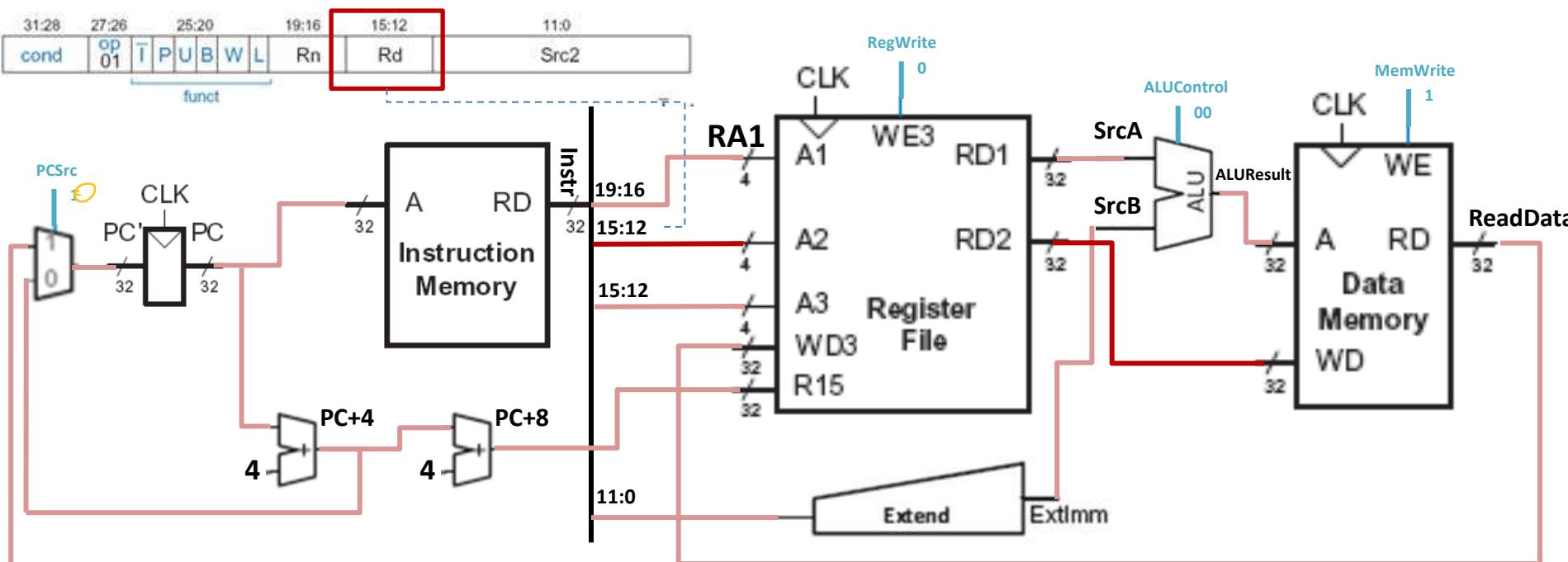


# Datapath STR

Il registro è specificato nel campo **Rd**, **Instr<sub>15:12</sub>**, che è collegato alla porta **A2** del register file.

Il valore del registro viene letto sulla porta **RD2**, che è collegata alla porta dati di scrittura (**WD**) della memoria dati.

L'abilitazione del segnale di scrittura **WE** è controllato da **MemWrite**, il quale è 1 se i dati devono essere scritti in memoria.

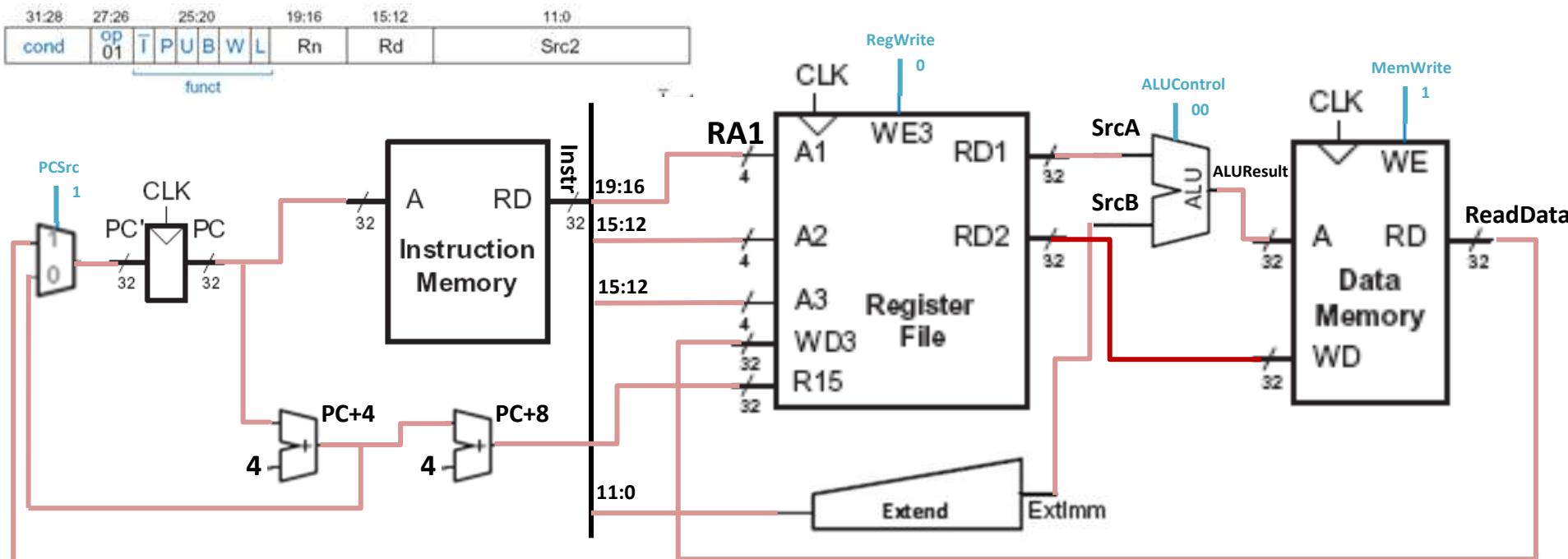


# Datapath STR

Il segnale **ALUControl** deve essere impostato a 00 per sommare l'indirizzo di base e l'offset.

Il segnale **RegWrite** è impostato a 0, perché nulla deve essere scritto nel register file.

Si noti che, pur essendo **Rd** associato anche a A3 e **ReadData** a WD3, questo non produce effetti sul File register, essendo **RegWrite** impostato a 0.



# Istruzioni di data processing

|            |                                                                                  |                    |                          |
|------------|----------------------------------------------------------------------------------|--------------------|--------------------------|
| <b>MOV</b> | Move a 32-bit value                                                              | <b>MOV Rd,n</b>    | Rd = n                   |
| <b>MVN</b> | Move negated (logical NOT) 32-bit value                                          | <b>MVN Rd,n</b>    | Rd = ~n                  |
| <b>ADD</b> | Add two 32-bit values                                                            | <b>ADD Rd,Rn,n</b> | Rd = Rn+n                |
| <b>ADC</b> | Add two 32-bit values and carry                                                  | <b>ADC Rd,Rn,n</b> | Rd = Rn+n+C              |
| <b>SUB</b> | Subtract two 32-bit values                                                       | <b>SUB Rd,Rn,n</b> | Rd = Rn-n                |
| <b>SBC</b> | Subtract with carry of two 32-bit values                                         | <b>SBC Rd,Rn,n</b> | Rd = Rn-n+C-1            |
| <b>RSB</b> | Reverse subtract of two 32-bit values                                            | <b>RSB Rd,Rn,n</b> | Rd = n-Rn                |
| <b>RSC</b> | Reverse subtract with carry of two 32-bit values                                 | <b>RSC Rd,Rn,n</b> | Rd = n-Rn+C-1            |
| <b>AND</b> | Bitwise AND of two 32-bit values                                                 | <b>AND Rd,Rn,n</b> | Rd = Rn AND n            |
| <b>ORR</b> | Bitwise OR of two 32-bit values                                                  | <b>ORR Rd,Rn,n</b> | Rd = Rn OR n             |
| <b>EOR</b> | Exclusive OR of two 32-bit values                                                | <b>EOR Rd,Rn,n</b> | Rd = Rn XOR n            |
| <b>BIC</b> | Bit clear. Every '1' in second operand clears corresponding bit of first operand | <b>BIC Rd,Rn,n</b> | Rd = Rn AND (NOT n)      |
| <b>CMP</b> | Compare                                                                          | <b>CMP Rd,n</b>    | Rd-n & change flags only |
| <b>CMN</b> | Compare Negative                                                                 | <b>CMN Rd,n</b>    | Rd+n & change flags only |
| <b>TST</b> | Test for a bit in a 32-bit value                                                 | <b>TST Rd,n</b>    | Rd AND n, change flags   |
| <b>TEQ</b> | Test for equality                                                                | <b>TEQ Rd,n</b>    | Rd XOR n, change flags   |

|            |                            |                        |                 |
|------------|----------------------------|------------------------|-----------------|
| <b>MUL</b> | Multiply two 32-bit values | <b>MUL Rd,Rm,Rs</b>    | Rd = Rm*Rs      |
| <b>MLA</b> | Multiple and accumulate    | <b>MLA Rd,Rm,Rs,Rn</b> | Rd = (Rm*Rs)+Rn |

# Istruzioni logiche

- AND
- ORR
- EOR (XOR)
- BIC (Bit Clear)
- MVN (MoVe and NOT)

| Source registers |           |           |           |           |
|------------------|-----------|-----------|-----------|-----------|
| R1               | 0100 0110 | 1010 0001 | 1111 0001 | 1011 0111 |
| R2               | 1111 1111 | 1111 1111 | 0000 0000 | 0000 0000 |

| Assembly code  | Result |           |           |           |
|----------------|--------|-----------|-----------|-----------|
| AND R3, R1, R2 | R3     | 0100 0110 | 1010 0001 | 0000 0000 |
| ORR R4, R1, R2 | R4     | 1111 1111 | 1111 1111 | 1111 0001 |
| EOR R5, R1, R2 | R5     | 1011 1001 | 0101 1110 | 1111 0001 |
| BIC R6, R1, R2 | R6     | 0000 0000 | 0000 0000 | 1011 0111 |
| MVN R7, R2     | R7     | 0000 0000 | 0000 0000 | 1111 1111 |

# Istruzioni logiche

- Le istruzioni AND or BIC sono utili per **mascherare** bit
  - **Esempio:** Maschera tutti i bit eccetto il byte meno significativo
  - $0xF234012F \text{ AND } 0x000000FF = 0x0000002F$
  - $0xF234012F \text{ BIC } 0xFFFFF00 = 0x0000002F$
- L'istruzione ORR: è utile per **combinare** bit fields
  - **Esempio:** combina **0xF2340000** con **0x000012BC**:
  - $0xF2340000 \text{ ORR } 0x000012BC = 0xF23412BC$

# Istruzioni di shifting

- LSL: logical shift left
  - Esempio: LSL R0, R7, #5 ; R0=R7 << 5
- LSR: logical shift right
  - Esempio: LSR R3, R2, #31 ; R3=R2 >> 31
- ASR: arithmetic shift right
  - Esempio: ASR R9, R11, #4 ; R9=R11 >>> 4
- ROR: rotate right
  - Esempio: ROR R8, R1, #3 ; R8=R1 ROR 3
- Lo shift aritmetico a sinistra si comporta come lo shift logico
- Non vi è un'istruzione per la rotazione a sinistra: la rotazione a sinistra di **N** posizioni corrisponde a una rotazione a destra di 32- **N** posizioni
- Lo shift di un valore a sinistra di **N** posizioni è equivalente a moltiplicare tale valore per  $2^N$ . Analogamente, uno shift aritmetico aritmetico a destra di un valore **N** è equivalente a dividere per  $2^N$ .

# Esempi istruzioni di shifting

| Source register |           |           |           |           |
|-----------------|-----------|-----------|-----------|-----------|
| R5              | 1111 1111 | 0001 1100 | 0001 0000 | 1110 0111 |

| Assembly Code   |    | Result    |           |           |           |
|-----------------|----|-----------|-----------|-----------|-----------|
| LSL R0, R5, #7  | R0 | 1000 1110 | 0000 1000 | 0111 0011 | 1000 0000 |
| LSR R1, R5, #17 | R1 | 0000 0000 | 0000 0000 | 0111 1111 | 1000 1110 |
| ASR R2, R5, #3  | R2 | 1111 1111 | 1110 0011 | 1000 0010 | 0001 1100 |
| ROR R3, R5, #21 | R3 | 1110 0000 | 1000 0111 | 0011 1111 | 1111 1000 |

| Source registers |           |           |           |           |
|------------------|-----------|-----------|-----------|-----------|
| R8               | 0000 1000 | 0001 1100 | 0001 0110 | 1110 0111 |
| R6               | 0000 0000 | 0000 0000 | 0000 0000 | 0001 0100 |

| Assembly code  |    | Result    |           |           |           |
|----------------|----|-----------|-----------|-----------|-----------|
| LSL R4, R8, R6 | R4 | 0110 1110 | 0111 0000 | 0000 0000 | 0000 0000 |
| ROR R5, R8, R6 | R5 | 1100 0001 | 0110 1110 | 0111 0000 | 1000 0001 |

# Istruzioni di moltiplicazione

- **MUL**: moltiplicazione  $32 \times 32$ -bit, risultato 32-bit
  - `MUL R1, R2, R3`
  - Risultato:  $R1 = (R2 \times R3)_{31:0}$
- **UMULL**: moltiplicazione unsigned  $32 \times 32$ -bit, risultato 64-bit
  - `UMULL R1, R2, R3, R4`
  - Risultato:  $\{R1, R4\} = R2 \times R3$  ( $R2, R3$  unsigned)
- **SMULL**: moltiplicazione signed  $32 \times 32$ -bit, risultato 64-bit
  - `SMULL R1, R2, R3, R4`
  - Risultato:  $\{R1, R4\} = R2 \times R3$  ( $R2, R3$  signed)

# Formato istruzioni di data processing

Il formato delle istruzioni di **data-processing** è il più comune.

Il primo operando sorgente è un registro. Il secondo operando sorgente può essere una costante o un registro. La destinazione è un registro.

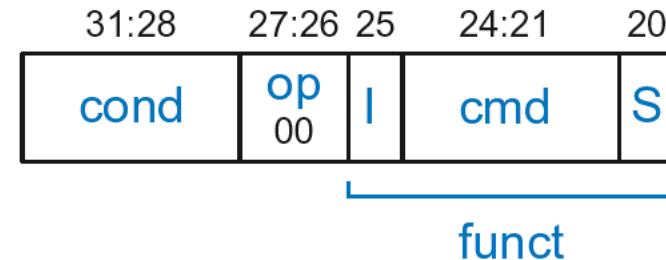
- Operandi:
  - *Rn*: primo registro sorgente
  - *Src2*: secondo registro sorgente o immediate
  - *Rd*: registro destinazione
- Campi di controllo:
  - *cond*: specifica l'esecuzione condizionale in base ai flag
  - *op*: 00 è l'opcode per istruzioni di data processing
  - *funct*: specifica il tipo di funzione da eseguire

## Data-processing

| 31:28       | 27:26     | 25:20        | 19:16     | 15:12     | 11:0        |
|-------------|-----------|--------------|-----------|-----------|-------------|
| <b>cond</b> | <b>op</b> | <b>funct</b> | <b>Rn</b> | <b>Rd</b> | <b>Src2</b> |
| 4 bits      | 2 bits    | 6 bits       | 4 bits    | 4 bits    | 12 bits     |

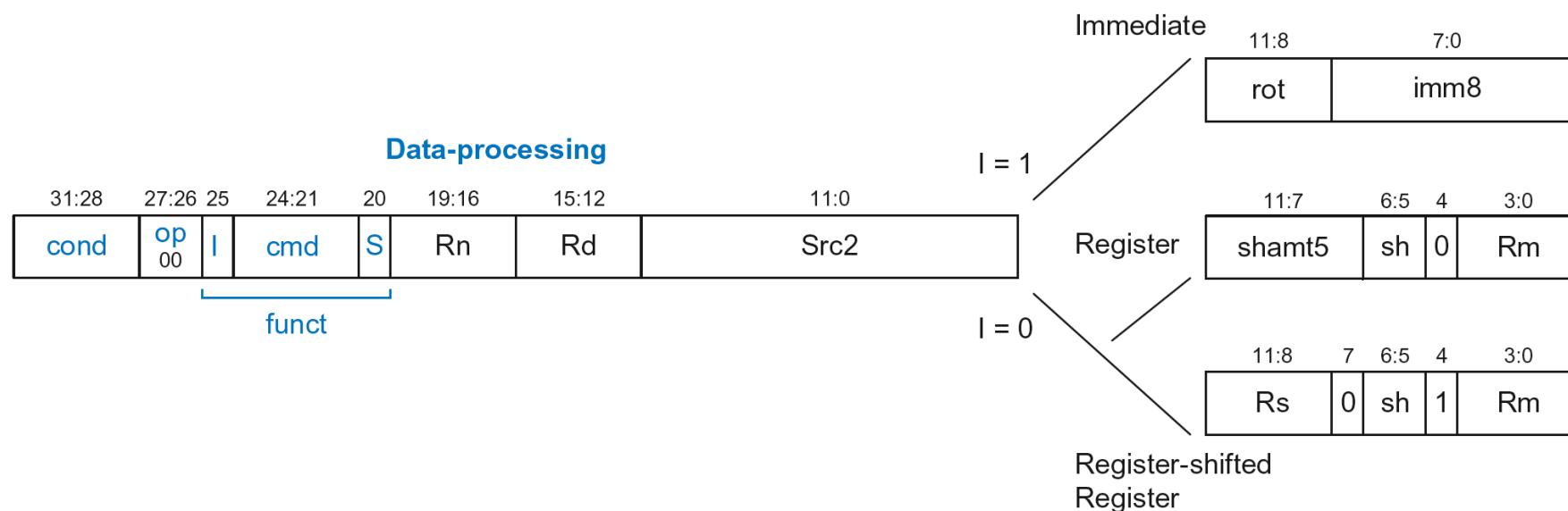
# Formato istruzioni di data processing

- *cmd*: indica l'istruzione specifica da eseguire
  - $cmd = 0100_2$  sta per ADD
  - $cmd = 0010_2$  sta per SUB
- I-bit
  - $I = 0$ : Src2 è un registro
  - $I = 1$ : Src2 è un immediate
- *S*-bit: quando è 1 allora vengono aggiornate le condition flags
  - $S = 0$ : SUB R0, R5, R7
  - $S = 1$ : ADDS R8, R2, R4 or CMPS R3, #10



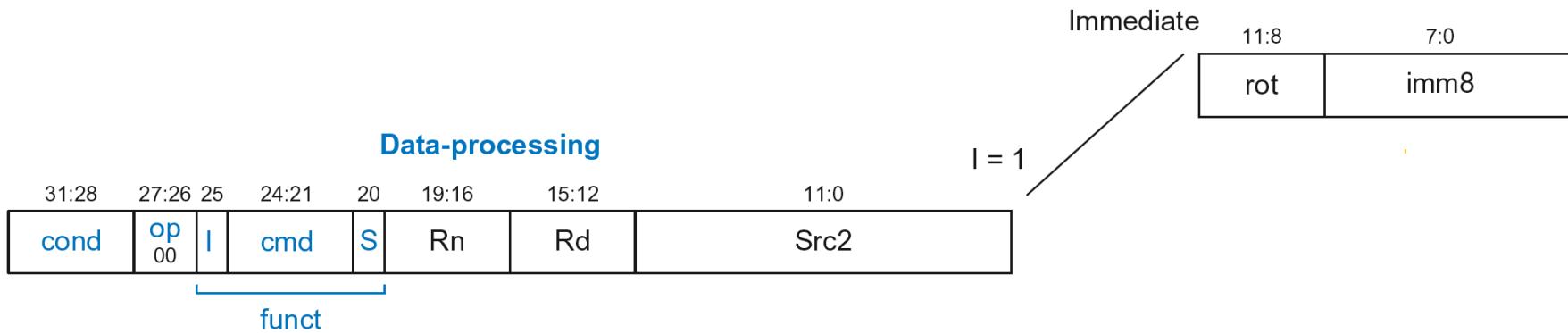
# Formato istruzioni di data processing

- $Src2$  può essere:
  - Immediate
  - Registro
  - Registro “shiftato” da un altro registro



# Formato immediate

- Un immediate è codificato come segue:
  - $imm8$ : 8-bit unsigned immediate
  - $rot$ : 4-bit rotation value
- Da cui si ricava una costante a 32-bit :  
 $imm8 \text{ ROR } (rot \times 2)$



# Formato immediate

- Un immediate è codificato come segue:
  - $imm8$ : 8-bit unsigned immediate
  - $rot$ : 4-bit rotation value
- Da cui si ricava una constant a 32-bit :

$$imm8 \text{ ROR } (rot \times 2)$$

Esempio:  $imm8 = 10001111$

| <b><i>rot</i></b> | <b>32-bit constant</b>                             |
|-------------------|----------------------------------------------------|
| 0000              | 0000 0000 0000 0000 0000 0000 <b>1000 1111</b>     |
| 0001              | <b>11</b> 00 0000 0000 0000 0000 00 <b>10 0011</b> |
| 0010              | <b>1111</b> 0000 0000 0000 0000 0000 <b>1000</b>   |
| ...               | ...                                                |
| 1111              | 0000 0000 0000 0000 0000 00 <b>10 0011 1100</b>    |

# Esempio: Add

ADD R0, R1, #42

*cond* =  $1110_2$  (14) indica un'esecuzione non condizionata

*op* =  $00_2$  (0) indica un'istruzione di data-processing

*cmd* =  $0100_2$  (4) è il codice di ADD

*I* = 1 indica che *Src2* è un immediate, *S* = 0,

*Rd* = 0, *Rn* = 1

*imm8* = 42, *rot* = 0

## Field Values

| 31:28    | 27:26  | 25 | 24:21    | 20 | 19:16 | 15:12 | 11:8   | 7:0      |
|----------|--------|----|----------|----|-------|-------|--------|----------|
| $1110_2$ | $00_2$ | 1  | $0100_2$ | 0  | 1     | 0     | 0      | 42       |
| cond     | op     | I  | cmd      | S  | Rn    | Rd    | shamt5 | sh       |
| 1110     | 00     | 1  | 0100     | 0  | 0001  | 0000  | 0000   | 00101010 |

**0xE281002A**

# Esempio: SUB

SUB R2, R3, #0xFF0

*cond* =  $1110_2$  (14)

*op* =  $00_2$  (0)

*cmd* =  $0010_2$  (2) è il codice di SUB

*I*=1, *S*=0

*Rd* = 2, *Rn* = 3

*imm8* = 0xFF

per produrre 0xFF0 *imm8* deve essere ruotato di 4 bit a sinistra,  
ovvero 28 bit a destra. Quindi, *rot* = 14

## Field Values

| 31:28    | 27:26  | 25 | 24:21    | 20 | 19:16 | 15:12 | 11:8 | 7:0      |
|----------|--------|----|----------|----|-------|-------|------|----------|
| $1110_2$ | $00_2$ | 1  | $0010_2$ | 0  | 3     | 2     | 14   | 255      |
| cond     | op     | I  | cmd      | S  | Rn    | Rd    | rot  | imm8     |
| 1110     | 00     | 1  | 0010     | 0  | 0011  | 0010  | 1110 | 11111111 |

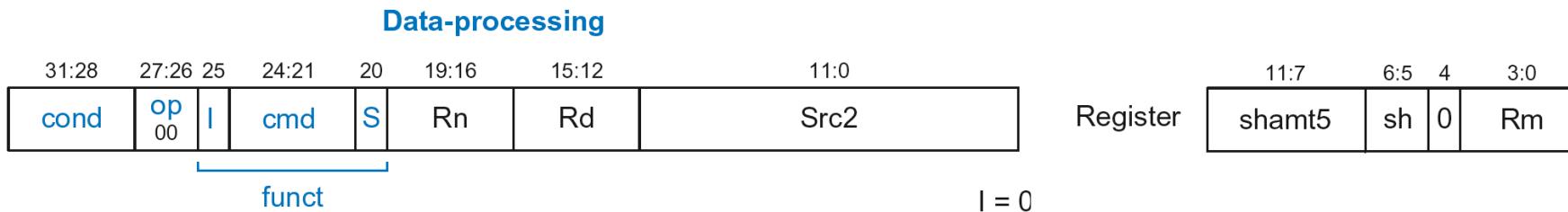
**0xE2432EFF**

# Formato register

*Rm*: indica il registro che fa da secondo operando

*shamt5*: indica di quanto il valore in Rm è shiftato

*sh*: indica il tipo di shift (i.e., >>, <<, >>>, ROR)



| Shift Type | <i>sh</i> |
|------------|-----------|
| LSL        | $00_2$    |
| LSR        | $01_2$    |
| ASR        | $10_2$    |
| ROR        | $11_2$    |

# Esempio: Add

ADD R5, R6, R7

Operation:  $R5 = R6 + R7$

$cond = 1110_2$  (14)

$op = 00_2$  (0)

$cmd = 0100_2$  (4)

$I=0$  indica che  $Src2$  è un registro, S=0,

$Rd = 5$ ,  $Rn = 6$ ,  $Rm = 7$

$shamt = 0$ ,  $sh = 0$

## Field Values

| 31:28    | 27:26  | 25 | 24:21    | 20 | 19:16 | 15:12 | 11:7   | 6:5 | 4 | 3:0  |
|----------|--------|----|----------|----|-------|-------|--------|-----|---|------|
| $1110_2$ | $00_2$ | 0  | $0100_2$ | 0  | 6     | 5     | 0      | 0   | 0 | 7    |
| cond     | op     | I  | cmd      | S  | Rn    | Rd    | shamt5 | sh  |   | Rm   |
| 1110     | 00     | 0  | 0100     | 0  | 0110  | 0101  | 00000  | 00  | 0 | 0111 |

0xE0865007

# Esempio: OR

ORR R9, R5, R3, LSR #2

Operation:  $R9 = R5 \text{ OR } (R3 >> 2)$

- $cond = 1110_2$  (14)
- $op = 00_2$  (0)
- $cmd = 1100_2$  (12) indica l'istruzione ORR  
 $\neq 0$

$Rd = 9$ ,  $Rn = 5$ ,  $Rm = 3$

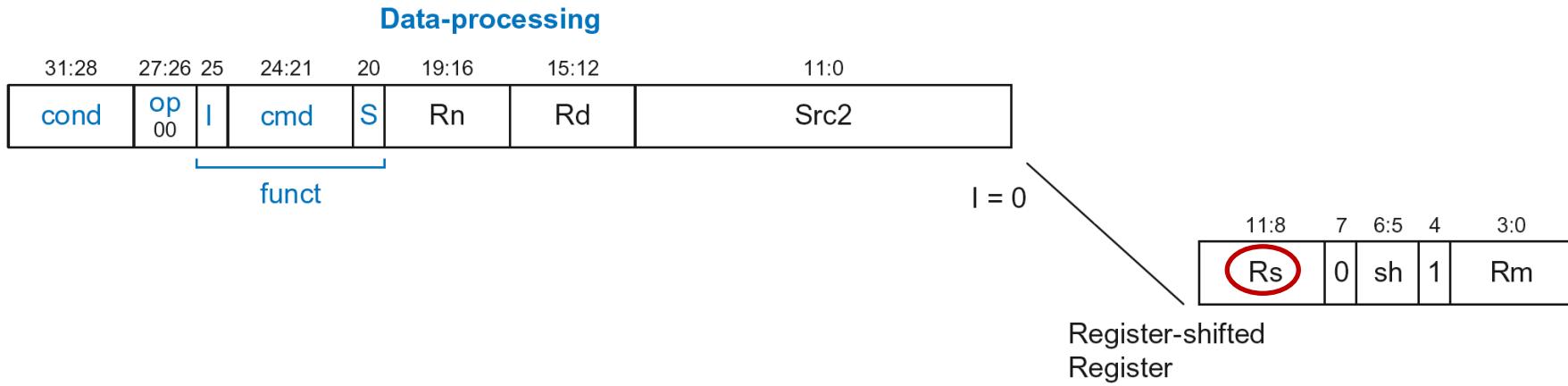
$shamt5 = 2$ ,  $sh = 01_2$  (LSR)



1110 00 0 1100 0 0101 1001 00010 01 0 0011

**0xE1859123**

# Formato register-shifted register



Simile al formato register, con la differenza che il numero di posizioni di cui Rm deve shiftare non è una costante ma è indicato dal registro Rs.

# Esempio: XOR

EOR R8, R9, R10, ROR R12

Operation:  $R8 = R9 \text{ XOR } (R10 \text{ ROR } R12)$

$cond = 1110_2$  (14)

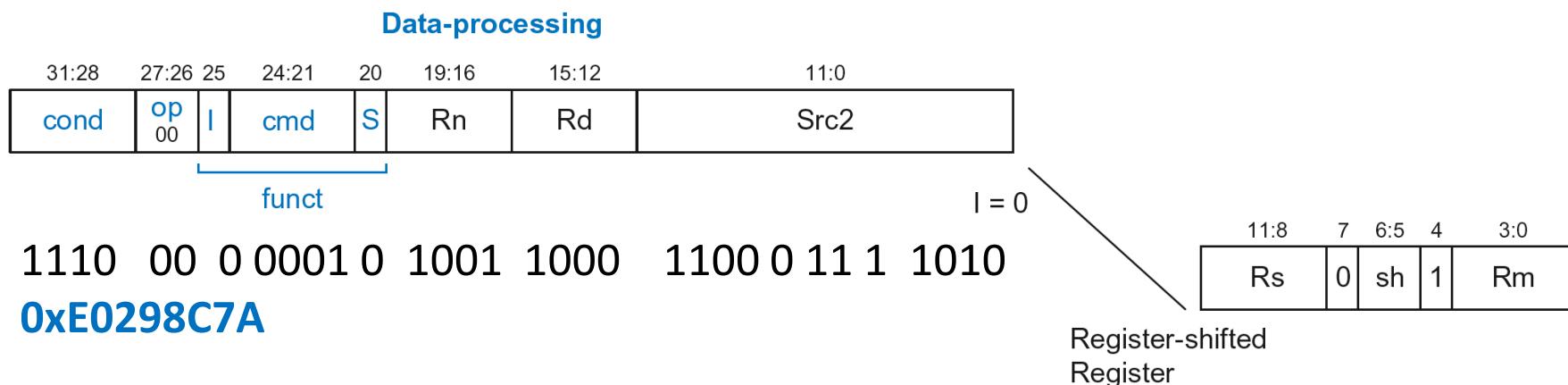
$op = 00_2$  (0)

$cmd = 0001_2$  (1) indica l'istruzione EOR

$I = 0$

$Rd = 8, Rn = 9, Rm = 10, Rs = 12$

$sh = 11_2$  (ROR)



# Esempio: ROR

ROR R1, R2, #23

Operation:  $R1 = R2 \text{ ROR } 23$

$cond = 1110_2$  (14)

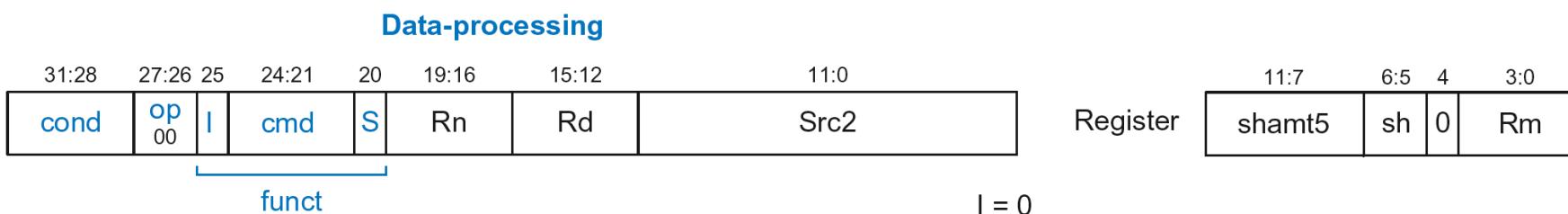
$op = 00_2$  (0)

$cmd = 1101_2$  (13) codice usato per tutti i tipi di shift (LSL, LSR, ASR, ROR)

$Src2$  è un immediate-shifted register quindi  $I=0$

$Rd = 1$ ,  $Rn = 0$ ,  $Rm = 2$

$shamt5 = 23$ ,  $sh = 11_2$  (ROR)



1110 00 011010 0000 0001 10111110 0010

**0xE1A01BE2**

# Datapath ADD, SUB, AND, ORR

Estendiamo il **datapath** per gestire le istruzioni di data processing **ADD**, **SUB**, **AND** e **ORR**, utilizzando la modalità immediate.

In tal caso, le istruzioni hanno come operandi un registro ed una costante contenuta nei bit dell'istruzione stessa. L'**ALU** esegue l'operazione e il risultato viene scritto in un terzo registro.

Esse differiscono solo nella specifica operazione eseguita dall'**ALU**. Quindi, possono essere implementate tutte con lo stesso hardware utilizzando diversi segnali **ALUControl**.

I valori per **ALUControl** sono:

- **ADD – 00;**
- **SUB – 01;**
- **AND – 10;**
- **ORR – 11.**

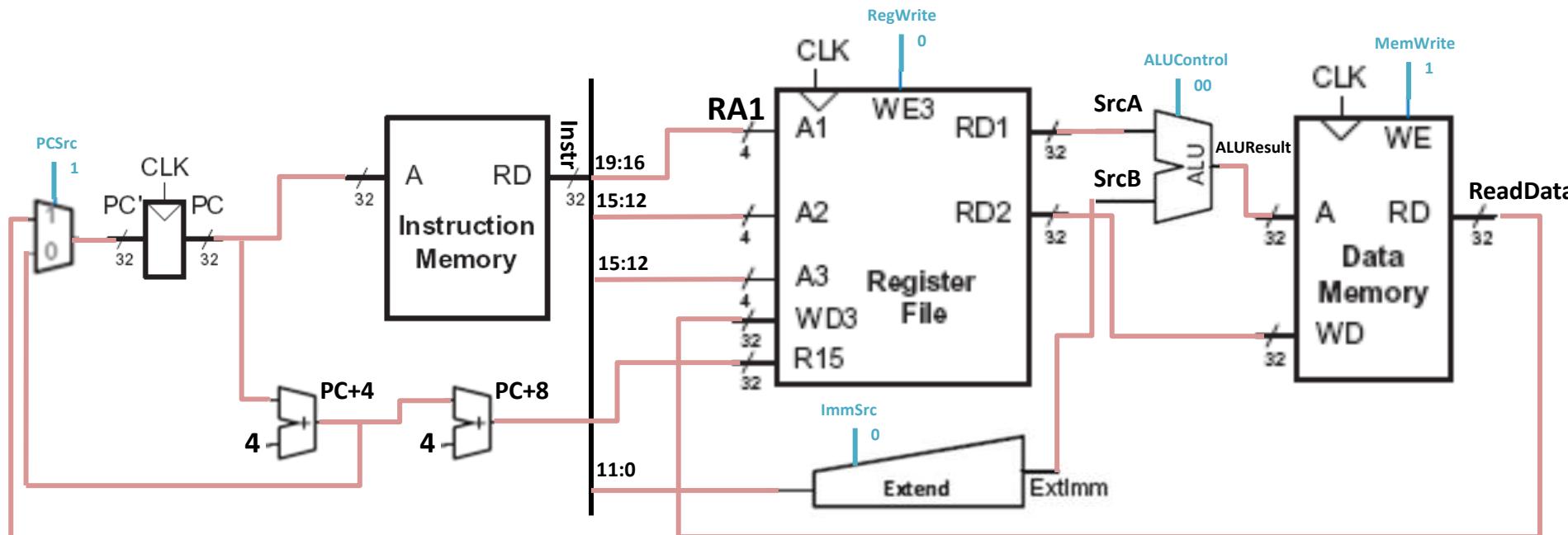
L'**ALU** imposta anche dei bit in **ALUFlags<sub>3:0</sub>**:

- **Zero;**
- **Negativo;**
- **Carry;**
- **oVerflow.**

# Datapath ADD, SUB, AND, ORR

Le istruzioni di data processing utilizzano costanti di 8 bit (non 12 bit), per cui il blocco **Extend** riceve in input un segnale di controllo **ImmSrc**:

- $\text{ImmSrc} = 0 \rightarrow \text{ExtImm}$  è esteso da  $\text{Instr}_{7:0}$ ;
- $\text{ImmSrc} = 1 \rightarrow \text{ExtImm}$  è esteso da  $\text{Instr}_{11:0}$  (per LDR o STR);



# Datapath ADD, SUB, AND, ORR

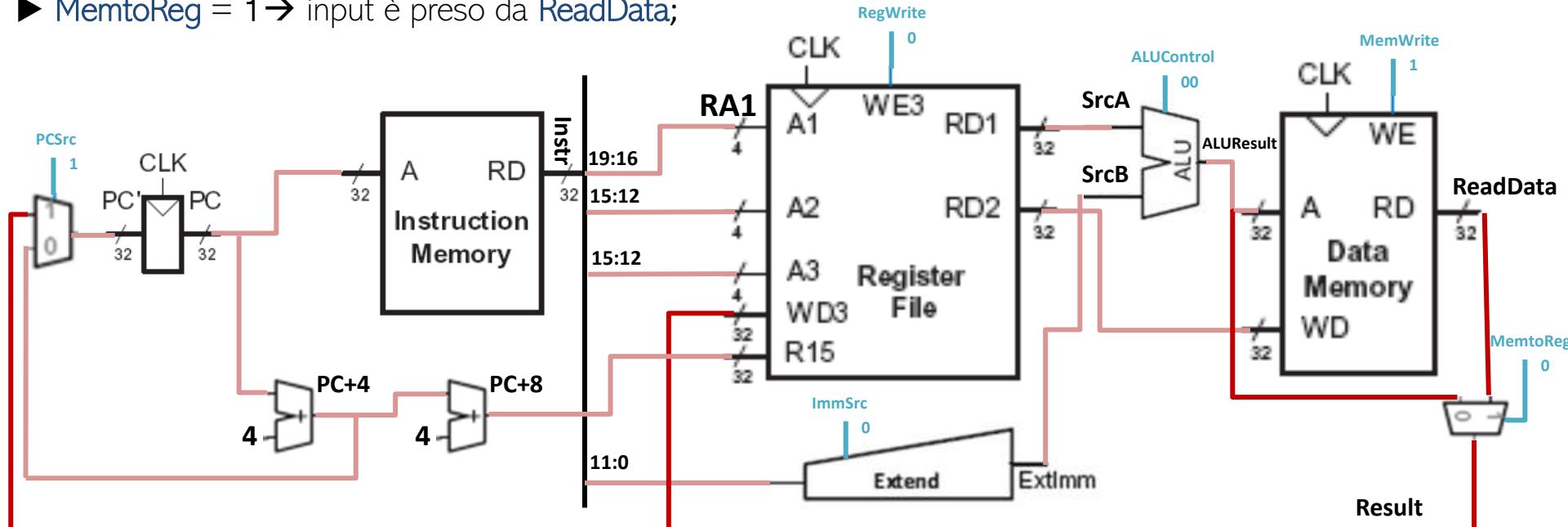
Un altro aspetto da disambiguare riguarda la scrittura nel register file.

Esso può ricevere l'input sia dalla **memoria dati** (LDR), che dall'**ALU** (operazioni aritmetiche).

Aggiungiamo un altro **multiplexer** che permette di selezionare la sorgente di input tra **ReadData** e **ALUResult**. L'uscita del multiplexer è indicata con **Result**.

Il multiplexer richiede un segnale di controllo, ovvero **MemtoReg**.

- **MemtoReg = 0** → input è preso da **ALUResult**;
- **MemtoReg = 1** → input è preso da **ReadData**;

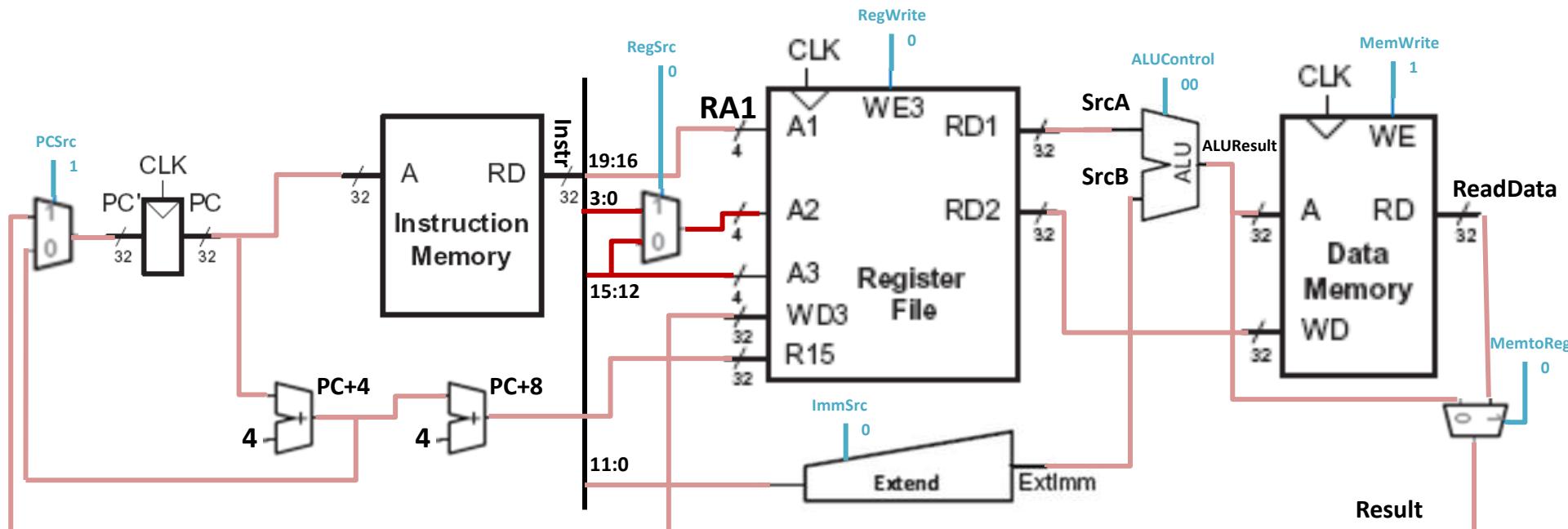


# Datapath ADD, SUB, AND, ORR

Le istruzioni di data processing con secondo argomento da registro ricevono quest'ultimo da **Rm**, specificato da  $\text{Instr}_{3:0}$ , piuttosto che da una costante.

Aggiungiamo un ulteriore **multiplexer** sugli ingressi del file registro. In base al valore del segnale di controllo **RegSrc**, **RA2** può essere selezionato fra:

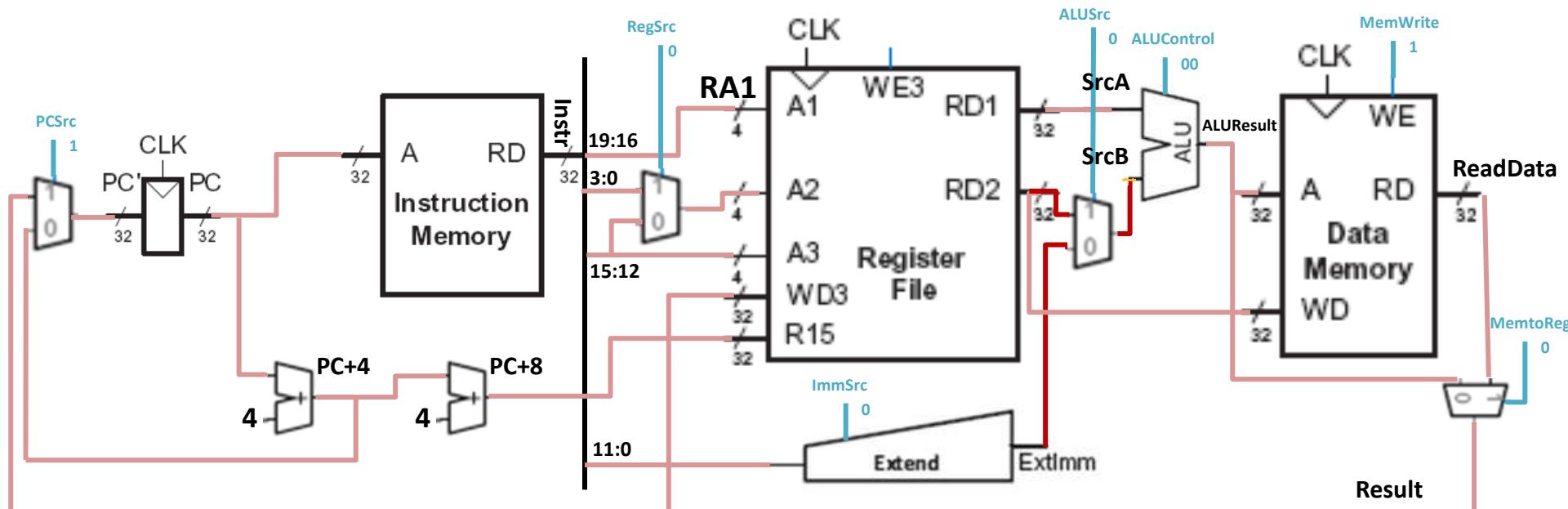
- **Rd** ( $\text{Instr}_{15:12}$ ) per **STR**;
- **Rm** ( $\text{Instr}_{3:0}$ ) per istruzioni di data processing con secondo argomento da registro.



# Datapath ADD, SUB, AND, ORR

Aggiungiamo un ulteriore **multiplexer** sugli ingressi dell'**ALU** per selezionare questo secondo registro sorgente. In base al valore del segnale di controllo **ALUSrc**, la seconda sorgente della ALU viene selezionata tra:

- ▶ **ExtImm** per istruzioni, che utilizzano costanti;
- ▶ dal **register file** per istruzioni di data processing con secondo argomento da registro.



# Esecuzione condizionata

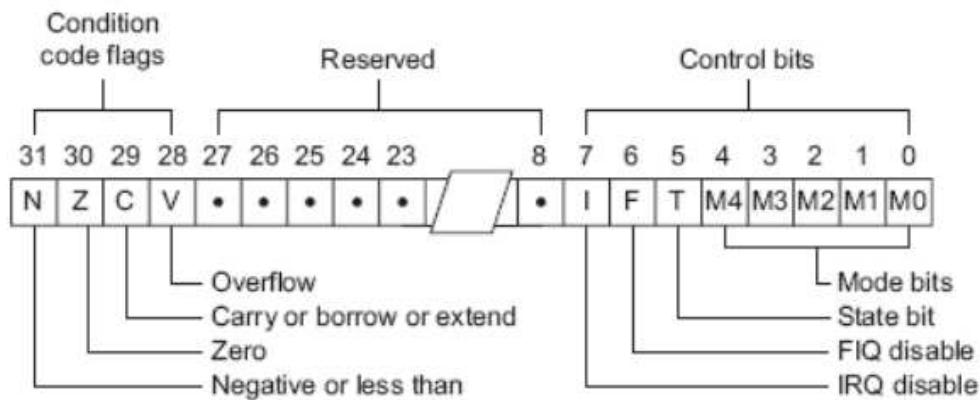
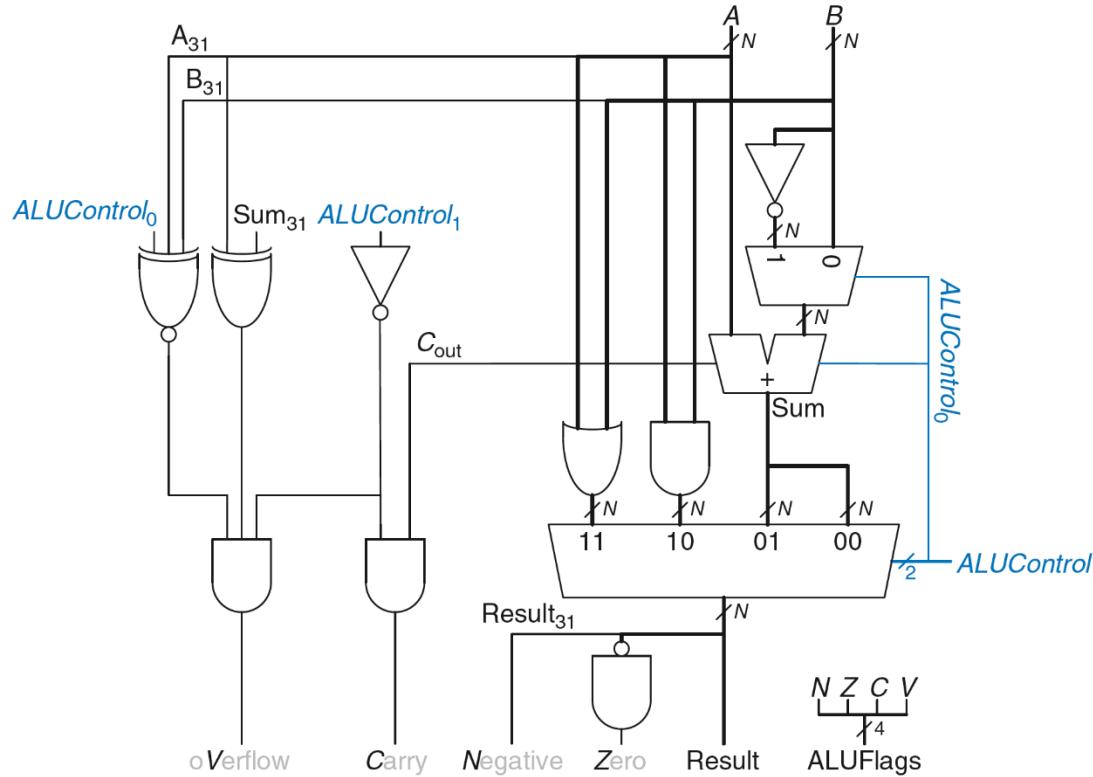
A volte si vuole che l'esecuzione di una istruzione dipenda dal verificarsi o meno di una certa condizione, per esempio:

- Istruzioni if/then/else, while, etc.: un blocco di istruzioni è eseguito solo se una data condizione valuta a true

ARM usa delle **condition flag** che sono

- memorizzate nel *Current Program Status Register (CPSR)*
- settate da una istruzione (es. CMP, ADDS)
- usate per eseguire in modo condizionato una istruzione

| Flag | Name            | Description                              |
|------|-----------------|------------------------------------------|
| N    | <b>Negative</b> | Instruction result is negative           |
| Z    | <b>Zero</b>     | Instruction results in zero              |
| C    | <b>Carry</b>    | Instruction causes an unsigned carry out |
| V    | <b>oVerflow</b> | Instruction causes an overflow           |



# Istruzione di comparazione: CMP

CMP R5, R6

- Esegue:  $R5 - R6$
- Non salva il risultato
- Aggiorna i flag nel Current Program Status Register. Se il risultato di  $R5 - R6$ :
  - è uguale a 0, allora  $Z=1$
  - è negativo, allora  $N=1$
  - causa un carry out,  $C=1$
  - causa un signed overflow,  $V=1$

# Suffisso S al nome di una istruzione

ADD**S** R1, R2, R3

- Esegue:  $R2 + R3$
- Salva il risultato in R1
- Aggiorna i flag nel Current Program Status Register. Se il risultato di  $R2 + R3$ :
  - è uguale a 0, allora  $Z=1$
  - è negativo, allora  $N=1$
  - causa un carry out,  $C=1$
  - causa un signed overflow,  $V=1$

# Istruzioni condizionate

- Un'istruzione può essere *eseguita condizionalmente* sulla base dei valori delle condition flags
- In assembly, un'esecuzione condizionata è indicata da un suffisso detto *condition mnemonic*.

```
CMP    R1, R2  
SUBNE R3, R5, R8
```

- **NE**: SUB verrà eseguito solo se  $Z = 0$
- Il valore di Z è determinato (in questo esempio) dall'istruzione CMP. Quindi sto verificando se  $R1 - R2 \neq 0$ , ovvero  $R1 \neq R2$

| cond | Mnemonic     | Name                                | CondEx                     |
|------|--------------|-------------------------------------|----------------------------|
| 0000 | EQ           | Equal                               | Z                          |
| 0001 | NE           | Not equal                           | $\overline{Z}$             |
| 0010 | CS/HS        | Carry set / unsigned higher or same | C                          |
| 0011 | CC/LO        | Carry clear / unsigned lower        | $\overline{C}$             |
| 0100 | MI           | Minus / negative                    | N                          |
| 0101 | PL           | Plus / positive or zero             | $\overline{N}$             |
| 0110 | VS           | Overflow / overflow set             | V                          |
| 0111 | VC           | No overflow / overflow clear        | $\overline{V}$             |
| 1000 | HI           | Unsigned higher                     | $\overline{Z}C$            |
| 1001 | LS           | Unsigned lower or same              | Z OR $\overline{C}$        |
| 1010 | GE           | Signed greater than or equal        | $\overline{N} \oplus V$    |
| 1011 | LT           | Signed less than                    | $N \oplus V$               |
| 1100 | GT           | Signed greater than                 | $\overline{Z}(N \oplus V)$ |
| 1101 | LE           | Signed less than or equal           | Z OR ( $N \oplus V$ )      |
| 1110 | AL (or none) | Always / unconditional              | Ignored                    |

**Unsigned**      **Signed**  
**A = 1001<sub>2</sub>**    A = 9      A = -7  
**B = 0010<sub>2</sub>**    B = 2      B = 2  
  

$$\begin{array}{r} A - B: \quad 1001 \\ \quad + 1110 \\ \hline (a) \quad 10111 \end{array} \quad \begin{array}{l} NZCV = 0011_2 \\ HS: \text{TRUE} \\ GE: \text{FALSE} \end{array}$$

**Unsigned**      **Signed**  
**A = 0101<sub>2</sub>**    A = 5      A = 5  
**B = 1101<sub>2</sub>**    B = 13     B = -3  
  

$$\begin{array}{r} A - B: \quad 0101 \\ \quad + 0011 \\ \hline (b) \quad 1000 \end{array} \quad \begin{array}{l} NZCV = 1001_2 \\ HS: \text{FALSE} \\ GE: \text{TRUE} \end{array}$$

# Esempi di istruzioni condizionate

```
CMP    R5, R9           ; performs R5-R9  
                  ; sets condition flags  
  
SUBEQ  R1, R2, R3       ; executes if R5==R9 (Z=1)  
ORRMI  R4, R0, R9       ; executes if R5-R9 is  
                  ; negative (N=1)
```

Si assume per esempio che  $R5 = 17$ ,  $R9 = 23$ :

CMP esegue:  $17 - 23 = -6$  (flags:  $N=1$ ,  $Z=0$ ,  $C=0$ ,  $V=0$ )

SUBEQ non è eseguita (non sono uguali:  $Z=0$ )

ORRMI è eseguita poiché il risultato è negativo ( $N=1$ )

# Istruzioni di Branching

*Un programma di solito esegue in sequenza, incrementando il Program Counter (PC) di 4 (32 bit) dopo ciascuna istruzione, in modo da puntare alla successiva istruzione.*

Le istruzioni branching permettono di cambiare il valore del PC. ARM include due tipi di branch: *simple branch* (B) e *branch and link* (BL).

Come altre istruzioni ARM, i branch possono essere condizionati o incondizionati.

Il codice assembly utilizza le etichette per indicare i blocchi di istruzioni nel programma.

Quando il codice assembly è tradotto in codice macchina, queste etichette vengono tradotte in indirizzi di istruzione.

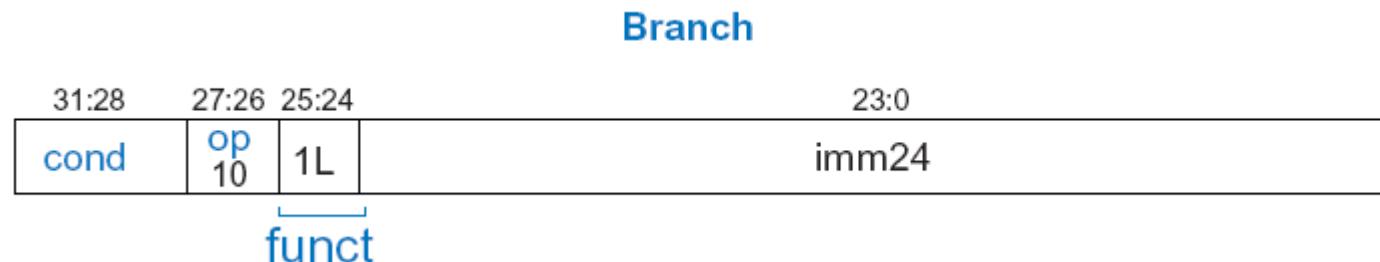
# Istruzioni di Branching

Le istruzioni di branching utilizzano un unico operando costante di 24 bit, **imm24**.

Esse hanno un campo **cond** di 4 bit e un campo **op** di 2 bit, il cui valore è  $10_2$ .

Il campo **funct** ha solo 2 bit. Il bit più significativo è sempre 1 per i branch. Il bit meno significativo, **L**, indica il tipo di operazione di branch: 1 per **BL** e 0 per **B**.

I restanti 24 bit, **imm24**, rappresentano un valore in complemento a due, che specifica la posizione dell'istruzione relativamente all'indirizzo **PC** + 8.



# Istruzioni di Branching

La costante a 24-bit viene moltiplicata per 4 ed estesa con segno. Pertanto, la logica Extend necessita di una ulteriore modalità. **ImmSrc** è, quindi, esteso a 2 bit.

L'istruzione di salto somma poi **ImmSrc** a **PC+8** e scrive il risultato di nuovo nel **PC**.

| ImmSrc | ExtImm                           | Description                                   |
|--------|----------------------------------|-----------------------------------------------|
| 00     | {24 0s} $Instr_{7:0}$            | 8-bit unsigned immediate for data-processing  |
| 01     | {20 0s} $Instr_{11:0}$           | 12-bit unsigned immediate for LDR/STR         |
| 10     | {6 $Instr_{23}$ } $Instr_{23:0}$ | 24-bit signed immediate multiplied by 4 for B |

# Istruzioni di Branching

L'istruzione **BL** (Branch and Link) è usata per la chiamata di una subroutine

- Salva l'indirizzo di ritorno (**R15**) in **R14**
- Il ritorno dalla routine si effettua copiando **R14** in **R15**:

**MOV R15, R14**

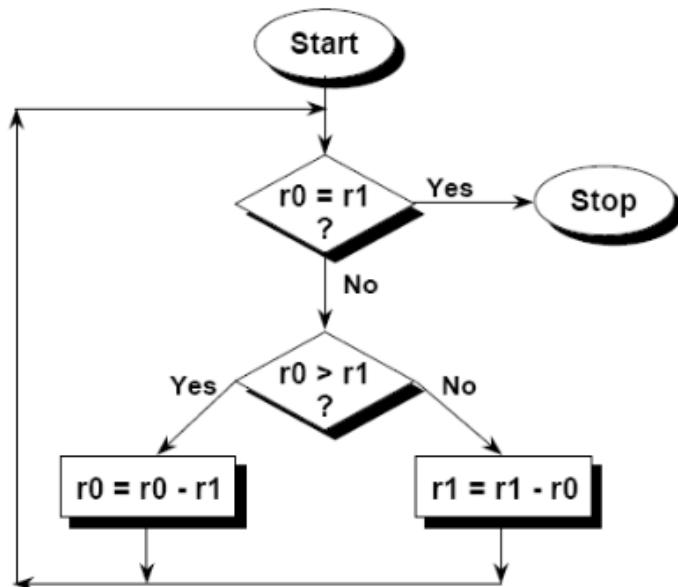
Il registro **R14** ha la funzione (architetturale) di subroutine Link Register (**LR**).

In esso viene salvato l'indirizzo di ritorno  
(ovvero il contenuto del registro **R15**)  
quando viene eseguita l'istruzione **BL**  
(Branch and Link).

```
....  
BL    function          ; call 'function'  
....  
; procedure returns to here  
....  
function                   ; function body  
....  
....  
MOV   PC, LR             ; Put R14 into PC to return
```

# Esempi di istruzioni di branching

Il processore ARM consente un codice molto compatto.



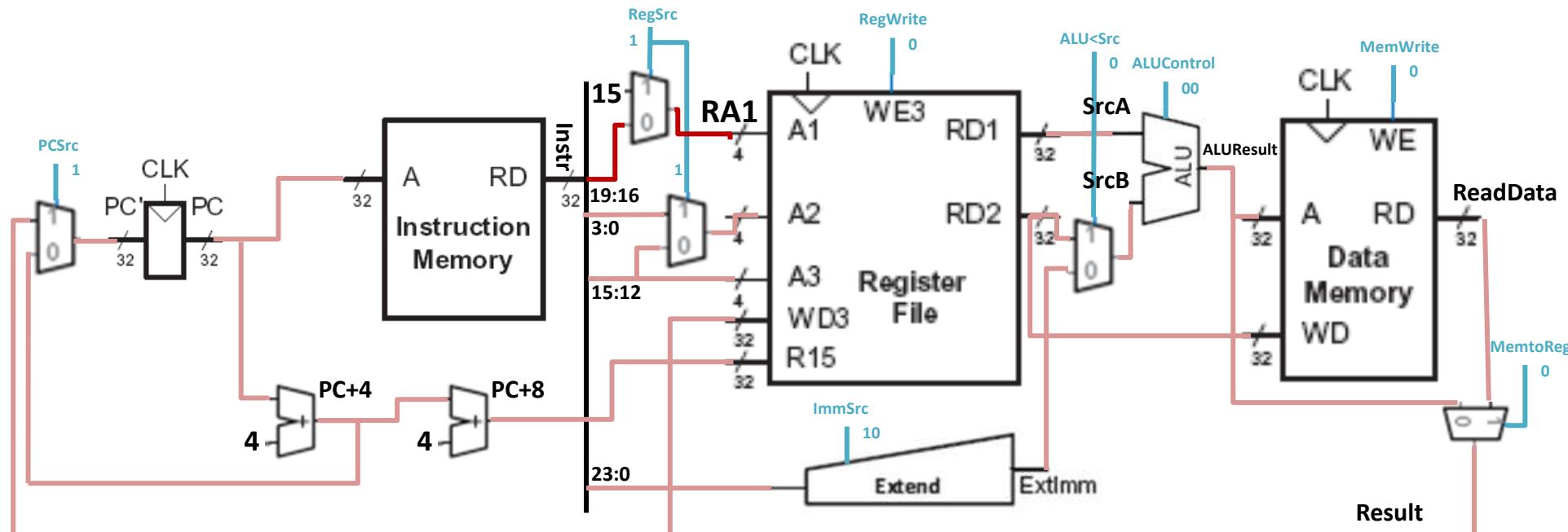
|     |                |                     |
|-----|----------------|---------------------|
| MCD | cmp r0,r1      | ;if r0 > r1         |
|     | subgt r0,r0,r1 | ;then r0 <- r0-r1   |
|     | sublt r1,r1,r0 | ;else r1 <- r1-r0   |
|     | bne MCD        | ;raggiunta la fine? |

|      |    |                     |                                                  |
|------|----|---------------------|--------------------------------------------------|
| 1011 | LT | Signed less than    | $N \oplus V$                                     |
| 1100 | GT | Signed greater than | $\overline{Z}(\overline{N} \oplus \overline{V})$ |

# Datapath istruzioni di branching

Dato che **PC+8** è letto dalla prima porta del register file, è necessario un **multiplexer** per selezionare R15 come ingresso di **RA1**. Il multiplexer è controllato dal segnale **RegSrc**, il cui valore è preso dai bit **Instr<sub>19:16</sub>**, per la maggior parte delle istruzioni ed è impostato a **15** per le istruzioni di branch (**B**).

**MemtoReg** è impostato a **0** e **PCSrc** è impostato a **1** per selezionare il nuovo **PC** da **ALUResult**.

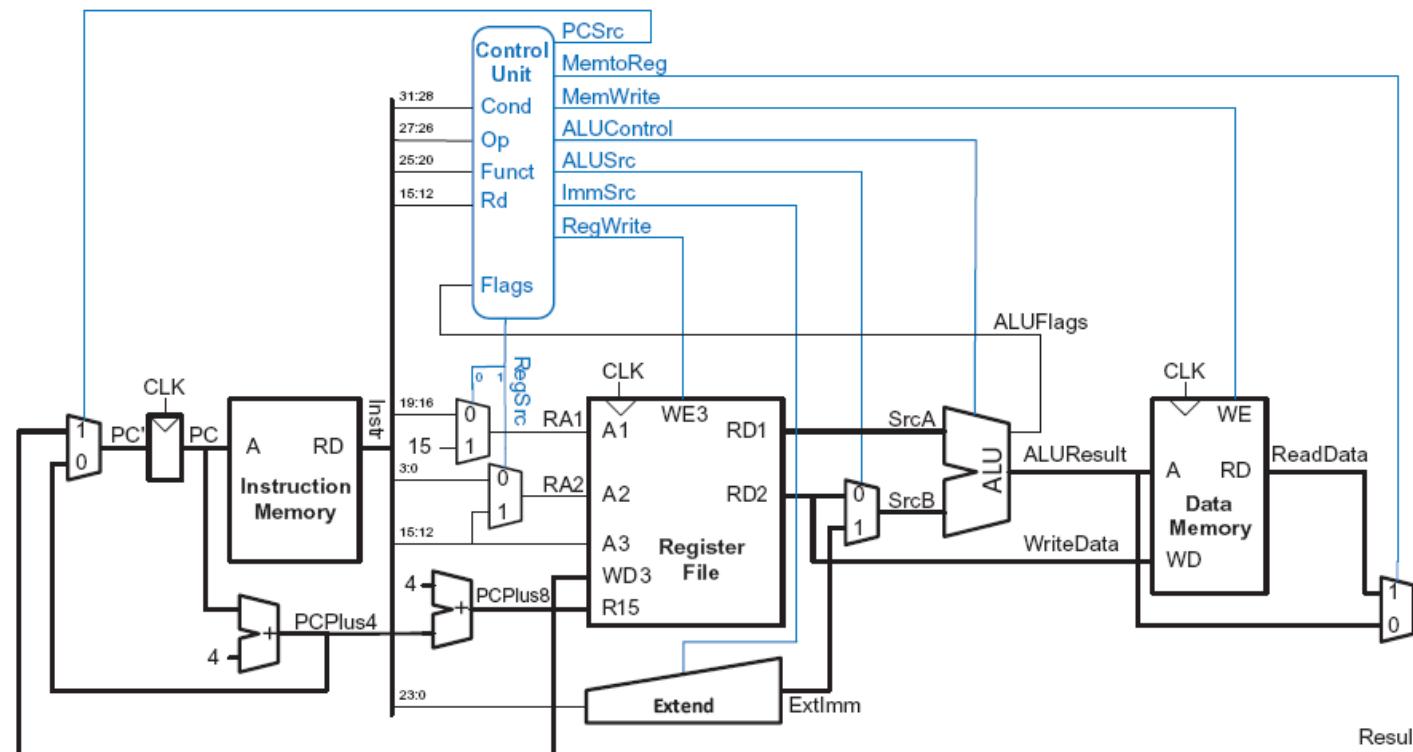


# Unità di controllo

L'unità di controllo calcola i segnali di controllo in base a:

- i campi **cond**, **op**, e **funct** dell'istruzione ( $\text{Instr}_{31:28}$ ,  $\text{Instr}_{27:26}$ , e  $\text{Instr}_{25:20}$ );
- i **flag**;
- se il registro di destinazione è il **PC**.

Il controller memorizza anche i **flag di stato** attuali nel *Current Program Status Register* e li aggiorna in modo appropriato.



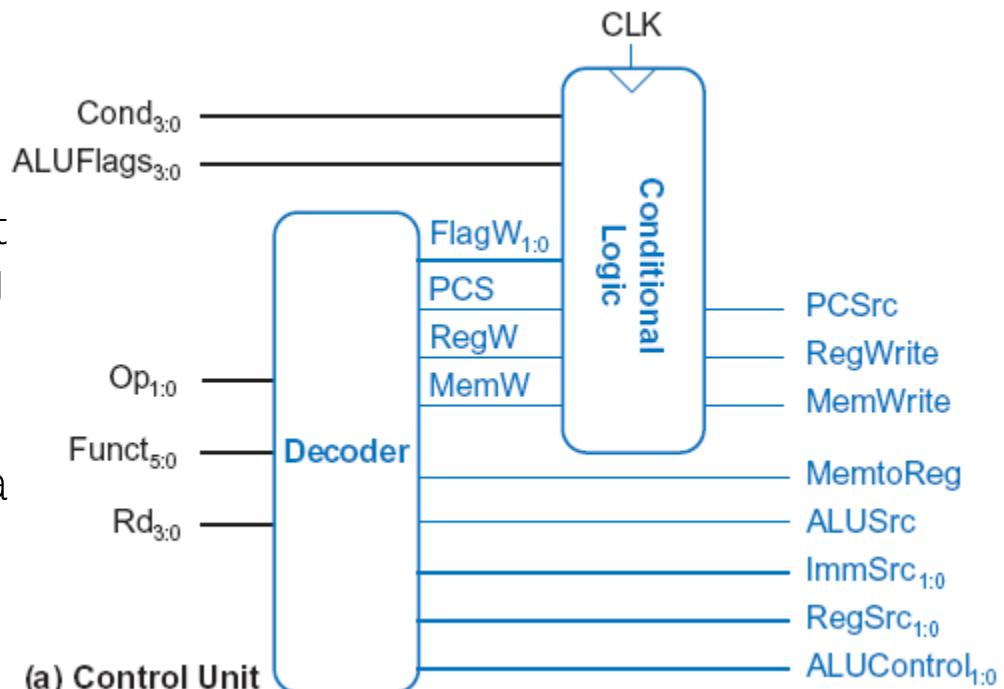
# Unità di controllo

Dividiamo l'unità di controllo in due parti principali:

- il **decoder** – genera i segnali di controllo sulla base dei campi di Instr;
- la **logica condizionale** – gestisce i flag di stato e li aggiorna quando l'istruzione deve essere eseguita su condizione.

Il **Decoder** è composto da:

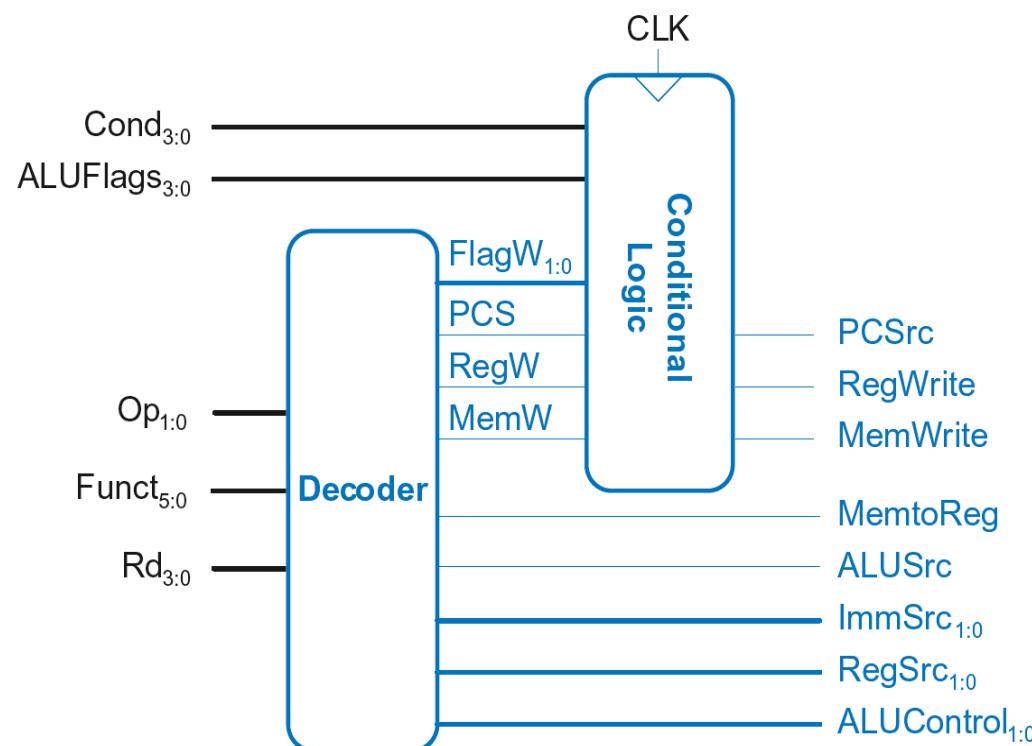
- un **decodificatore principale**, che produce la maggior parte dei segnali di controllo;
- un **decoder ALU**, che utilizza il campo Funct per determinare il tipo di operazione che l'ALU deve eseguire;
- la **logica di controllo del PC**, che determina se il PC deve essere aggiornato a causa di una istruzione di branch o di una scrittura in R15.



# Unità di controllo

Dividiamo l'unità di controllo in due parti principali:

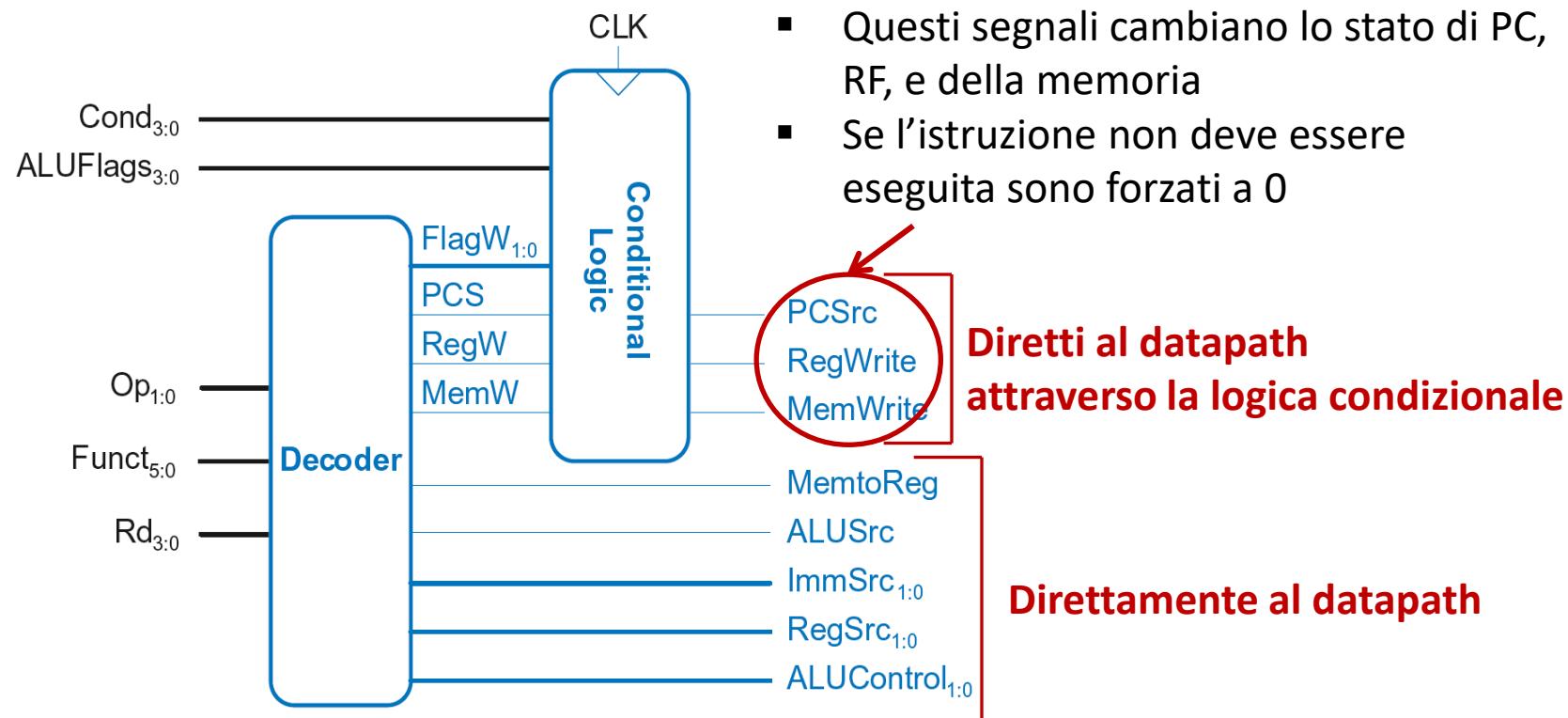
- il **decoder** – genera i segnali di controllo sulla base dei campi di Instr;
- la **logica condizionale** – gestisce i flag di stato e li aggiorna quando l'istruzione deve essere eseguita su condizione.



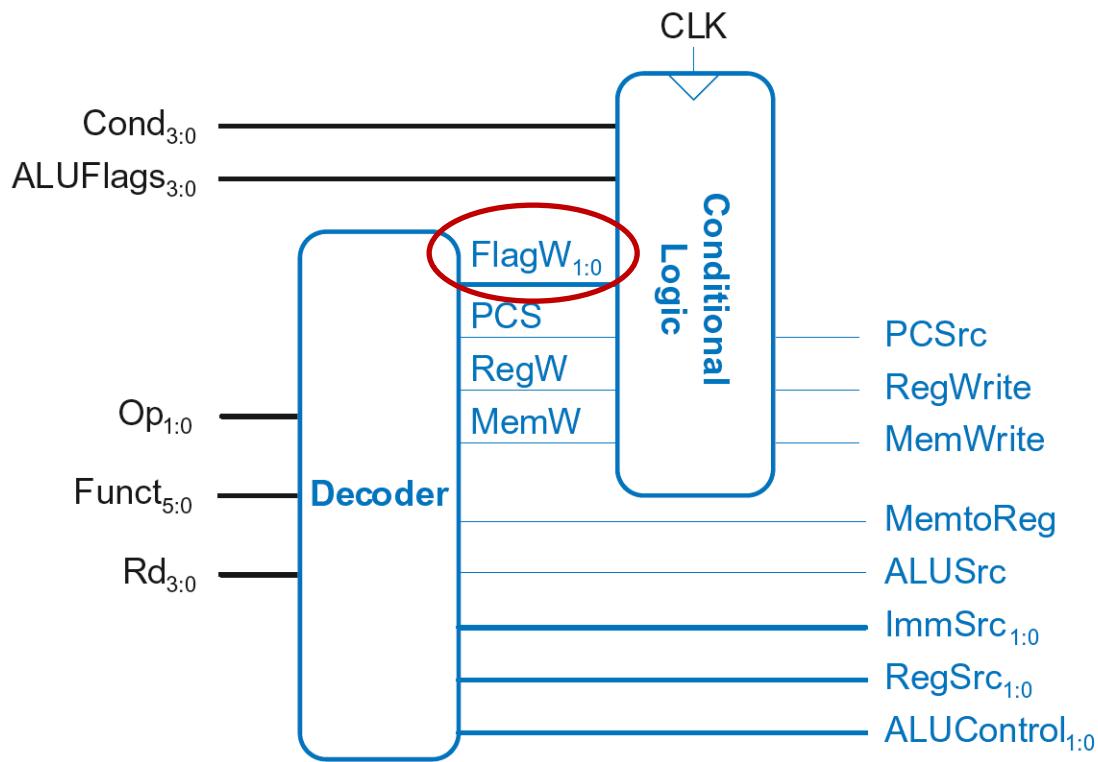
# Unità di controllo

Dividiamo l'unità di controllo in due parti principali:

- il **decoder** – genera i segnali di controllo sulla base dei campi di Instr;
- la **logica condizionale** – gestisce i flag di stato e li aggiorna quando l'istruzione deve essere eseguita su condizione.



# Unità di controllo



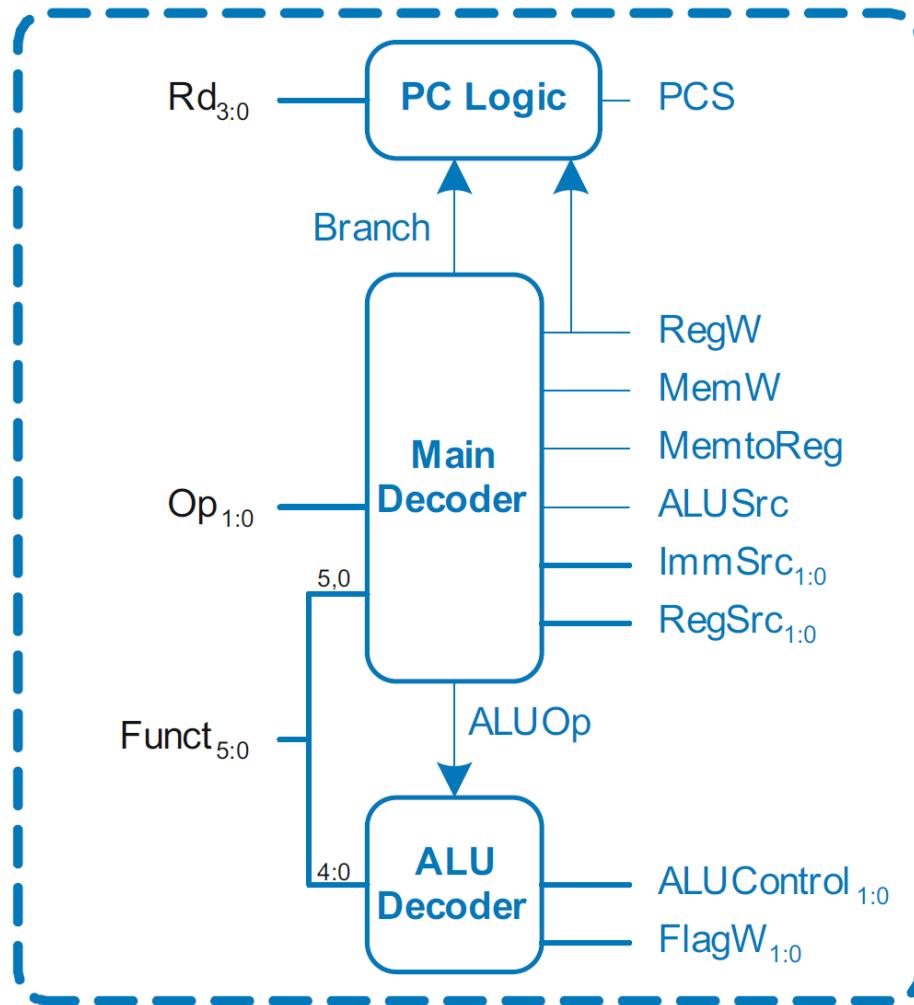
- $\text{FlagW}_{1:0}$ : Flag Write signal, indica quando le *ALUFlags* devono essere aggiornate, ovvero quando in una istruzione  $S=1$
- ADD, SUB aggiornano tutti i flag (**NZCV**)
- AND, ORR aggiornano solo N e Z
- Quindi sono necessari due bit:
  - $\text{FlagW}_1 = 1$ : NZ  
( $\text{ALUFlags}_{3:2}$  saved)
  - $\text{FlagW}_0 = 1$ : CV  
( $\text{ALUFlags}_{1:0}$  saved)

# Decoder

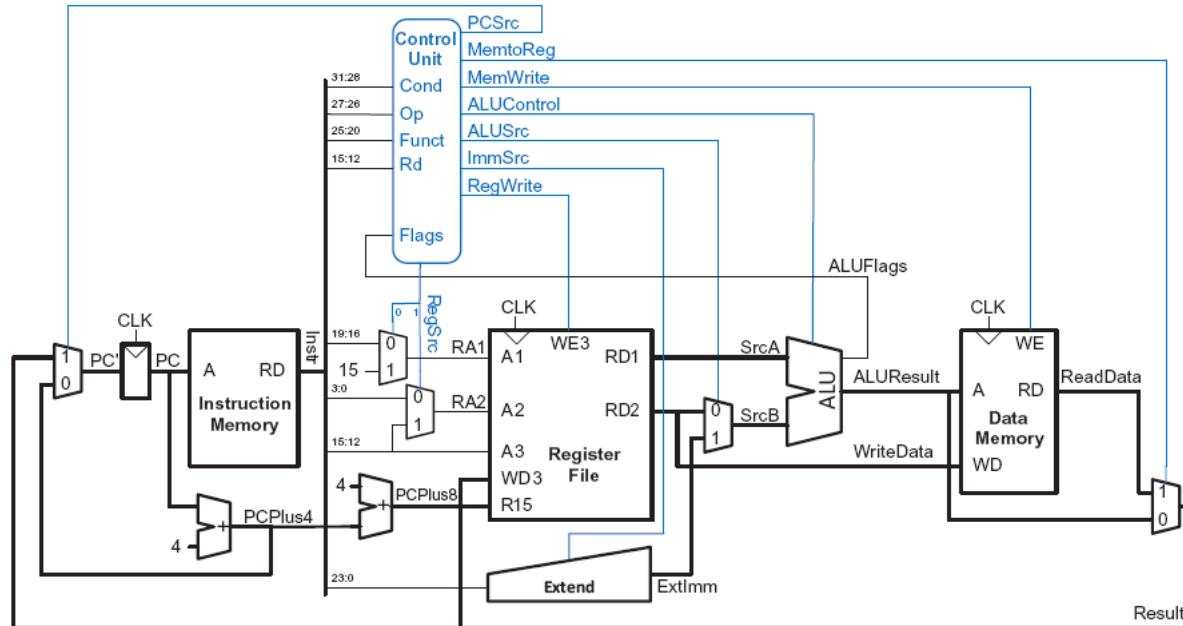
- ▶ determinare il tipo di istruzione: data processing con registro o costante, STR, LDR, o B.
- ▶ produrre i segnali di controllo adeguati per il datapath. Alcuni segnali sono inviati direttamente al datapath: **MemtoReg**, **ALUSrc**, **ImmSrc1:0**, e **RegSrc1:0**.
- ▶ generare i segnali che abilitano la scrittura (**MemW** e **RegW**), i quali devono passare attraverso la logica condizionale prima di diventare segnali datapath (**MemWrite** e **RegWrite**). Tali segnali possono essere azzerati dalla logica condizionale, se la condizione non è soddisfatta.
- ▶ generare i segnali **Branch** e **ALUOp**, utilizzati rispettivamente per indicare l'istruzione B o il tipo di istruzione data processing.

La logica per il decoder principale può essere sviluppata dalla tabella di verità utilizzando le tecniche standard per la progettazione della logica combinatoria.

- Main Decoder
- ALU Decoder
- PC Logic

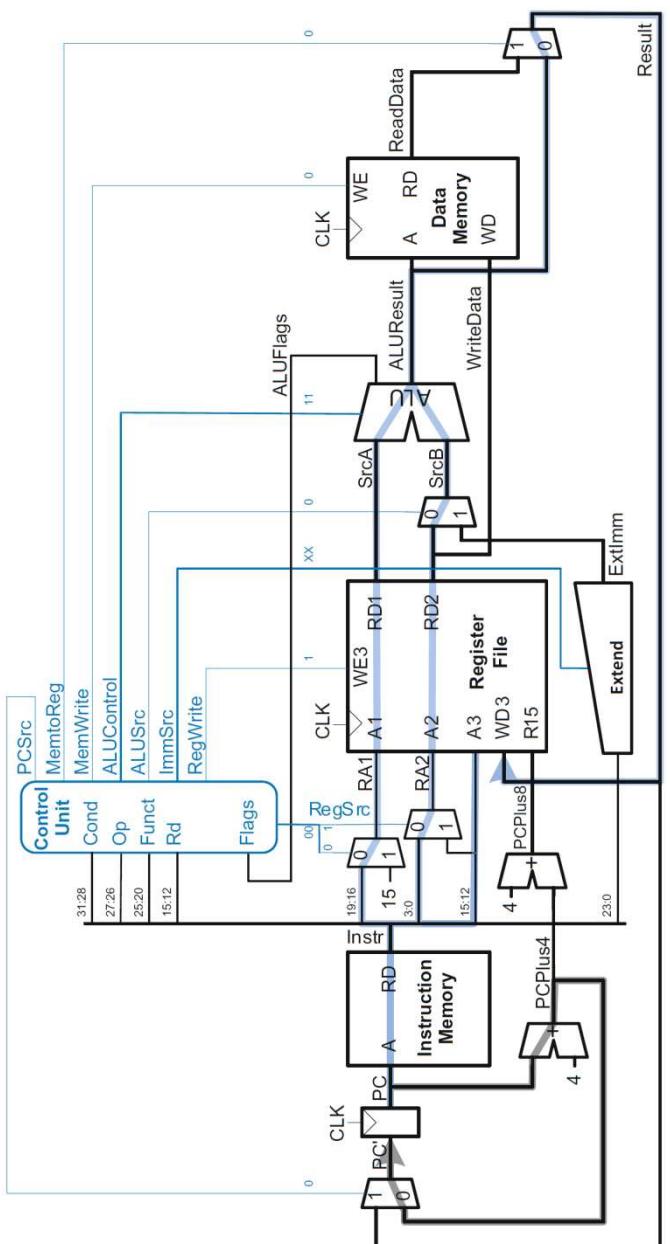


| Op | Funct <sub>5</sub> | Type | ALUOp  | RegSrc | ImmSrc | RegW | ALUSrc | MemW | MemtoReg | Branch |   |
|----|--------------------|------|--------|--------|--------|------|--------|------|----------|--------|---|
| 00 | 0                  | X    | DP Reg | 0      | 0      | 0    | 0      | XX   | 1        | 00     | 1 |
| 00 | 1                  | X    | DP Imm | 0      | 0      | 0    | 1      | 00   | 1        | X0     | 1 |
| 01 | X                  | 0    | STR    | 0      | X      | 1    | 1      | 01   | 0        | 10     | 0 |
| 01 | X                  | 1    | LDR    | 0      | 1      | 0    | 1      | 01   | 1        | X0     | 0 |
| 11 | X                  | X    | B      | 1      | 0      | 0    | 1      | 10   | 0        | X1     | 0 |

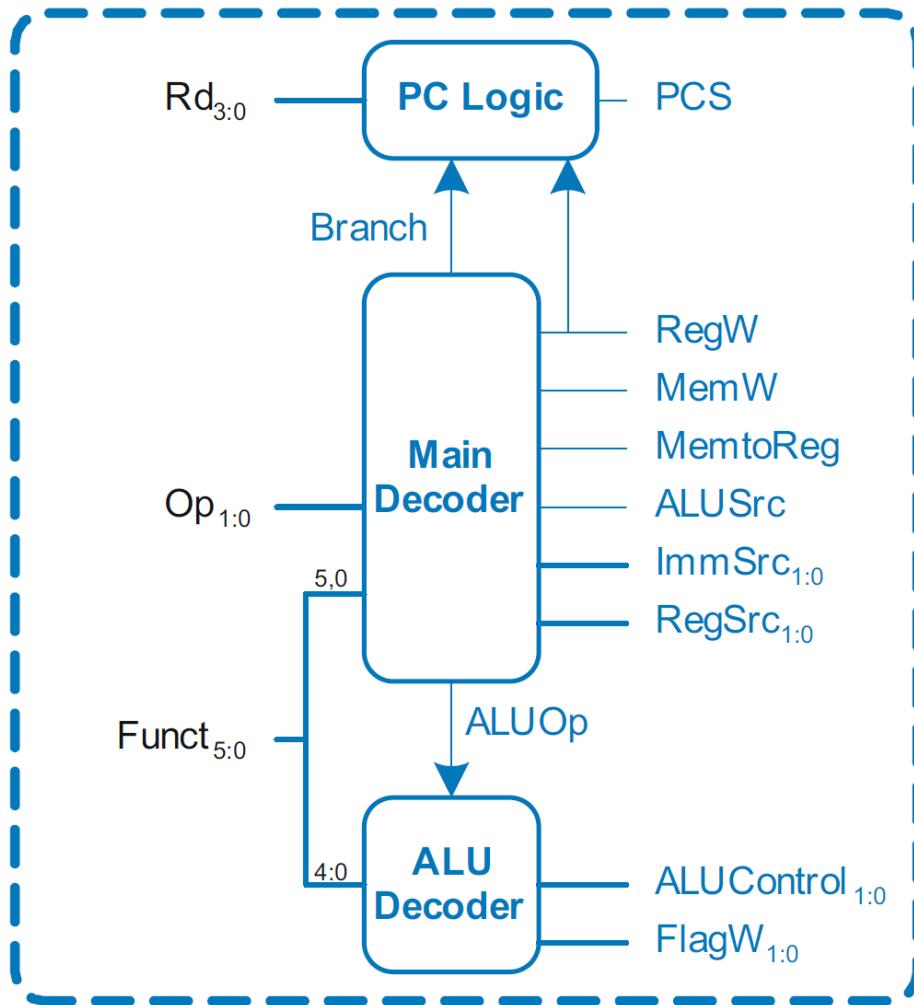


# Esempio: DPReg

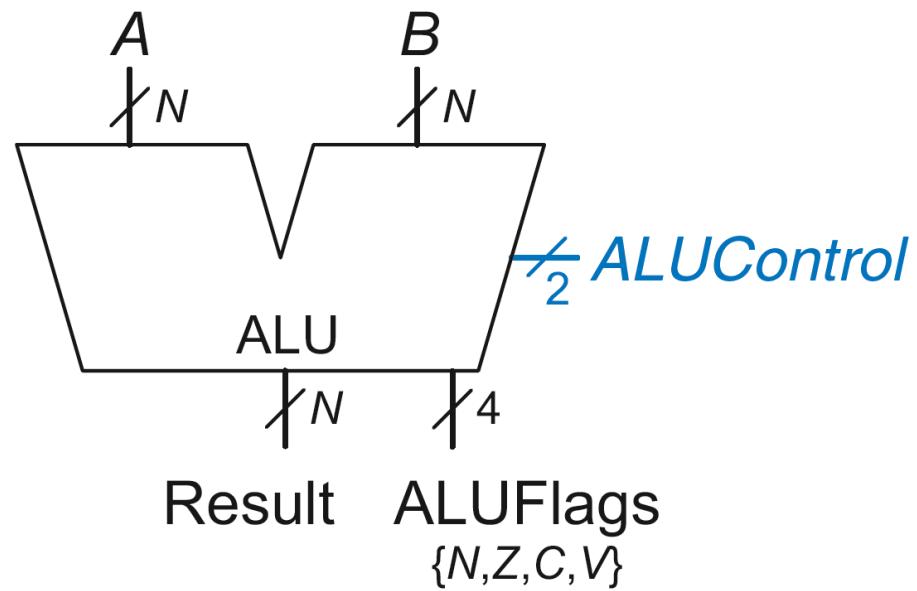
| ALUOp              |    |   |   |        |   |   |   |   |
|--------------------|----|---|---|--------|---|---|---|---|
| RegSrc             |    |   |   |        |   |   |   |   |
| RegW               |    |   |   |        |   |   |   |   |
| ImmSrc             |    |   |   |        |   |   |   |   |
| ALUSrc             |    |   |   |        |   |   |   |   |
| MemW               |    |   |   |        |   |   |   |   |
| MemtoReg           |    |   |   |        |   |   |   |   |
| Branch             |    |   |   |        |   |   |   |   |
| Type               |    |   |   |        |   |   |   |   |
| Funct <sub>0</sub> |    |   |   |        |   |   |   |   |
| Funct <sub>5</sub> |    |   |   |        |   |   |   |   |
| Op                 | 00 | 0 | X | DP Reg | 0 | 0 | 0 | 1 |



- Main Decoder
- ALU Decoder
- PC Logic



| ALUControl <sub>1:0</sub> | Function |
|---------------------------|----------|
| 00                        | Add      |
| 01                        | Subtract |
| 10                        | AND      |
| 11                        | OR       |

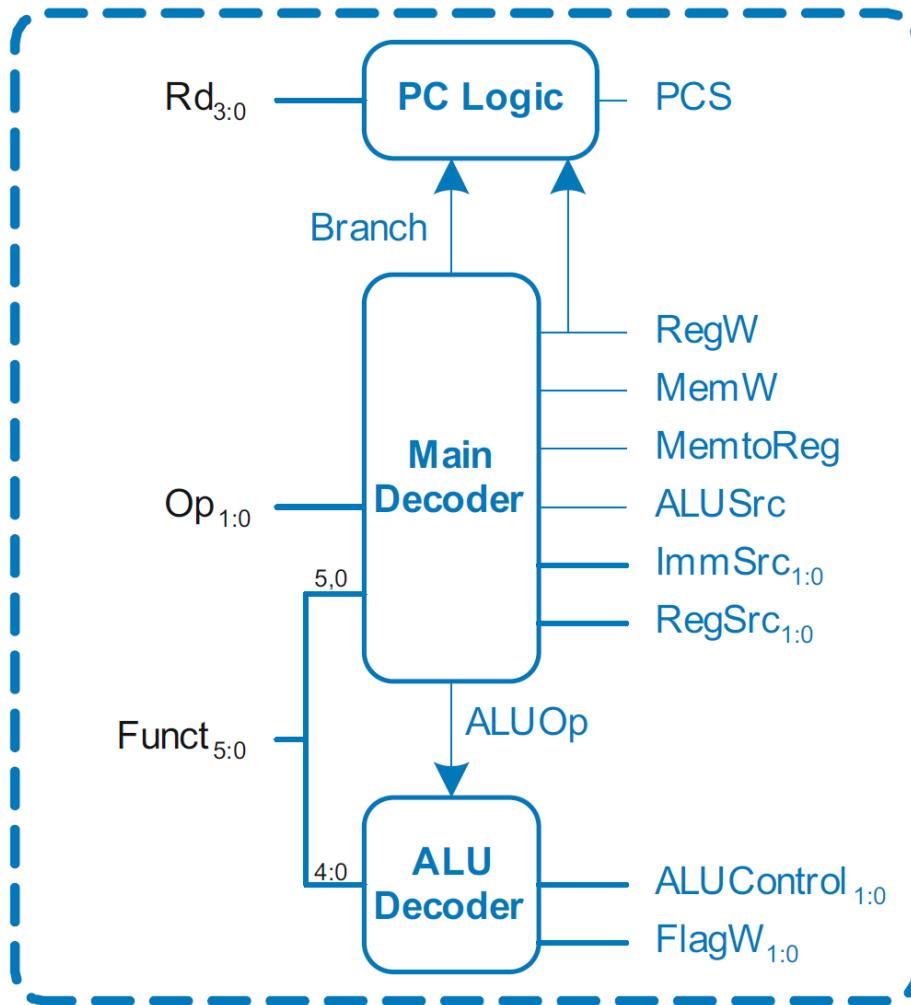


| <b>ALUOp</b> | <b>Funct<sub>4:1</sub><br/>(cmd)</b> | <b>Funct<sub>0</sub><br/>(S)</b> | <b>Type</b> | <b>ALUControl<sub>1:0</sub></b> | <b>FlagW<sub>1:0</sub></b> |
|--------------|--------------------------------------|----------------------------------|-------------|---------------------------------|----------------------------|
| 0            | X                                    | X                                | Not DP      | 00                              | 00                         |
| 1            | 0100                                 | 0                                | ADD         | 00                              | 00                         |
|              |                                      | 1                                |             |                                 | 11                         |
|              | 0010                                 | 0                                | SUB         | 01                              | 00                         |
|              |                                      | 1                                |             |                                 | 11                         |
|              | 0000                                 | 0                                | AND         | 10                              | 00                         |
|              |                                      | 1                                |             |                                 | 10                         |
|              | 1100                                 | 0                                | ORR         | 11                              | 00                         |
|              |                                      | 1                                |             |                                 | 10                         |

*FlagW<sub>1</sub>* = 1: NZ (Flags<sub>3:2</sub>) devono essere salvate

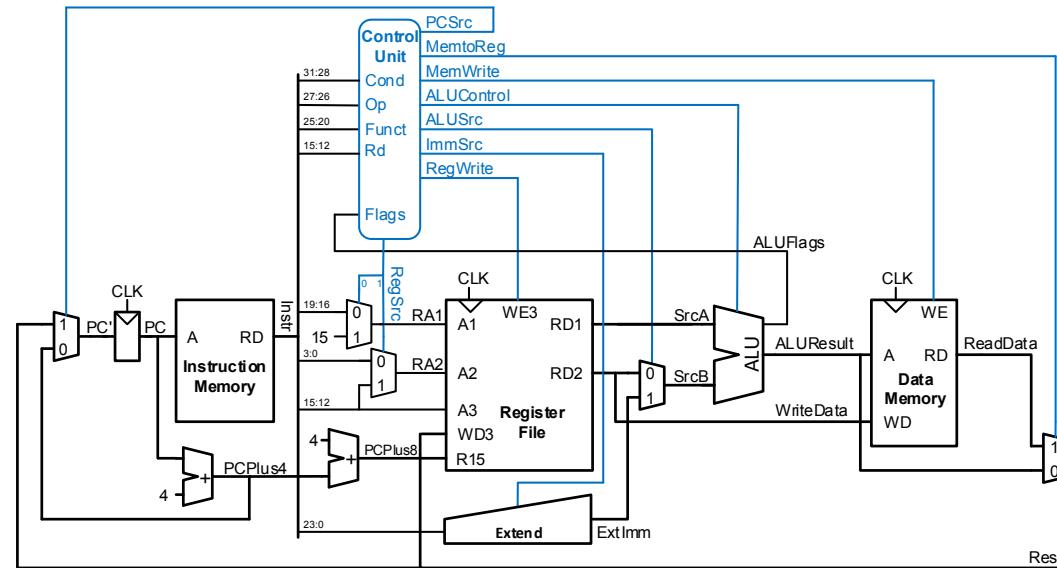
*FlagW<sub>0</sub>* = 1: CV (Flags<sub>1:0</sub>) devono essere salvate

- Main Decoder
- ALU Decoder
- **PC Logic**



La **logica del PC** controlla se l'istruzione è una scrittura in **R15** o un branch secondo la condizione:

$$PCS = ((Rd == 15) \text{ AND } RegW) \text{ OR Branch}$$

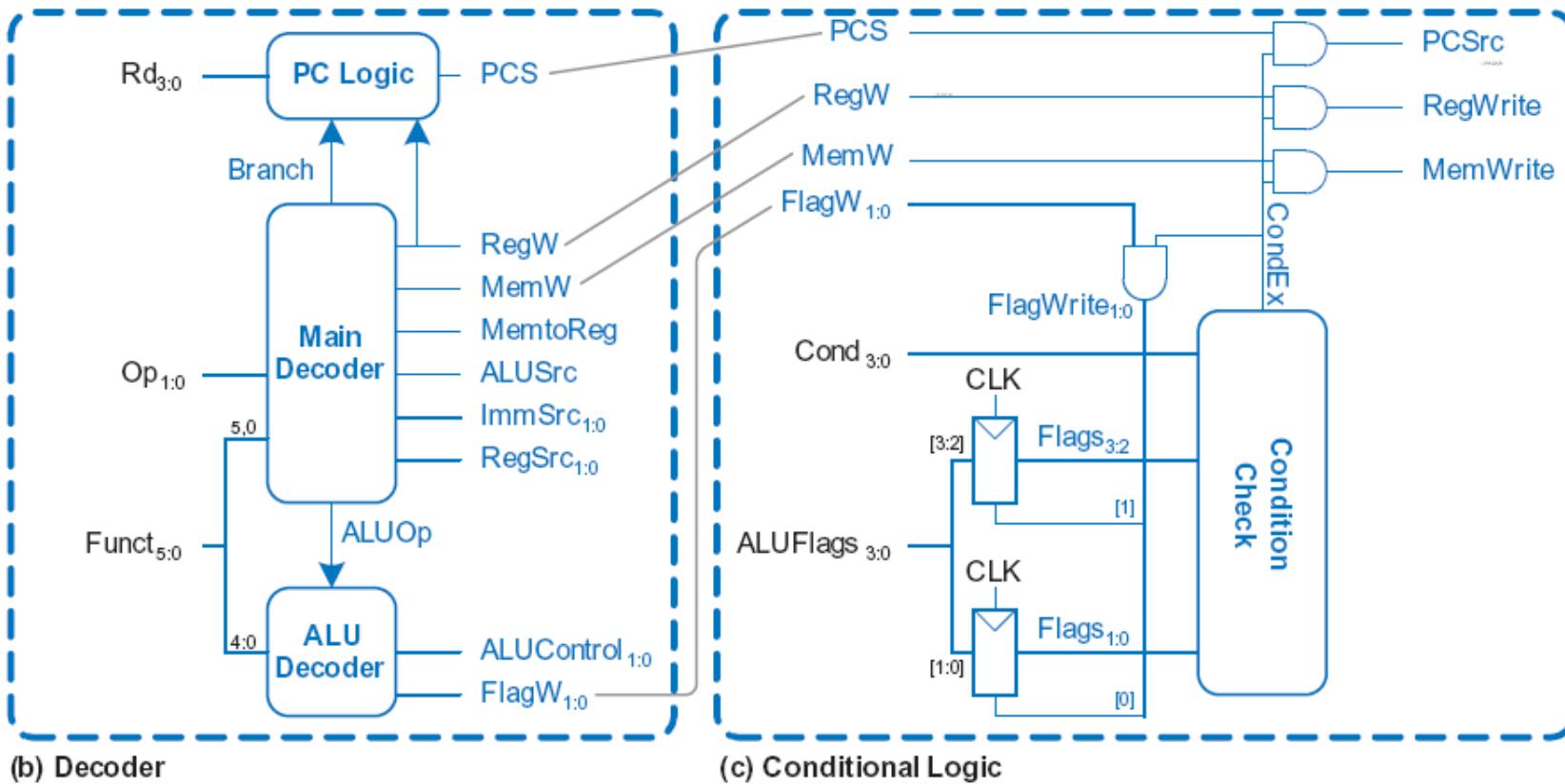


Se l'istruzione è eseguita:  $PCSrc = PCS$

Altrimenti:  $PCSrc = 0$  (e quindi,  $PC = PC + 4$ )

# Logica condizionale

I segnali che abilitano la scrittura (**MemW** and **RegW**) e l'aggiornamento dei flag (**FlagWrite**) e del PC (**PCS**) devono *passare* attraverso la logica condizionale prima di diventare operativi (e.g. segnali datapath **MemWrite**, **RegWrite** e **PCSrc**). Tali segnali possono essere azzerati dalla logica condizionale, se la condizione non è soddisfatta.



# Condizioni su flags di stato

| Mnemonic | Name                                | CondEx                       |
|----------|-------------------------------------|------------------------------|
| EQ       | Equal                               | $Z$                          |
| NE       | Not equal                           | $\bar{Z}$                    |
| CS / HS  | Carry set / Unsigned higher or same | $C$                          |
| CC / LO  | Carry clear / Unsigned lower        | $\bar{C}$                    |
| MI       | Minus / Negative                    | $N$                          |
| PL       | Plus / Positive or zero             | $\bar{N}$                    |
| VS       | Overflow / Overflow set             | $V$                          |
| VC       | No overflow / Overflow clear        | $\bar{V}$                    |
| HI       | Unsigned higher                     | $\bar{Z}C$                   |
| LS       | Unsigned lower or same              | $Z \text{ OR } \bar{C}$      |
| GE       | Signed greater than or equal        | $\overline{N \oplus V}$      |
| LT       | Signed less than                    | $N \oplus V$                 |
| GT       | Signed greater than                 | $\bar{Z}(N \oplus V)$        |
| LE       | Signed less than or equal           | $Z \text{ OR } (N \oplus V)$ |
| AL       | Always / unconditional              | ignored                      |

# Condizioni su flags di stato

Condizioni sul risultato di una operazione aritmetica in complemento a 2

| Mnemonic | Name                         | CondEx    |
|----------|------------------------------|-----------|
| MI       | Minus / Negative             | $N$       |
| PL       | Plus / Positive or zero      | $\bar{N}$ |
| VS       | Overflow / Overflow set      | $V$       |
| VC       | No overflow / Overflow clear | $\bar{V}$ |
| AL       | Always / unconditional       | ignored   |

# Condizioni su flags di stato

Per confrontare due interi (signed o unsigned) A e B si esegue la differenza A-B e si verificano le seguenti condizioni sui flag

| Mnemonic | Name                                | CondEx                           |
|----------|-------------------------------------|----------------------------------|
| EQ       | Equal                               | $Z$                              |
| NE       | Not equal                           | $\bar{Z}$                        |
| CS / HS  | Carry set / Unsigned higher or same | $C$                              |
| CC / LO  | Carry clear / Unsigned lower        | $\bar{C}$                        |
| HI       | Unsigned higher                     | $\bar{Z}C$                       |
| LS       | Unsigned lower or same              | $Z \text{ OR } \bar{C}$          |
| GE       | Signed greater than or equal        | $\overline{N \oplus V}$          |
| LT       | Signed less than                    | $N \oplus V$                     |
| GT       | Signed greater than                 | $\bar{Z}(\overline{N \oplus V})$ |
| LE       | Signed less than or equal           | $Z \text{ OR } (N \oplus V)$     |

# Condizioni su flags di stato

| Mnemonic | Name      | CondEx    |
|----------|-----------|-----------|
| EQ       | Equal     | Z         |
| NE       | Not equal | $\bar{Z}$ |

$A == B$  sse  $A - B == 0$  sse  $Z = 1$

$A \neq B$  sse  $A - B \neq 0$  sse  $Z = 0$

Vale sia per la rappresentazione in complemento a 2 (interi con segno) che nella rappresentazione senza segno

# Condizioni su flags di stato

| Mnemonic | Name                         | CondEx                       |
|----------|------------------------------|------------------------------|
| GE       | Signed greater than or equal | $\bar{N} \oplus V$           |
| LT       | Signed less than             | $N \oplus V$                 |
| GT       | Signed greater than          | $\bar{Z}(\bar{N} \oplus V)$  |
| LE       | Signed less than or equal    | $Z \text{ OR } (N \oplus V)$ |

#

$A < B \text{ sse } A - B < 0 \text{ sse } (N \oplus V)$

- Quando  $A - B$  non genera overflow allora deve essere negativo
  - Questo accade sempre quando A e B hanno lo stesso segno
- Quando  $A - B$  genera overflow?
  - Allora A e B hanno segni diversi
  - $A < B$  è vera sse A negativo e B positivo sse A negativo e  $(-B)$  negativo, e quindi, essendoci overflow il risultato deve essere positive (ovvero  $N=0$ )

$A \leq B \text{ sse } A - B < 0 \text{ or } A - B == 0 \text{ sse } Z + (N \oplus V)$

# Condizioni su flags di stato

| Mnemonic | Name                                | CondEx                  |
|----------|-------------------------------------|-------------------------|
| CS / HS  | Carry set / Unsigned higher or same | $C$                     |
| CC / LO  | Carry clear / Unsigned lower        | $\bar{C}$               |
| HI       | Unsigned higher                     | $\bar{Z}C$              |
| LS       | Unsigned lower or same              | $Z \text{ OR } \bar{C}$ |

Problema A-B che semantica ha nella rappresentazione senza segno?

- Sappiamo che  $A-B$  è in complemento a due  $A+(\sim B+1)$ , dove  $\sim B$  è la negazione bit a bit di  $B$
- Nella rappresentazione senza segno invece
  - $\sim(X_{N-1}\dots X_0) = (1-X_{N-1})\dots(1-X_0) = 1\dots 1 - (X_{N-1}\dots X_0) = 2^N - 1 - B$

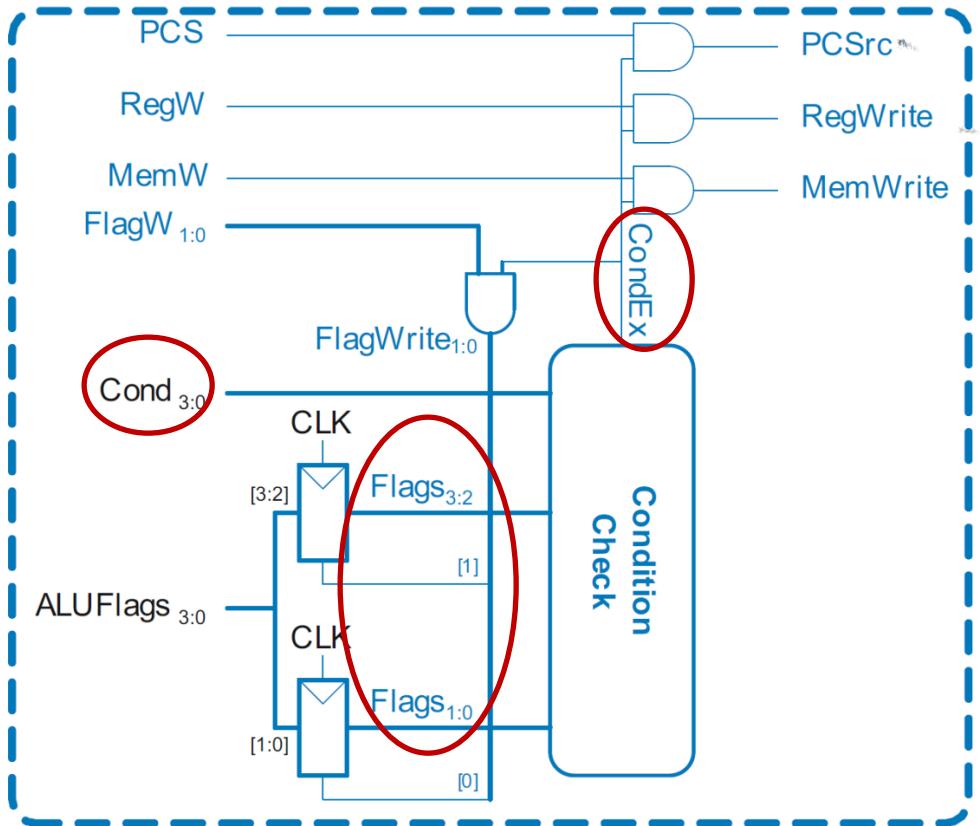
|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 |

# Condizioni su flags di stato

- Quindi  $A + (\sim B + 1)$  corrisponde nella rappresentazione senza segno a  $2^N + A - B$
- Ora  $2^N + A - B$  non genera un carry out sse  $2^N + A - B \leq 2^N - 1$  sse  $A + 1 \leq B$  sse  $A < B$
- Quindi  $A < B$  è verificato dalla condizione  $\bar{C}$
- $A \leq B$  sse  $A < B$  o  $A == B$  sse  $Z + \bar{C}$
- $A > B$  sse  $!(A \leq B)$  sse  $\bar{Z}C$

# Esempi

$\text{Flags}_{3:0} = \text{NZCV}$

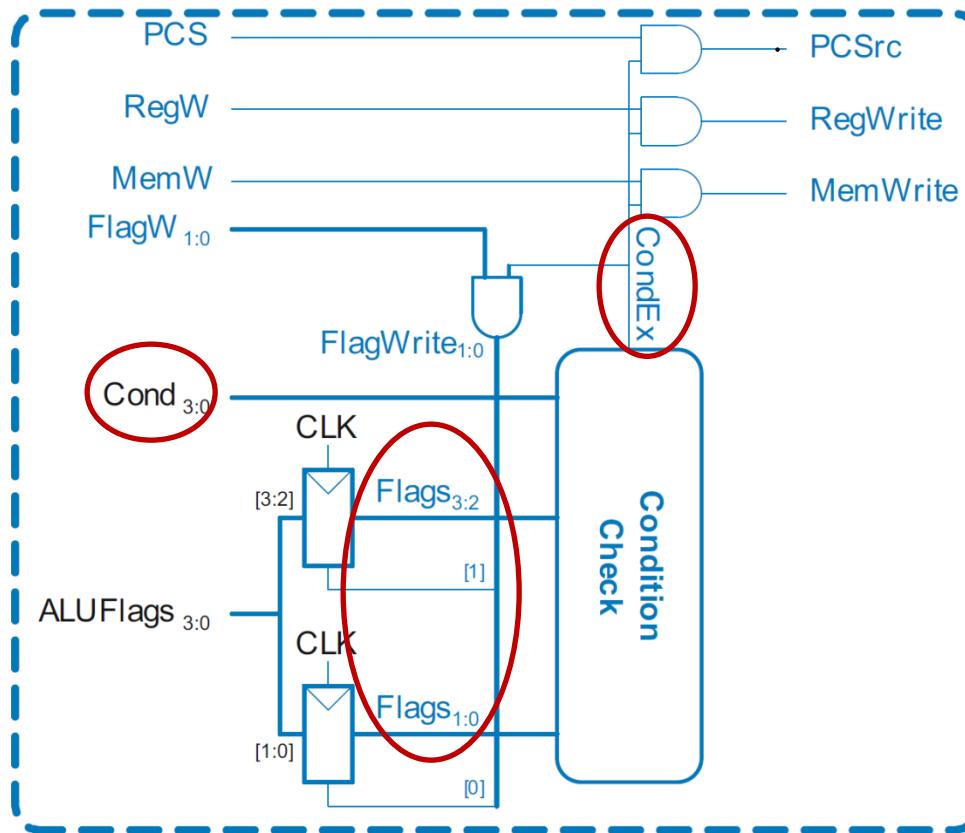


AND R1, R2, R3

Cond<sub>3:0</sub>=1110 (istruzione non condizionata) =>  
CondEx = 1

# Esempi

$\text{Flags}_{3:0} = \text{NZCV}$

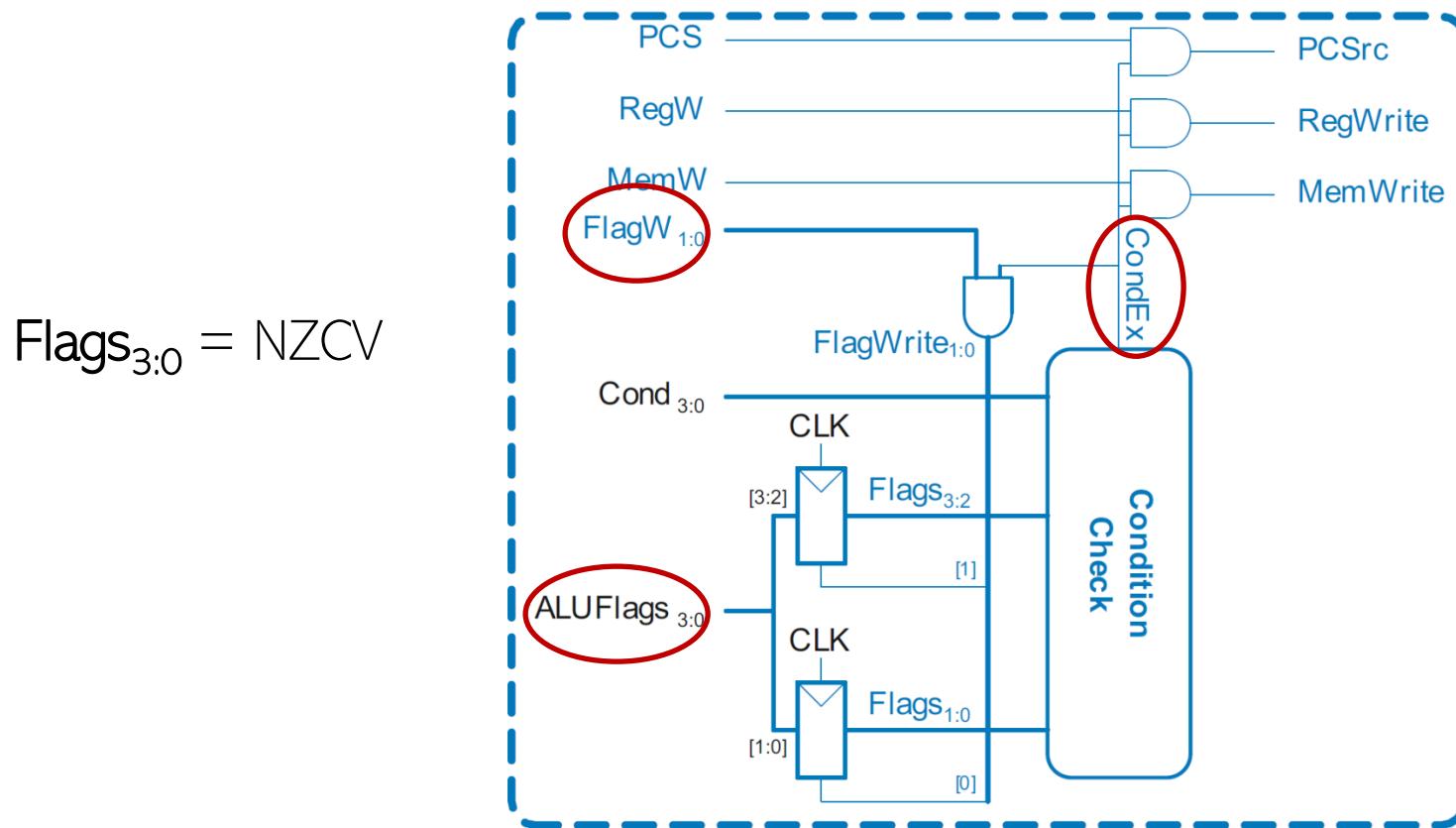


EOREQ R5, R6, R7

Cond<sub>3:0</sub>=0000 (EQ): if Flags<sub>3:2</sub>=X1XX

$\Rightarrow \text{CondEx} = 1$

# Update dei flags di stato

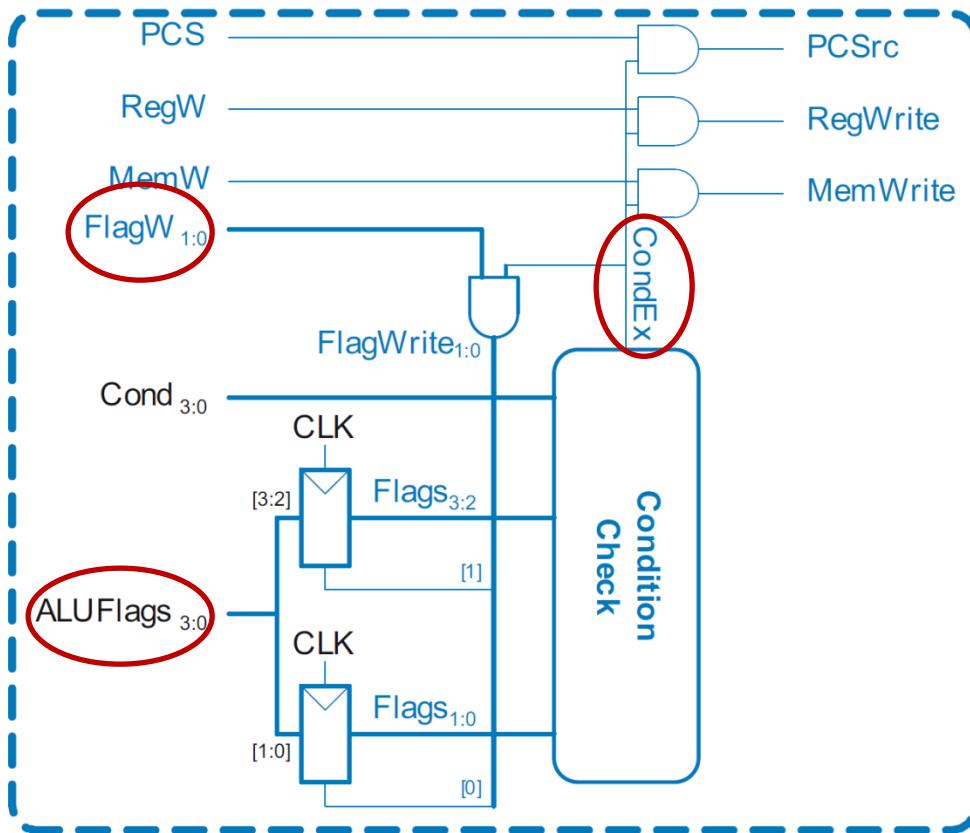


$$\text{Flags}_{3:0} = \text{NZCV}$$

$\text{Flags}_{3:0}$  vengono aggiornati con i valori di  $\text{ALUFlags}_{3:0}$  se si verificano le seguenti condizioni:

- $\text{FlagW}$  è 1 (es. S-bit dell'istruzione corrente è 1)
- $\text{CondEx}$  è 1 (l'istruzione deve essere eseguita)

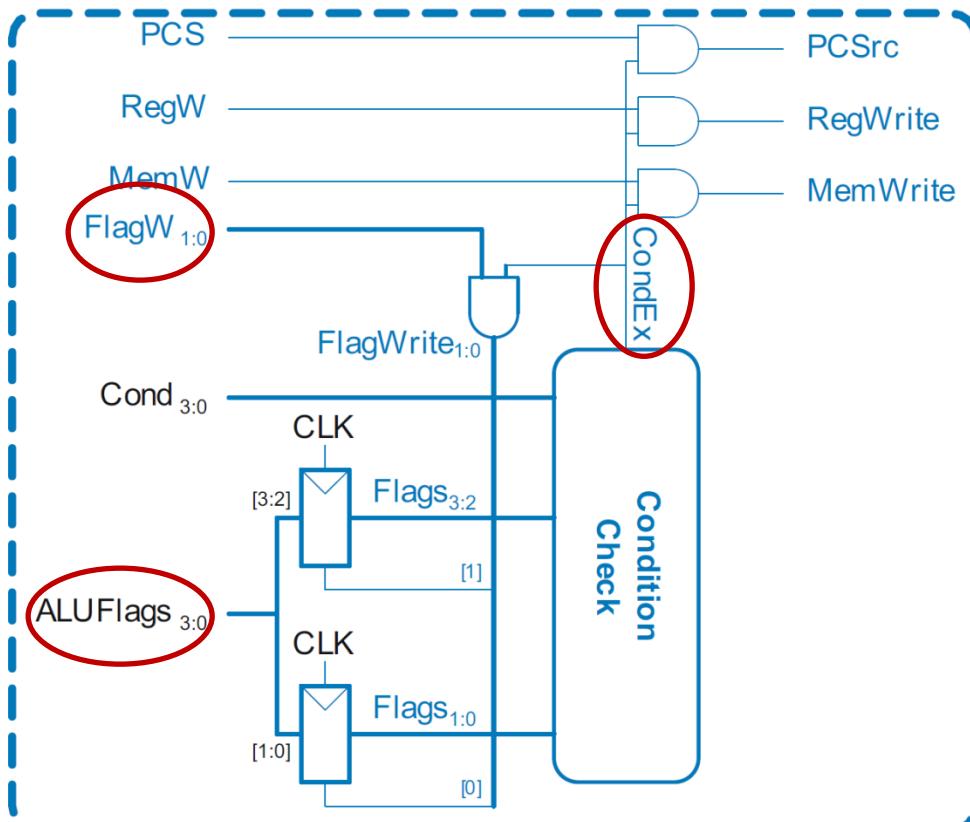
# Esempi



SUBS R5, R6, R7

FlagW<sub>1:0</sub> = 11 e CondEx = 1 (istruzione incondizionata) =>  
FlagWrite<sub>1:0</sub> = 11      Tutti i flag vengono aggiornati

# Esempi



ANDS R7, R1, R3

FlagW<sub>1:0</sub> = 10 E CondEx = 1 (istruzione incondizionata) =>

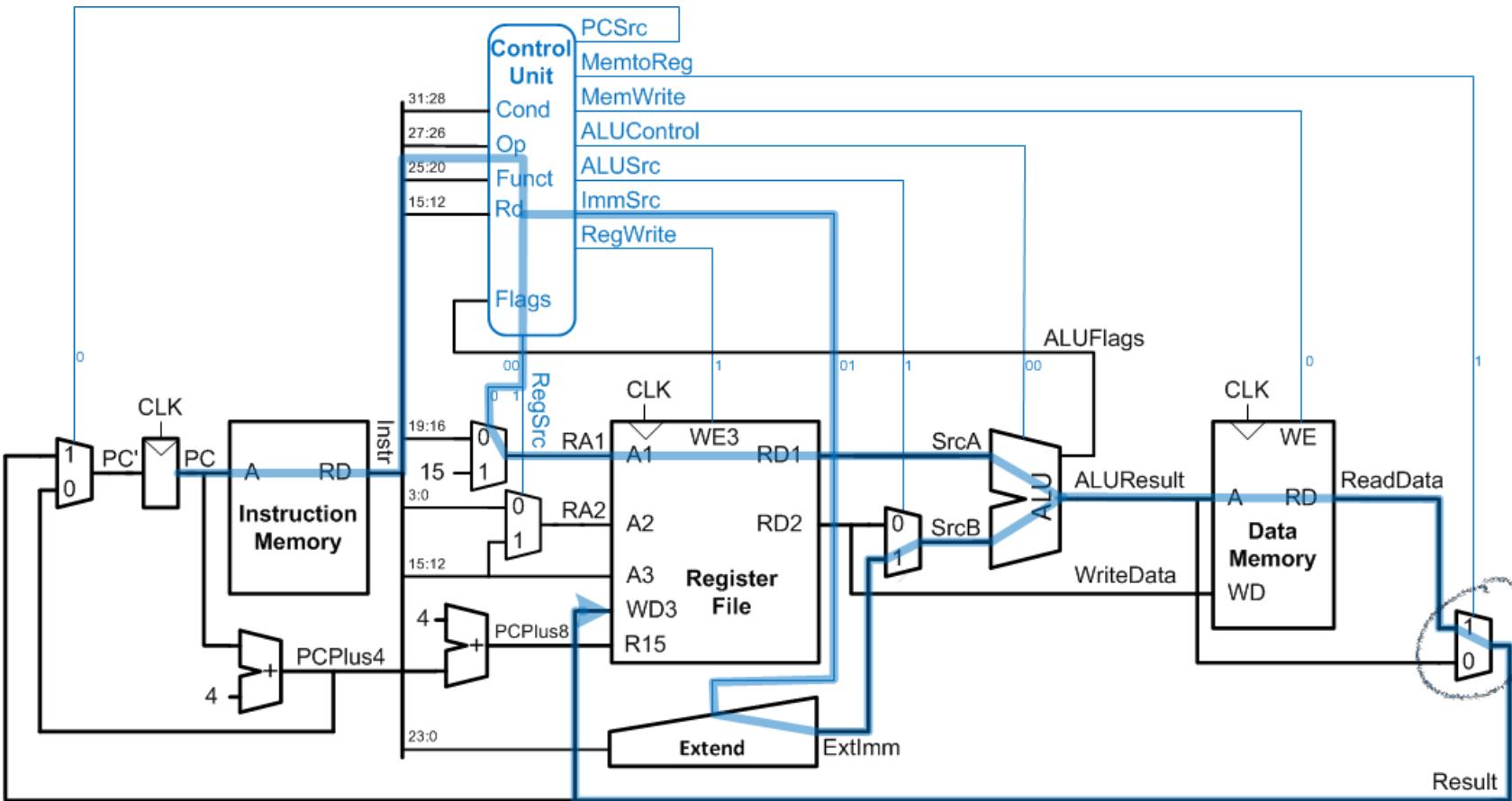
FlagWrite<sub>1:0</sub> = 10 Only Flags<sub>3:2</sub> sono aggiornati

# Analisi delle prestazioni

Ogni istruzione nel processore a ciclo singolo impiega un ciclo di clock, quindi il CPI è 1.

I critical path per l'istruzione LDR sono:

- ▶ ( $t_{pcq\_PC}$ ) – caricamento di un nuovo indirizzo (PC) sul fronte di salita del clock;
- ▶ ( $t_{mem}$ ) – lettura dell'istruzione in memoria;
- ▶ ( $t_{dec}$ ) – il Decoder principale calcola RegSrc0, che induce il multiplexer a scegliere Instr<sub>19:16</sub> come RA1, e il register file legge questo registro come srcA;
- ▶ ( $\max[t_{mux} + t_{RFread}, t_{ext} + t_{mux}]$ ) – mentre il register file viene letto, il campo costante viene esteso e viene selezionata dal multiplexer ALUSrc per determinare srcB.
- ▶ ( $t_{ALU}$ ) – l'ALU somma srcA e srcB per trovare l'indirizzo effettivo.
- ▶ ( $t_{mem}$ ) – La memoria di dati legge da questo indirizzo.
- ▶ ( $t_{mux}$ ) – il multiplexer MemtoReg seleziona ReadData.
- ▶ ( $t_{RFsetup}$ ) – viene impostato il segnale Result ed il risultato viene scritto nel register file.



$T_c$  limited by critical path (LDR)

# Analisi delle prestazioni

Il tempo totale è dato dalla somma dei parziali:

$$T_{c1} = t_{pcq\_PC} + t_{mem} + t_{dec} + \max[t_{mux} + t_{RFread}, t_{ext} + t_{mux}] + t_{ALU} + t_{mem} + t_{mux} + t_{RFsetup};$$

Nella maggior parte delle implementazioni, l'ALU, la memoria ed il register file sono sostanzialmente più lenti di altri blocchi combinatori. Pertanto, il tempo di ciclo può essere semplificato come:

$$T_{c1} = t_{pcq\_PC} + 2t_{mem} + t_{dec} + t_{RFread} + t_{ALU} + 2t_{mux} + t_{RFsetup};$$

Table 7.5 Delay of circuit elements

| Element             | Parameter     | Delay (ps) |
|---------------------|---------------|------------|
| Register clk-to-Q   | $t_{pcq}$     | 40         |
| Register setup      | $t_{setup}$   | 50         |
| Multiplexer         | $t_{mux}$     | 25         |
| ALU                 | $t_{ALU}$     | 120        |
| Decoder             | $t_{dec}$     | 70         |
| Memory read         | $t_{mem}$     | 200        |
| Register file read  | $t_{RFread}$  | 100        |
| Register file setup | $t_{RFsetup}$ | 60         |

# Analisi delle prestazioni

**Domanda:** qual è il tempo di esecuzione per un programma con 100 miliardi di istruzioni?

**Risposta:**

secondo l'equazione

$$T_{c1} = t_{pcq\_PC} + 2t_{mem} + t_{dec} + t_{RFread} + t_{ALU} + 2t_{mux} + t_{RFsetup}$$

il tempo di ciclo del processore singolo ciclo è

$$\begin{aligned} T_{c1} &= 40 + 2(200) + 70 + 100 + 120 + 2(25) \\ &= 840 \text{ ps.} \end{aligned}$$

Secondo l'equazione

$$\text{Tempo di esecuzione} = (\# \text{istruzioni}) \left( \frac{\text{cicli}}{\text{istruzione}} \right) \left( \frac{\text{secondi}}{\text{ciclo}} \right)$$

il tempo di esecuzione totale è

$$T_1 = (100 \times 10^9 \text{ istruzioni}) (1 \text{ ciclo / di istruzione}) (840 \times 10^{-12} \text{ s / ciclo}) = 84 \text{ secondi.}$$

Table 7.5 Delay of circuit elements

| Element             | Parameter     | Delay (ps) |
|---------------------|---------------|------------|
| Register clk-to-Q   | $t_{pcq}$     | 40         |
| Register setup      | $t_{setup}$   | 50         |
| Multiplexer         | $t_{mux}$     | 25         |
| ALU                 | $t_{ALU}$     | 120        |
| Decoder             | $t_{dec}$     | 70         |
| Memory read         | $t_{mem}$     | 200        |
| Register file read  | $t_{RFread}$  | 100        |
| Register file setup | $t_{RFsetup}$ | 60         |

# Limiti del ciclo singolo

Le architetture a ciclo singolo hanno tre principali limiti:

- ▶ **separazione delle memorie**: la memoria istruzioni e la memoria dati devono essere necessariamente separate, poiché dati e istruzioni devono essere gestiti all'interno dello stesso ciclo.
- ▶ **inefficienza temporale**: il ciclo di clock deve avere una durata pari al tempo impiegato dall'istruzione più lenta, sprecando tempo per tutte quelle istruzioni molto più veloci.
- ▶ **duplicazione delle componenti**: lo stesso componente non può essere riutilizzato per scopi distinti; ad esempio sono necessarie tre ALU, due per la gestione del PC e una per l'esecuzione delle istruzioni.

# Vantaggi del ciclo multiplo

Le architetture a ciclo multiplo risolvono tali problemi, partizionando una intera istruzione in più passi, ciascuno dei quali viene eseguito in un ciclo di clock differente.

- ▶ È possibile utilizzare una sola memoria comune sia per le istruzioni, che per i dati. Infatti, l'istruzione viene letta in un ciclo, mentre i dati vengono letti o scritti in memoria in un ciclo differente.
- ▶ Istruzioni meno complesse richiedono un minor numero di cicli di clock, evitando sprechi di tempo.
- ▶ È possibile utilizzare un'unica ALU sia per gestire il PC, che per eseguire le istruzioni, purché tali operazioni siano effettuate in cicli di clock differenti.

# Datapath

Il **datapath** è sviluppato in modo incrementale, i nuovi collegamenti sono evidenziati in **nero** (o **blu**), mentre quanto già introdotto è rappresentato in **grigio**.

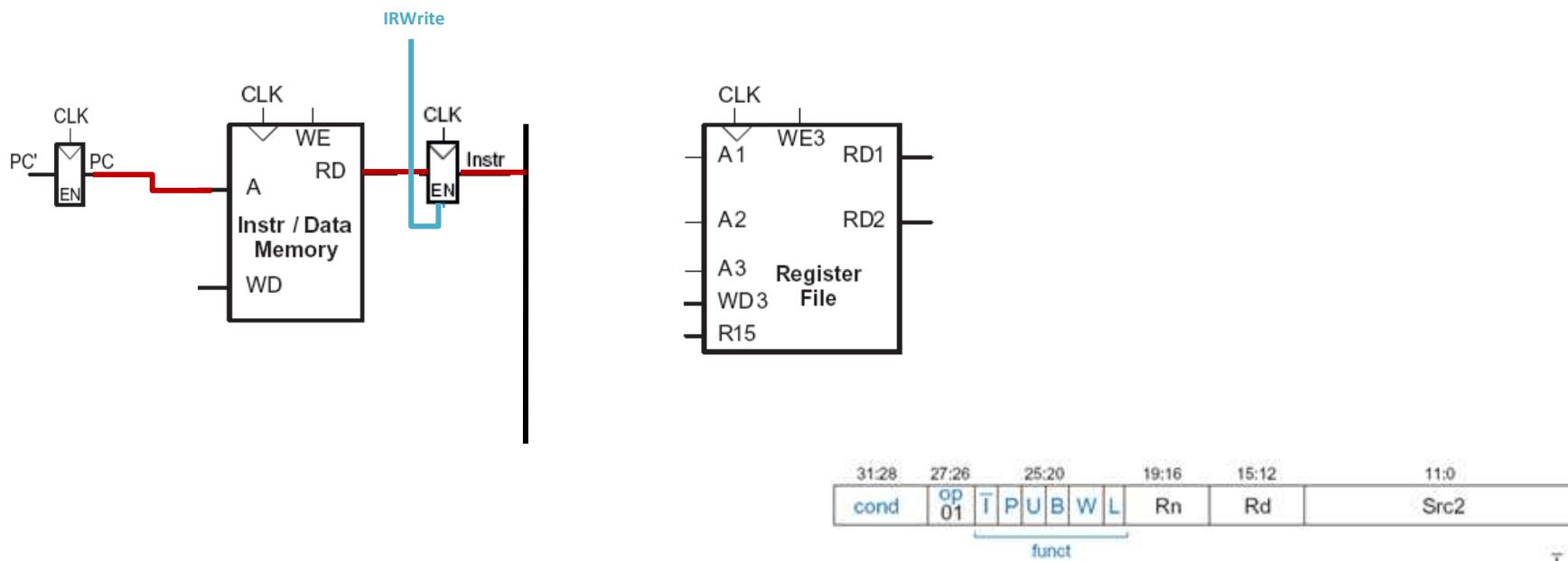
Al fine di facilitare la comprensione dell'architettura di un processore ARM, considereremo un limitato set di istruzioni:

- ▶ le istruzioni di elaborazione dati: **ADD, SUB, AND, ORR** (con registro e modalità di indirizzamento diretto e senza shift);
- ▶ le istruzioni di Memoria: **LDR, STR** (diretto e con offset positivo);
- ▶ le istruzioni di salto (branch): **B**.

# Datapath LDR

Il **PC** contiene l'indirizzo dell'istruzione da eseguire. Il primo passo è quello di leggere questa istruzione dalla memoria istruzioni, per cui il PC viene collegato all'indirizzo di ingresso della memoria.

L'istruzione a 32 bit viene letta e memorizzata in un registro **IR**. Il registro **IR** riceve un segnale **IRWrite** che indica quando caricare una istruzione.

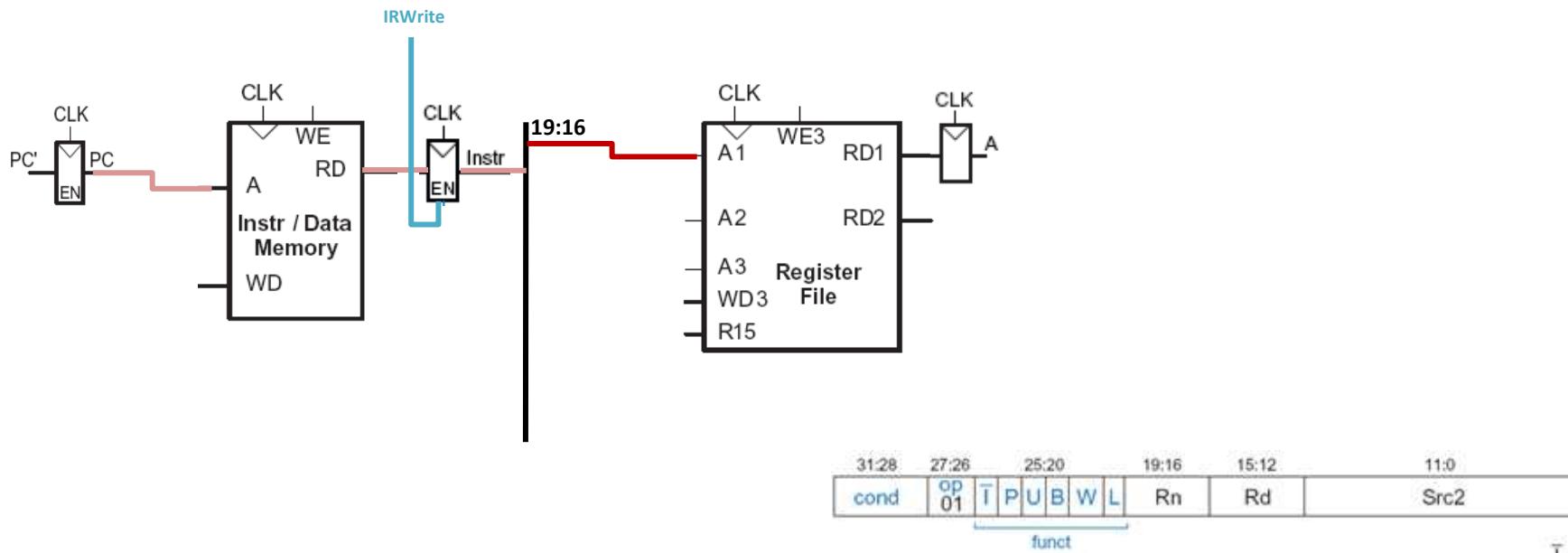


# Datapath LDR

Il passo successivo è quello di leggere il registro sorgente contenente l'indirizzo di base. Questo registro è specificato nel campo **Rn** dell'istruzione, **Instr<sub>19:16</sub>**

Questi bit vengono collegati all'ingresso indirizzo di una delle porte del file register (**A1**).

Il register file legge il valore di registro in **RD1** e lo memorizza in un registro **A**.



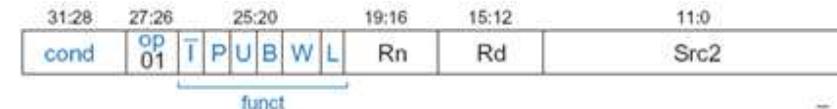
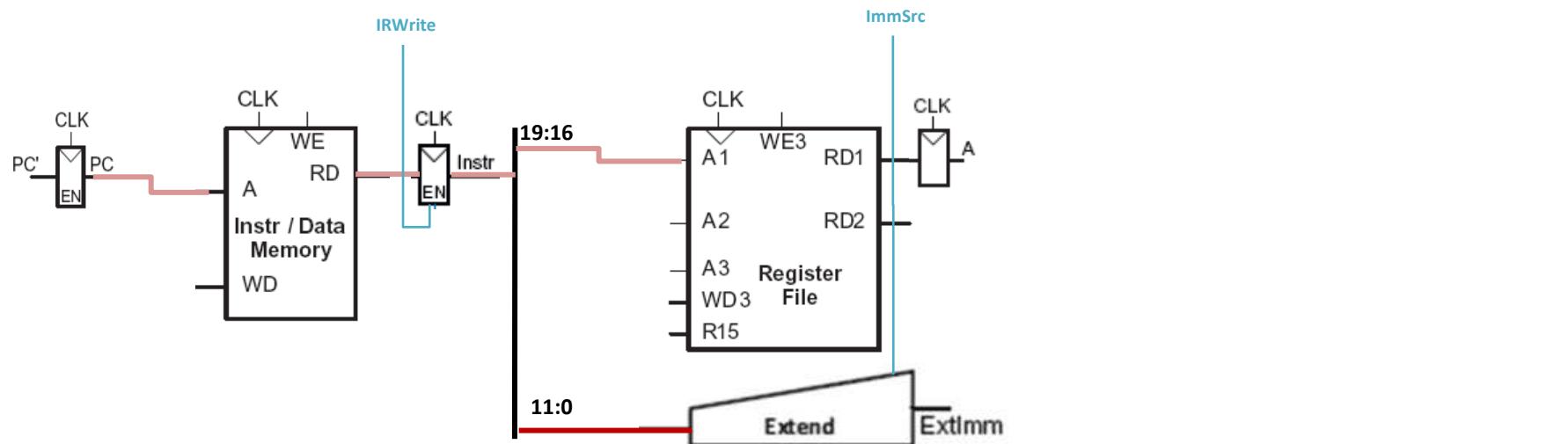
# Datapath LDR

L'istruzione **LDR** richiede anche un **offset**, il quale è memorizzato nell'istruzione stessa e corrisponde ai bit  $\text{Instr}_{11:0}$ .

L'offset è un valore senza segno, quindi deve essere esteso a 32 bit.

Il valore a 32 bit (**ExtImm**) è tale che  $\text{ExtImm}_{31:12} = 0$  e  $\text{ExtImm}_{11:0} = \text{Instr}_{11:0}$ .

**ExtImm** estende a 32 bit costanti a 8, 12 e 24 bit. Non viene memorizzato in un registro, poiché dipende solo da **Instr**, che non cambia durante l'esecuzione dell'istruzione.

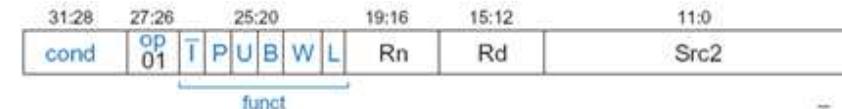
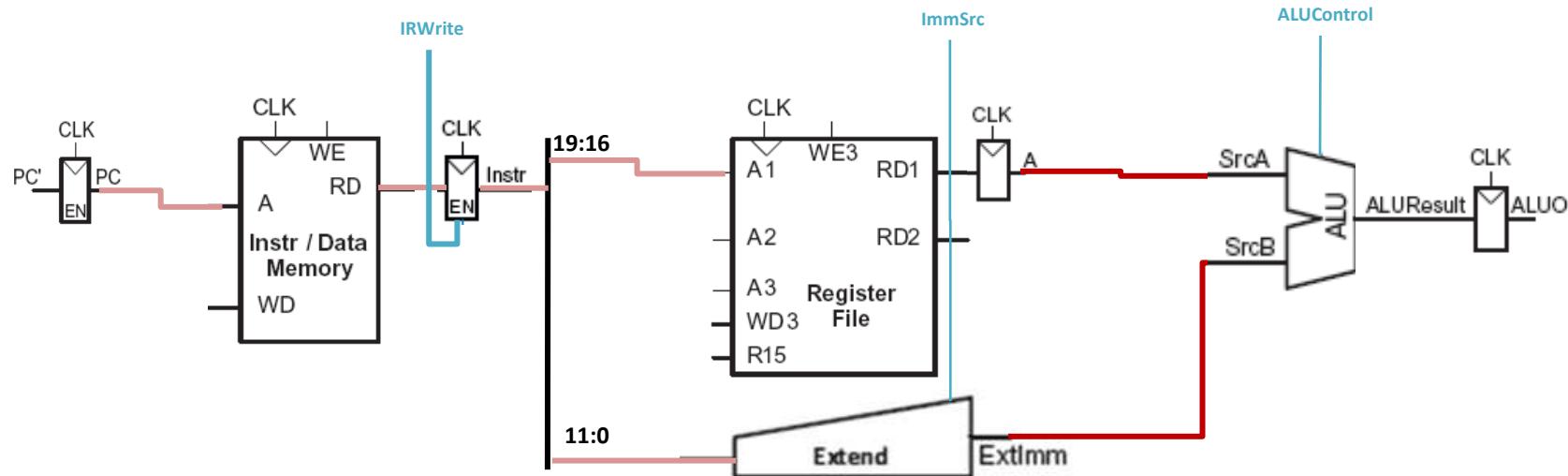


# Datapath LDR

Il processore deve aggiungere l'**indirizzo di base** all'offset per trovare l'indirizzo di memoria a cui leggere. La somma è effettuata per mezzo di una **ALU**.

La **ALU** riceve due operandi (**srcA** e **srcB**). **srcA** proviene dal register file, mentre **srcB** da ExtImm. Inoltre, il segnale a 2-bit **ALUControl** specifica l'operazione (00 per somma, 01 sottrazione).

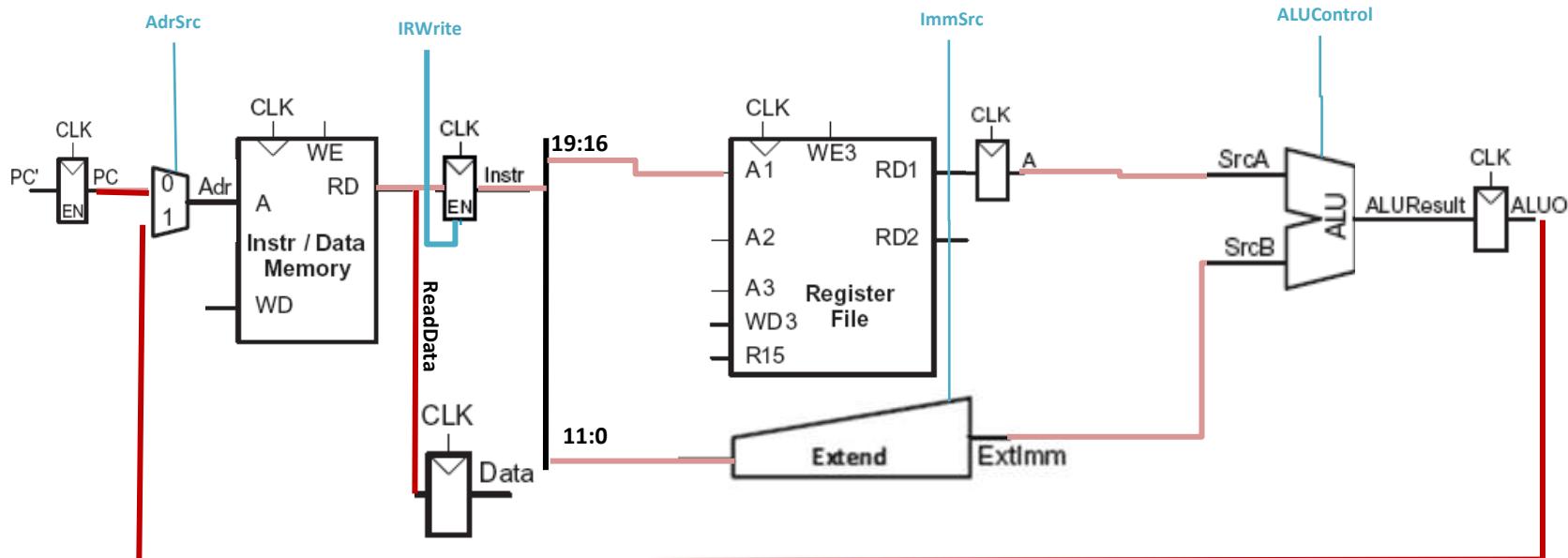
La **ALU** genera un valore a 32 bit **ALUResult**, che viene memorizzato in un registro **ALUOut**.



# Datapath LDR

L'indirizzo calcolato dall'**ALU** deve essere inviato alla porta **A** della memoria. Serve un multiplexer per disambiguare l'accesso con **ALUOut** o **PC**. Il multiplexer è controllato dal segnale **AdrSrc**.

I dati vengono letti dalla memoria dati sul bus **ReadData**, e poi vengono memorizzati in un registro chiamato **Data**.



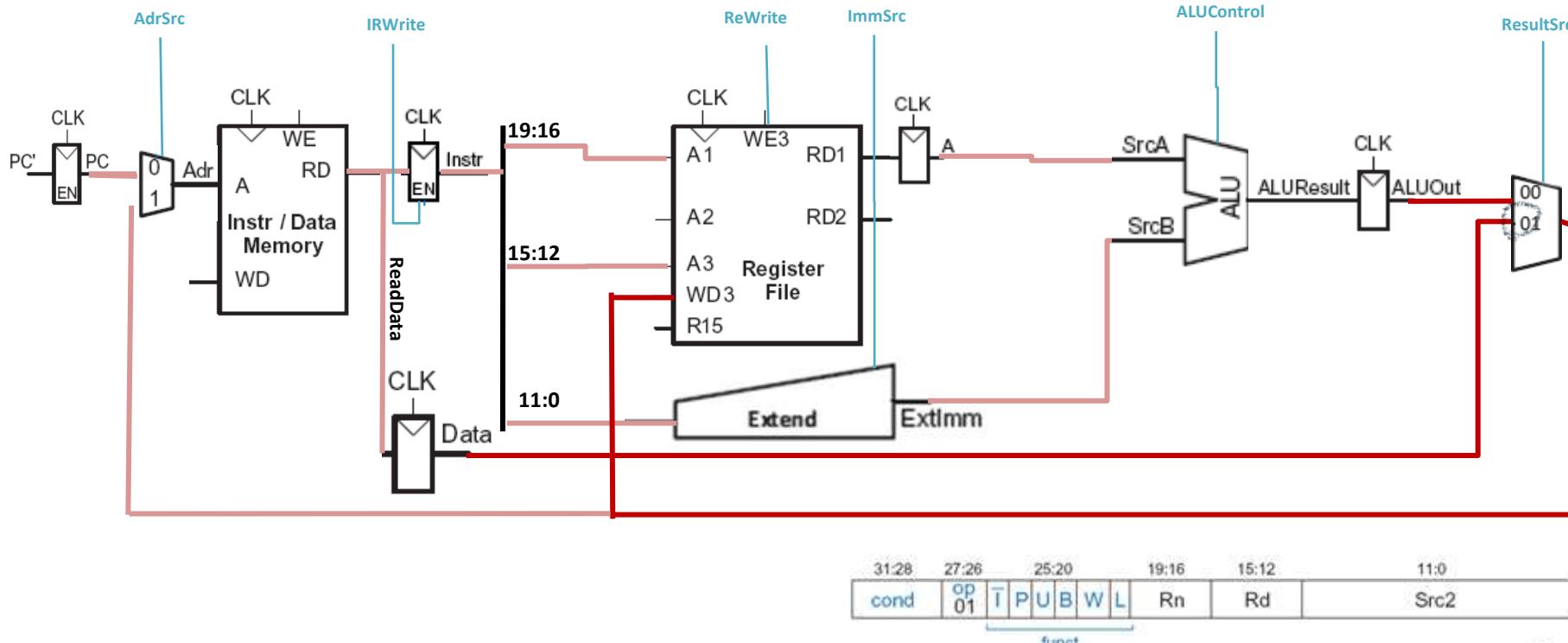
| 31:28 | 27:26    | 25:20 | 19:16 | 15:12 | 11:0 |
|-------|----------|-------|-------|-------|------|
| cond  | op<br>01 | T     | PUBW  | L     | Rn   |

# Datapath LDR

Inoltre, i dati appena letti devono essere scritti nel registro **Rd** specificato dai bit **15:12** dell'istruzione **Instr.**

Piuttosto che collegare direttamente **Data** alla porta **WD3**, si consideri che anche il risultato dell'ALU potrebbe dover essere scritto in **Rd**. Quindi aggiungiamo un multiplexer che seleziona fra **ALUOut** e **Data**.

Il segnale **RegWrite** deve essere impostato a **1**, per permettere la scrittura nel registro.

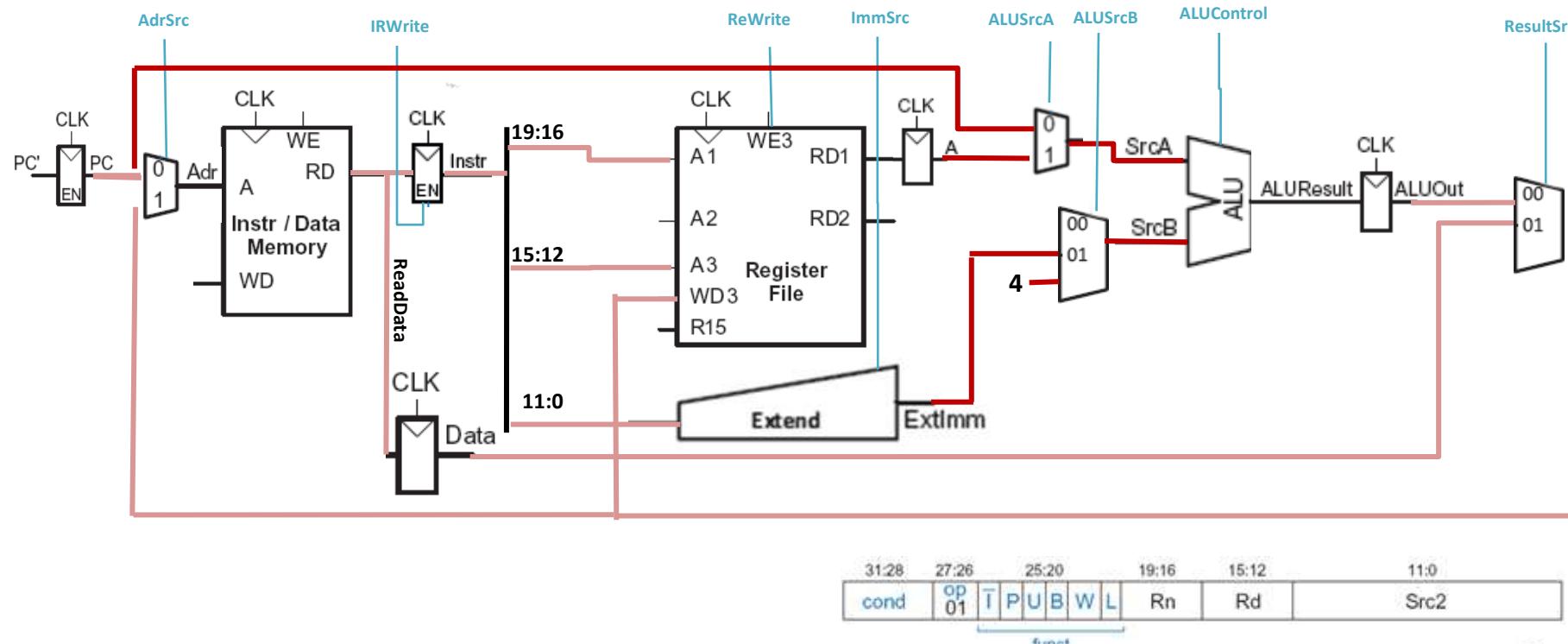


# Datapath LDR

Un ulteriore compito a carico dell'**ALU** è l'incremento del **PC**, operazione che prima era svolta da una ALU diversa.

Aggiungiamo un multiplexer sul primo ingresso dell'**ALU**, che permette di scegliere fra il contenuto del registro **A** e il **PC**.

Sul secondo ingresso dell'ALU aggiungiamo un ulteriore multiplexer che permetta di selezionare fra ExtImm e la costante 4.

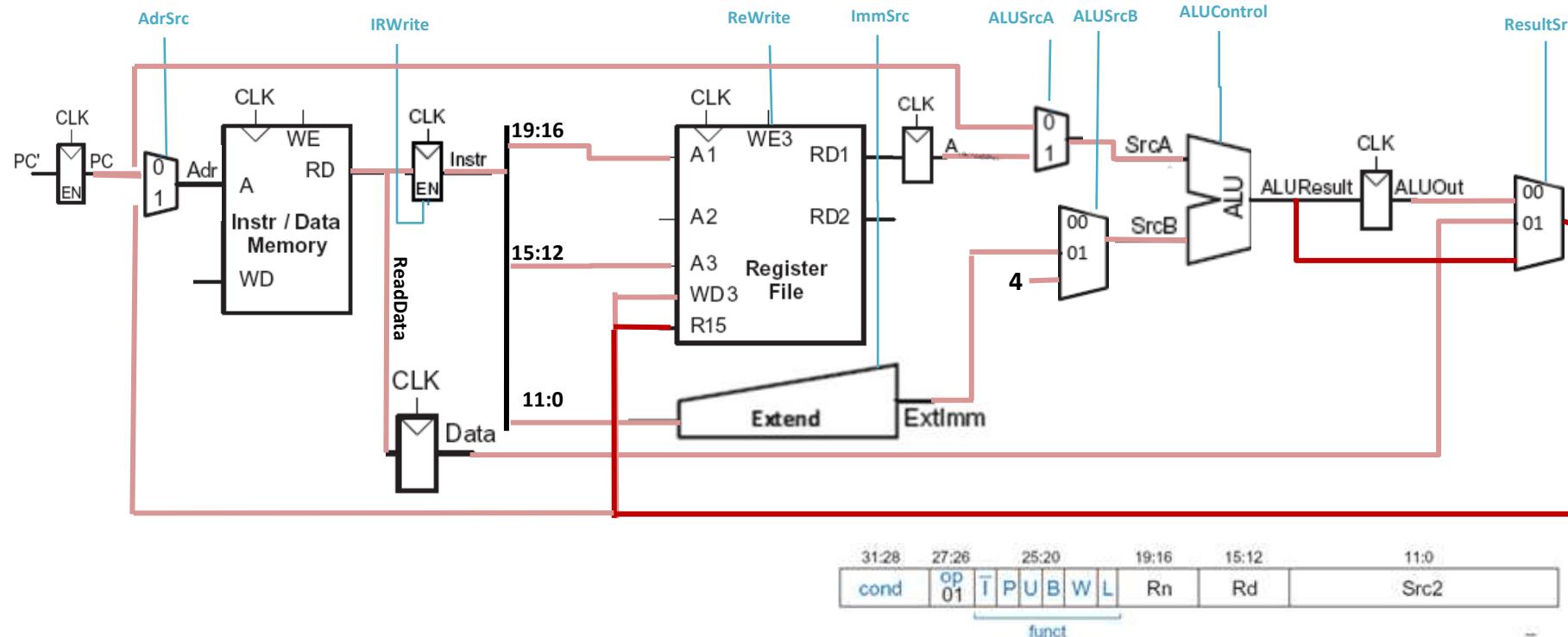


# Datapath LDR

Si consideri infine, che il contenuto del registro **R15** nelle architetture ARM corrisponde a **PC+8**.

Durante il passo di fetch, il **PC** è stato aggiornato a **PC+4**, per cui sommare 4 al nuovo contenuto di **PC** produce **PC+8**, che viene memorizzato in **R15**.

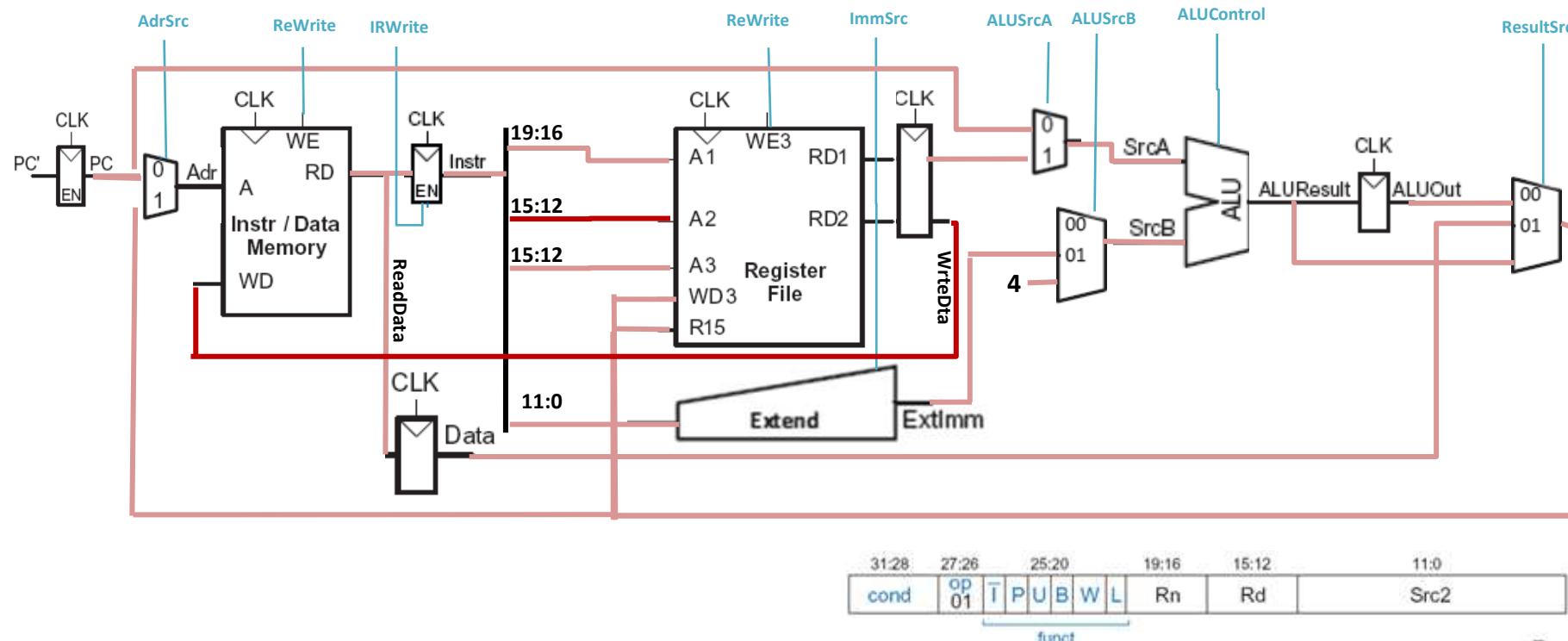
Scrivere **PC+8** in **R15**, richiede che il risultato dell'ALU possa essere collegato a tale registro. A tal fine, colleghiamo **ALUResult** con uno dei tre ingressi del multiplexer.



# Datapath STR

Analogamente all'istruzione di caricamento, **STR** legge l'indirizzo di base dalla porta **RD1** del register file, estende la costante e l'**ALU** somma i due valori per calcolare l'indirizzo di memoria. Tutte queste operazioni sono già supportate.

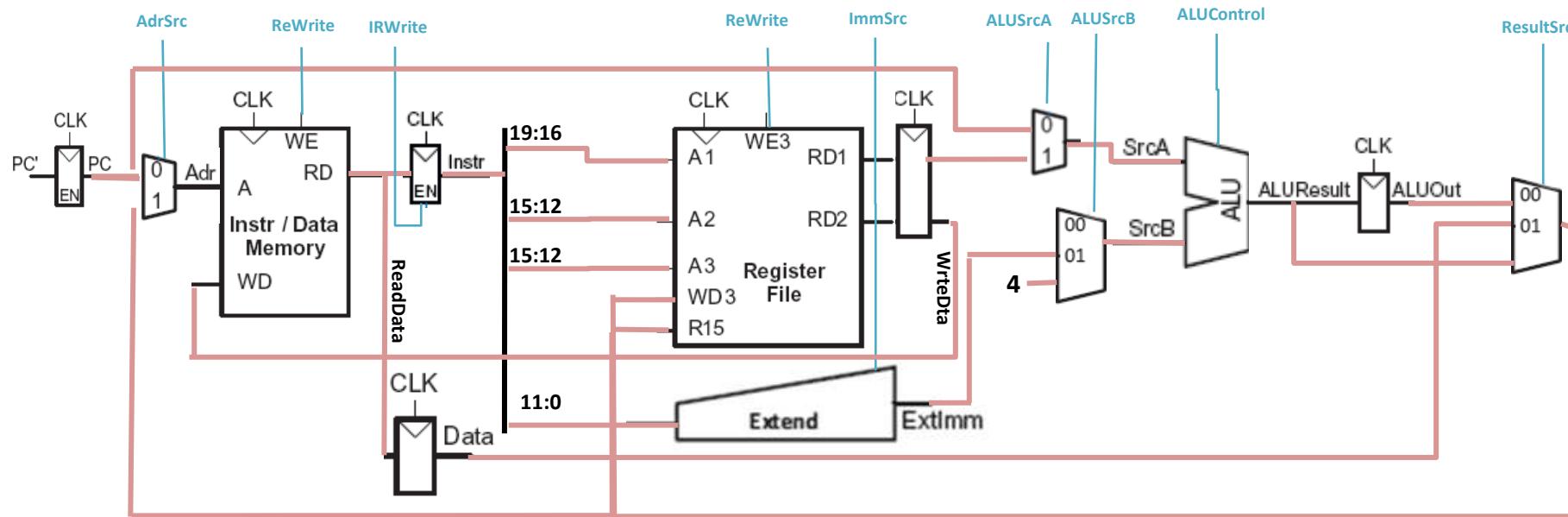
In aggiunta, **STR** legge il registro **Rd** da cui scrivere, che è specificato nei bit **Instr<sub>15:12</sub>**. Il contenuto di **RD2** è inserito in un registro temporaneo WriteData e al passo successivo è inviato alla memoria, con segnale **MemWrite** attivo.



# Datapath DataProcessing

Per le istruzioni di data processing con costante (**ADD, SUB, AND, OR**), il datapath legge il primo operando specificato da **Rn**, estende la costante da **8** a **32** bit, esegue l'operazione mediante l'**ALU** e scrive il risultato in un registro del register file. Tutte queste operazioni sono già supportate dal datapath.

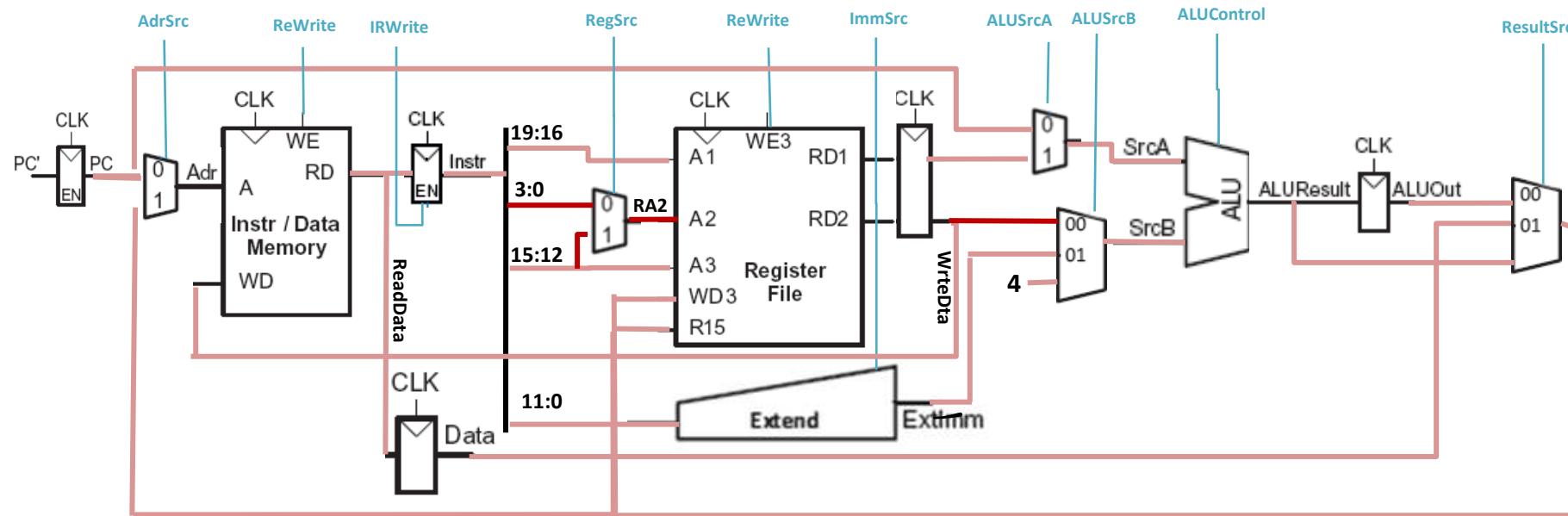
L'operazione da effettuare è specificata dal segnale **ALUControl**, mentre gli **ALUFlags** permettono di aggiornare il registro di stato.



# Datapath DataProcessing

Per le istruzioni di data processing con registro (**ADD, SUB, AND, OR**), il datapath legge il secondo operando specificato da **Rm**, indicato nei bit **Instr<sub>3:0</sub>**.

Inseriamo un multiplexer per selezionare tale campo sulla porta **A2** del register file. Il mutiplexer è controllato dal segnale **RegSrc**. Inoltre, estendiamo il multiplexer in **SrcB** in modo da considerare questo caso.

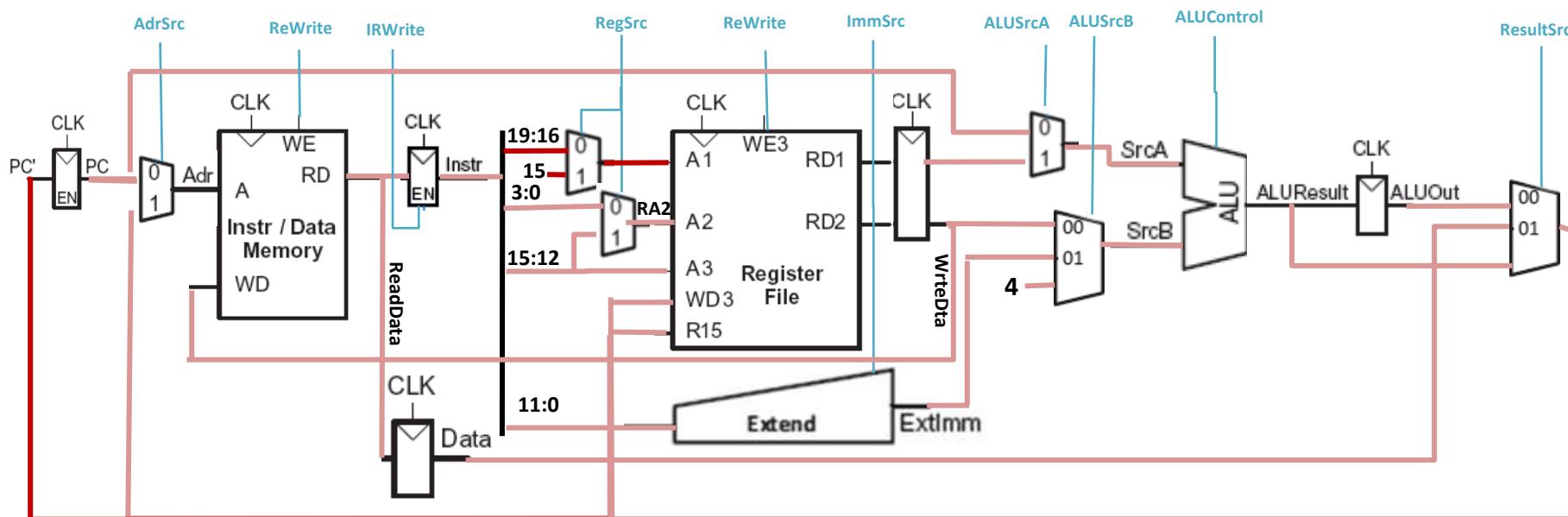


# Datapath Branch

Per le istruzioni di branch, il datapath legge **PC+8** e una costante a 24 bit, che viene estesa a 32 bit. La somma di questi due valori è addizionata al **PC**.

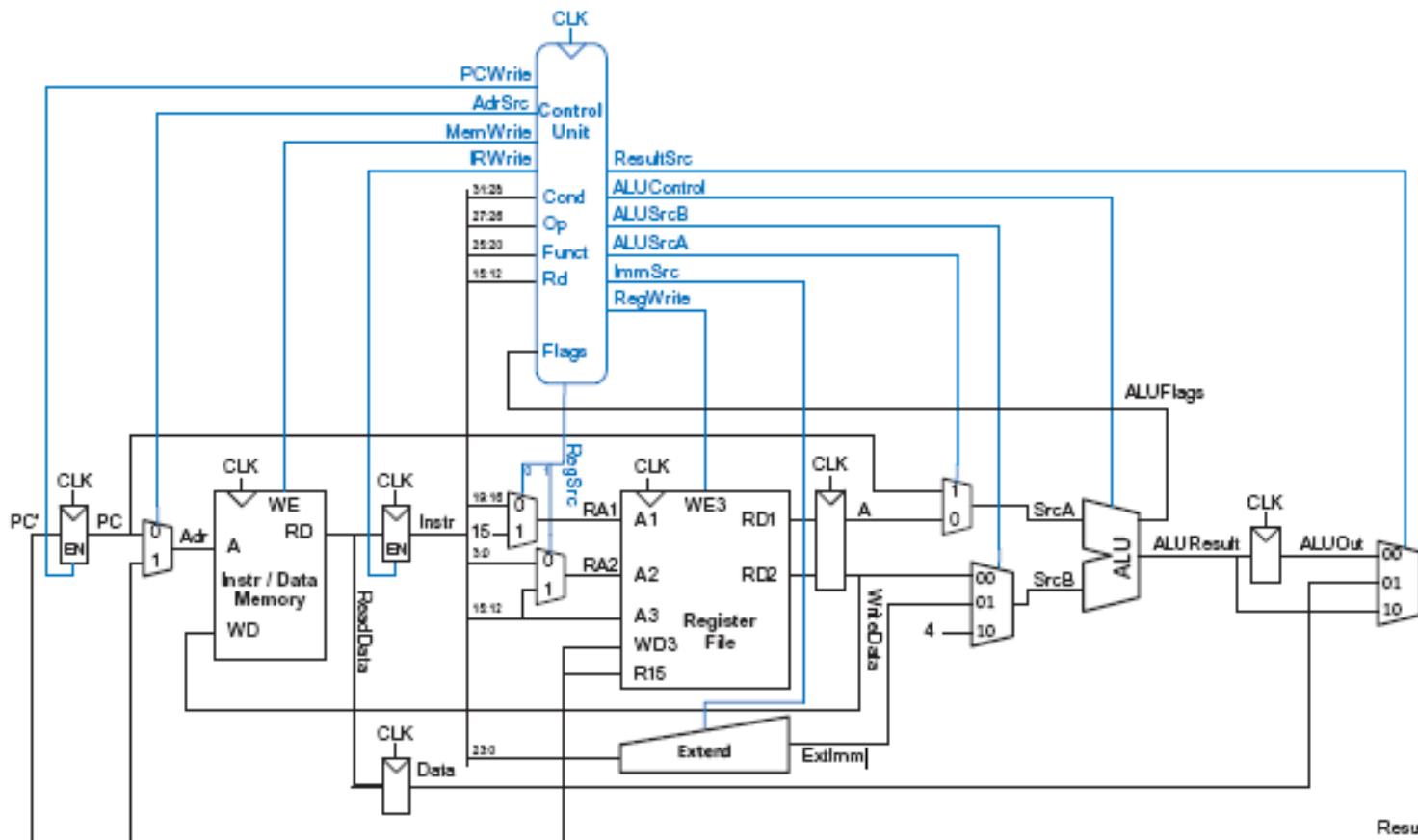
Si ricorda, inoltre, che il registro **R15** contiene il valore **PC+8** e deve essere letto per tornare da un salto. È sufficiente aggiungere un multiplexer per selezionare **R15** come input sulla porta **A1**.

Il multiplexer è controllato dal segnale **RegSrc**.



# Unità di controllo

Come nel processore a ciclo singolo, l'unità di controllo genera i segnali di controllo in base ai campi **cond**, **op** e **funct** dell'istruzione ( $\text{Instr}_{31:28}$ ,  $\text{Instr}_{27:26}$ , e  $\text{Instr}_{25:20}$ ), ai flag e al fatto che il registro destinazione sia o meno il PC.

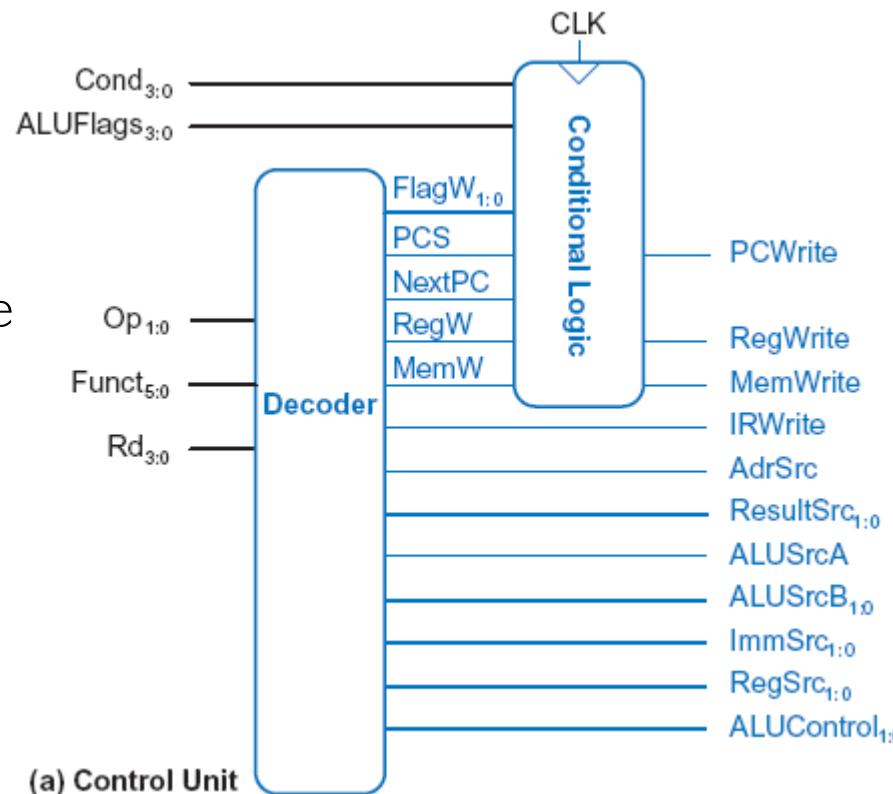


L'unità di controllo memorizza  
e aggiorna i flag di stato.

# Unità di controllo

Come nel processore a ciclo singolo, l'unità di controllo è suddivisa in decodificatore e logica condizionale. Il decodificatore è progettato come una FSM, che produce i segnali appropriati per i diversi cicli, sulla base del proprio stato.

Il decodificatore è realizzato con una macchina di Moore, in tal modo le uscite dipendono solo dello stato attuale.



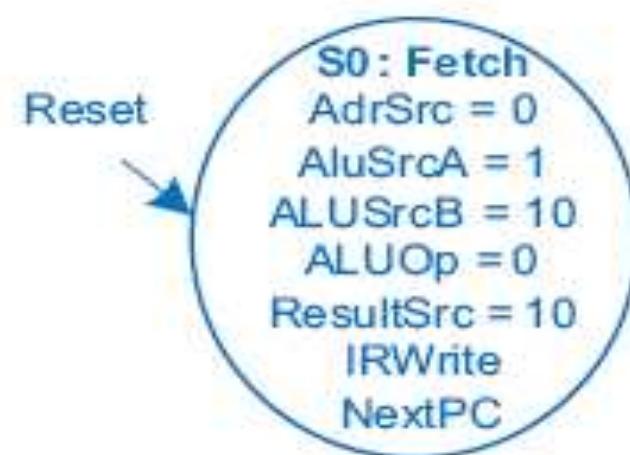
# Dataflow di una istruzione

L'unità di controllo produce i segnali di attivazione per tutto il datapath (selezione nei multiplexer, abilitazione dei registri e scrittura in memoria).

Uno stato dell'automa che implementa il **main decoder** non è altro che lo stato dei segnali in un determinato momento.

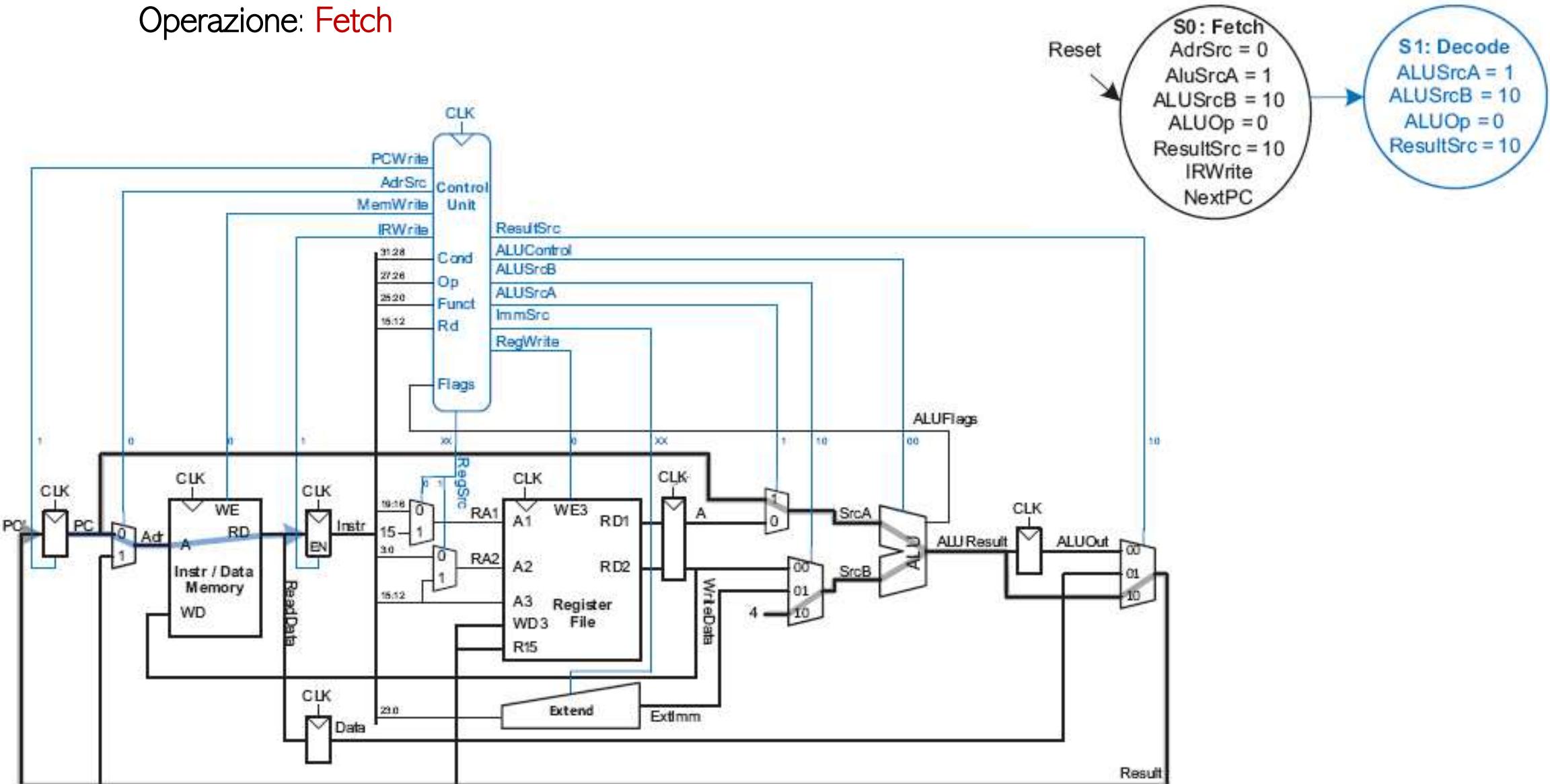
Per avere una visione più chiara di quali siano gli stati e di come vengano effettuate le transizioni da uno stato all'altro è utile considerare il **data flow** del processore.

In altri termini, data una istruzione osserviamo il comportamento del processore nei diversi cicli, in cui avvengono i quattro passi **fetch**, **decode**, **execute** e **store**.



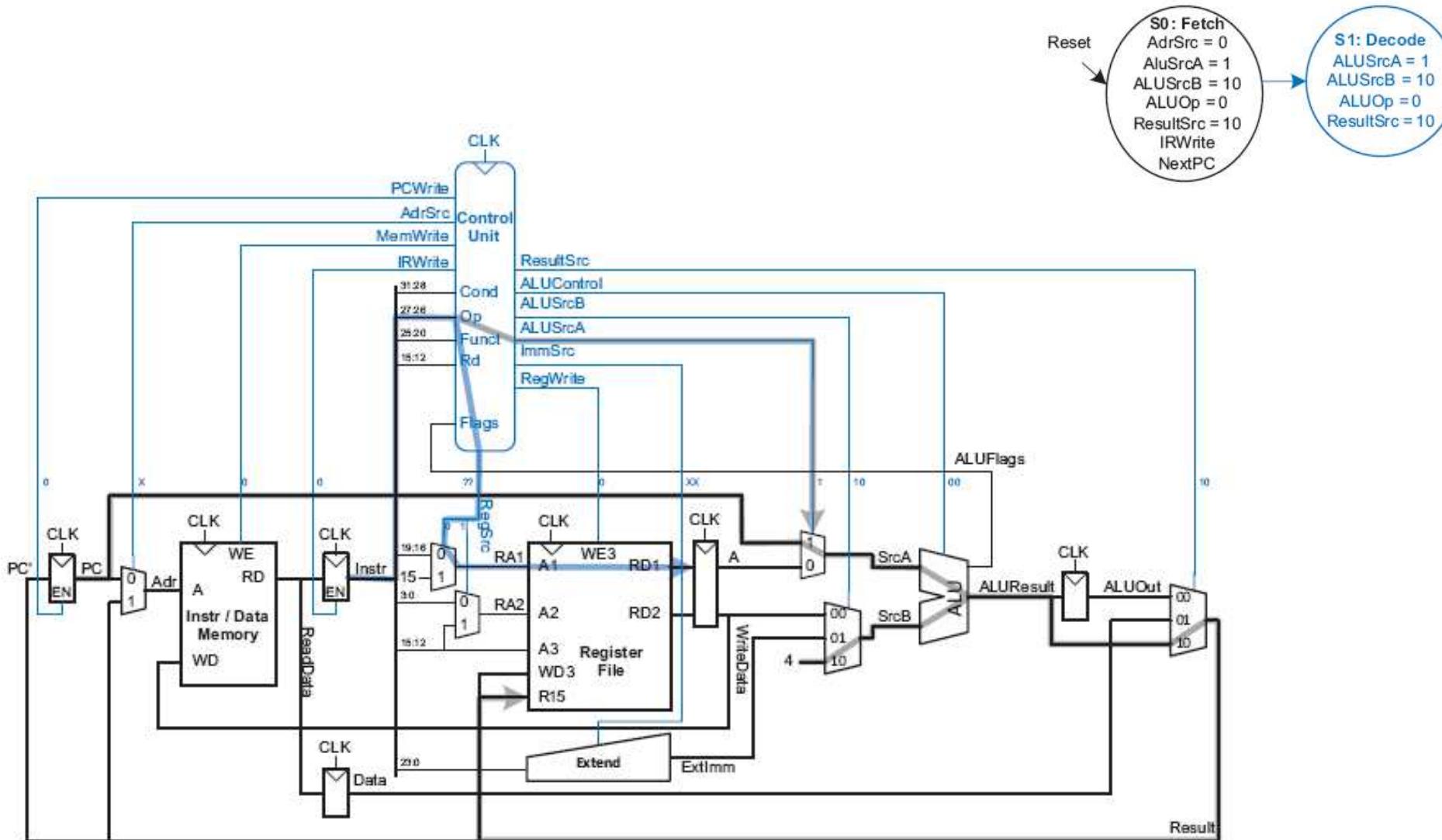
# Dataflow di LDR

## Operazione: Fetch



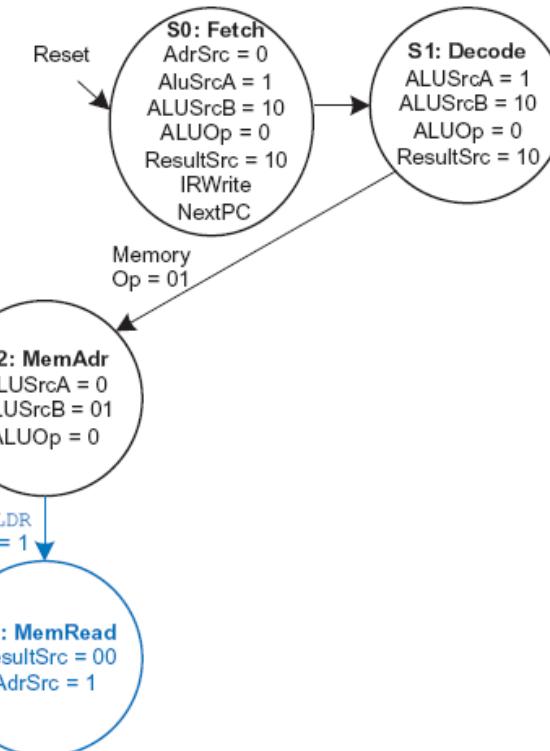
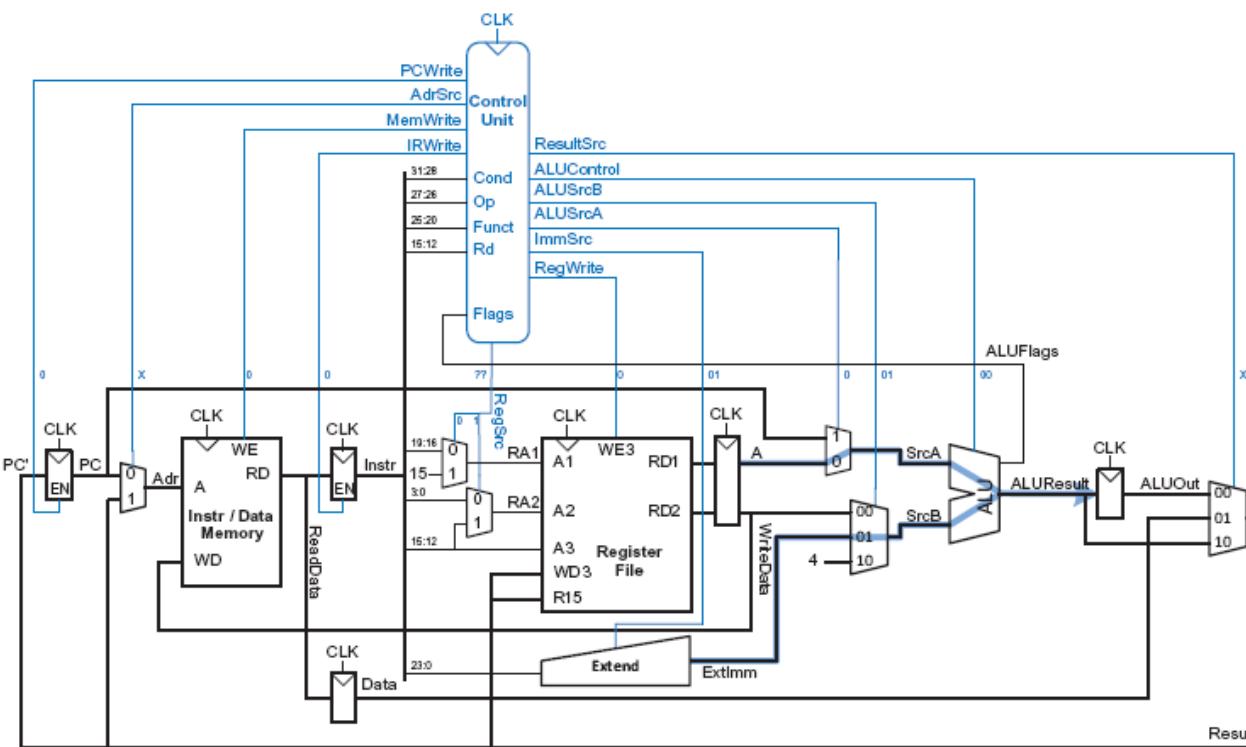
# Dataflow di LDR

Operazione: **Decode**



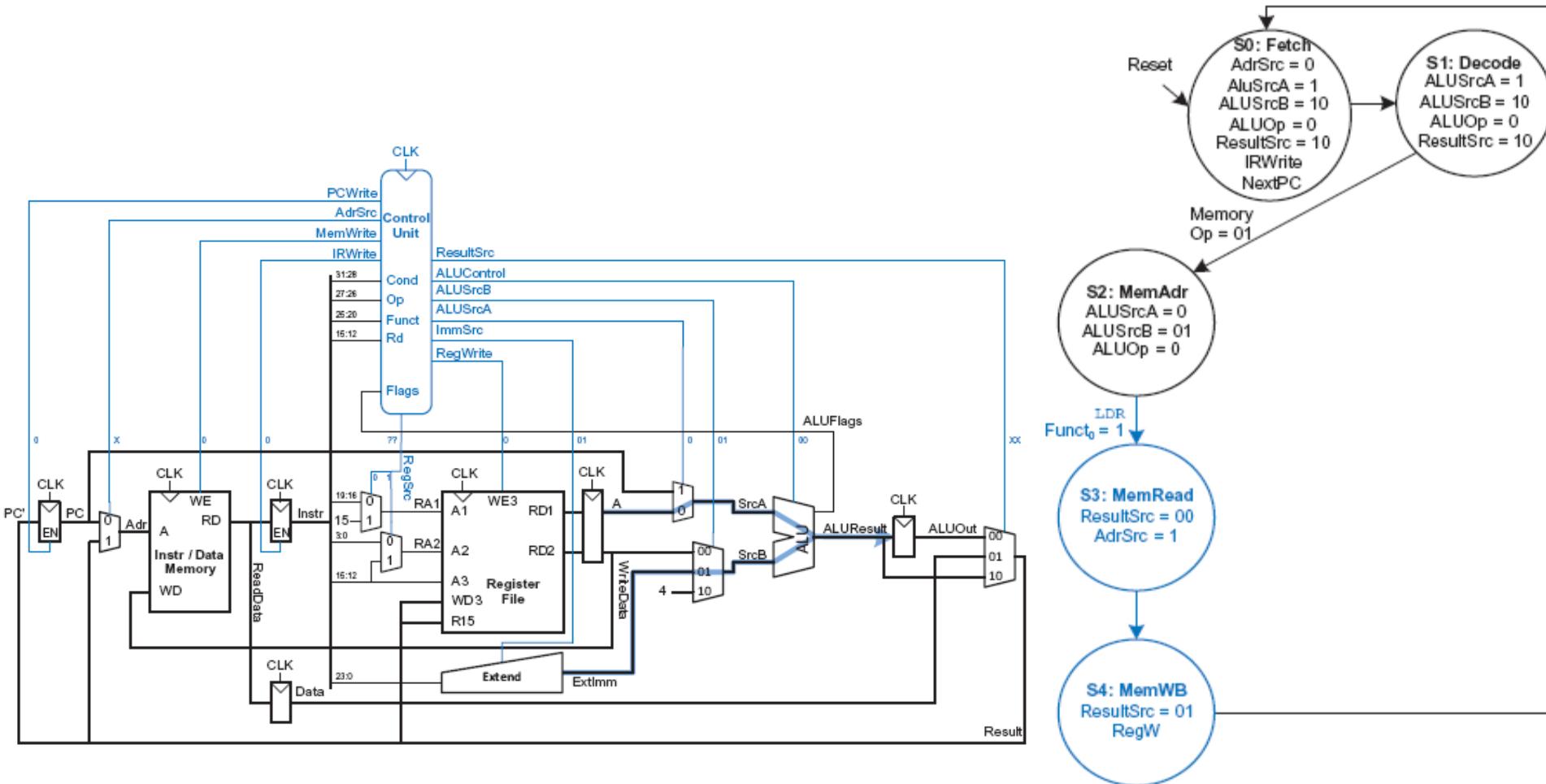
# Dataflow di LDR

Operazione: **Execute** (memory address computation)



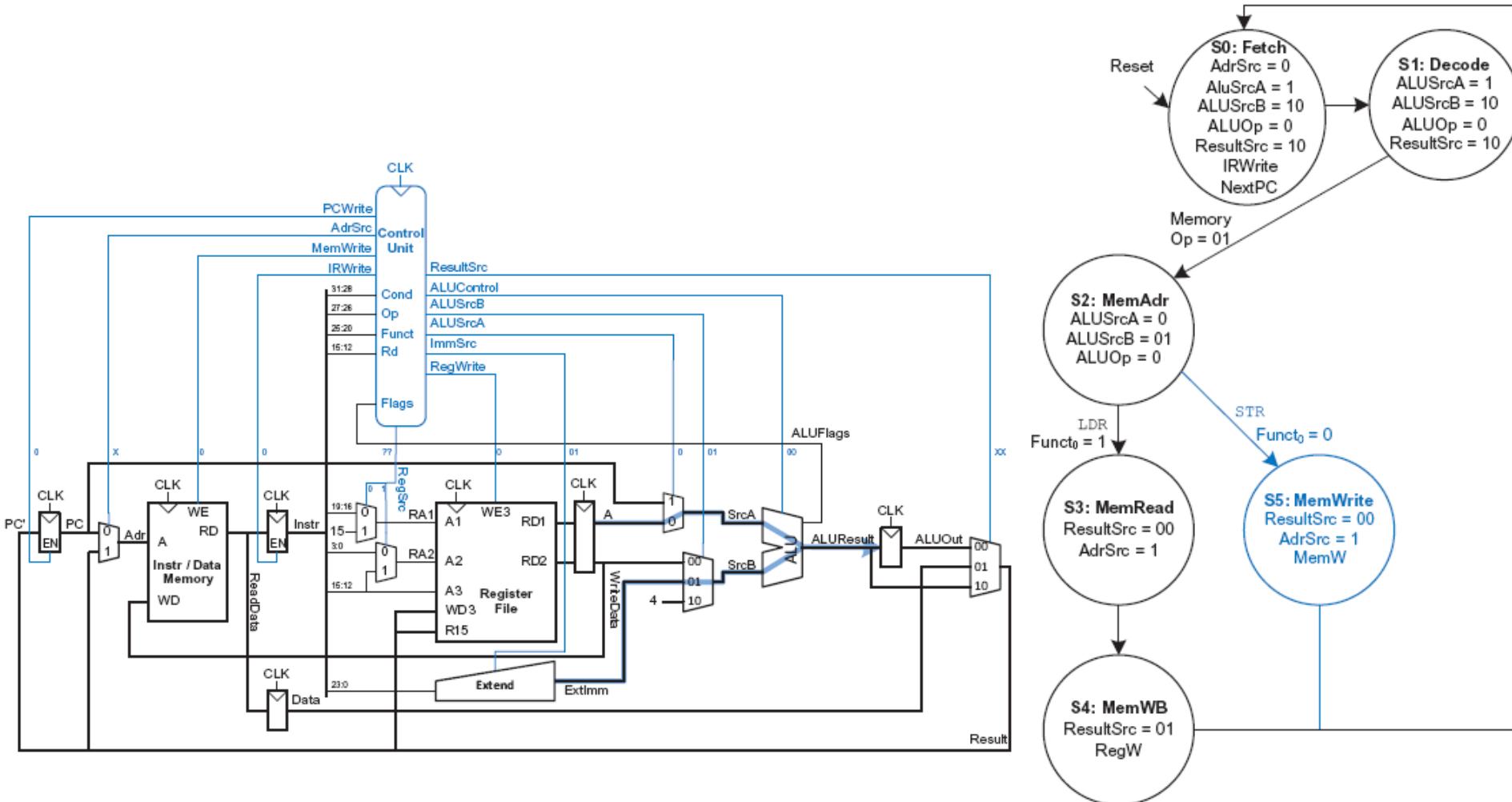
# Dataflow di LDR

Operazione: **Store (memory read)**



# Dataflow di STR

Operazione: **Execute (memory write)**



# Analisi delle prestazioni

In un processore a ciclo multiplo, il tempo impiegato per eseguire una istruzione dipende dal numero di cicli di clock, di cui necessita e dalla durata di un singolo ciclo di clock.

Il numero di cicli di clock necessario ad eseguire le diverse istruzioni è di:

- **branch** – 3 cicli;
- **data processing** – 4 cicli;
- **memory store** – 4 cicli;
- **memory load** – 5 cicli;

Valutiamo le prestazioni del processore a ciclo multiplo rispetto al benchmark SPECINT2000, che prevede:

- **LDR** – 25%;
- **STR** – 10%;
- **B** – 13%;
- **Data Processing** – 52%;

# Analisi delle prestazioni

Il numero medio di cicli per istruzione è dato da:

$$\begin{aligned}\text{CPI} &= (\text{perc. di B})(\#\text{cicli B}) + (\text{perc. di DP} + \text{perc. di STR})(\#\text{cili DP e STR}) + (\text{perc. di LDR})(\#\text{cili LDR}) = \\ &= (0.13) \quad (3) \quad + \quad (0.52 + 0.10) \quad (4) \quad + \quad (0.25) \quad (5) \quad = 4.12\end{aligned}$$

I percorsi critici nel datapath, che richiedono maggior tempo e che quindi sono predominanti, sono due:

- ▶ Dal **PC**, attraverso il multiplexer **SrcA**, attraverso l'ALU, attraverso il multiplexer **Result**, attraverso la porta **R15**, fino al registro **A**.
- ▶ Da **ALUOut**, attraverso il registro **Result**, attraverso il multiplexer **Adr**, attraverso la memoria (read), fino al registro **Data**.

- ▶ ( $t_{pcq\_PC}$ ) – caricamento di un nuovo indirizzo (PC) sul fronte di salita del clock;
- ▶ ( $t_{mux}$ ) – selezione di un output da parte del multiplexer.
- ▶ ( $t_{ALU}$ ) – l'ALU esegue su srcA e srcB una operazione.
- ▶ ( $t_{setup}$ ) – viene impostato un segnale.

$$T_{c2} = t_{pcq\_PC} + 2t_{mux} + \max[t_{ALU} + t_{mux}, t_{mem}] + t_{setup};$$

# Analisi delle prestazioni

**Domanda:** qual è il tempo di esecuzione per un programma con 100 miliardi di istruzioni?

**Risposta:**

secondo l'equazione

$$T_{c2} = t_{pcq\_PC} + 2t_{mux} + \max[t_{ALU} + t_{mux}, t_{mem}] + t_{setup};$$

il tempo di ciclo del processore a ciclo multiplo è

$$T_{c2} = 40 + 2(25) + 200 + 50 = 340 \text{ ps.}$$

Secondo l'equazione

$$\text{Tempo di esecuzione} = (\# \text{istruzioni}) \left( \frac{\text{cicli}}{\text{istruzione}} \right) \left( \frac{\text{secondi}}{\text{ciclo}} \right)$$

il tempo di esecuzione totale è

$$T_1 = (100 \times 10^9 \text{ istruzioni}) (4.12 \text{ cicli / istruzione}) (340 \times 10^{-12} \text{ s / ciclo}) = 140 \text{ secondi.}$$

Table 7.5 Delay of circuit elements

| Element             | Parameter     | Delay (ps) |
|---------------------|---------------|------------|
| Register clk-to-Q   | $t_{pcq}$     | 40         |
| Register setup      | $t_{setup}$   | 50         |
| Multiplexer         | $t_{mux}$     | 25         |
| ALU                 | $t_{ALU}$     | 120        |
| Decoder             | $t_{dec}$     | 70         |
| Memory read         | $t_{mem}$     | 200        |
| Register file read  | $t_{RFread}$  | 100        |
| Register file setup | $t_{RFsetup}$ | 60         |

**Nota:** il tempo impiegato dal processore a ciclo singolo per lo stesso benchmark era di 84 sec.

# Considerazioni finali

Una delle motivazioni alla base della progettazione di un processore a ciclo multiplo è stata quella di evitare che il ciclo durasse quanto quello necessario all'istruzione più lenta.

Questo esempio dimostra che il processore a ciclo multiplo è più lento di quello a ciclo singolo a causa delle latenze di propagazione.

Infatti, sebbene l'istruzione più lenta (LDR) sia stata suddivisa in cinque fasi, la frequenza di ciclo del processore non è aumentata di cinque volte.

Questo in parte perché:

- ▶ non tutti i passaggi hanno esattamente la stessa lunghezza;
- ▶ i tempi di setup sono necessari ad ogni passo, e non solo la prima volta per l'intera istruzione.

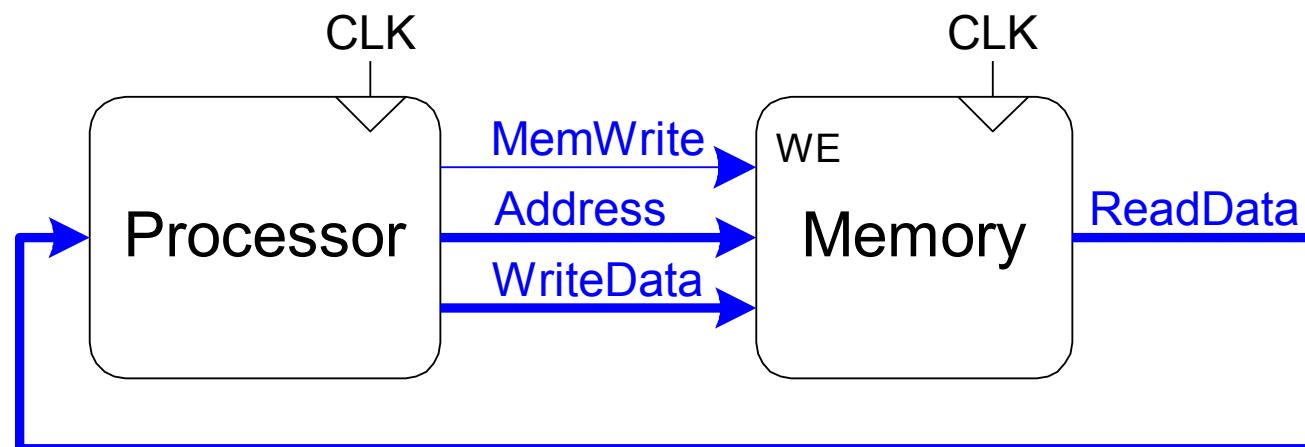
In generale, ci si è resi conto che il fatto che alcuni calcoli siano più veloci di altri è difficile da sfruttare, a meno che le differenze sono grandi.

# Memorie

# Memorie vs CPU

Nell'architettura di Von Neumann il canale di comunicazione tra la CPU e la memoria è il punto critico (collo di bottiglia) del sistema.

La tecnologia consente di realizzare CPU sempre più veloci e memorie sempre più grandi ma la velocità di accesso delle memorie non cresce così rapidamente come la velocità della CPU.



# Memorie vs CPU

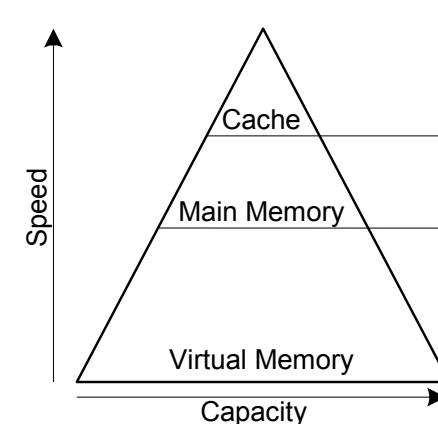
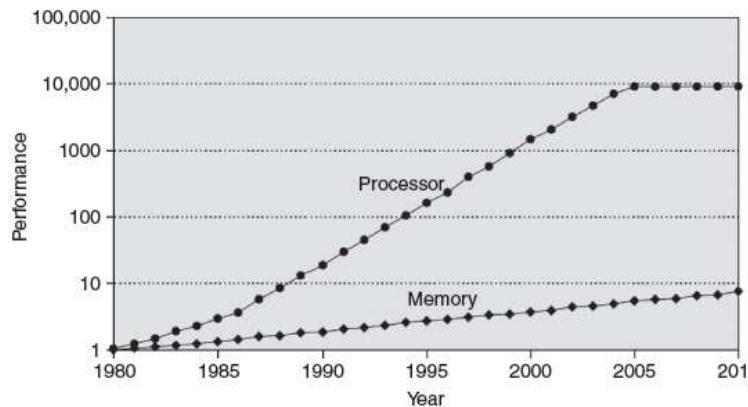
Storicamente le **CPU** sono sempre state più veloci delle **memorie**.

Al giorno d'oggi siamo in grado di produrre delle memorie veloci quanto una CPU moderna, il reale problema è dato da:

- queste memorie hanno un costo elevatissimo.
- le memorie dovrebbero essere piazzate in gran parte sugli stessi chip delle CPU, il che non è possibile.

Per ovviare a questo problema, gli ingegneri ricorrono ad uno schema chiamato “**a gerarchia di memoria**” in cui si combinano:

- una quantità molto **piccola** di memoria estremamente **veloce** (la cache)
- una quantità molto **grande** di memoria **lenta**.



| Technology | Price / GB | Access Time (ns) | Bandwidth (GB/s) |
|------------|------------|------------------|------------------|
| SRAM       | \$10,000   | 1                | 25+              |
| DRAM       | \$10       | 10 - 50          | 10               |
| SSD        | \$1        | 100,000          | 0.5              |
| HDD        | \$0.1      | 10,000,000       | 0.1              |

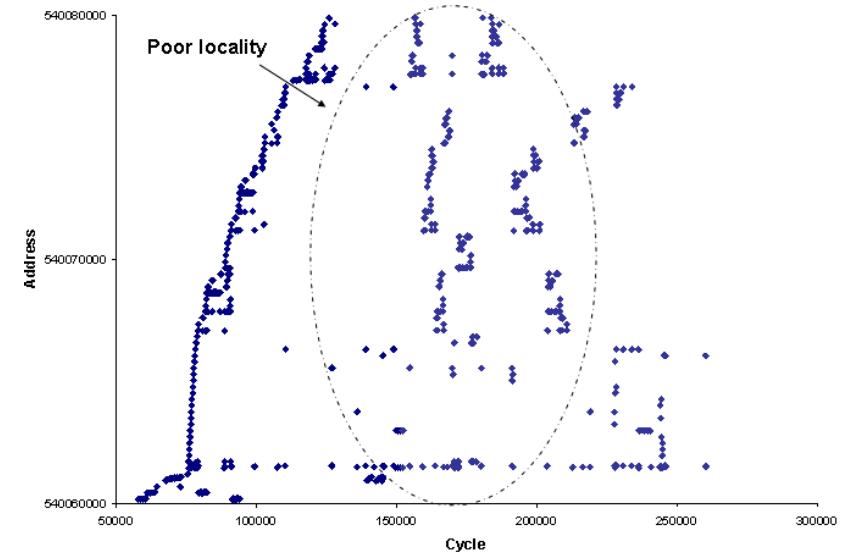
# Inquadramento del Problema

La maggior parte del tempo di esecuzione di una CPU è, di solito, impegnato da procedure **in cui vengono eseguite ripetutamente le stesse istruzioni**.

In realtà, non è importante conoscere lo schema dettagliato della sequenza di istruzioni, il punto chiave è che molte istruzioni in aree ben localizzate del programma vengono eseguite ripetutamente in un determinato periodo, e si accede al resto del programma relativamente di rado.

Questa proprietà viene chiamata *località dei riferimenti*. Si manifesta in due modi: *località temporale* e *località spaziale*.

- La *località temporale* rappresenta la probabilità che un'istruzione eseguita di recente venga eseguita nuovamente, entro breve tempo.
- La *località spaziale* rappresenta invece la probabilità che istruzioni vicine ad un'istruzione eseguita di recente (dove la vicinanza è espressa in termini di indirizzi delle istruzioni) siano anch'esse eseguite nel prossimo futuro.



# Utilità di avere una cache memory

La velocità con cui la memoria risponde alle richieste di istruzioni e dati della CPU ha un peso determinante sulle prestazioni di un sistema.

Se tra la memoria principale e la CPU si potesse interporre una memoria molto veloce, contenente le parti di programma e di dati che, volta per volta, interessano l'elaborazione, il tempo totale di esecuzione verrebbe ridotto in modo significativo.

Questa è esattamente la funzione della "*cache memory*" che in inglese significa letteralmente "*schermo della memoria*".

Si tratta quindi di un elemento che inserito tra CPU e memoria principale impedisce alla prima di "*vedere*" i tempi di risposta reali della memoria.

La struttura della CPU, infatti, non viene influenzata dalla presenza o meno di una *cache memory*. L'interfaccia verso la memoria applica il "*protocollo*" di trasferimento *dalla* memoria o *verso* la memoria ignorando la presenza di una struttura intermedia.

Agli albori di questa tecnica, infatti, la cache memory era solo un accessorio a pagamento.

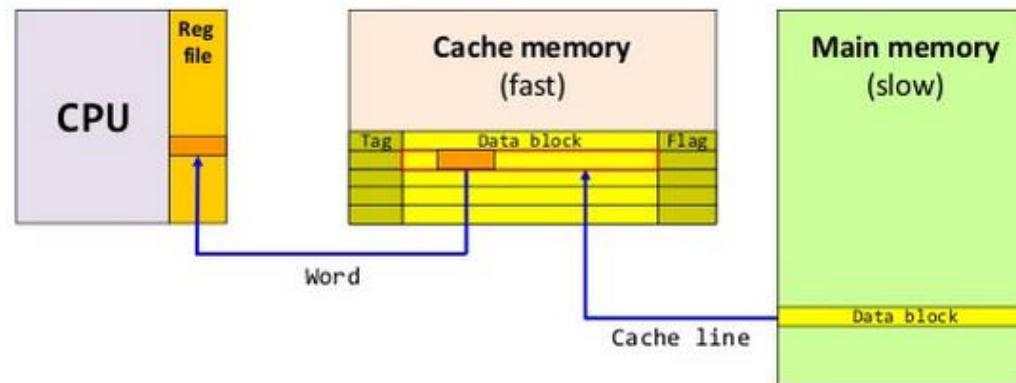
Una traduzione possibile in italiano del concetto è "*memoria tamponcino*".

# Principi di funzionamento

Concettualmente, le operazioni svolte da una memoria cache sono molto semplici. I circuiti di controllo della memoria cache sono progettati per avvantaggiarsi della proprietà della località dei riferimenti.

L'aspetto temporale di questa proprietà suggerisce di portare un elemento (istruzioni o dati) nella cache quando viene richiesto per la prima volta, in modo tale che rimanga a disposizione nel caso di una nuova richiesta.

L'aspetto spaziale suggerisce che, invece di portare dalla memoria principale alla cache un elemento alla volta, è conveniente portare un insieme di elementi che risiedono in indirizzi adiacenti.



Si farà riferimento al termine *blocco* per indicare un insieme di indirizzi contigui di una qualche dimensione.

Un altro termine utilizzato di frequente per indicare un blocco della cache è *linea di cache*.

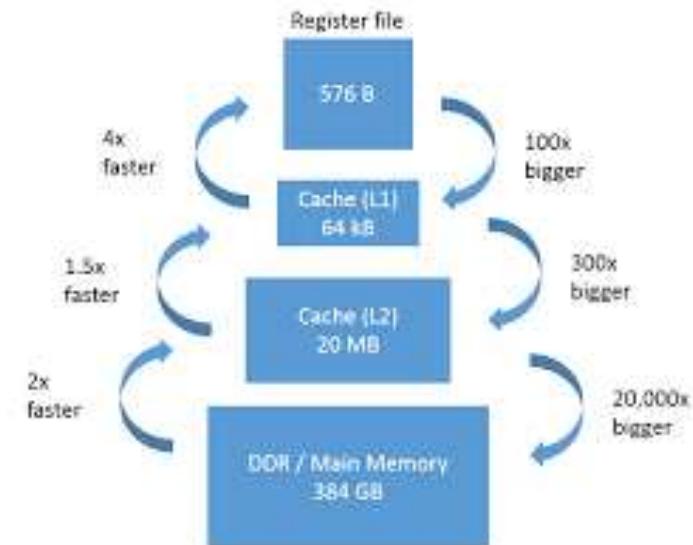
# Dimensione della cache

Quando si riceve una richiesta di lettura fatta dalla CPU, il contenuto di un blocco di parole di memoria contenenti la locazione specificata viene trasferito nella cache, una o più parole alla volta.

In seguito, ognqualvolta il programma fa riferimento a una di queste locazioni del blocco, i valori desiderati vengono letti direttamente dalla cache.

Affinché la cache svolga efficacemente il suo compito deve avere tempi di accesso molto più brevi di quelli della memoria principale e ciò come vedremo impone che sia piccola.

Se è piccola non potrà che contenere una frazione ridotta delle istruzioni ed i dati della memoria principale.



# Tecniche di gestione

La corrispondenza tra i blocchi della memoria principale e quelli della cache è specificata dalla funzione di *posizionamento (mapping)*.

Quando la cache è piena e si fa riferimento a una parola di memoria (istruzione o dato) che non è presente nella cache, l'hardware di controllo della cache deve decidere quale blocco della cache debba essere rimosso per far spazio al nuovo blocco, che contiene la parola a cui si fa riferimento.

L'insieme di regole in base alle quali viene fatta questa scelta costituisce *l'algoritmo di sostituzione*.

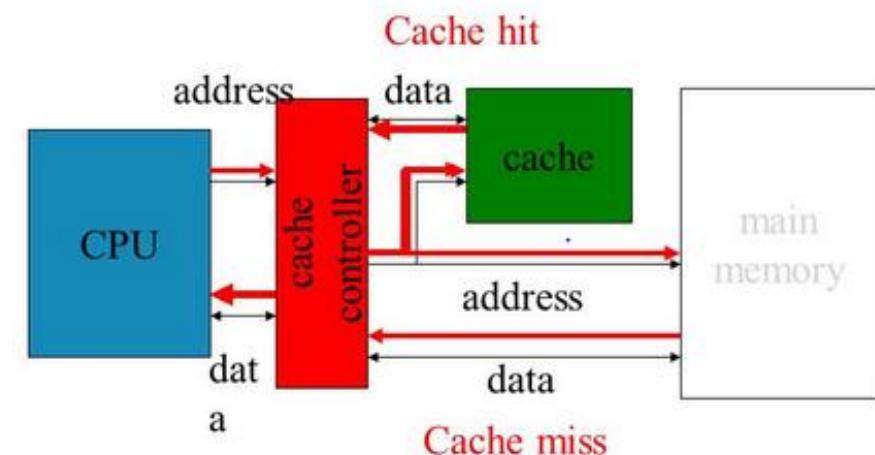
# Tecniche di gestione

La struttura dei sistemi è, di solito, organizzata in modo che la presenza di una cache non influenzi il funzionamento della CPU.

Quest'ultima, infatti, nell'esecuzione del programma effettua le richieste di lettura e scrittura utilizzando gli indirizzi delle locazioni nella memoria principale.

E' la logica di controllo della cache che si fa carico di determinare se la parola richiesta è presente o meno nella cache.

Se è presente, viene effettuata l'operazione di lettura o scrittura della locazione di memoria appropriata. In questo caso si dice che l'accesso in lettura o in scrittura ha avuto successo (*read hit* o *write hit*).



# Possibili gestioni di un “cache hit”

Se l'accesso alla cache ha avuto successo bisogna distinguere due tipi di situazioni:

*Operazione di lettura:* la memoria principale non viene coinvolta

*Operazione di scrittura:* il sistema può procedere in due modi:

- Nel primo (*write-through*), la locazione della cache e quella della memoria principale **vengono entrambe aggiornate**. Il *write-through* è più semplice, ma implica inutili operazioni di scrittura nella memoria principale, se una parola viene aggiornata più volte durante il periodo in cui risiede nella cache.
- Nel secondo (*write-back*) si aggiorna **soltanto la locazione della memoria cache**, segnandola come aggiornata con un **bit di modifica** o *dirty*. La locazione della memoria principale viene aggiornata in seguito, quando il blocco contenente la parola marcata deve essere rimosso dalla memoria cache per far posto a un nuovo blocco. Si noti che anche il protocollo *write-back* può causare inutili scritture nella memoria principale, visto che, quando si procede con la scrittura nella memoria principale di un blocco, **tutte le parole del blocco vengono scritte**, anche se solo una delle parole del blocco della cache è stata modificata.

# Gestione di un “read miss”

Quando la parola indirizzata durante un'operazione di lettura non è presente nella cache si dice che l'accesso in lettura è fallito (*read miss*).

Il blocco di parole contenente la parola richiesta deve essere copiato dalla memoria principale nella memoria cache.

Due sono le possibilità:

- Dopo aver caricato l'intero blocco nella memoria cache, la parola richiesta viene inviata alla CPU.
- In alternativa, è possibile inviare immediatamente la parola alla CPU, non appena la si legge dalla memoria principale.

Quest'ultimo approccio, chiamato *load-through*, o anche *early restart*, riduce in qualche modo il tempo di attesa della CPU, a discapito di una maggiore complessità del circuito di controllo della cache.

# Gestione di un “write miss”

Durante un'operazione di scrittura, se la parola indirizzata non è nella cache si dice che l'accesso in scrittura è fallito (*write miss*).

Anche in questo caso ci sono le due alternative,  
se si utilizza un protocollo *write-through*, le informazioni vengono scritte direttamente  
nella memoria principale,  
nel caso invece di un protocollo *write-back*, il blocco, contenente la parola indirizzata,  
viene prima caricato nella cache, poi viene soprascritto con le nuove informazioni.

# Prestazioni

$$\begin{aligned}\textbf{Hit Rate} &= \# \text{ hits} / \# \text{ memory accesses} \\ &= 1 - \text{Miss Rate}\end{aligned}$$

$$\begin{aligned}\textbf{Miss Rate} &= \# \text{ misses} / \# \text{ memory accesses} \\ &= 1 - \text{Hit Rate}\end{aligned}$$

**Average memory access time (AMAT):** average time for processor to access data

$$\textbf{AMAT} = t_{\text{cache}} + MR_{\text{cache}}[t_{MM} + MR_{MM}(t_{VM})]$$

# Prestazioni

- Un programma ha 2,000 loads e stores
- 1,250 di questi dati sono presenti nella cache
- Il resto è fornito da altri livelli nella gerarchia di memoria
- Quale è l'hit e il miss rate per la cache?

$$\text{Hit Rate} = 1250/2000 = 0.625$$

$$\text{Miss Rate} = 750/2000 = 0.375 = 1 - \text{Hit Rate}$$

# Prestazioni

- Assumi che un processore ha 2 livelli di memoria: cache e main memory
- $t_{\text{cache}} = 1 \text{ cycle}$ ,  $t_{MM} = 100 \text{ cycles}$
- Qual'è l'AMAT dati gli hit e miss rare precedenti?

$$\begin{aligned}\text{AMAT} &= t_{\text{cache}} + MR_{\text{cache}}(t_{MM}) \\ &= [1 + 0.375(100)] \text{ cycles} \\ &= 38.5 \text{ cycles}\end{aligned}$$

# Terminologia Cache

- **Capacity ( $C$ ):**
  - Numero di byte nella cache
- **Block size ( $b$ ):**
  - Grandezza di un blocco, ovvero numero di bytes che sono trascritti nella cache per volta
- **Number of blocks ( $B = C/b$ ):**
  - Numero di blocchi nella cache
- **Degree of associativity ( $M$ ):**
  - Numero di blocchi in un set
- **Number of sets ( $S = B/M$ ):**
  - Ogni indirizzo di memoria è mappato in esattamente un set della cache

# Tipologie di memorie cache

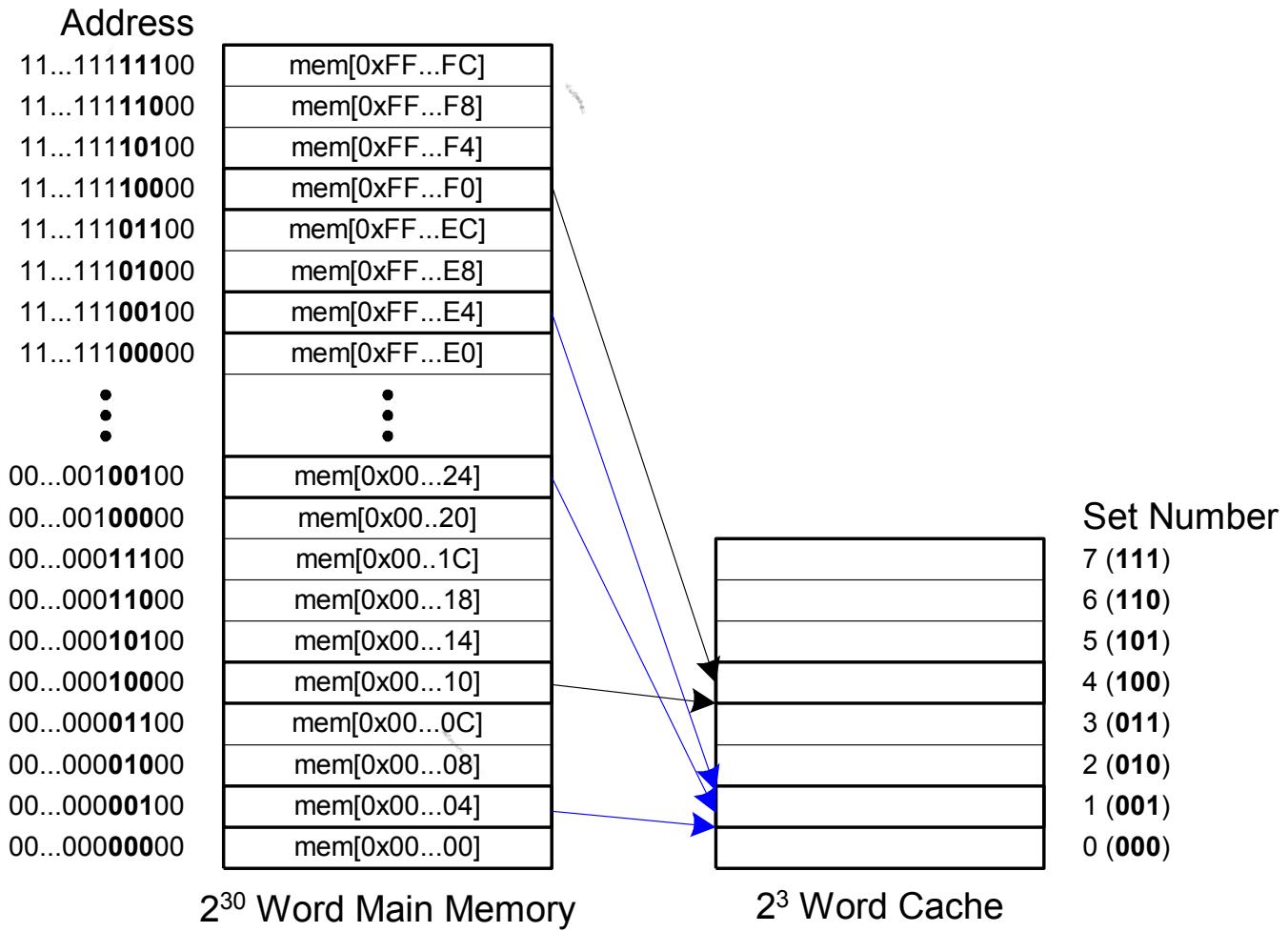
- La memoria Cache è organizzata in  $S$  set
- Ogni indirizzo di memoria mappa in esattamente un unico set
- Diverse categorie di memoria cache dipendono dal numero di blocchi che un set contiene:
  - **Direct mapped:** 1 blocco per ogni set
  - **$N$ -way set associative:**  $N$  blocchi per set
  - **Fully associative:** tutti i blocchi di una cache sono in un unico set
- Esaminiamo ogni tipologia con una cache con:
  - Capacity ( $C = 8$  words)
  - Block size ( $b = 1$  word)
  - Quindi, il numero di blocchi è ( $B = 8$ )

*Ovviamente una memoria del genere è ridicolmente piccola, serve solo per scopi didattici*

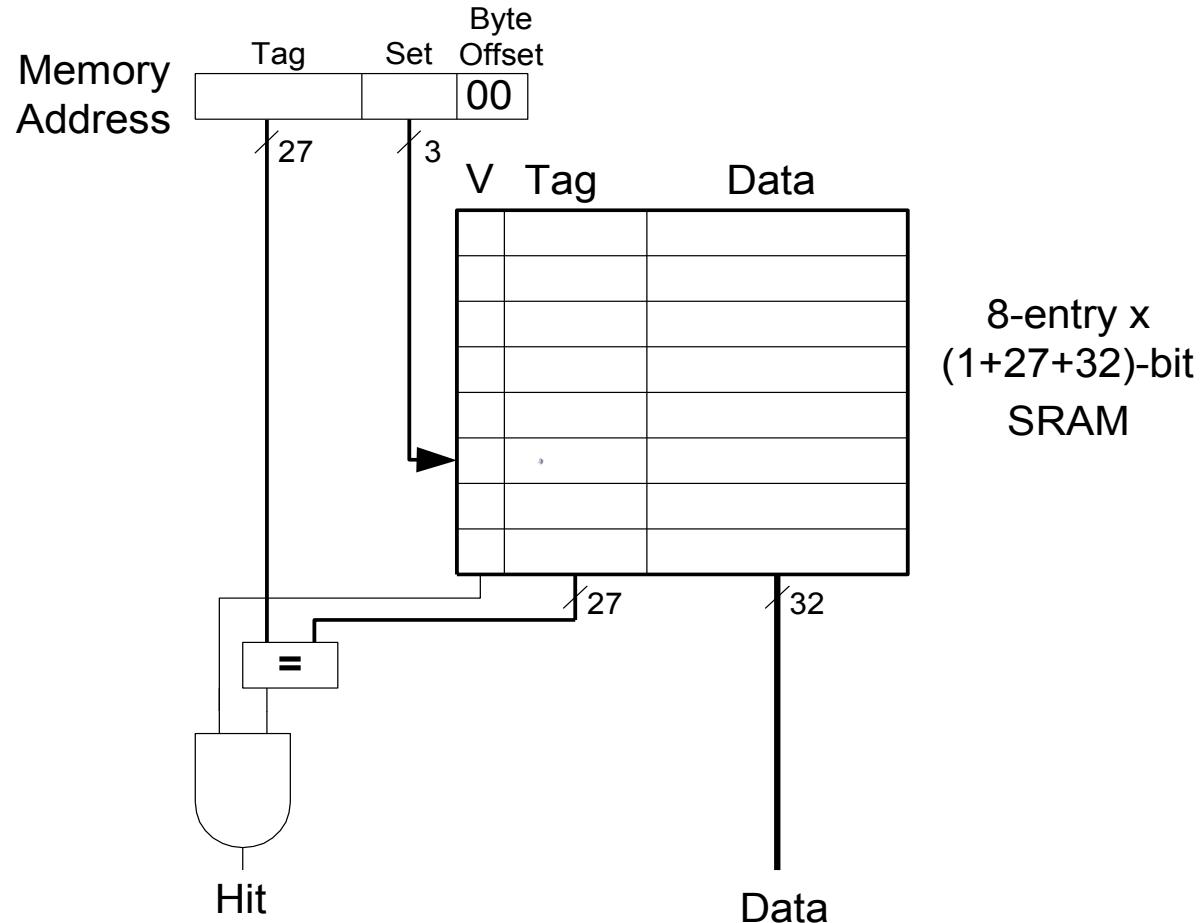
- Capacity:  $C$
- Block size:  $b$
- Number of blocks in cache:  $B = C/b$
- Number of blocks in a set:  $N$
- Number of sets:  $S = B/N$

| Organization          | Number of Ways<br>( $N$ ) | Number of Sets<br>( $S = B/N$ ) |
|-----------------------|---------------------------|---------------------------------|
| Direct Mapped         | 1                         | $B$                             |
| N-Way Set Associative | $1 < N < B$               | $B / N$                         |
| Fully Associative     | $B$                       | 1                               |

# Direct Mapped Cache

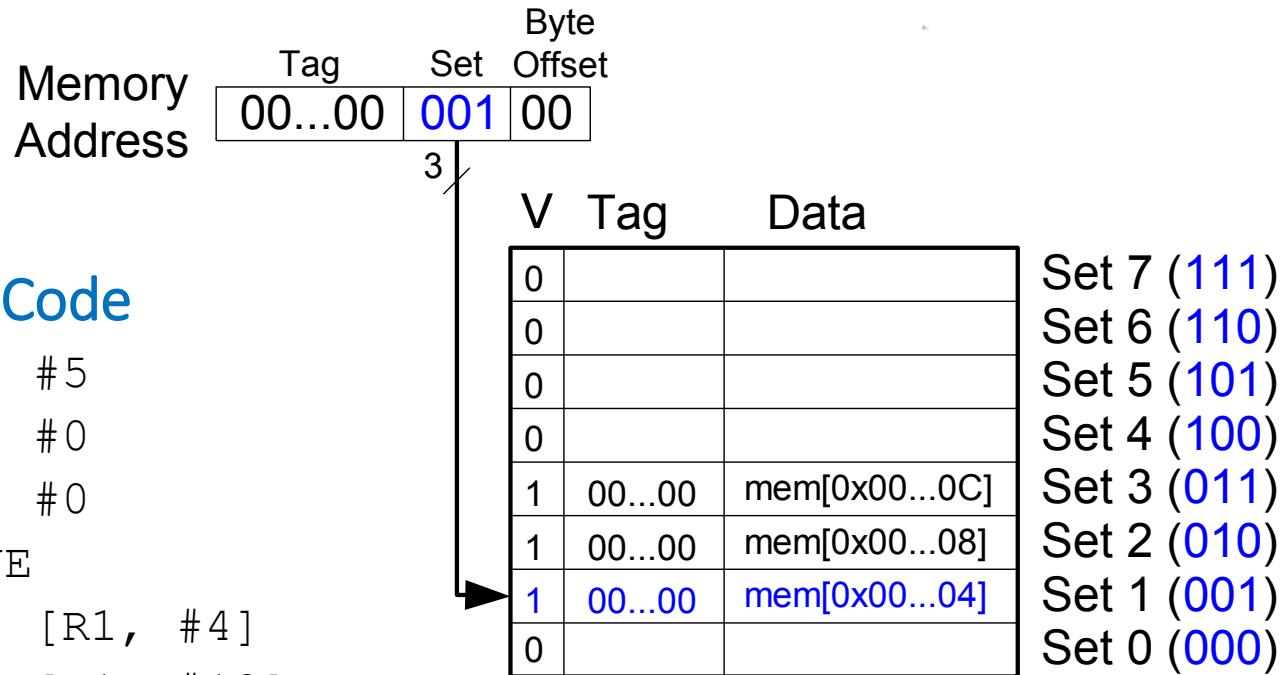


# Direct Mapped Cache Hardware



## ARM Assembly Code

```
MOV R0, #5  
MOV R1, #0  
LOOP  CMP R0, #0  
      BEQ DONE  
      LDR R2, [R1, #4]  
      LDR R3, [R1, #12]  
      LDR R4, [R1, #8]  
      SUB R0, R0, #1  
      B    LOOP  
  
DONE
```

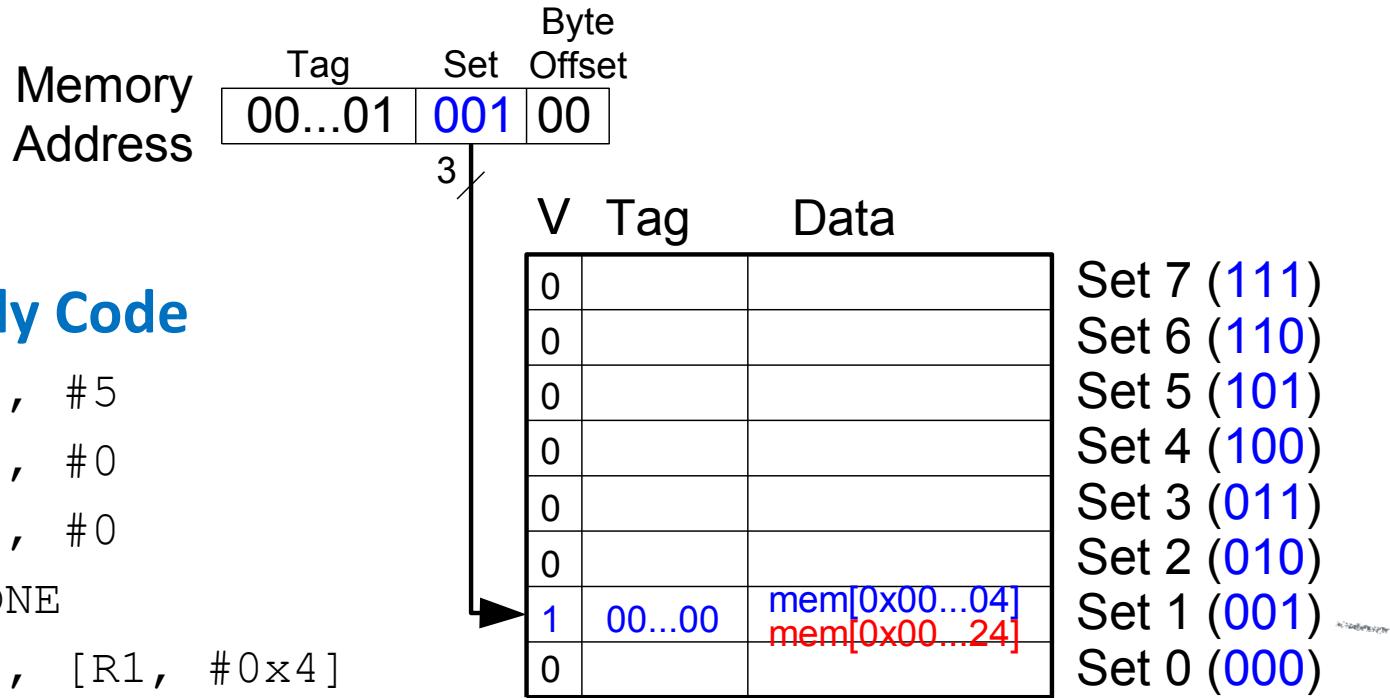


$$\begin{aligned}\text{Miss Rate} &= 3/15 \\ &= 20\%\end{aligned}$$

Miss obbligatori

## ARM Assembly Code

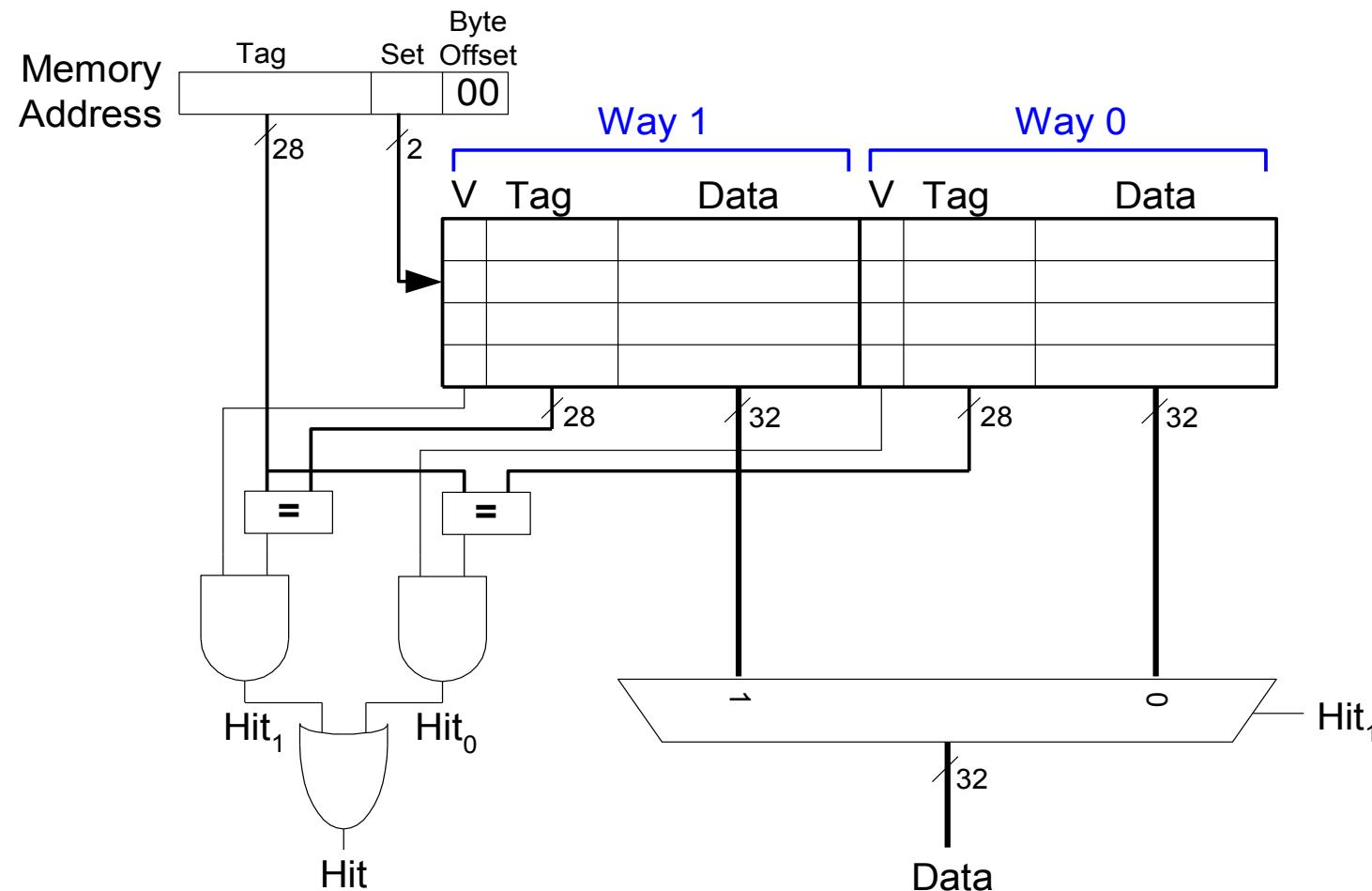
```
MOV R0, #5  
MOV R1, #0  
LOOP  CMP R0, #0  
      BEQ DONE  
      LDR R2, [R1, #0x4]  
      LDR R3, [R1, #0x24]  
      SUB R0, R0, #1  
      B    LOOP  
  
DONE
```



**Miss Rate = 10/10  
= 100%**

**Miss di conflitto**

# $N$ -Way Set Associative Cache



## ARM Assembly Code

```
MOV R0, #5  
MOV R1, #0  
LOOP  CMP R0, 0  
      BEQ DONE  
      LDR R2, [R1, #0x4]  
      LDR R3, [R1, #0x24]  
      SUB R0, R0, #1  
      B    LOOP
```

DONE

$$\begin{aligned}\text{Miss Rate} &= 2/10 \\ &= 20\%\end{aligned}$$

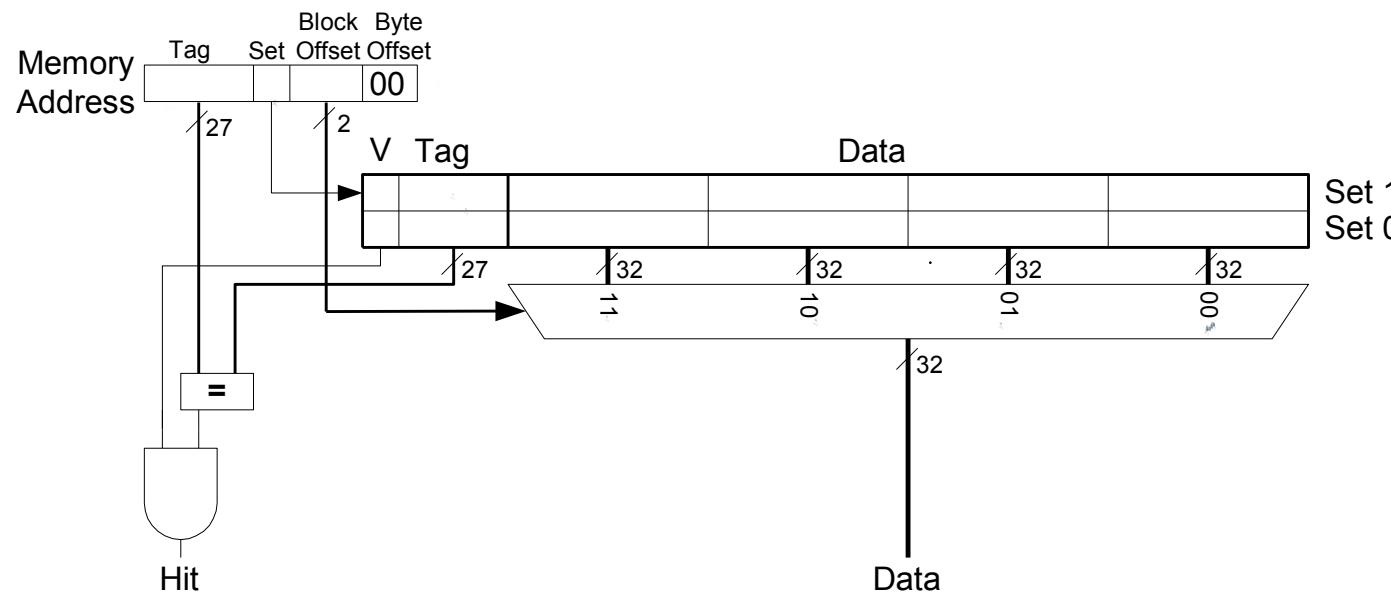
L'associatività riduce i miss di conflitto

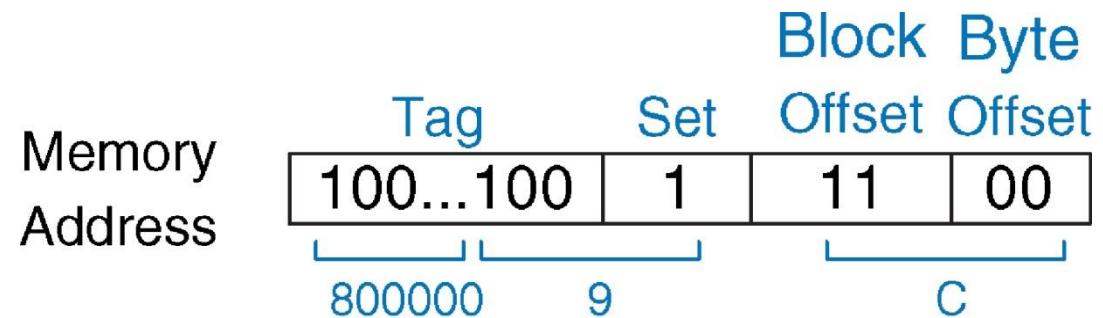
| Way 1 |         |                | Way 0 |         |                |
|-------|---------|----------------|-------|---------|----------------|
| V     | Tag     | Data           | V     | Tag     | Data           |
| 0     |         |                | 0     |         |                |
| 0     |         |                | 0     |         |                |
| 1     | 00...10 | mem[0x00...24] | 1     | 00...00 | mem[0x00...04] |
| 0     |         |                | 0     |         |                |

Set 3  
Set 2  
Set 1  
Set 0

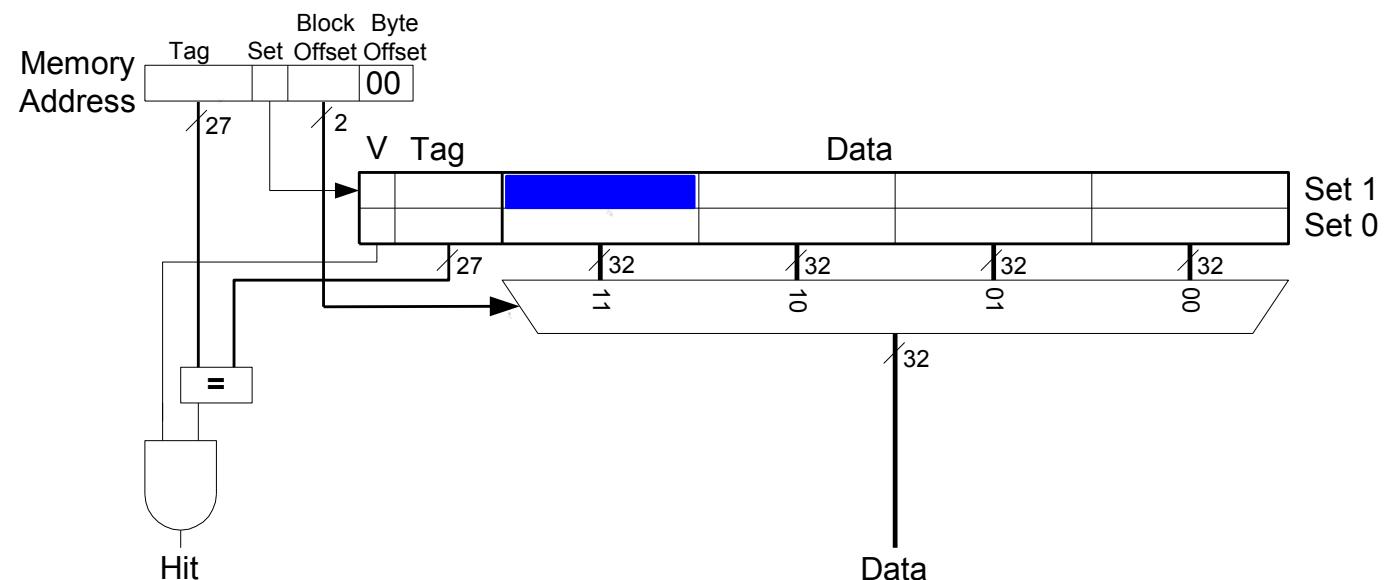
# Come sfruttare la località spaziale?

- Incrementare il block size:
  - Block size,  $b = 4$  words
  - $C = 8$  words
  - Direct mapped (1 block per set)
  - Number of blocks,  $B = 2$  ( $C/b = 8/4 = 2$ )





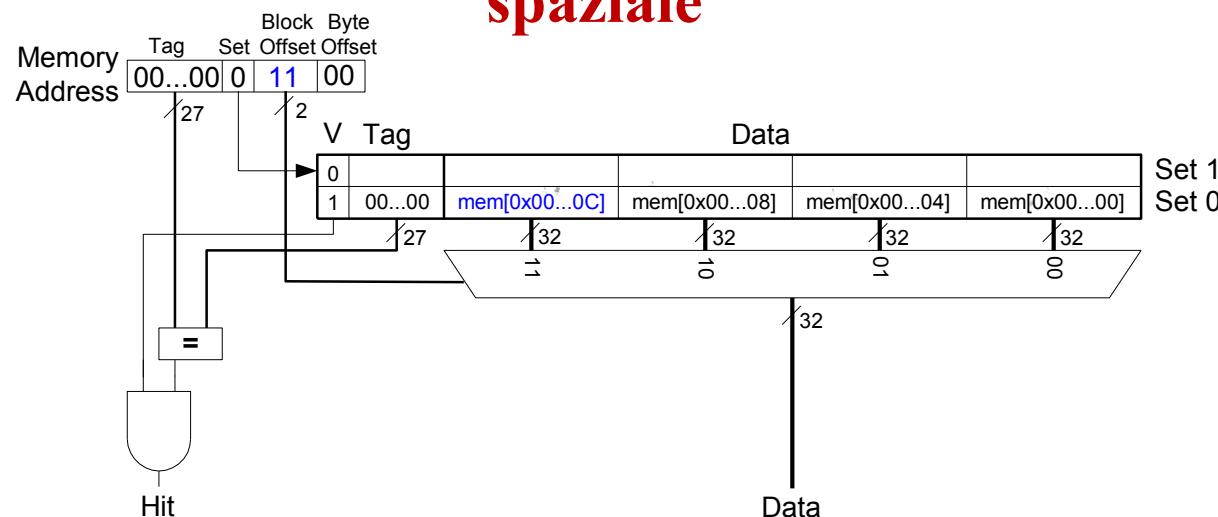
© 2007 Elsevier, Inc. All rights reserved



# ARM assembly code

```
MOV R0, #5
MOV R1, #0
LOOP  CMP R0, 0
      BEQ DONE
      LDR R2, [R1, #4]
      LDR R3, [R1, #12]
      LDR R4, [R1, #8]
      SUB R0, R0, #1
      B   LOOP
```

DONE

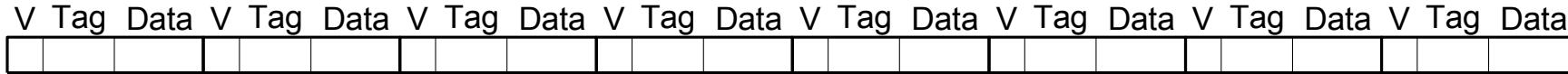


## Miss Rate = 1/15

$$= 6.67\%$$

**Blocchi più grandi riducono  
i miss obbligatori  
attraverso la località  
spaziale**

# Fully Associative Cache



- Consente la massima flessibilità nella scelta della locazione nella cache in cui posizionare un blocco di memoria
- Lo spazio della cache può essere utilizzato in modo più efficiente e questo comporta una riduzione dei miss di conflitto
- Il costo della ricerca di un blocco in cache è notevole, mentre nel caso di indirizzamento diretto è praticamente nullo
  - Una operazione di questo tipo si chiama ricerca associativa o per contenuto, in quanto corrisponde all'operazione di ricerca di un item in un elenco
- E' più costosa da costruire

# Il bit di validità

- Un ulteriore bit di controllo, chiamato ***bit di validità***, è necessario per ogni blocco.
- **Questo bit indica se il blocco contiene o meno dati validi.**
- Non deve però essere confuso con il ***bit di modifica***, menzionato in precedenza, il bit di modifica, che indica se il blocco è stato modificato o meno durante il periodo in cui è stato nella cache, serve soltanto nei sistemi che non fanno uso del metodo ***write-through***.

I bit di validità dei blocchi che si trovano nella cache vengono tutti posti a 0 nell'istante iniziale di accensione del sistema e, in seguito, ogni volta che nella memoria principale vengono caricati programmi e dati nuovi dal disco.

# Il Direct Memory Access - DMA

I trasferimenti dal disco alla memoria principale vengono effettuati mediante un meccanismo detto di DMA (Direct Memory Access). Questa sigla specifica che durante il trasferimento la CPU non interviene, perché è la logica di controllo del disco che gestisce le linee indirizzo e le linee dati, fornendo i segnali di controllo necessari.

Come vedremo successivamente, in una operazione di questo genere vengono trasferite grandi quantità di istruzioni e dati organizzati in entità dette “*pagine*” ciascuna delle quali di solito contiene migliaia di *locazioni* e quindi centinaia di *blocchi*.

Normalmente, le pagine di programma e quelle di dati vanno nella memoria centrale senza coinvolgere la cache.

# Gestione del bit di validità

**Il bit di validità di un certo blocco della cache viene posto a “1” la prima volta che esso è chiamato a contenere istruzioni o dati trasferiti dalla memoria principale;** poi, ogni volta che le locazioni di un blocco della memoria principale vengono interessate da un trasferimento di nuove istruzioni o di nuovi dati dal disco, viene effettuato un controllo per determinare se qualcuno dei blocchi che stanno per essere sovrascritti fossero o meno presenti nella cache.

**Se nella cache c’è una copia del blocco, il suo bit di validità viene posto a “0”; in tal modo, si garantisce che nella cache non ci siano dati obsoleti.**

Il bit di validità a “0” candida immediatamente il blocco ad essere sovrascritto.

# Svuotamento (flush) della cache

Una situazione dello stesso tipo si determina quando viene effettuato un trasferimento tramite DMA dalla memoria al disco e la cache utilizza un protocollo write-back.

**In questo caso, i dati nella memoria potrebbero non riflettere i cambiamenti che possono essere stati fatti sulla copia dei dati presenti nella cache.**

Una possibile soluzione a tale problema consiste nello ***svuotamento (flush)*** della cache forzando i dati con il bit di modifica attivo a essere ricopiatati nella memoria principale prima che venga effettuato il trasferimento tramite DMA.

# “Coerenza” della cache

Il sistema operativo è in grado di fare tutto tutte le operazioni necessarie in modo molto efficiente, senza peggiorare significativamente le prestazioni, visto che le operazioni di ingresso/uscita di scrittura non sono molto frequenti.

Questa necessità di garantire che due diverse entità (i sottosistemi della CPU e del DMA nel caso presente) utilizzino la stessa copia dei dati viene chiamata problema della ***coerenza della cache***.

# Algoritmi di sostituzione ( 1 / 2 )

- In una cache ad indirizzamento diretto, la posizione di ogni blocco è predefinita, quindi non esiste alcuna strategia di sostituzione.
- Nelle memorie cache *associative* e *set-associative*, esiste invece una certa flessibilità.

**Quando un nuovo blocco deve essere portato nella cache, e tutte le posizioni che potrebbe occupare contengono dati validi, il controllore della cache deve decidere quale blocco, tra quelli esistenti, soprascrivere.**

Questa è una decisione importante, perché la scelta potrebbe essere determinante per le prestazioni del sistema.

## Algoritmi di sostituzione ( 2 / 2 )

In generale, l'obiettivo è quello di **mantenere nella cache quei blocchi che hanno una maggior possibilità di essere nuovamente utilizzati nel prossimo futuro.** ma, non è facile prevedere i blocchi a cui si farà riferimento.

Una possibile strategia è di tener presente che la ***località dei riferimenti*** suggerisce che i blocchi a cui c'è stato accesso di recente hanno elavata probabilità di essere utilizzati nuovamente entro breve tempo.

# Sostituzione con algoritmo LRU

Quando bisogna eliminare dalla cache un blocco, è sensato soprascrivere quello a cui non si accede da più tempo. Tale blocco prende il nome di blocco *utilizzato meno di recente (Least Recently Used, LRU)*, e la tecnica si chiama *algoritmo di sostituzione LRU*.

Per utilizzare l'algoritmo LRU, il controllore della cache deve mantenere traccia di tutti gli accessi ai blocchi mentre l'elaborazione prosegue.

Il blocco LRU di una memoria cache *set-associativa* con insiemi di quattro blocchi, può, ad esempio, essere indicato da un contatore di 2 bit associato a ciascun blocco, il cui contenuto sia opportunamente gestito secondo un protocollo automatico

# Gestione del contatore di accessi

**Quando si ha un successo nell'accesso al blocco (*hit*):**

- il contatore di quel blocco viene posto a 0. I contatori con valori originariamente inferiori a quelli del blocco a cui si accede, vengono incrementati di uno, mentre tutti gli altri rimangono invariati.

**Quando, invece, l'accesso fallisce (*miss*) si aprono due possibilità:**

- l'insieme non è pieno, il contatore associato al nuovo blocco caricato dalla memoria principale viene posto a 0 e il valore di tutti gli altri contatori viene incrementato di 1.
- l'insieme è pieno, si rimuove il blocco, il cui contatore ha valore 3, e si pone il nuovo blocco al suo posto, mettendo il contatore a 0. Gli altri tre contatori dell'insieme vengono incrementati di un'unità.

**E' facile verificare che i valori dei contatori dei blocchi occupati sono sempre diversi.**

# Alternative all'algoritmo LRU

L'algoritmo LRU è stato utilizzato ampiamente, con buone prestazioni, in numerosi schemi di accesso. Ma in alcune situazioni, per esempio nel caso di accessi sequenziali a un vettore di elementi che è leggermente troppo grande per poter stare tutto nella cache, le prestazioni sono molto scadenti

Le prestazioni dell'algoritmo LRU possono essere migliorate introducendo una piccola dose di casualità nella scelta del blocco da sostituire.

Sono stati proposti molti altri algoritmi di sostituzione più o meno complessi, che richiedono talvolta anche notevoli complicazioni hardware, ma l'algoritmo più semplice, e spesso anche piuttosto efficace, consiste nello **scegliere in modo completamente casuale il blocco da sostituire**.

## ARM Assembly Code

```
MOV R0, #0  
LDR R1, [R0, #4]  
LDR R2, [R0, #0x24]  
LDR R3, [R0, #0x54]
```

Way 1                          Way 0

| V | U | Tag      | Data           | V | Tag      | Data           |
|---|---|----------|----------------|---|----------|----------------|
| 0 | 0 |          |                | 0 |          |                |
| 0 | 0 |          |                | 0 |          |                |
| 1 | 0 | 00...010 | mem[0x00...24] | 1 | 00...000 | mem[0x00...04] |
| 0 | 0 |          |                | 0 |          |                |

Set 3 (11)  
Set 2 (10)  
Set 1 (01)  
Set 0 (00)

(a)

Way 1                          Way 0

| V | U | Tag      | Data           | V | Tag      | Data           |
|---|---|----------|----------------|---|----------|----------------|
| 0 | 0 |          |                | 0 |          |                |
| 0 | 0 |          |                | 0 |          |                |
| 1 | 1 | 00...010 | mem[0x00...24] | 1 | 00...101 | mem[0x00...54] |
| 0 | 0 |          |                | 0 |          |                |

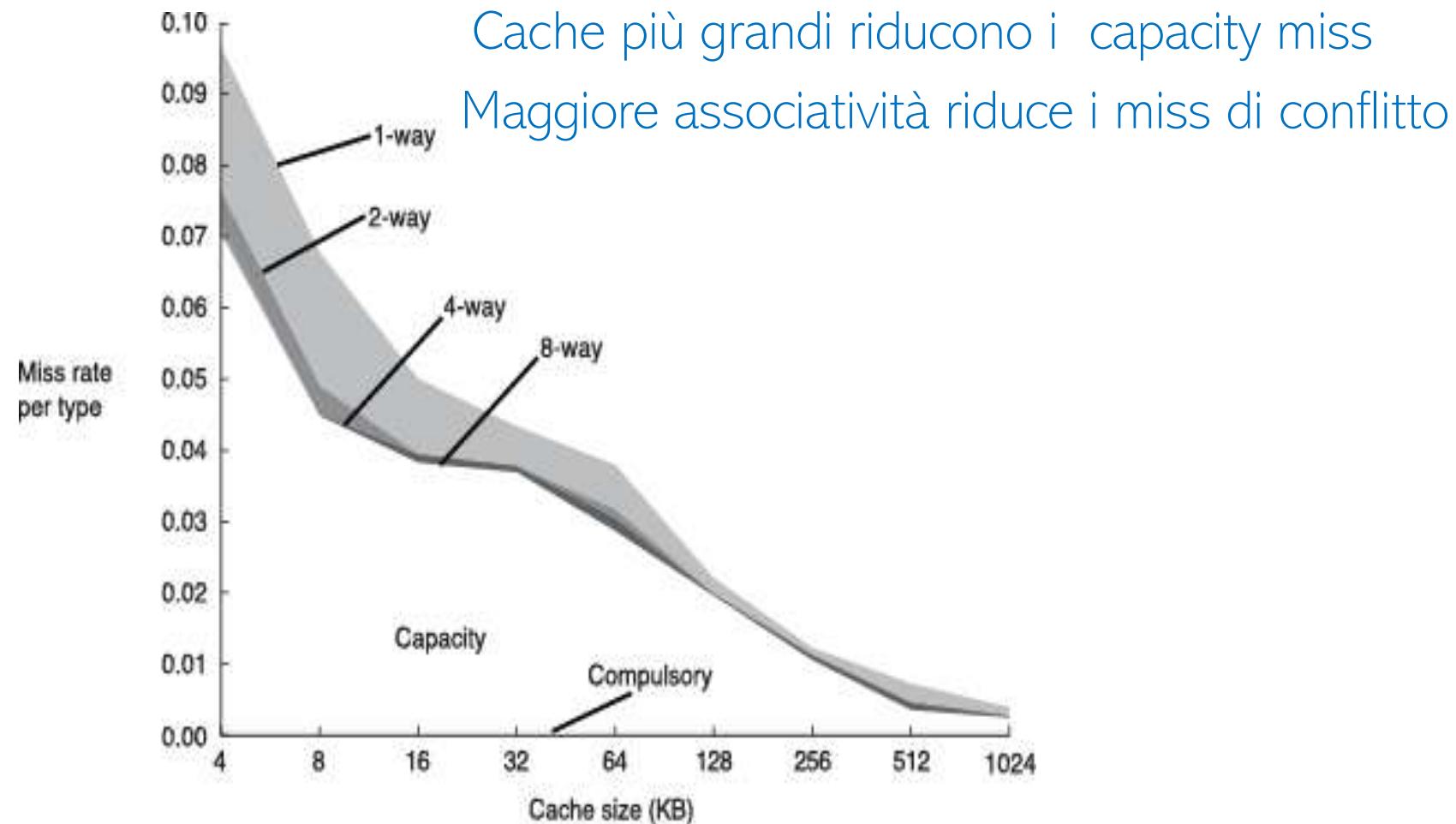
Set 3 (11)  
Set 2 (10)  
Set 1 (01)  
Set 0 (00)

(b)

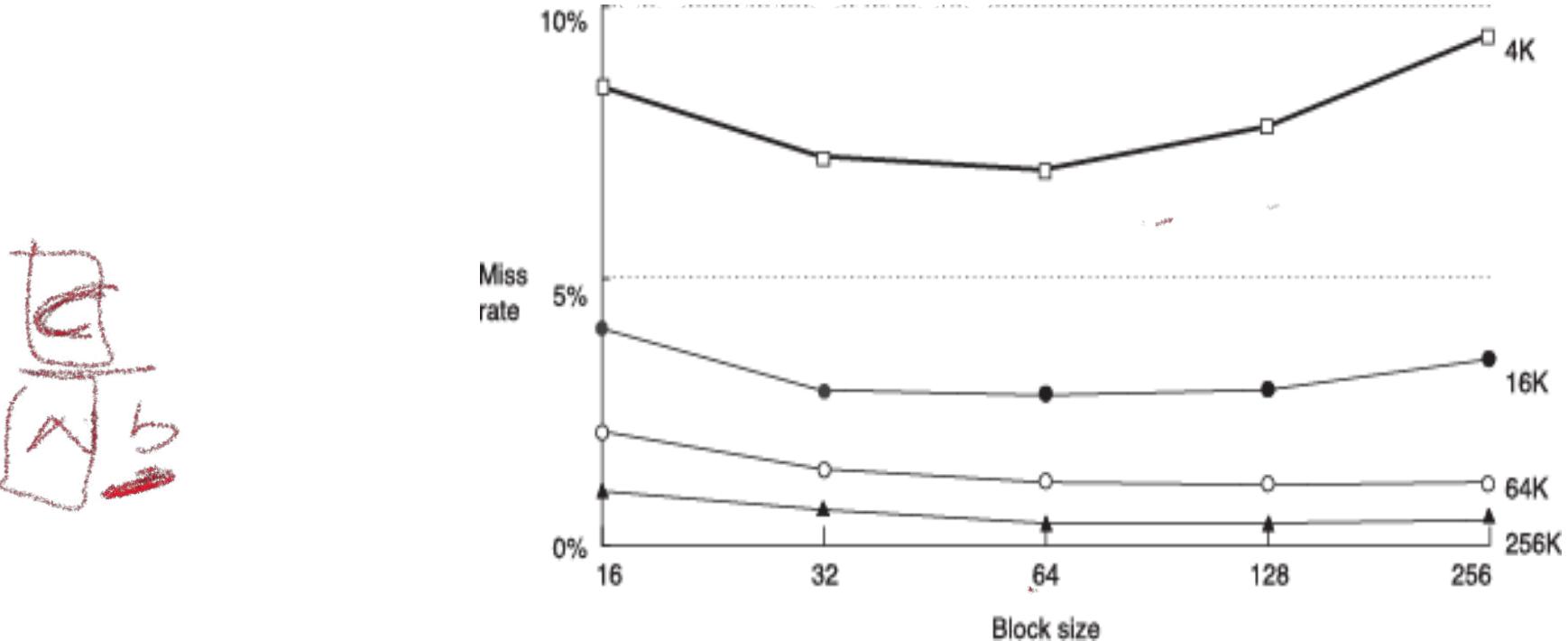
# Tipi di Miss

- **Compulsory:** primo accesso ad un dato
- **Capacity:** la cache è troppo piccola per contenere tutti i dati di interesse
- **Conflict:** due dati sono indirizzati nella stessa locazione nella cache

**Miss penalty:** il tempo necessario per recuperare un blocco da un livello più basso nella gerarchia delle memorie



Adapted from Patterson & Hennessy, *Computer Architecture: A Quantitative Approach*, 2011



- Blocchi più grandi riducono i miss obbligatori sfruttando la località spaziale
- Data una cache di dimensioni fissate, all'aumentare del block size il numero di set decrementa ( $S \times N_{xb} = C$ ). Quindi aumentano i miss di conflitto

## Frazionamento della memoria in moduli

**E utile che i trasferimenti da/verso le unità più veloci possano essere effettuati alla stessa frequenza dell'unità più veloce.**

Ciò non può avvenire se si accede nello stesso modo all'unità più lenta e a quella più veloce.

**Tale risultato può invece essere raggiunto se si sfrutta il parallelismo nell'organizzazione dell'unità più lenta.**

Un modo efficiente per introdurre il parallelismo nella gestione della memoria principale consiste nell'impiego di un'organizzazione *interallacciata*.

Se la memoria principale di un calcolatore è strutturata come una collezione di moduli fisicamente separati, ognuno con il proprio registro degli indirizzi (Address Register, AR) e registro dei dati (Data Register, DR), le operazioni di accesso alla memoria possono procedere contemporaneamente in più di un modulo.

**Così, si può incrementare la frequenza aggregata di trasmissione delle parole da o verso il sistema di memoria principale.**

# Distribuzione degli indirizzi nei moduli

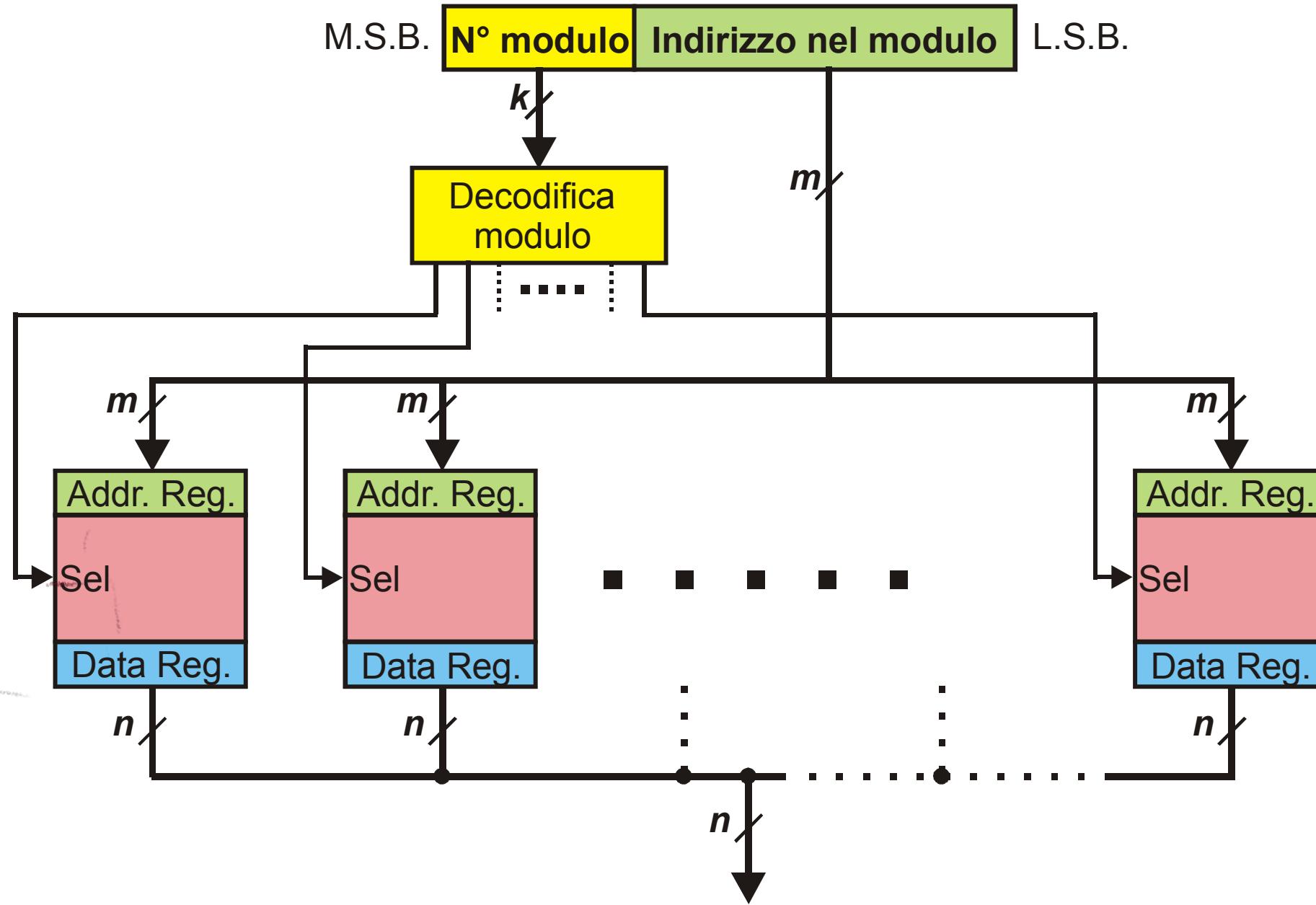
Il modo in cui gli indirizzi individuali sono distribuiti attraverso i moduli è importante per stabilire il numero medio di moduli che può essere tenuto occupato mentre le elaborazioni procedono.

**Esistono due metodi di distribuzione degli indirizzi.**

Nel primo, l'indirizzo di memoria generato dalla CPU viene decodificato nel seguente modo:

I  $k$  bit più significativi indicano uno degli  $n$  moduli, e gli  $m$  bit meno significativi indicano una particolare parola all'interno del modulo. Quando si accede a locazioni consecutive, come accade quando si copia un blocco di dati nella cache, viene coinvolto soltanto un modulo. Nello stesso tempo, tuttavia, dispositivi con possibilità di accesso diretto alla memoria (DMA) possono aver accesso alle informazioni contenute in altri moduli della memoria.

## Indirizzo di memoria



# Distribuzione degli indirizzi nei moduli

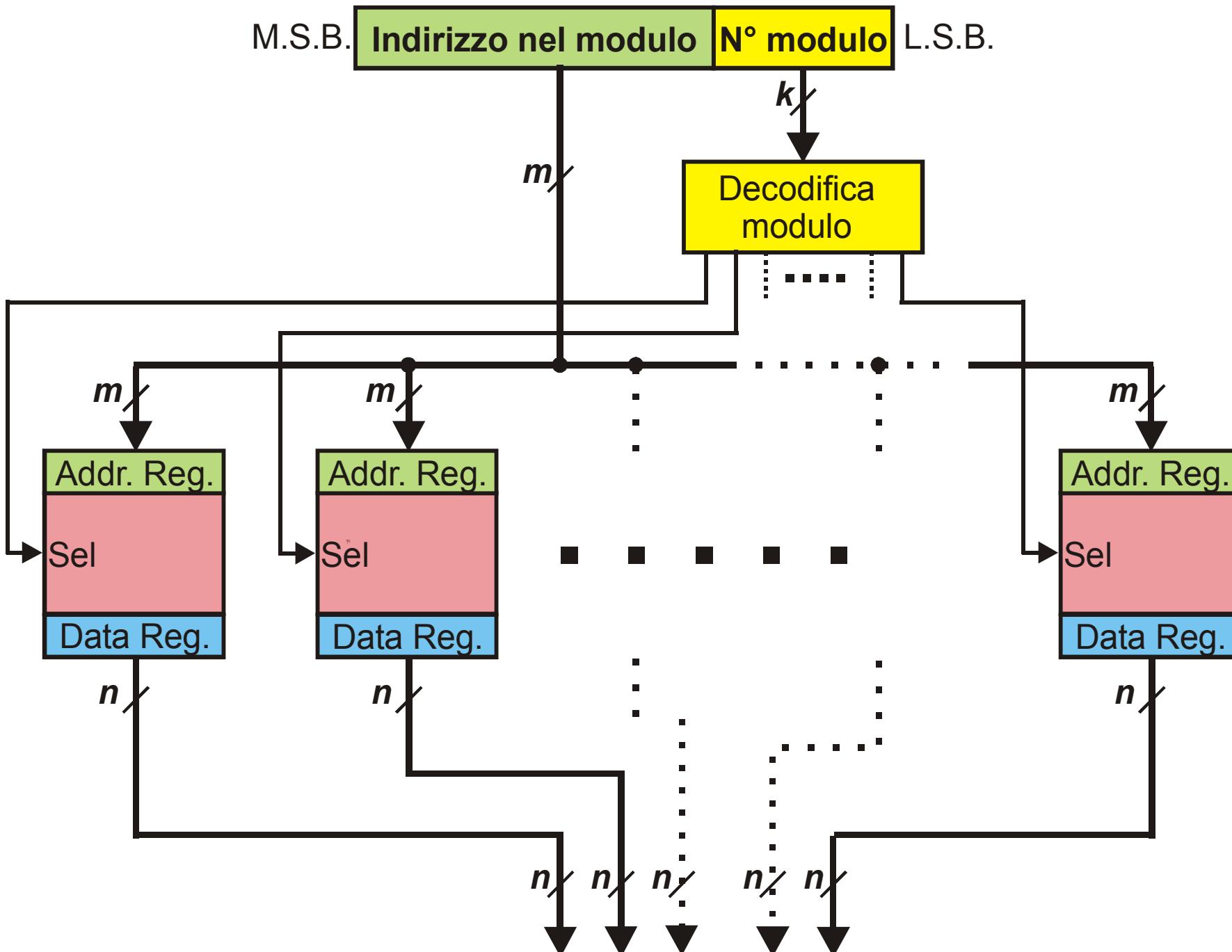
Il secondo e più efficiente modo di indirizzare i moduli è chiamato ***interallacciamento della memoria***.

I  $k$  bit meno significativi dell'indirizzo di memoria selezionano un modulo, mentre gli  $m$  bit più significativi indicano una locazione all'interno del modulo. In questo modo, indirizzi consecutivi sono posizionati in moduli successivi; così facendo, ogni componente del sistema che faccia richiesta di accesso a locazioni di memoria consecutive può tenere occupati vari moduli in un qualsiasi istante. Ciò consente di ottenere sia accessi a un blocco di dati più veloci, sia una più elevata media complessiva di utilizzo della memoria.

**Per realizzare la struttura interallacciata, devono esserci  $2^k$  moduli.**

In caso contrario, ci sono degli stati vuoti associati a locazioni inesistenti nello spazio di indirizzamento della memoria.

# Indirizzo di memoria

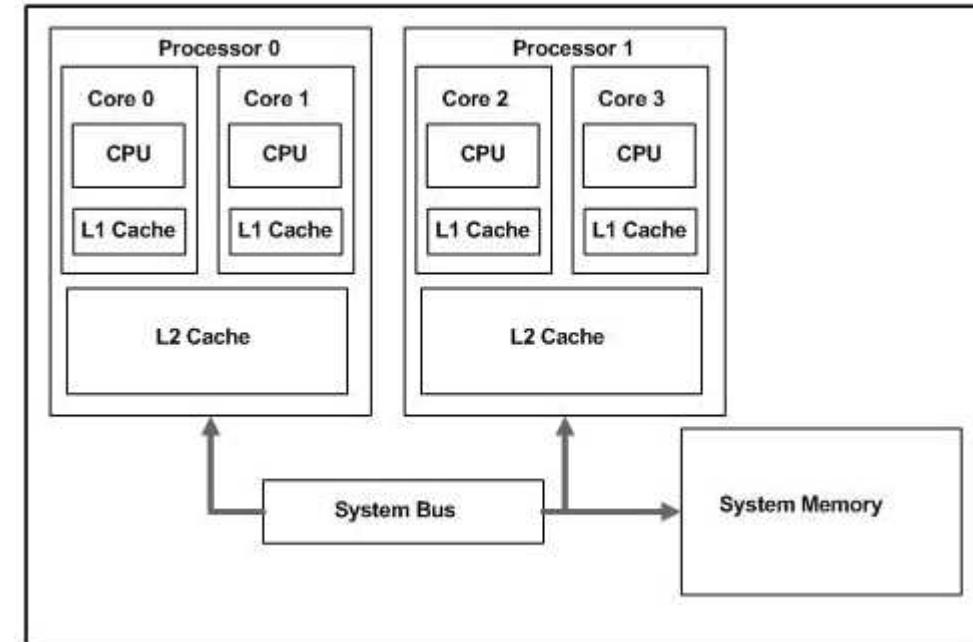


# Cache multi livello

- Oggigiorno la memoria cache costituisce una gerarchia di 2 o 3 livelli: L1, L2, L3
- Vi possono essere due tipologie di cache multi livello: inclusiva o esclusiva
- Supponiamo due livelli.
- Inclusiva: L2 contiene i dati presenti in L1 (similmente L2 è contenuta in L3)
  - L2 contiene sia i dati correnti di L1, sia quelli che erano presenti in precedenza e sono stati sovrascritti mediante l'algoritmo di sostituzione (copy back)
  - Per mantenere la coerenza L1 ha una politica di write-through verso L2, e così L2 verso L3
  - Miss in L1 e hit in L2: il blocco relativo viene copiato da L2 a L1 (in caso di sostituzione un dato va da L2 a L1 e un'altro da L1 a L2)
  - Miss in L1 e L2: il blocco dalla main memory viene copiato sia in L1 che L2
- Esclusiva: L2 contiene solo i dati di copy back da L1 (victim cache)
  - La memoria complessiva della cache è data dalla somma delle capacità dei vari livelli, in realtà nelle moderne architetture la capacità di L2 è un ordine di grandezza rispetto a quella di L1, e così fra L3 e L2, il guadagno rispetto alle cache inclusive non è molto significativo).
  - Miss in L1 e hit in L2: Le linee di cache di L1 e L2 vengono scambiati fra loro, cioè la linea di cache di L1 viene memorizzata in L2 e la linea di L2 in L1.
  - Miss in L1 e L2: Il dato letto dalla memoria è memorizzato direttamente in L1 e la linea di cache rimpiazzata di L1 (*victim data*) è trasferita in L2 rimpiazzando un'altra linea di cache secondo la politica di rimpiazzo usata.

# Cache condivise

- Nelle architetture multi core L2 o L3 possono essere condivisi fra più core di uno stesso processore, questo consente:
  - Un uso più efficiente del livello condiviso
    - Se un core è inattivo, allora l'altro può utilizzare per sé il livello condiviso
  - Nella programmazione multi-threading consente di utilizzare in maniera più efficiente dati condivisi
    - Un core può eseguire un pre-/post-processing di dati forniti dall'altro core
  - Riduce il front-side bus traffic
    - Un unico livello condiviso che si interfaccia con la main memory



# La Memoria Virtuale

Agli albori dell'informatica la memoria centrale di un computer raramente superava le 4k parole ed i sistemi operativi erano, a dir poco, essenziali.

A quell'epoca, chiunque si disponesse a scrivere un programma doveva prima informarsi di quale fosse la disponibilità di spazio nella memoria centrale del sistema da adoperare, al netto delle necessità della CPU e della "parte residente" del sistema operativo.

Fatto il conteggio doveva scrivere un programma fatto di pezzi da mandare in esecuzione uno per volta (*overlay, sovrapposizione*), registrando i risultati parziali in gruppi di memoria i cui indirizzi fossero noti anche ai successivi pezzi di programma.

Questi li prelevavano per usarli ed, a loro volta registravano i risultati in locazioni prefissate.

Senza questa procedura il programma non poteva essere eseguito.

# La Memoria Virtuale

Tutto questo è oggi completamente ignoto agli utilizzatori di computer.

Il fatto che talvolta siano indicati dei requisiti minimi del sistema per usare un certo programma, è un semplice suggerimento per ottenere dal programma un comportamento ragionevole, senza lunghissimi tempi di attesa tra due operazioni successive.

Oggi è assolutamente normale che programmi che tra occupazione diretta ed indiretta avrebbero bisogno di 25-30 GB girino su macchine con memoria centrale da 16 GB, grazie alla tecnica della “*memoria virtuale*”, inventata da un gruppo di ricercatori inglesi di Manchester nel 1961 ed adottata dall'IBM negli anni '70.

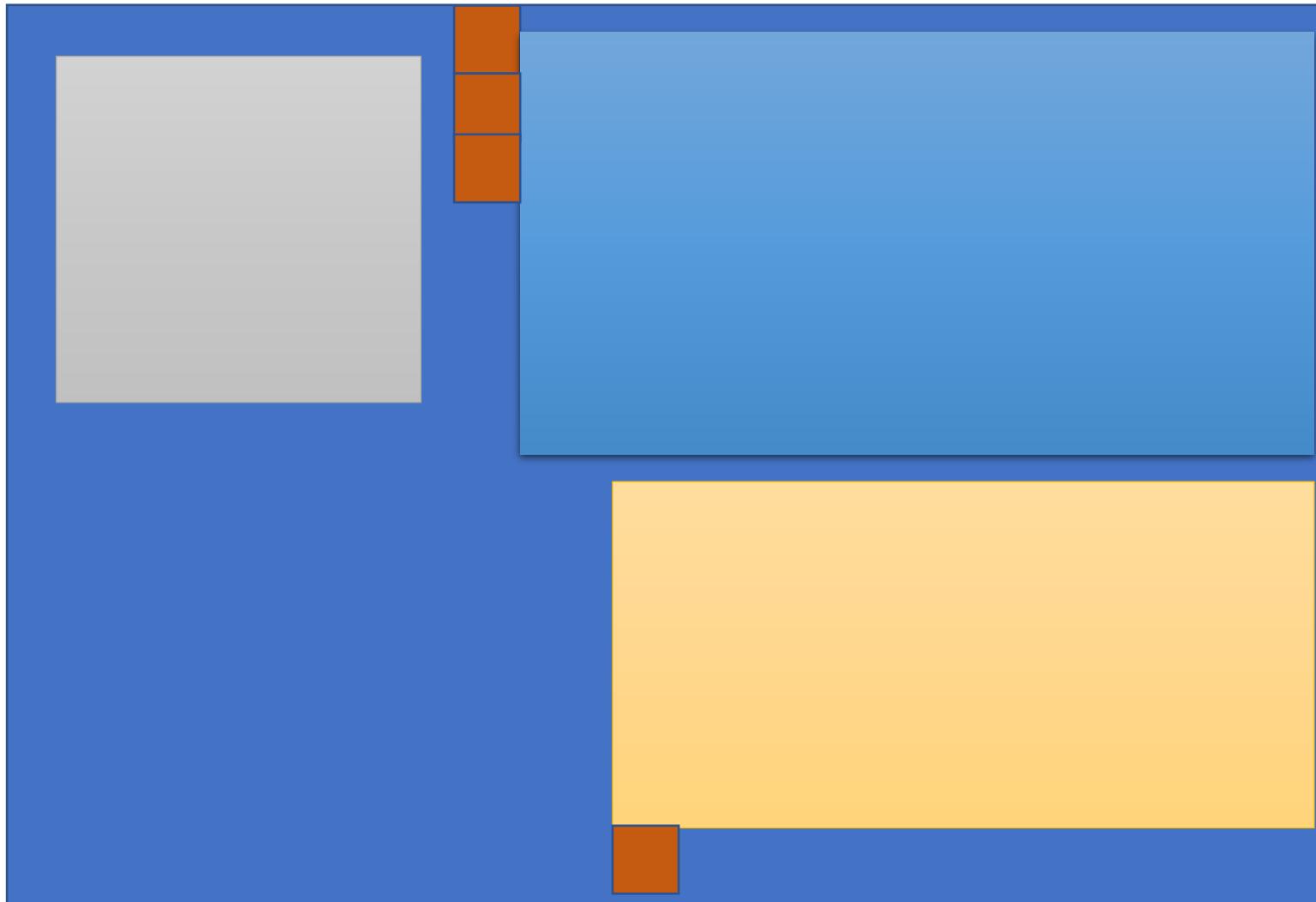
## La gestione è affidata al sistema operativo

In estrema sintesi la tecnica della *memoria virtuale* prevede che il sistema operativo assegni una parte degli spazi liberi nella memoria centrale ai segmenti di un programma installato su disco.

Il sistema operativo seguirà criteri prefissati, assegnando uno spazio compreso tra un minimo ed un massimo, in funzione della dimensione del programma.

Gli spazi potranno essere anche non continui.

Il risultato è che programmi piccoli saranno trasferiti interamente ma, naturalmente, in funzione della mappa di occupazione corrente potranno essere dislocati in maniera frammentata in qualsiasi parte della memoria.



# La memoria è gestita dinamicamente

Programmi di grandi dimensioni avranno a disposizione spazi insufficienti a contenerli completamente e saranno soggetti a continue operazioni di “*swapping*”, cioè di sostituzione dinamica dei loro segmenti nello spazio di memoria centrale assegnato.

L'operatore può imporre al sistema operativo di dedicare più spazio ad un programma, ma se esso usa una gran parte del campo d'indirizzamento del processore, non potrà in ogni caso essere completamente trasferito in memoria.

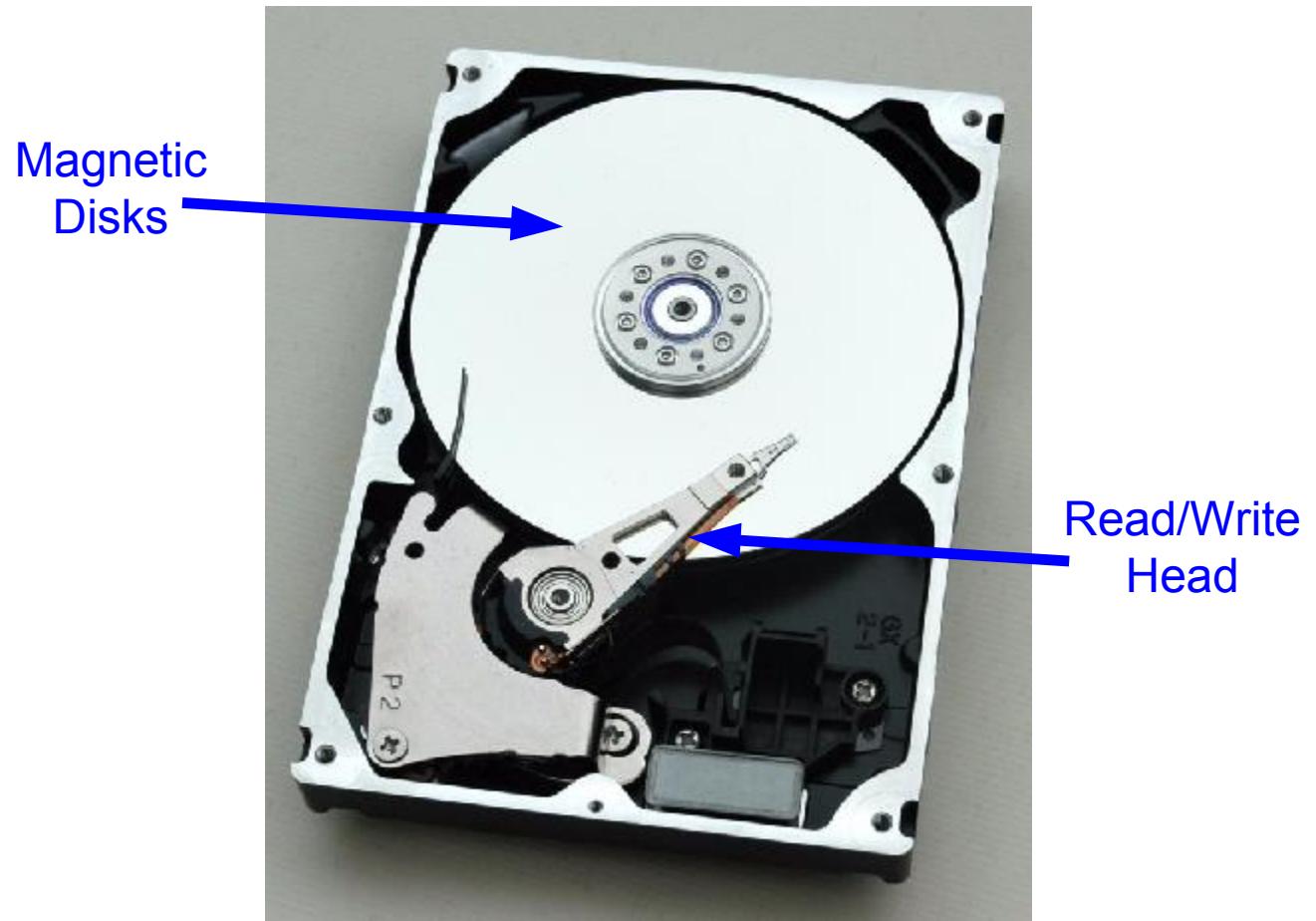
La tecnica di gestione degli spazi ha molti punti in comune con il funzionamento di una memoria cache.

Anche in questo caso, infatti, se gli spazi a disposizione nella memoria principale sono saturati, e il programma accede ad una locazione di memoria che non corrisponde ad un indirizzo esistente nella memoria centrale, si deve ricorrere ad un algoritmo di sostituzione.

# Dimensioni della pagina di memoria

La parte di programma contenente l'indirizzo richiesto viene prelevato dal disco (trasferimento in DMA) e sostituita in memoria ad una delle parti che vi risiedevano.

Poiché il tempo di accesso al disco costituisce una penalità di fallimento molto più pesante, in termini di tempo rispetto a quella di una cache, l'entità minima da trasferire è presa molto più grande del blocco della cache ed è una *pagina di memoria* di solito compresa tra 2k e 16k; normalmente in un accesso al disco vengono scambiate molte pagine.



Takes milliseconds to *seek* correct location on disk

# Overview

- **Indirizzi virtuali**
  - I programmi usano indirizzi virtuali
  - L'intero spazio degli indirizzi virtuali è memorizzato nell'hard drive (memoria di massa)
  - Essendo la memoria fisica (main memory) più piccola della memoria di massa, solo un sottoinsieme degli indirizzi virtuali è presente nella main memory
  - gli indirizzi virtuali vengono tradotti in indirizzi fisici della main memory
  - I dati che non vengono ritrovati nella main memory sono prelevati dalla memoria di massa (operazione molto dispendiosa in termini di tempo)
- **Memory Protection**
  - Il Sistema operativo associa ad ogni programma un suo spazio di indirizzi virtuali
  - Così un programma può non essere consapevole che altri programmi sono in esecuzione

# Analogie tra Cache e memoria virtuale

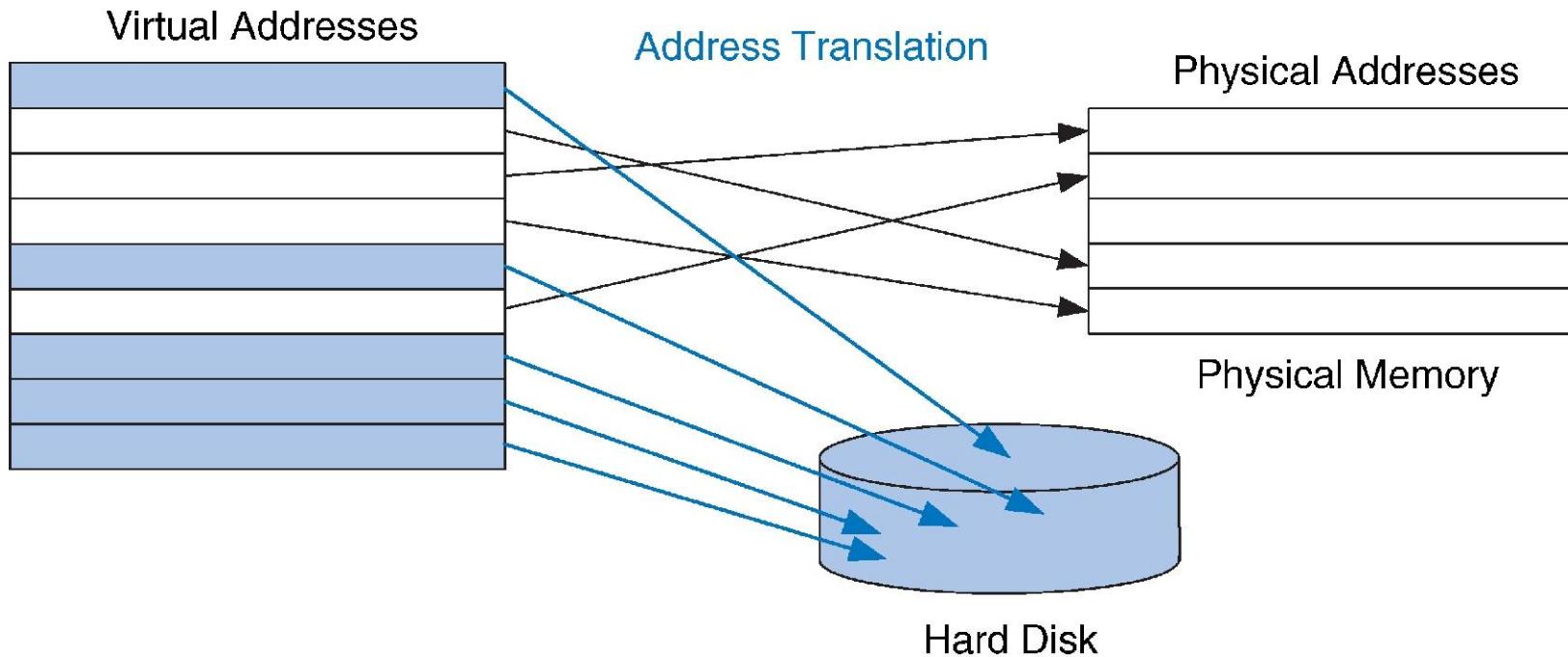
| Cache        | Virtual Memory      |
|--------------|---------------------|
| Block        | Page                |
| Block Size   | Page Size           |
| Block Offset | Page Offset         |
| Miss         | Page Fault          |
| Tag          | Virtual Page Number |

La memoria fisica (main memory) agisce come cache per la memoria virtuale

# Definizioni per la memoria virtuale

- **Page size:** dimensione dei segmenti di memoria trasferiti dalla memoria di massa alla main memory (2K-16K)
- **Address translation:** determinare l'indirizzo fisico nella main memory a partire da un indirizzo virtuale
- **Page table:** lookup table usata per tradurre gli indirizzi virtuali in indirizzi fisici

# Indirizzi fisici e virtuali



- Data la grandezza delle pagine e la località spaziale e temporale durante l'esecuzione di un programma, la maggior parte degli accessi sono ritrovati nella main memory
- Ma i programmi sfruttano la maggior capacità della memoria virtuale

# Esempio di memoria virtuale

## System:

Virtual memory size:  $2 \text{ GB} = 2^{31}$  bytes

Physical memory size:  $128 \text{ MB} = 2^{27}$  bytes

Page size:  $4 \text{ KB} = 2^{12}$  bytes

## Organization:

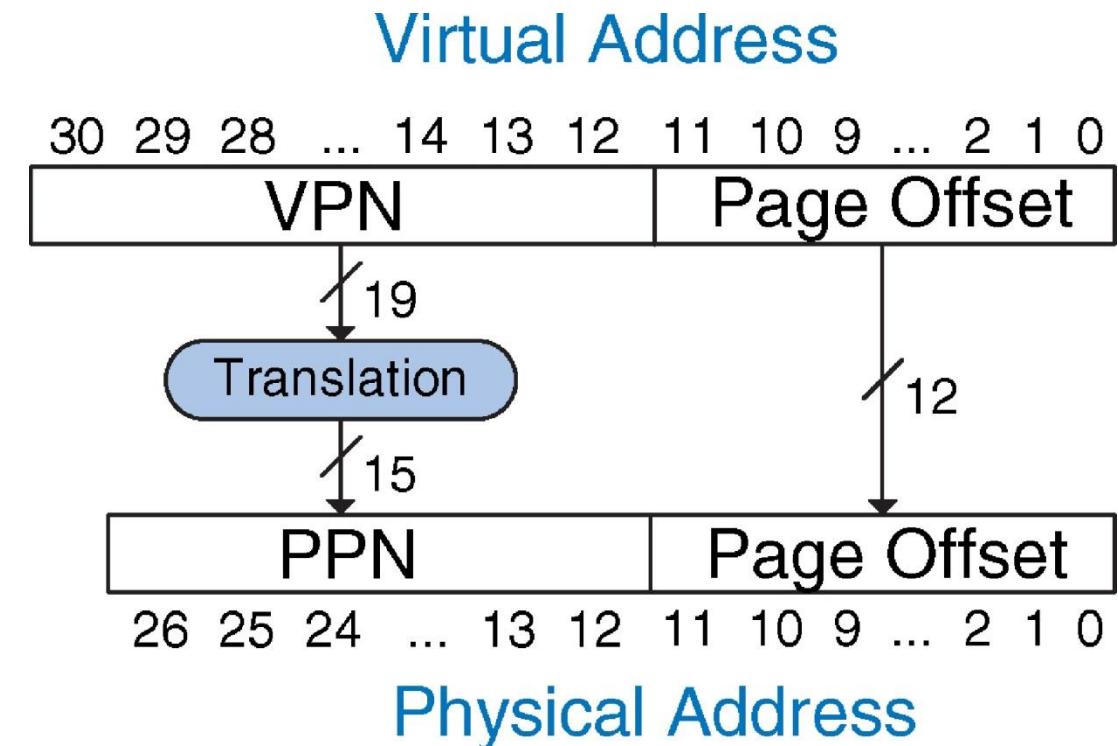
Virtual address: **31** bits

Physical address: **27** bits

Page offset: **12** bits

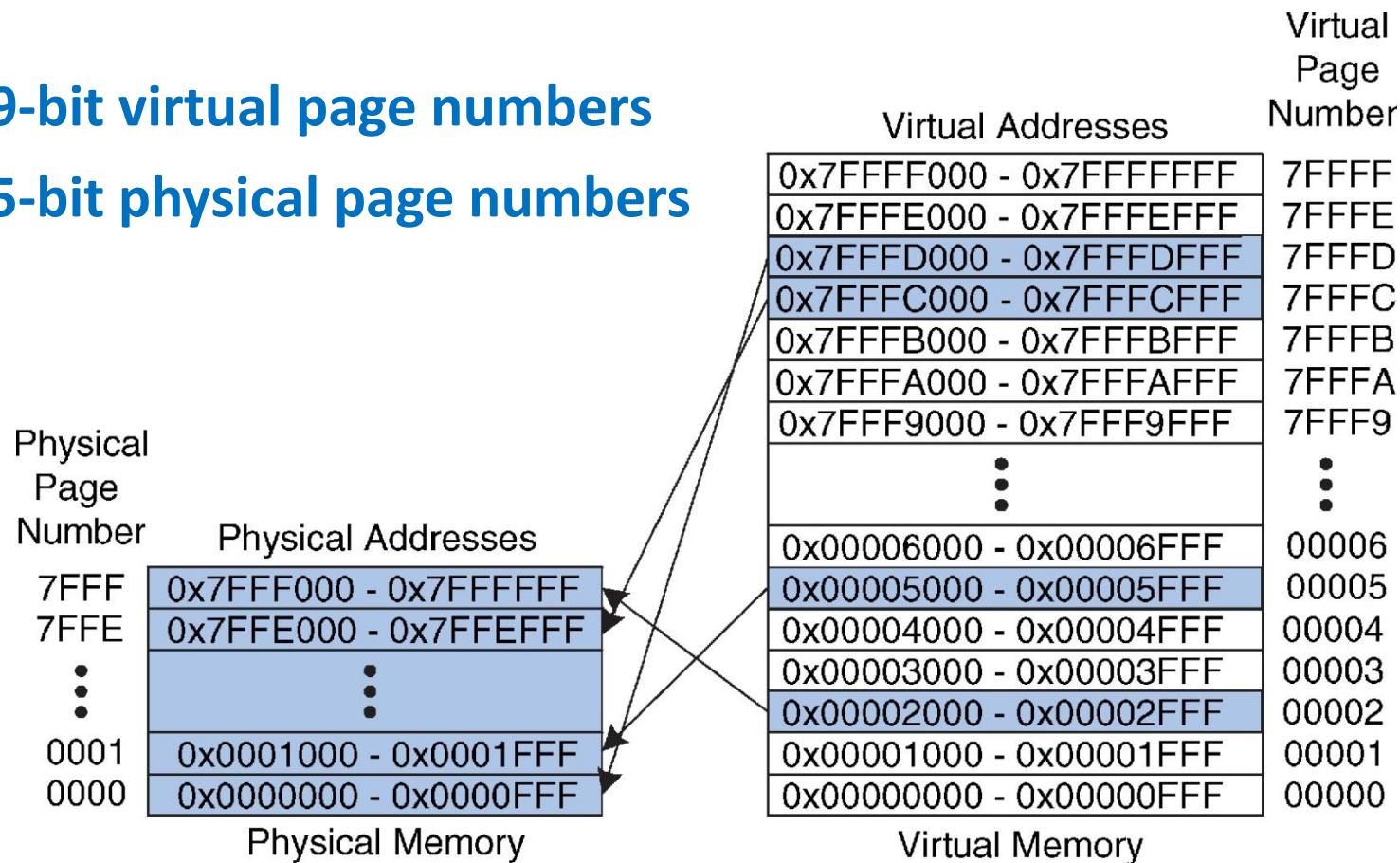
# Virtual pages =  $2^{31}/2^{12} = 2^{19}$  (VPN = **19** bits)

# Physical pages =  $2^{27}/2^{12} = 2^{15}$  (PPN = **15** bits)



# Esempio di memoria virtuale

- 19-bit virtual page numbers
- 15-bit physical page numbers



# Esempio di memoria virtuale

- 19-bit virtual page numbers
- 15-bit physical page numbers

| Physical<br>Page<br>Number | Physical Addresses     |
|----------------------------|------------------------|
| 7FFF                       | 0x7FFF000 - 0x7FFFFFFF |
| 7FFE                       | 0x7FFE000 - 0x7FFEFFFF |
| ⋮                          | ⋮                      |
| 0001                       | 0x0001000 - 0x0001FFF  |
| 0000                       | 0x0000000 - 0x0000FFF  |

Physical Memory

Virtual Memory size  $2G = 31$  bit, quindi 8 esadecimali con il più significativo che arriva fino a 7

| Virtual Addresses       | Virtual<br>Page<br>Number |
|-------------------------|---------------------------|
| 0x7FFF000 - 0x7FFFFFFF  | 7FFFF                     |
| 0x7FFE000 - 0x7FFEFFFF  | 7FFFE                     |
| 0x7FFFD000 - 0x7FFFDFFF | 7FFFD                     |
| 0x7FFFC000 - 0x7FFFCFFF | 7FFFC                     |
| 0x7FFFB000 - 0x7FFFBFFF | 7FFFB                     |
| 0x7FFFA000 - 0x7FFFAFFF | 7FFFA                     |
| 0x7FFF9000 - 0x7FFF9FFF | 7FFF9                     |
| ⋮                       | ⋮                         |
| 0x00006000 - 0x00006FFF | 00006                     |
| 0x00005000 - 0x00005FFF | 00005                     |
| 0x00004000 - 0x00004FFF | 00004                     |
| 0x00003000 - 0x00003FFF | 00003                     |
| 0x00002000 - 0x00002FFF | 00002                     |
| 0x00001000 - 0x00001FFF | 00001                     |
| 0x00000000 - 0x00000FFF | 00000                     |

Virtual Memory

Page 4K= 12 bit=3 esadecimali

# Esempio memoria virtuale

Quale è l'indirizzo fisico  
dell'indirizzo **0x247C**?

- VPN = **0x00002**
- VPN 0x00002 maps to PPN **0x7FFF**
- 12-bit page offset: **0x47C**
- Physical address = **0x7FFF47C**

| Physical Page Number | Physical Addresses     |
|----------------------|------------------------|
| 7FFF                 | 0x7FFF000 - 0x7FFFFFFF |
| 7FFE                 | 0x7FFE000 - 0x7FFEFFFF |
| ⋮                    | ⋮                      |
| 0001                 | 0x0001000 - 0x0001FFF  |
| 0000                 | 0x0000000 - 0x0000FFF  |

Physical Memory

| Virtual Page Number | Virtual Addresses       |
|---------------------|-------------------------|
| 7FFF                | 0x7FFFF000 - 0x7FFFFFFF |
| 7FFE                | 0x7FFFE000 - 0x7FFEFFFF |
| 7FFD                | 0x7FFFD000 - 0x7FFFDFFF |
| 7FFC                | 0x7FFFC000 - 0x7FFFCFFF |
| 7FFB                | 0x7FFFB000 - 0x7FFFBFFF |
| 7FFA                | 0x7FFFA000 - 0x7FFFAFFF |
| 7FF9                | 0x7FFF9000 - 0x7FFF9FFF |
| ⋮                   | ⋮                       |
| 0006                | 0x00006000 - 0x00006FFF |
| 0005                | 0x00005000 - 0x00005FFF |
| 0004                | 0x00004000 - 0x00004FFF |
| 0003                | 0x00003000 - 0x00003FFF |
| 0002                | 0x00002000 - 0x00002FFF |
| 0001                | 0x00001000 - 0x00001FFF |
| 0000                | 0x00000000 - 0x00000FFF |

Virtual Memory

# La Memory Management Unit (MMU)

Quello a cui fa riferimento il processore nell'elaborazione è detto indirizzo “*virtuale*” o logico ed ha come unica limitazione di essere entro il campo di indirizzamento massimo del processore.

Il compito di tradurre questo indirizzo in un indirizzo che punti ad una locazione effettivamente esistente nel sistema (*indirizzo fisico*) è demandato ad una struttura logica contenuta nel chip del processore, che si chiama **Memory Management Unit (MMU)** e che opera sotto il controllo software del sistema operativo.

# Traduzione degli indirizzi

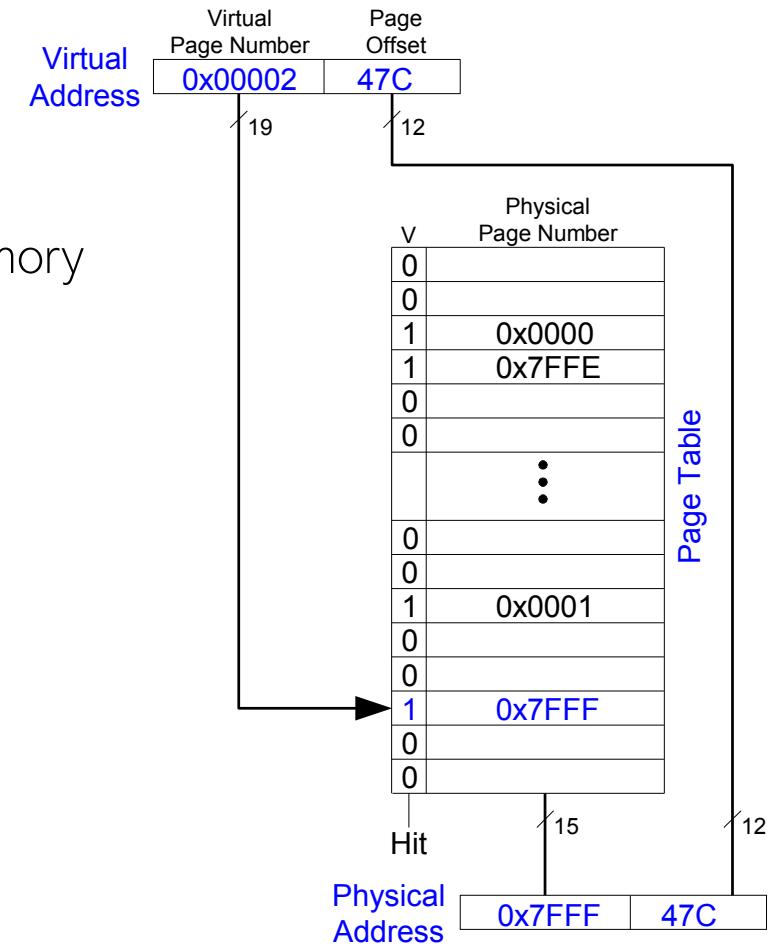
La memoria virtuale si comporta come una sorta di cache completamente associativa per il disco.

Tuttavia, se si gestisse la memoria virtuale come le cache, ossia tramite un comparatore sui bit più significativi per verificare la corrispondenza delle pagine, servirebbe un numero improponibile di comparatori, dato l'elevato numero di pagine in cui è organizzata la RAM.

In alternativa, il processo di traduzione degli indirizzi è gestito mediante **tabelle di paginazione**.

# La page table

- Contiene un entry per ogni pagina virtuale
- Campi di una entry:
  - **Valid bit:** 1 se la pagina è presente nella main memory
  - **Physical page number:** dove la pagina è locata nella main memory



# Esempio 1

Qual'è indirizzo fisico  
dell'indirizzo virtuale **0x5F20**?

- VPN = **5**
- Entry 5 in page table VPN 5 =>  
physical page **1**
- Physical address: **0x1F20**

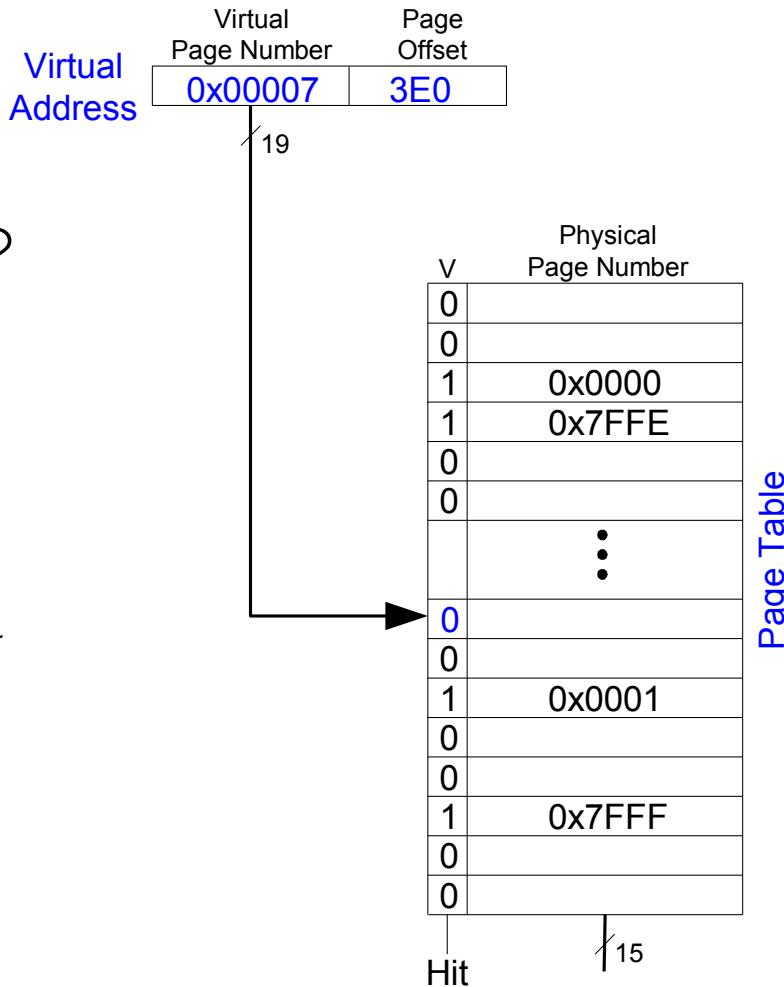
| V | Physical Page Number |
|---|----------------------|
| 0 |                      |
| 0 |                      |
| 1 | 0x0000               |
| 1 | 0x7FFE               |
| 0 |                      |
| 0 |                      |
| ⋮ |                      |
| 0 |                      |
| 0 |                      |
| 1 | 0x0001               |
| 0 |                      |
| 0 |                      |
| 1 | 0x7FFF               |
| 0 |                      |
| 0 |                      |

Page Table

# Esempio 2

Quale è l'indirizzo fisico  
dell'indirizzo virtuale **0x73E0**?

- VPN = **7**
- L'entry 7 non è valida
- La pagina virtuale deve essere caricata nella main memory dalla memoria di massa



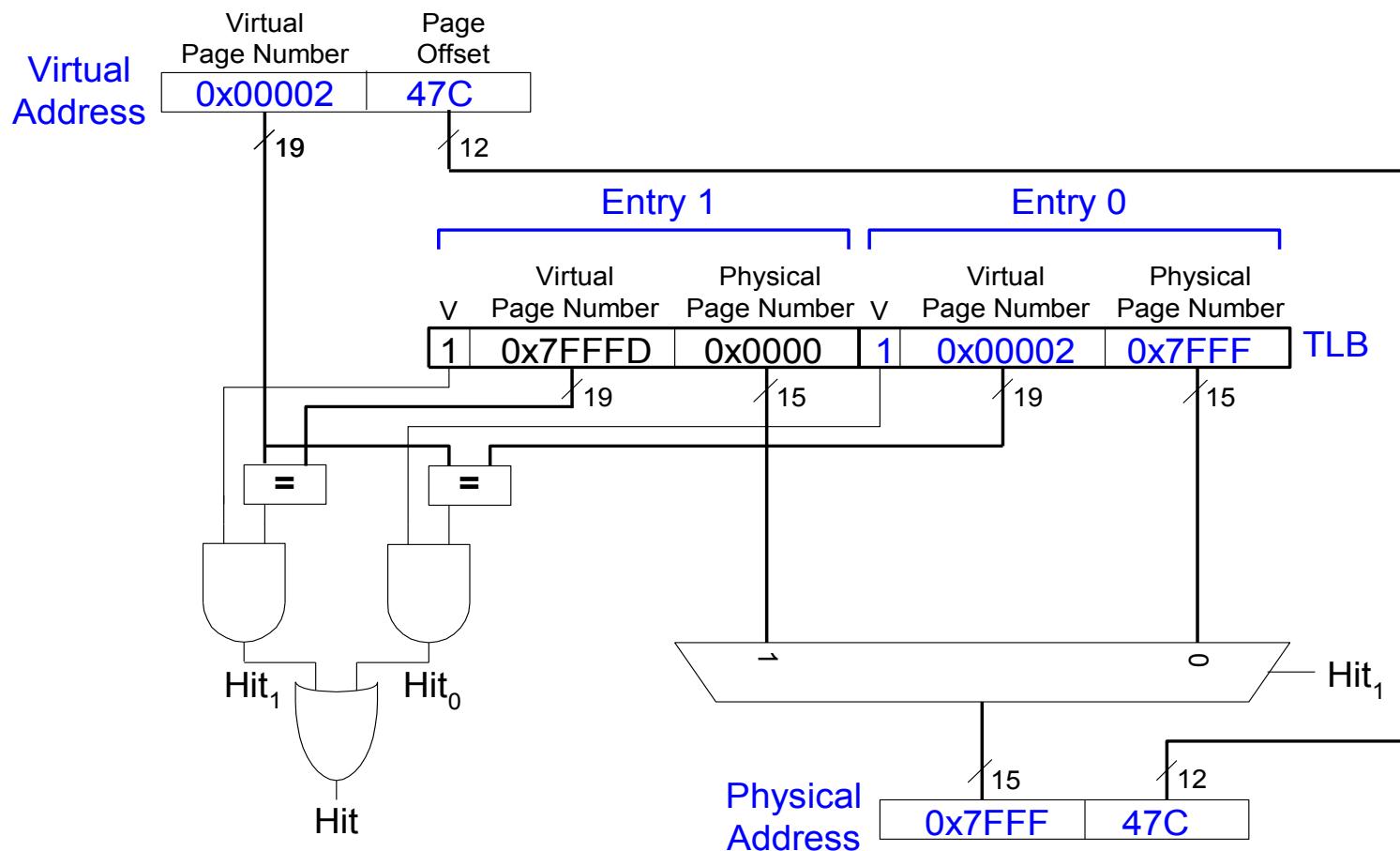
# Caratteristiche della Page Table

- La page table è solitamente voluminosa
  - Di solito è essa stessa allocata nella main memory
- Le operazioni di load/store richiedono 2 accessi alla main memory:
  - Il primo per la traduzione (lettura della page table)
  - Il secondo è l'accesso vero e proprio al dato dopo la traduzione
- Questo dimezzerebbe le performance della main memory...

# Translation Lookaside Buffer (TLB)

- Per migliorare le performance e far sì che la maggior parte delle operazioni di load e store avvenga con un solo accesso in memoria si usa una piccola cache dei traduzioni più recenti (TLB)
- TLB sfrutta il fatto che gli accessi alla page table godono di una elevate località temporale:
  - Essendo le dimensioni di una pagina relativamente grandi vi è un'alta probabilità che operazioni di load o store consecutive richiedano un accesso alla medesima pagina
- TLB
  - Veloce: accesso in circa 1 ciclo
  - Piccola: tipicamente 16 - 512 entry
  - Fully associative
  - Efficace: > 99 % hit rates

# Esempio di 2-Entry TLB



# Esercizi sulle cache

Domanda: Si ipotizzi che il 30% delle istruzioni in un programma tipico effettui un'operazione di scrittura o di lettura in RAM, e che la frequenza di successo di lettura in cache sia del 95% per le istruzioni e del 90% per i dati. Si supponga, inoltre, che la penalità di fallimento sia la stessa per operazioni di scrittura e operazioni di lettura. Si supponga che, in cicli di clock, una lettura diretta in memoria costi 10, una in cache costi 1 e una sia in cache che in memoria costi 16. Quale sarebbe il guadagno utilizzando la cache?

# Esercizi sulle cache

Domanda: Si ipotizzi che il 30% delle istruzioni in un programma tipico effettui un'operazione di scrittura o di lettura in RAM, e che la frequenza di successo di lettura in cache sia del 95% per le istruzioni e del 90% per i dati. Si supponga, inoltre, che la penalità di fallimento sia la stessa per operazioni di scrittura e operazioni di lettura. Si supponga che, in cicli di clock, una lettura diretta in memoria costi 10, una in cache costi 1 e una sia in cache che in memoria costi 16. Quale sarebbe il guadagno utilizzando la cache?

Date n istruzioni, costo senza cache è:  $(1 + 0,3) n \cdot 10$

Costo con cache:  $n (0,95 * 1 + 0,05 * 16) + 0,3 n (0,9 * 1 + 0,1 * 16)$

$$G = \frac{(1,3 * 10)n}{[(0,95+16 * 0,05)+(0,9+0,1*16)]n} = \frac{13}{2,5} = 5,2$$

# Esercizi sulle cache

Domanda: Si ipotizzi che **4/5** delle istruzioni in un programma effettui un'operazione di scrittura o di lettura in RAM, e che la frequenza di successo di lettura in cache per le istruzioni sia **2/3** di quella per i dati. Si supponga, inoltre, che la penalità di fallimento sia la stessa per operazioni di scrittura e operazioni di lettura. Si supponga che, in cicli di clock, una lettura diretta in memoria costi **10**, una in cache costi **1** e che una in entrambe (sia in memoria, che in cache) costi **16**. Quale deve essere il tasso di successo dei dati per avere un guadagno maggiore di **1**?

Date n istruzioni, costo senza cache è:  $(1 + 4/5) * n * 10 = 18 n$

Costo con cache:  $n (2/3 * x * 1 + (1 - 2/3 * x) 16) + 4/5 n (x * 1 + (1-x) * 16)$

$$G = \frac{18}{[16 - 10x + 0,8(16 - 15x)]} = \frac{18}{28.8 - 22x} > 1$$

$$\begin{aligned} 22x &> 10.8 \\ x &> 0.49 \end{aligned}$$

# Esercizi cache

Domanda 2: Si consideri una memoria cache set-associativa da **2048** parole, suddivisa in **128** blocchi da **16** locazioni ciascuna, in cui si sceglie di avere **4** blocchi per insieme, ossia **32** insiemi. Sia data l'indirizzo **0x000244F4**, in quale insieme della cache essa sarà inserita e con quale etichetta? Analogamente, quali sono insieme ed etichetta se i blocchi per insieme fossero **2**?

**0x000244F4 = 0000 0000 0000 0010 0100 0100 1111 0100**

**0000 0000 0000 0010 0100 0100 1111 0100**

**11 01** → **Parola nel set = 13**

**10011** → **Set = 19**

**0000 0000 0000 0010 0100 0** → **Etichetta = 72**

# Esercizi cache

Domanda : Si consideri una memoria cache set-associativa da **2048** parole, suddivisa in **64** blocchi da **32** locazioni ciascuna, in cui si sceglie di avere **2** blocchi per insieme, ossia **32** insiemi. Sia data la parola **0x0000D33C** , in quale insieme della cache essa sarà inserita e con quale etichetta?

**0x0000D33F = 0000 0000 0000 0000 1101 0011 0011 1100**

**011 11** → Parola nel set = 15

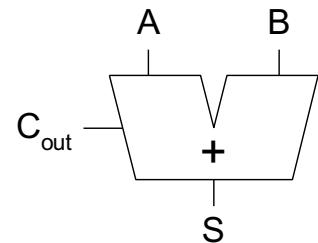
**0011 0** → Set = 6

**0000 0000 0000 0000 1101** → Etichetta = 13

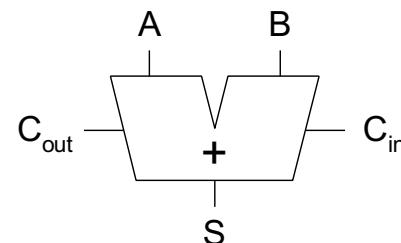
# Circuiti aritmetici e memorie

# 1-Bit Adders

Half  
Adder



Full  
Adder



| A | B | C <sub>out</sub> | S |
|---|---|------------------|---|
| 0 | 0 | 0                | 0 |
| 0 | 1 | 0                | 1 |
| 1 | 0 | 0                | 1 |
| 1 | 1 | 1                | 0 |

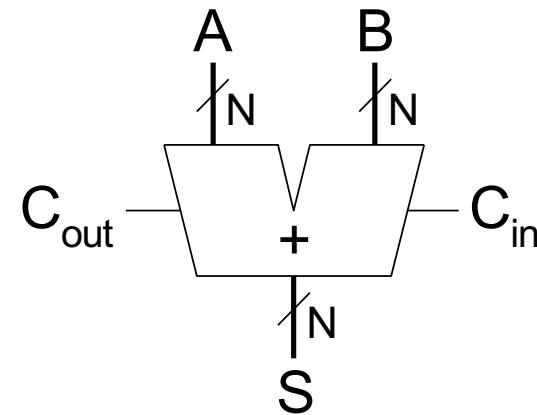
$$\begin{aligned}S &= A \oplus B \\C_{\text{out}} &= AB\end{aligned}$$

| C <sub>in</sub> | A | B | C <sub>out</sub> | S |
|-----------------|---|---|------------------|---|
| 0               | 0 | 0 | 0                | 0 |
| 0               | 0 | 1 | 0                | 1 |
| 0               | 1 | 0 | 0                | 1 |
| 0               | 1 | 1 | 1                | 0 |
| 1               | 0 | 0 | 0                | 1 |
| 1               | 0 | 1 | 1                | 0 |
| 1               | 1 | 0 | 1                | 0 |
| 1               | 1 | 1 | 1                | 1 |

$$\begin{aligned}S &= A \oplus B \oplus C_{\text{in}} \\C_{\text{out}} &= AB + AC_{\text{in}} + BC_{\text{in}}\end{aligned}$$

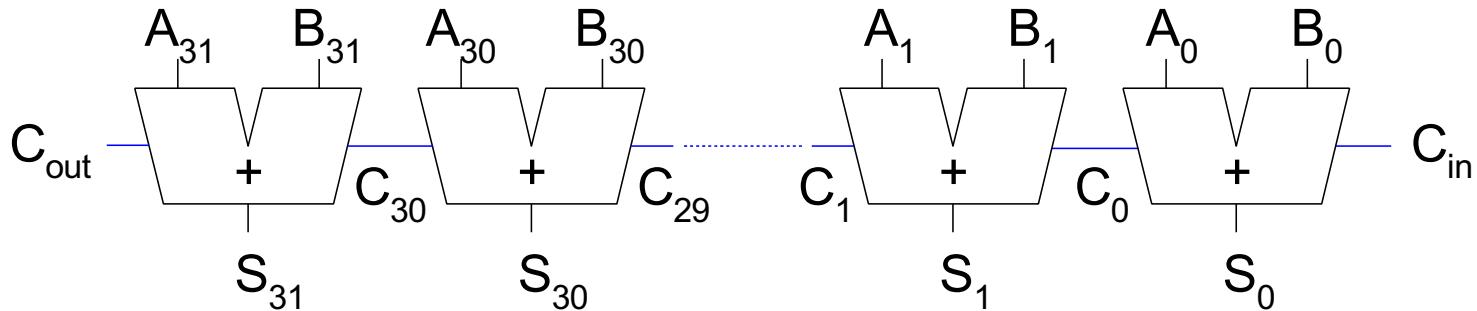
# Multibit Adders (CPAs)

- Tipologie carry propagate adders (CPAs):
  - Ripple-carry (lento, ma più semplice da realizzare)
  - Carry-lookahead (veloce, ma richiede più hardware)



# Ripple-Carry Adder

- 1-bit adder concatenate insieme
- Il riporto si propaga lungo la catena
- Svantaggio: **lento**



# Ritardo del Ripple-Carry

$$t_{\text{ripple}} = N t_{FA}$$

$t_{FA}$  è il ritardo di un singolo 1-bit full adder

N è il numero di 1-bit full adders

# Carry-Lookahead Adder

Calcolare  $C_{\text{out}}$  di un blocco di  $k$ -bit usando le funzioni *generate* e *propagate*

| $C_{\text{in}}$ | $A_i$ | $B_i$ | $C_{\text{out}}$ |
|-----------------|-------|-------|------------------|
| 0               | 0     | 0     | 0                |
| 1               |       |       | 0                |
| 0               | 0     | 1     | 0                |
| 1               |       |       | 1                |
| 0               | 1     | 0     | 0                |
| 1               |       |       | 1                |
| 0               | 1     | 1     | 1                |
| 1               |       |       | 1                |

# Carry-Lookahead Adder

| $C_{in}$ | $A_i$ | $B_i$ | $C_{out}$ |
|----------|-------|-------|-----------|
| 0        | 0     | 0     | 0         |
| 1        |       |       | 0         |
| 0        | 0     | 1     | 0         |
| 1        |       |       | 1         |
| 0        | 1     | 0     | 0         |
| 1        |       |       | 1         |
| 0        | 1     | 1     | 1         |
| 1        |       |       | 1         |

# Carry-Lookahead Adder

- Il bit *i-esimo* produce un riporto (carry out) per generazione o per propagazione di un riporto del bit *(i-1)-esimo* (carry in)
- **Generate:** Il bit *i-esimo* genera un carry out se  $A_i$  e  $B_i$  sono entrambi 1.

$$G_i = A_i B_i$$

- **Propagate:** Il bit *i-esimo* propaga un carry al carry out se  $A_i$  o  $B_i$  sono 1.

$$P_i = A_i + B_i$$

- **Carry out:** il carry *i* ( $C_i$ ) è dato da:

$$C_i = A_i B_i + (A_i + B_i) C_{i-1} = G_i + P_i C_{i-1}$$

# Block Propagate and Generate

Ora dobbiamo estendere le funzioni Propagate and Generate a blocchi di  $k$ -bits, ovvero:

- calcolare se un blocco di  $k$ -bit  $(i, i+1, \dots, i+k-1)$  propaga il carry out del blocco precedente (ovvero da  $C_{i-1}$  a  $C_{i+k-1}$ )
- calcolare se un blocco di  $k$ -bit  $(i, i+1, \dots, i+k-1)$  genera un carry out (in  $C_{i+k-1}$ )

# Esempi

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |

blocco di 4 propagate = block propagate

# Esempi

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |   |

2 propagate, 1 annihilator, 1 propagate= block annihilator

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |   |

.

1 propagate, 1 generator, 2 propagate= block generator

# Esempi

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |

1 propagate, 1 generator, 1 propagate, 1 annihilator = block annihilator

# Block Propagate and Generate Signals

- Block propagate and generate per un blocco di 4-bit ( $P_{3:0}$  and  $G_{3:0}$ ):

$$P_{3:0} = P_3 P_2 P_1 P_0$$

$$\begin{aligned} G_{3:0} &= G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 = \\ &G_3 + P_3 (G_2 + P_2 (G_1 + P_1 G_0)) \end{aligned}$$

- In generale,

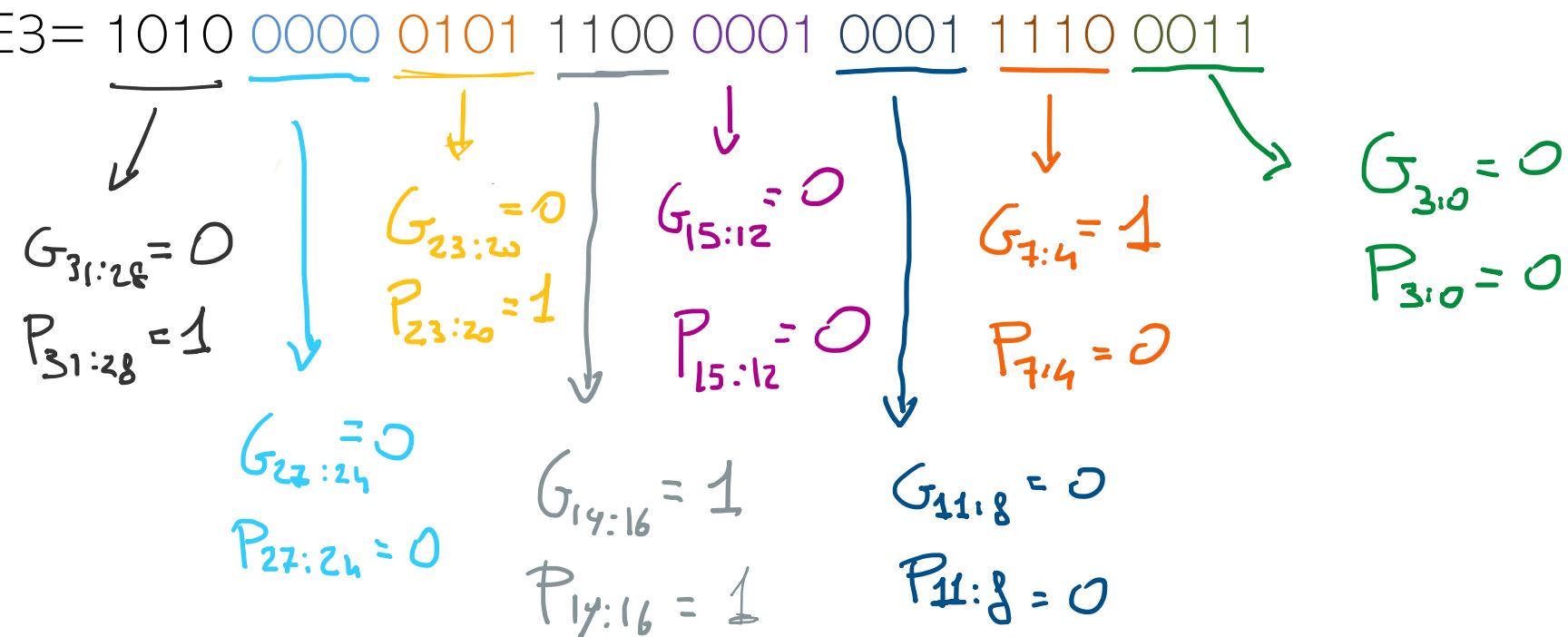
$$P_{i:j} = P_i P_{i-1} P_{i-2} \dots P_j$$

$$G_{i:j} = G_i + P_i (G_{i-1} + P_{i-1} (G_{i-2} + P_{i-2} (\dots G_j) \dots))$$

$$C_i = G_{i:j} + P_{i:j} C_{j-1}$$

# Esempio

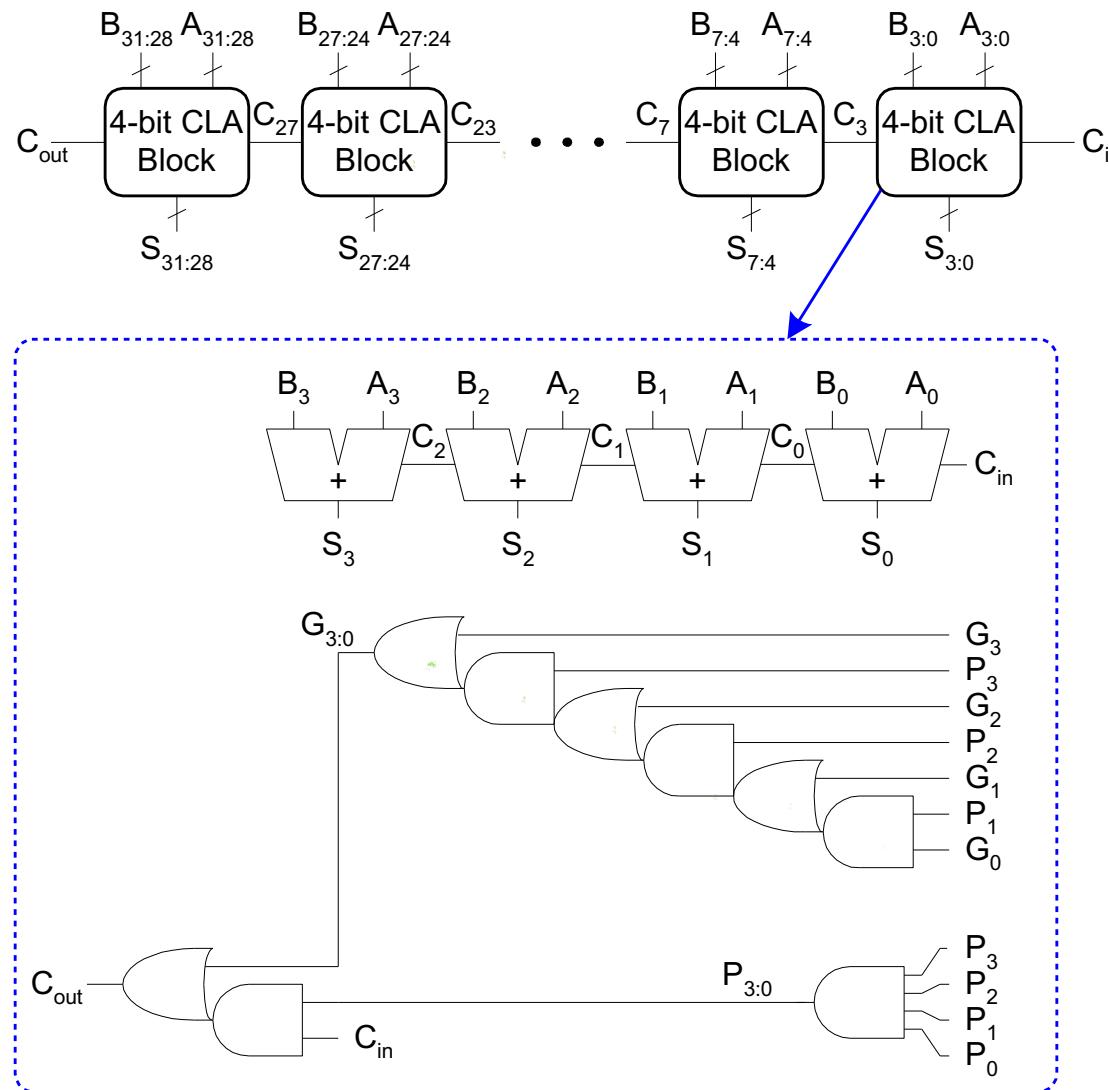
- Consideriamo le due parole a 32 bit
  - 0x52A703C1
  - 0xA05C11E3
- $0x52A703C1 = 0101\ 0010\ 1010\ 0111\ 0000\ 0011\ 1100\ 0001$
- $0xA05C11E3 = 1010\ \underline{0000}\ \underline{0101}\ \underline{1100}\ \underline{0001}\ \underline{0001}\ \underline{1110}\ \underline{0011}$



# Carry-Lookahead Addition

- Step 1: calcola tutti i  $G_i$  and  $P_i$
- Step 2: calcola  $G$  and  $P$  per blocchi di  $k$ -bit
- Step 3:  $C_{in}$  si propaga mediante la logica propagate/generate dei vari blocchi di  $k$ -bit

# 32-bit CLA with 4-bit Blocks



# Ritardo del Carry-Lookahead Adder

Per un  $N$ -bit CLA con blocchi di  $k$  bit:

$$t_{CLA} = t_{pg} + t_{pg\_block} + (N/k - 1)t_{AND\_OR} + kt_{FA}$$

- $t_{pg}$  : ritardo per generare  $P_i, G_i$
- $t_{pg\_block}$  : ritardo per generare  $P_{i:j}, G_{i:j}$
- $t_{AND\_OR}$  : ritardo delle porte AND/OR a monte della logica propagate/generate, questo ritardo si propaga da  $C_{in}$  a  $C_{out}$  che si propaga lungo i  $N/k - 1$  blocchi

Un  $N$ -bit carry-lookahead adder è generalmente più veloce di un ripple-carry adder per  $N > 16$

# Confronto

- 32-bit ripple-carry vs 32-bit CLA con blocchi di 4 bit
- 2-input gate delay = 100 ps; full adder delay = 300 ps

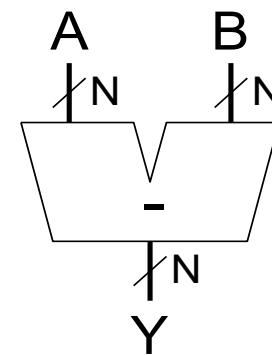
$$\begin{aligned} t_{\text{ripple}} &= Nt_{FA} = 32(300 \text{ ps}) \\ &= \mathbf{9.6 \text{ ns}} \end{aligned}$$

$$\begin{aligned} t_{\text{CLA}} &= t_{pg} + t_{pg\_block} + (N/k - 1)t_{\text{AND\_OR}} + kt_{FA} \\ &= [100 + 600 + (7)200 + 4(300)] \text{ ps} \\ &= \mathbf{3.3 \text{ ns}} \end{aligned}$$

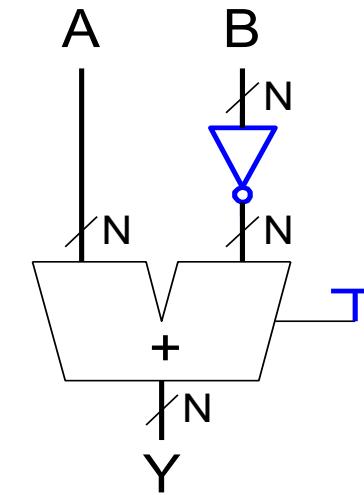
# Sottrattore

- Nella rappresentazione a complemento a 2 per complementare un numero occorre invertire ogni cifra e sommare 1
  - Es:  $2 = 0010_2 \rightarrow -2 = 1101_2 + 0001_2 = 1110_2$

## Symbol

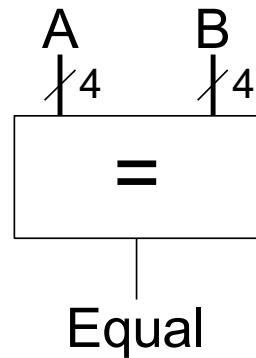


## Implementation

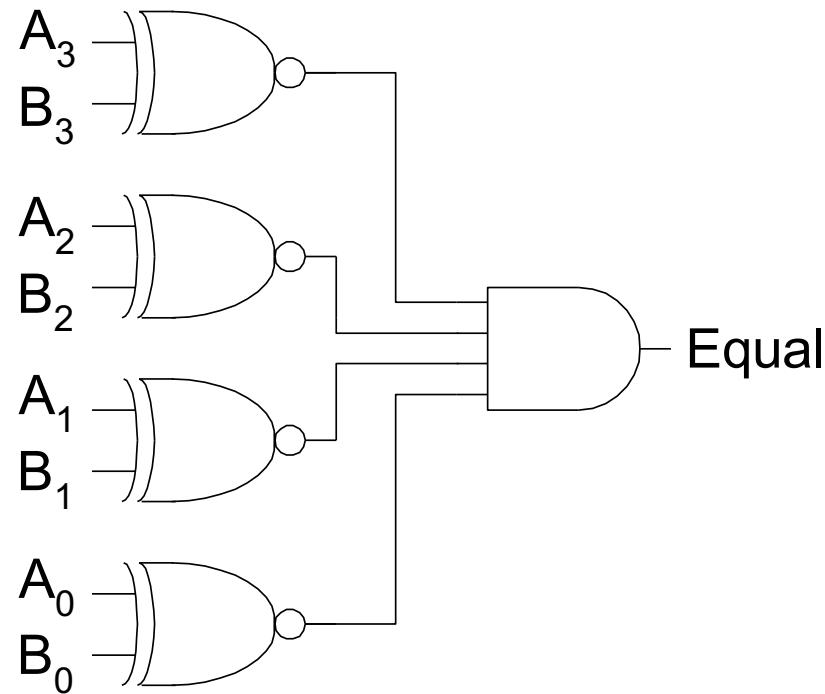


# Comparatore: uguaglianza

## Symbol

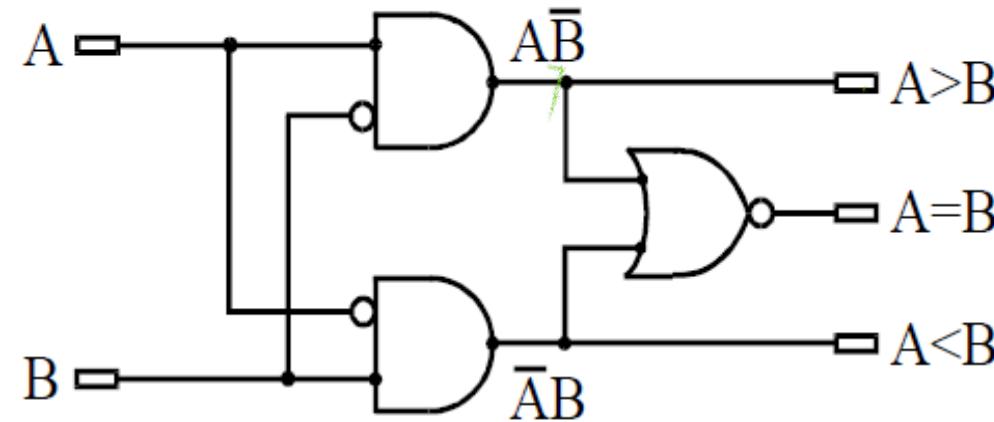


## Implementation

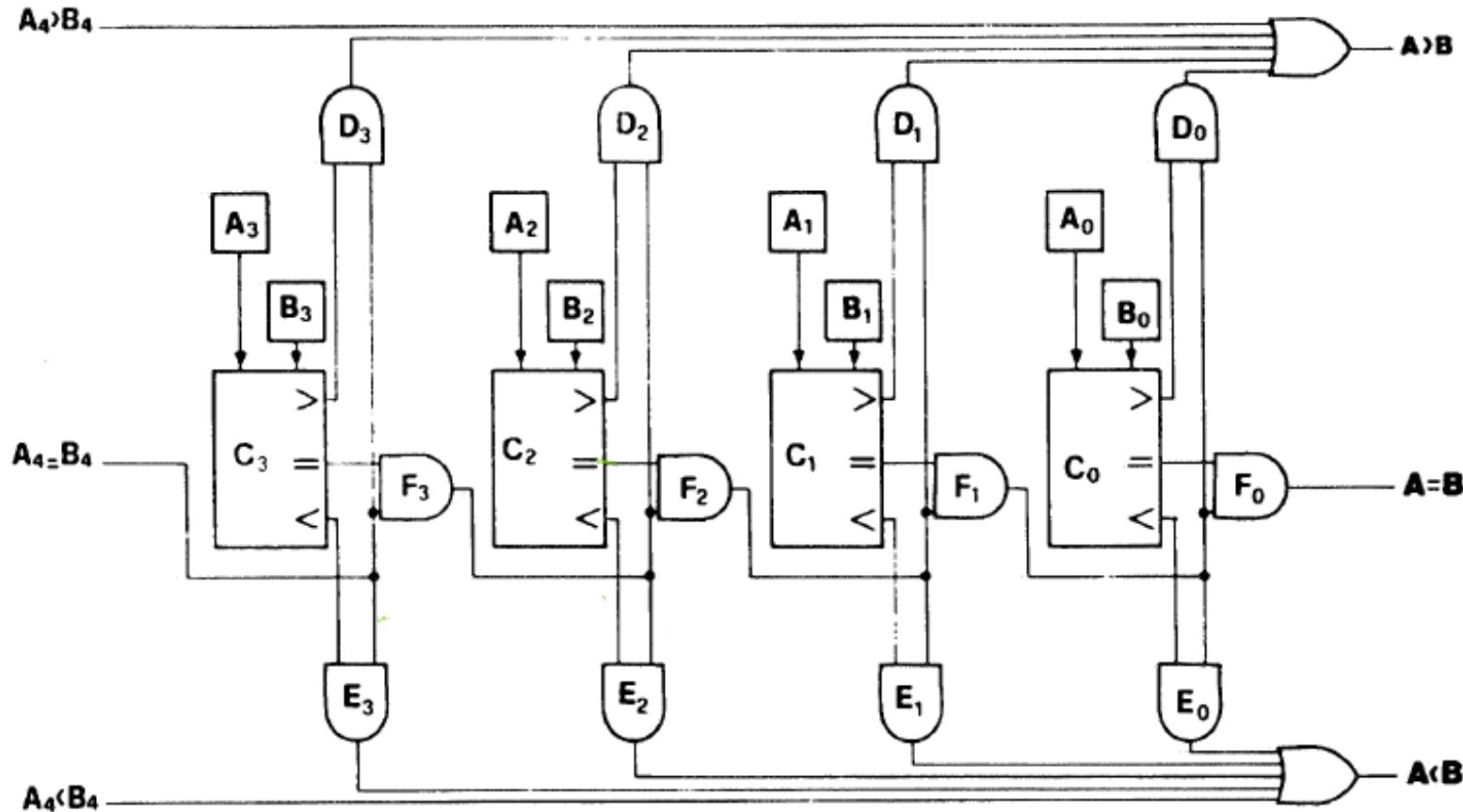


# Comparatore completo a un bit

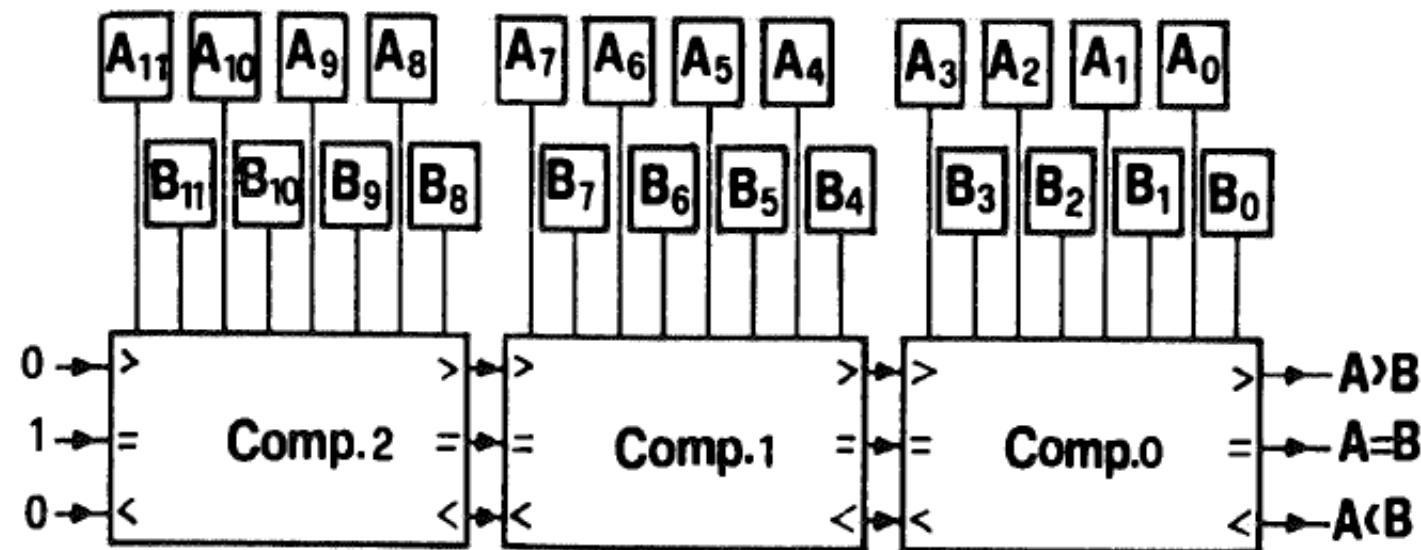
| A | B | A>B | A=B | A<B |
|---|---|-----|-----|-----|
| 0 | 0 | 0   | 1   | 0   |
| 0 | 1 | 0   | 0   | 1   |
| 1 | 0 | 1   | 0   | 0   |
| 1 | 1 | 0   | 1   | 0   |



# Comparatore completo a 4 bit



# Comparatore completo a 12 bit



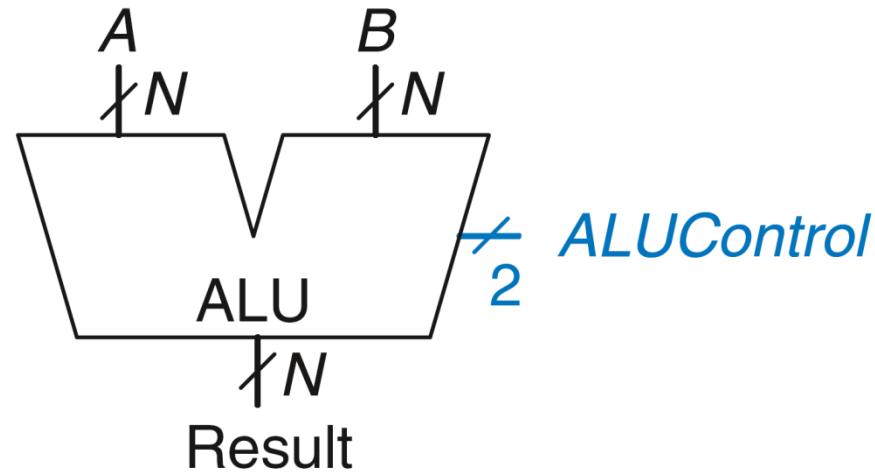
# ALU: Arithmetic Logic Unit

Operazioni che ALU tipicamente esegue:

- Addizione
- Sottrazione
- AND (bit a bit)
- OR (bit a bit)

# ALU: Arithmetic Logic Unit

| ALUControl <sub>1:0</sub> | Function |
|---------------------------|----------|
| 00                        | Add      |
| 01                        | Subtract |
| 10                        | AND      |
| 11                        | OR       |

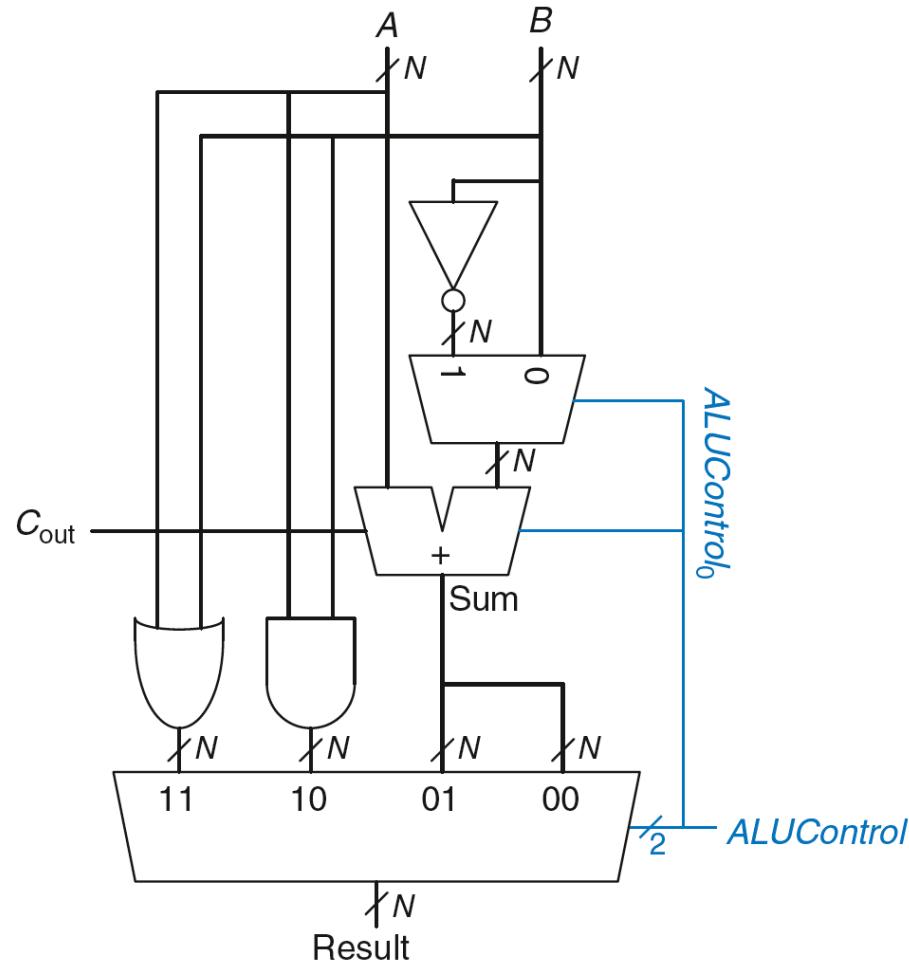


# ALU: Arithmetic Logic Unit

| ALUControl <sub>1:0</sub> | Function |
|---------------------------|----------|
| 00                        | Add      |
| 01                        | Subtract |
| 10                        | AND      |
| 11                        | OR       |

**Esempio: A OR B**

$ALUControl_{1:0} = 11$



# ALU: Arithmetic Logic Unit

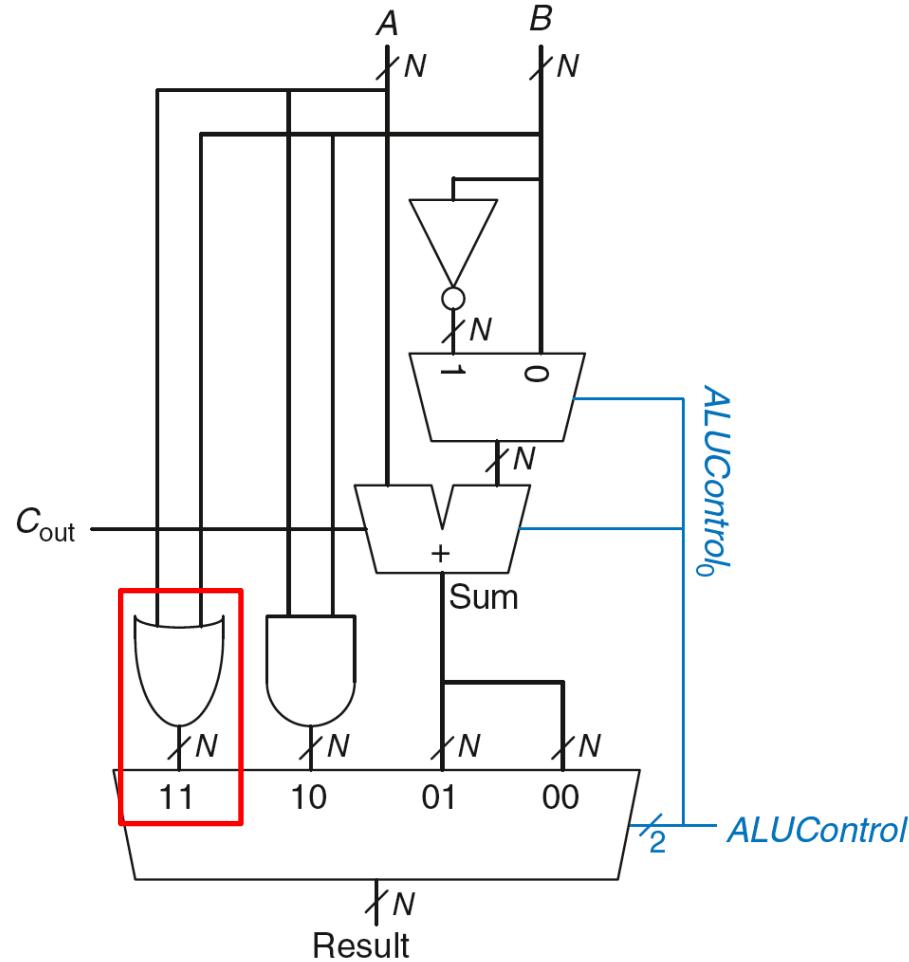
| ALUControl <sub>1:0</sub> | Function |
|---------------------------|----------|
| 00                        | Add      |
| 01                        | Subtract |
| 10                        | AND      |
| 11                        | OR       |

**Esempio: A OR B**

$ALUControl_{1:0} = 11$

Mux seleziona l'output della porta OR

**Result = A OR B**



# ALU: Arithmetic Logic Unit

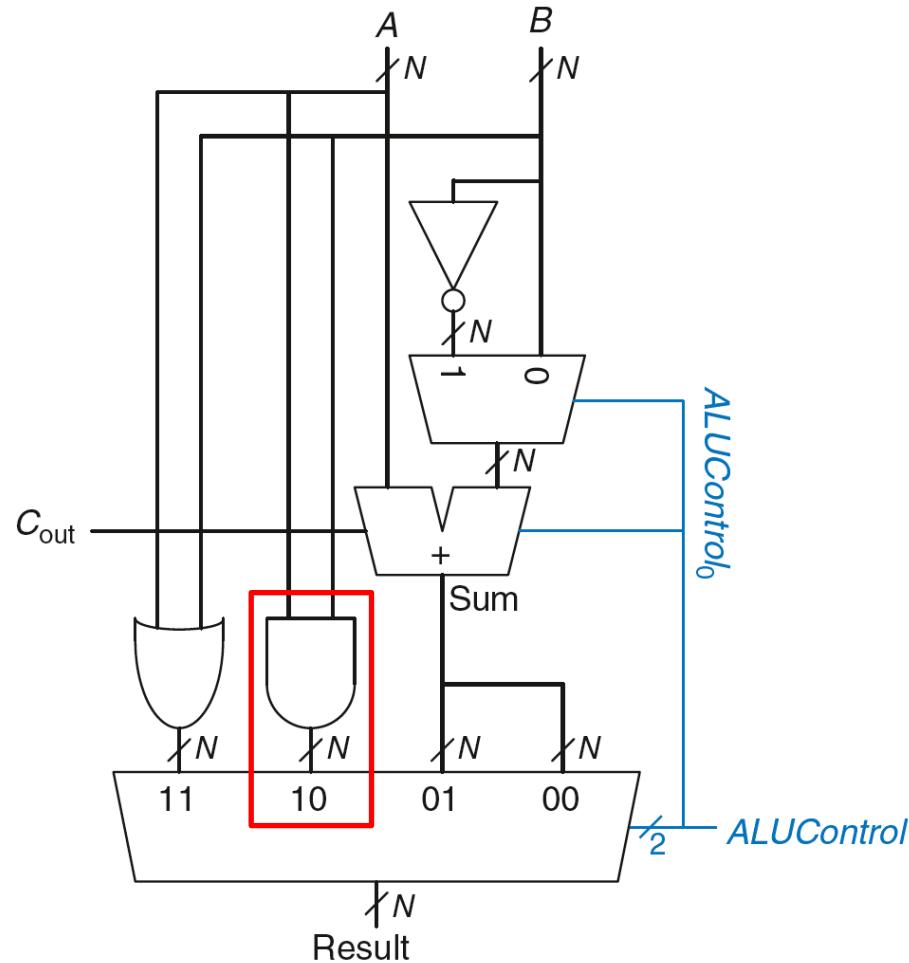
| ALUControl <sub>1:0</sub> | Function |
|---------------------------|----------|
| 00                        | Add      |
| 01                        | Subtract |
| 10                        | AND      |
| 11                        | OR       |

**Esempio: A AND B**

$ALUControl_{1:0} = 10$

Mux seleziona l'output della porta AND

**Result = A AND B**

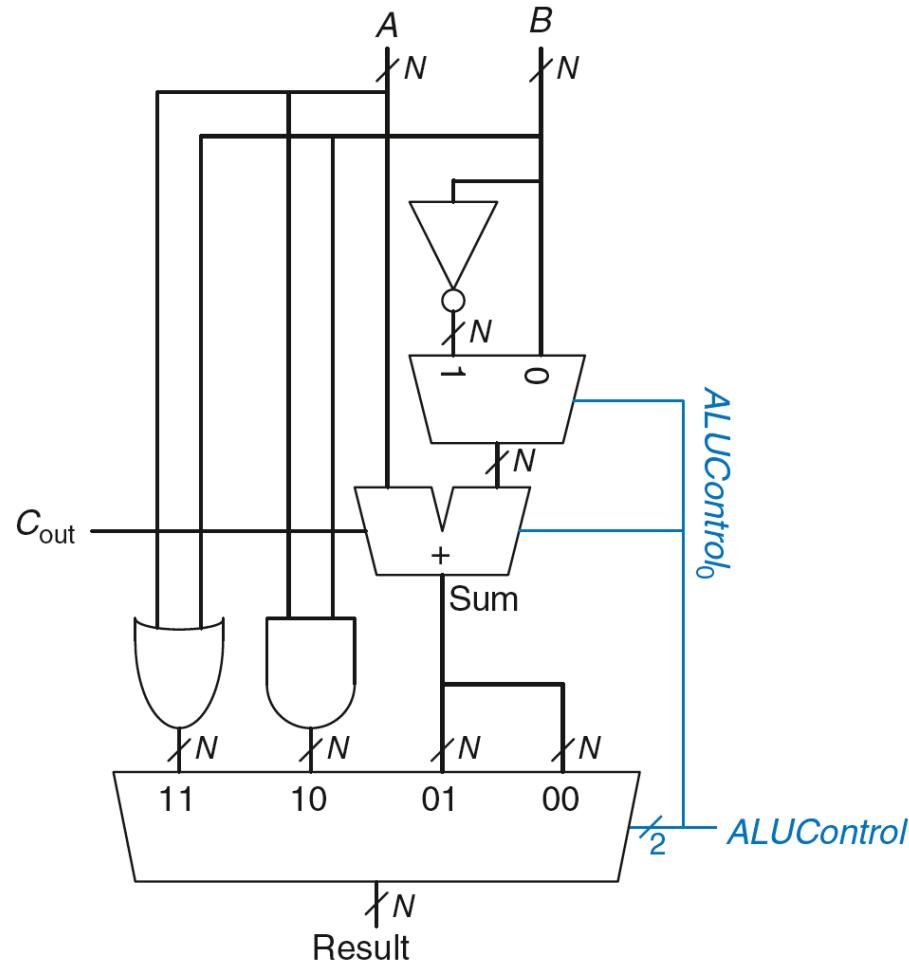


# ALU: Arithmetic Logic Unit

| ALUControl <sub>1:0</sub> | Function |
|---------------------------|----------|
| 00                        | Add      |
| 01                        | Subtract |
| 10                        | AND      |
| 11                        | OR       |

## Esempio: $A + B$

$$ALUControl_{1:0} = 00$$



# ALU: Arithmetic Logic Unit

| ALUControl <sub>1:0</sub> | Function |
|---------------------------|----------|
| 00                        | Add      |
| 01                        | Subtract |
| 10                        | AND      |
| 11                        | OR       |

**Esempio:  $A + B$**

$ALUControl_{1:0} = 00$

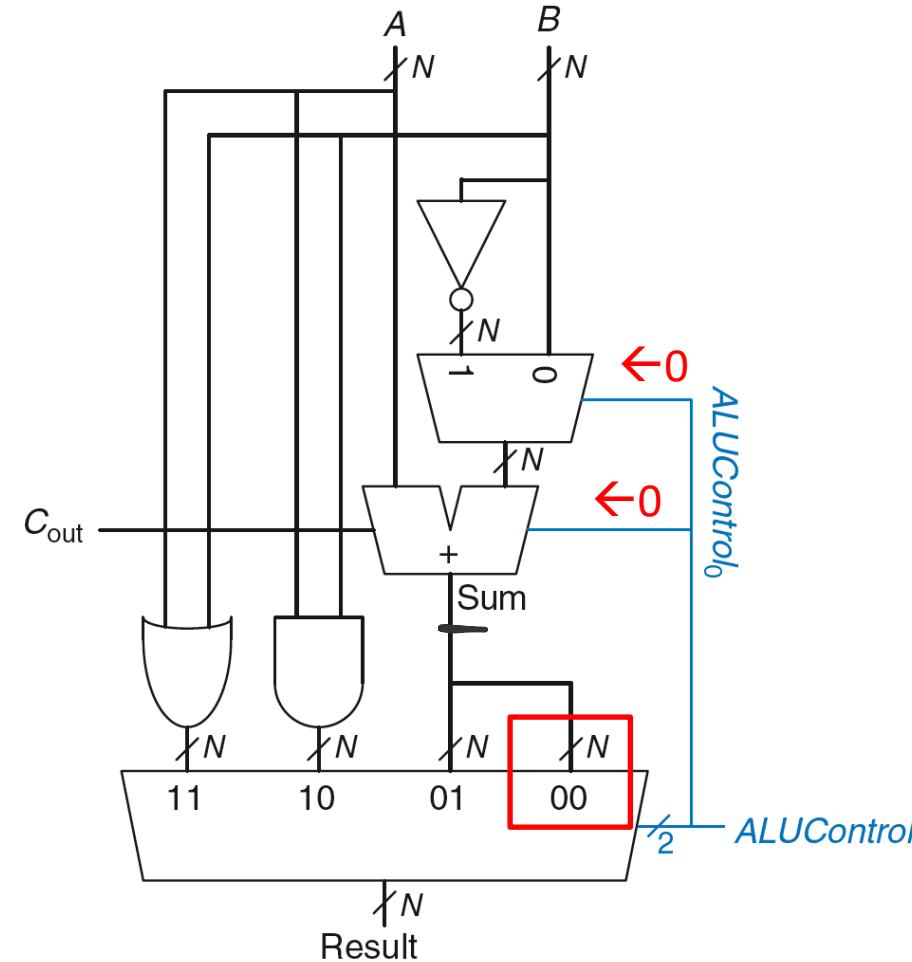
$ALUControl_0 = 0$ , quindi:

Cin = 0

il 2<sup>nd</sup> input dell'adder è  $B$

Mux seleziona *Sum* come *Result*

**Result =  $A + B$**



# ALU: Arithmetic Logic Unit

| ALUControl <sub>1:0</sub> | Function |
|---------------------------|----------|
| 00                        | Add      |
| 01                        | Subtract |
| 10                        | AND      |
| 11                        | OR       |

**Esempio:  $A - B$**

$ALUControl_{1:0} = 01$

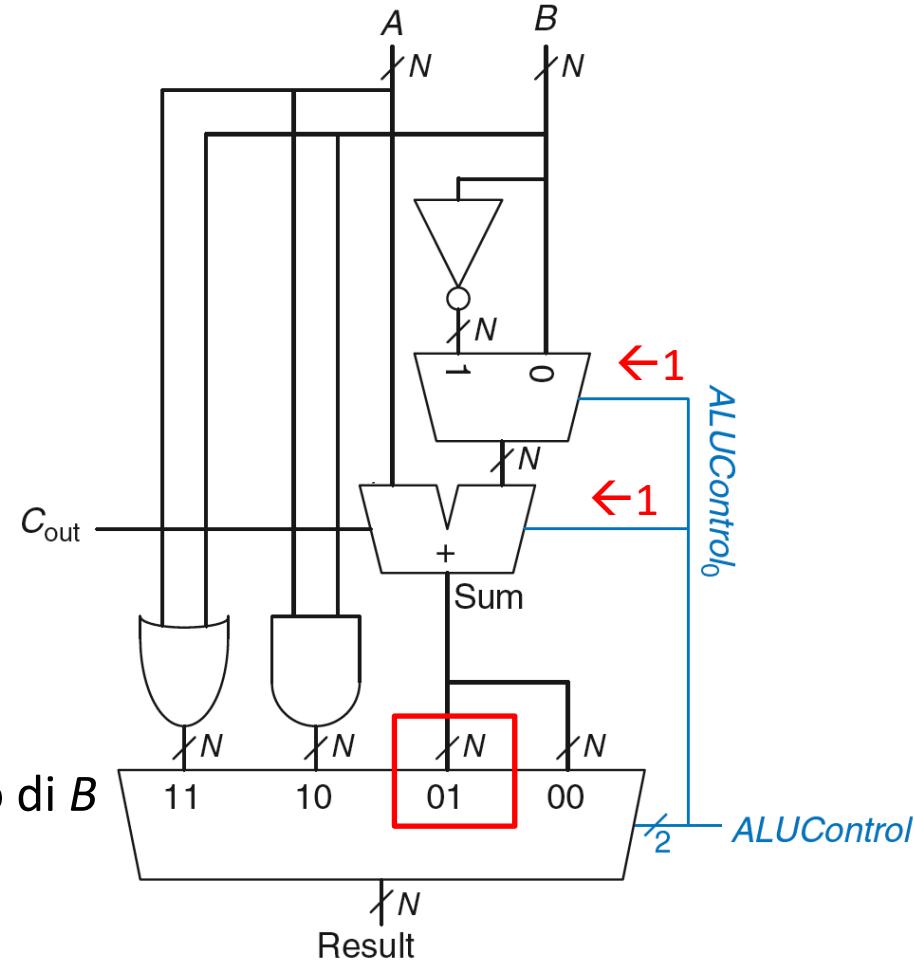
$ALUControl_0 = 1$ , quindi:

Cin = 1

2<sup>nd</sup> input dell'adder è il complemento di B

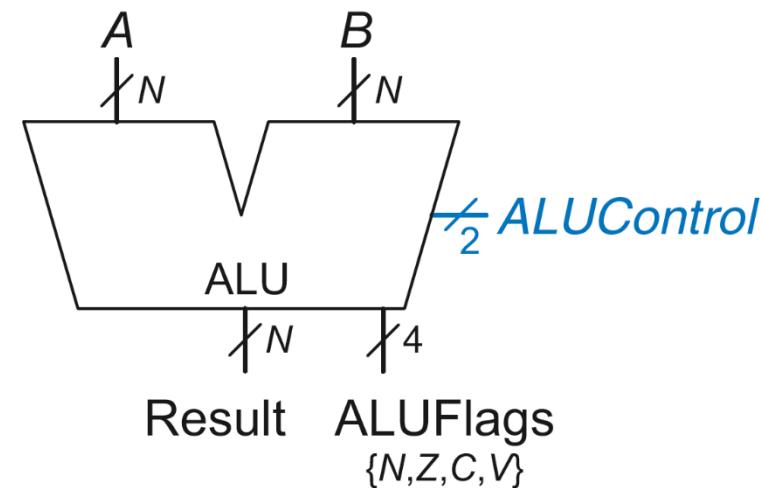
Mux seleziona Sum come Result

**Result =  $A - B$**

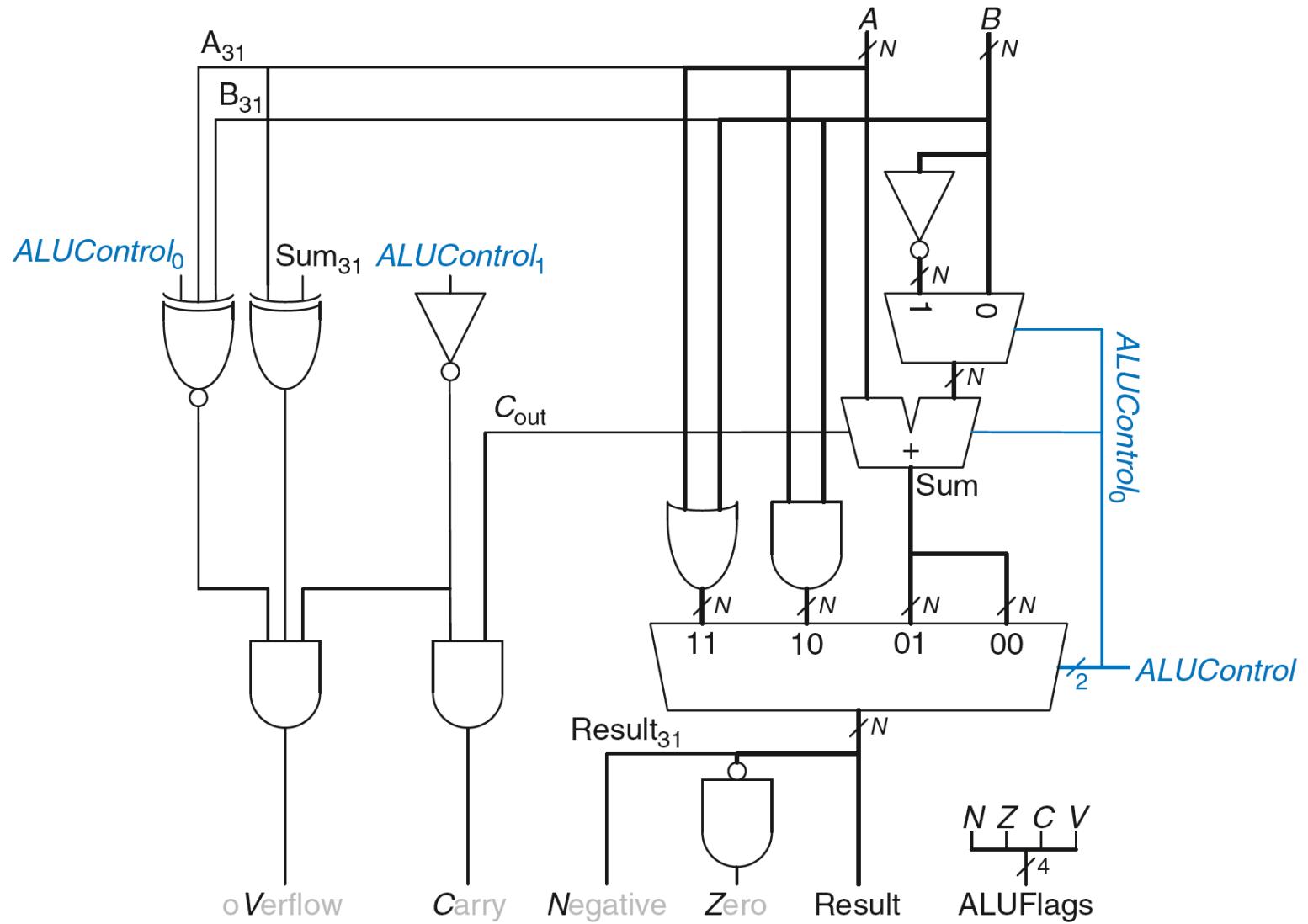


# ALU con flags di stato

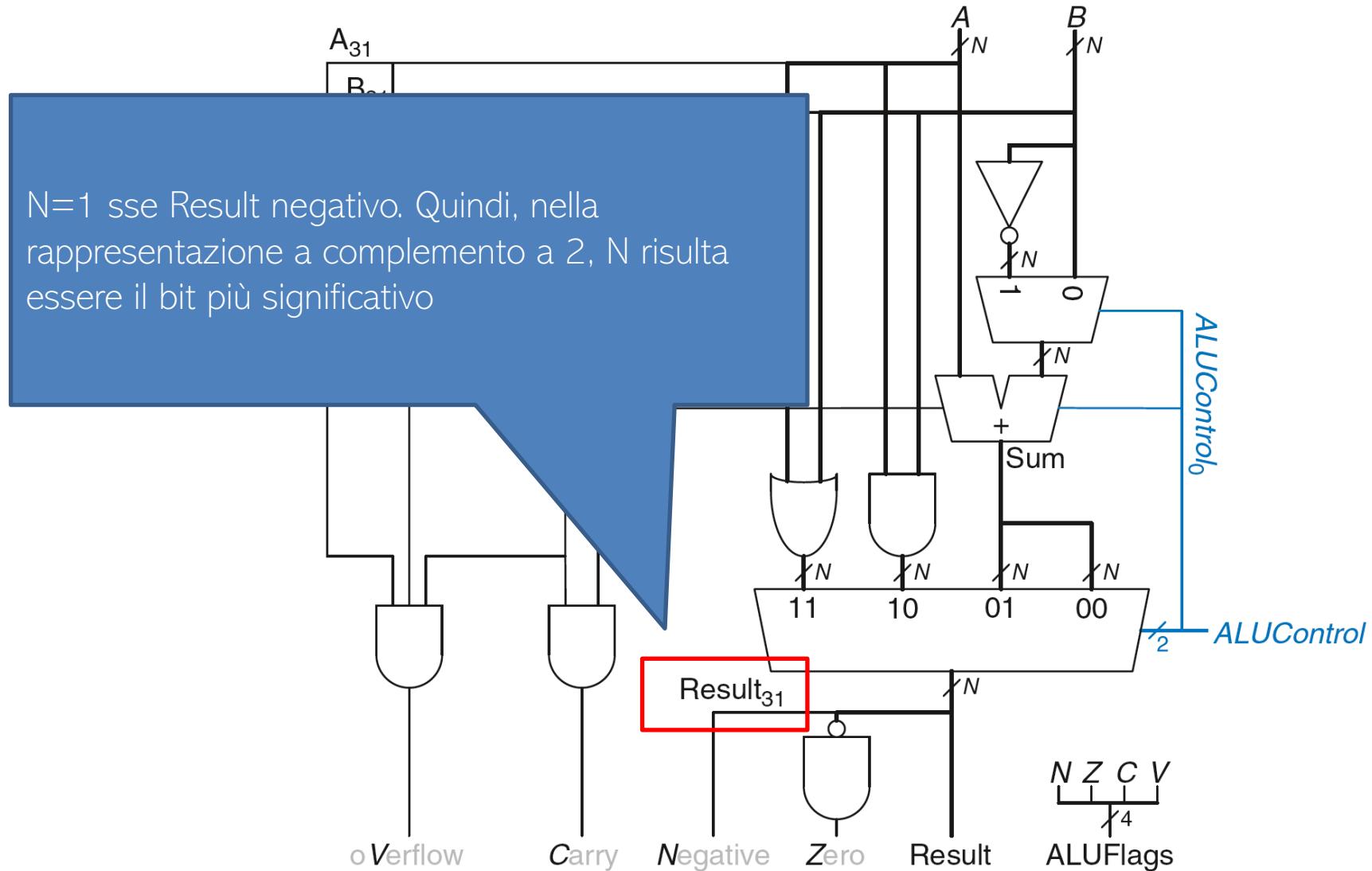
| Flag | Description                      |
|------|----------------------------------|
| $N$  | <i>Result</i> is <b>Negative</b> |
| $Z$  | <i>Result</i> is <b>Zero</b>     |
| $C$  | Adder produces <b>Carry out</b>  |
| $V$  | Adder <b>oVerflowed</b>          |



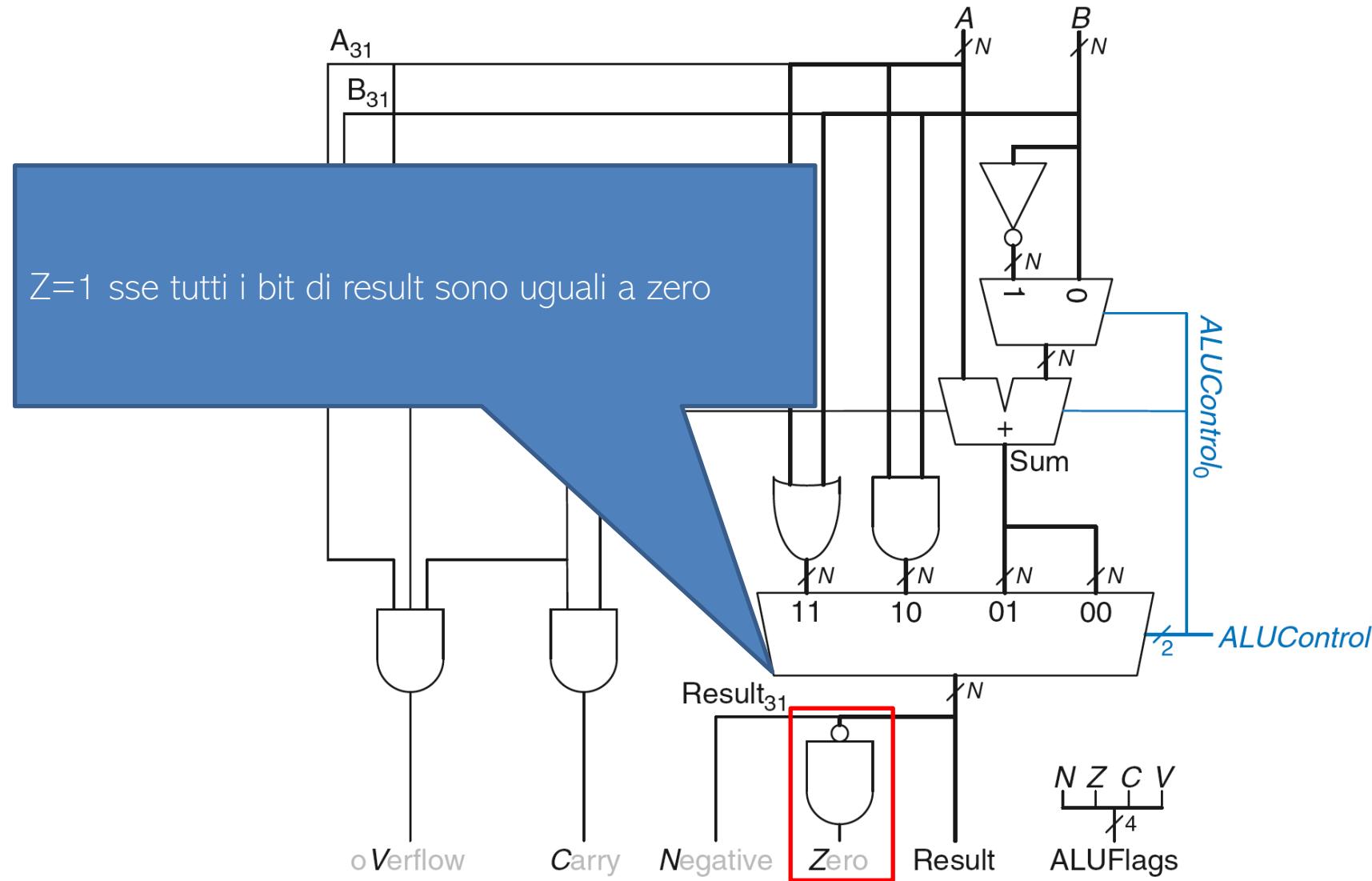
# ALU con flags di stato



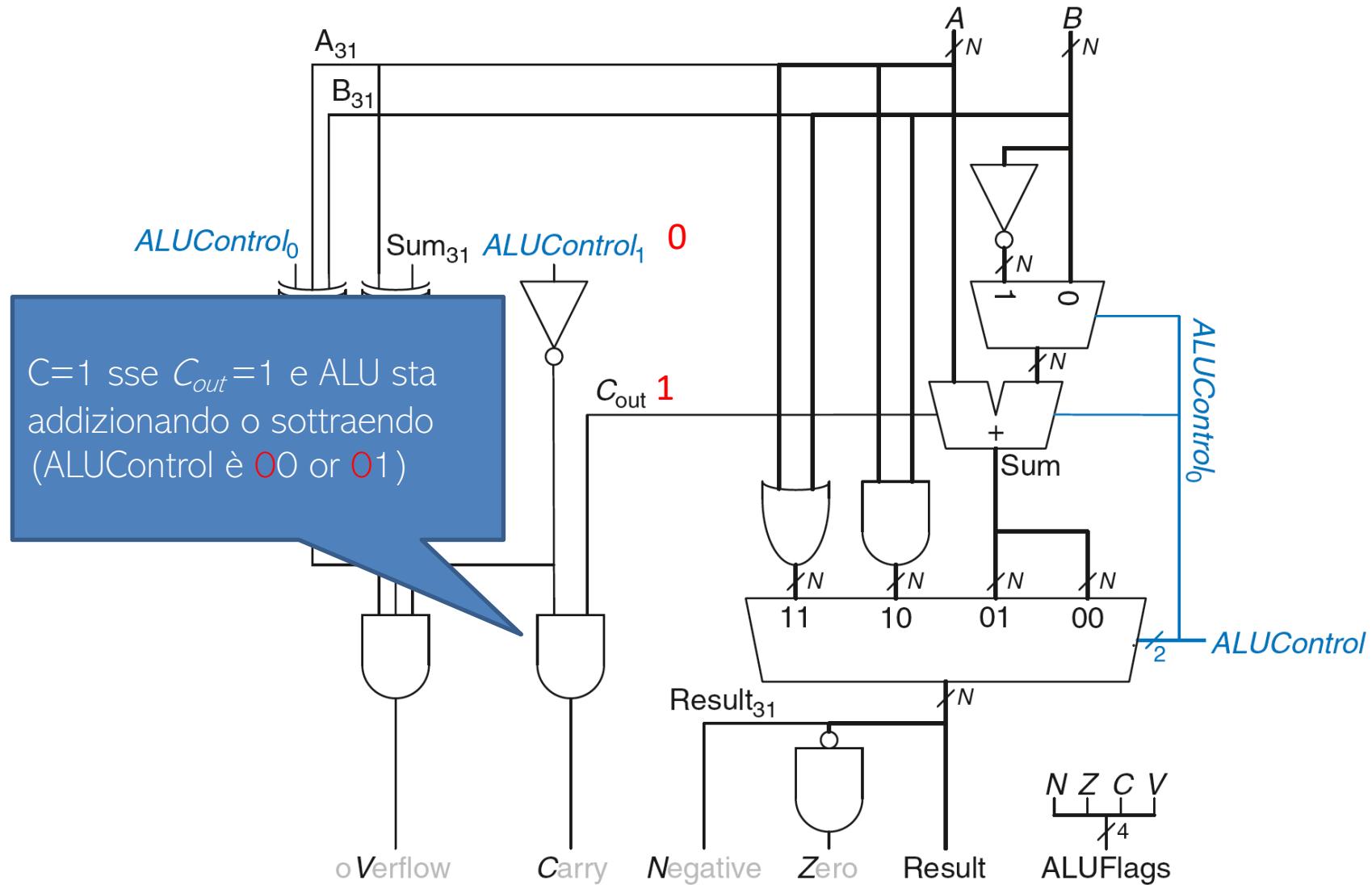
# ALU con flags di stato: Negative



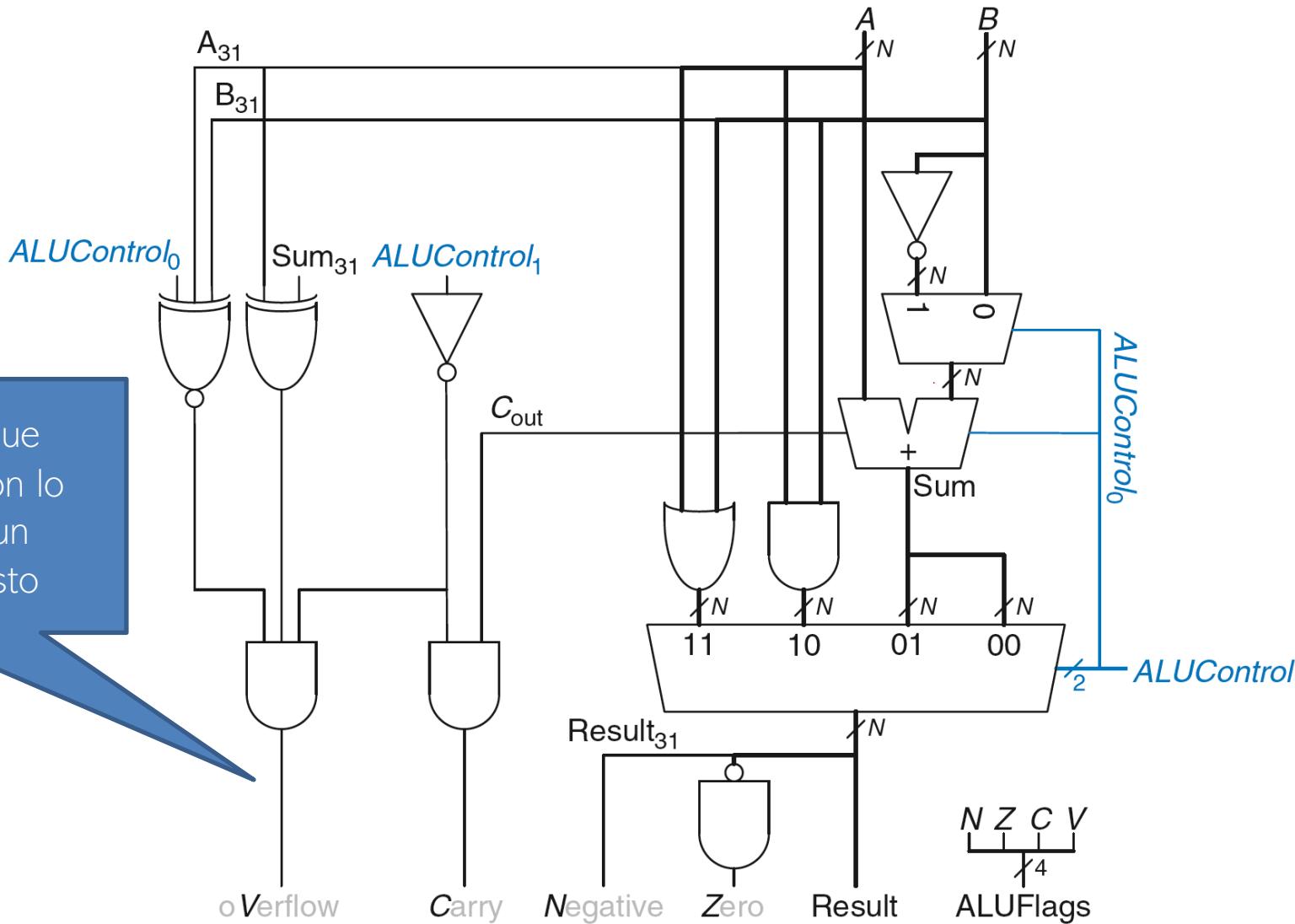
# ALU con flags di stato: Zero



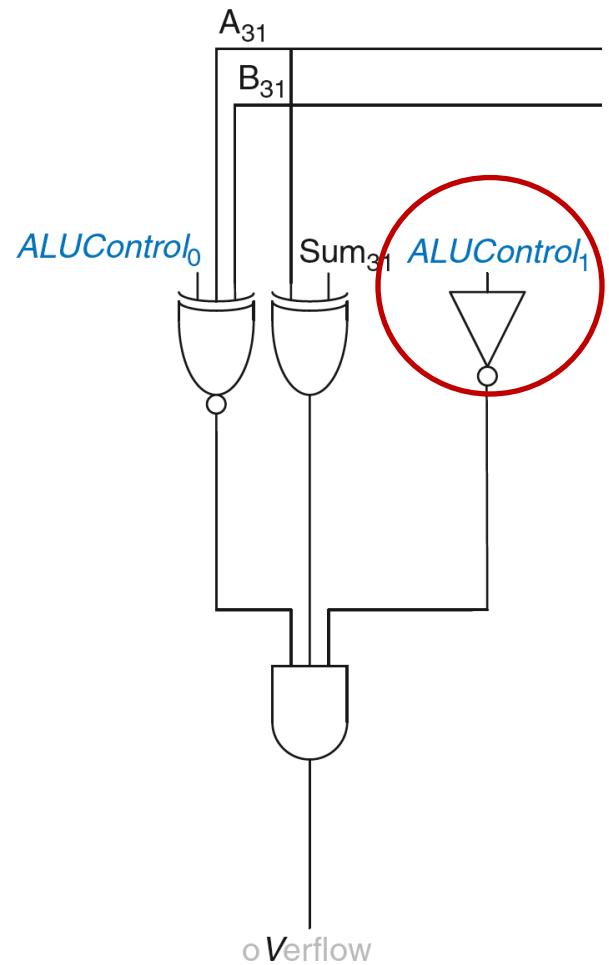
# ALU con flags di stato: Carry



# ALU con flags di stato: Overflow



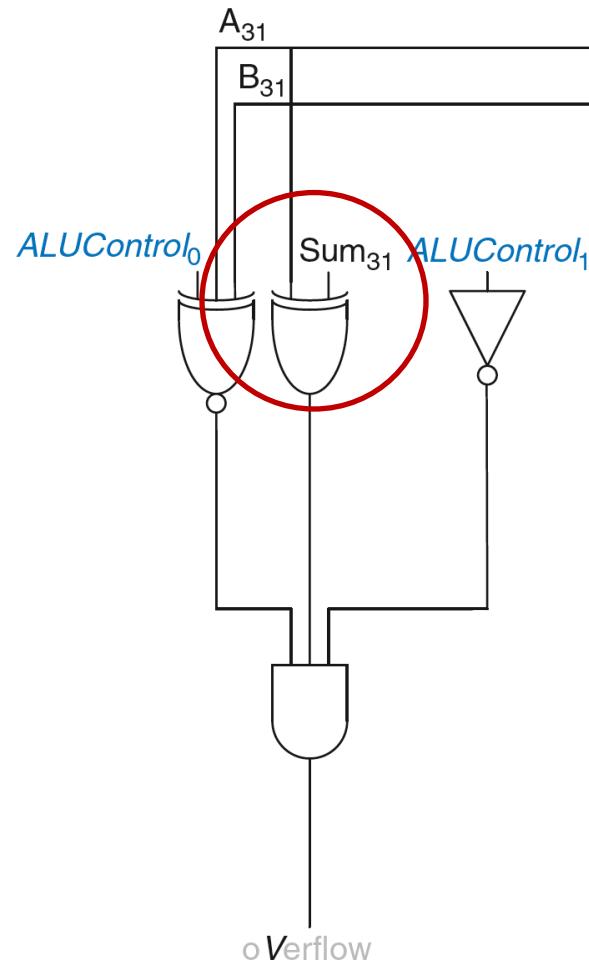
# ALU con flags di stato: Overflow



**V = 1** sse:

ALU esegue una addizione o sottrazione  
(*ALUControl*<sub>1</sub> = 0)

# ALU con flags di stato: Overflow



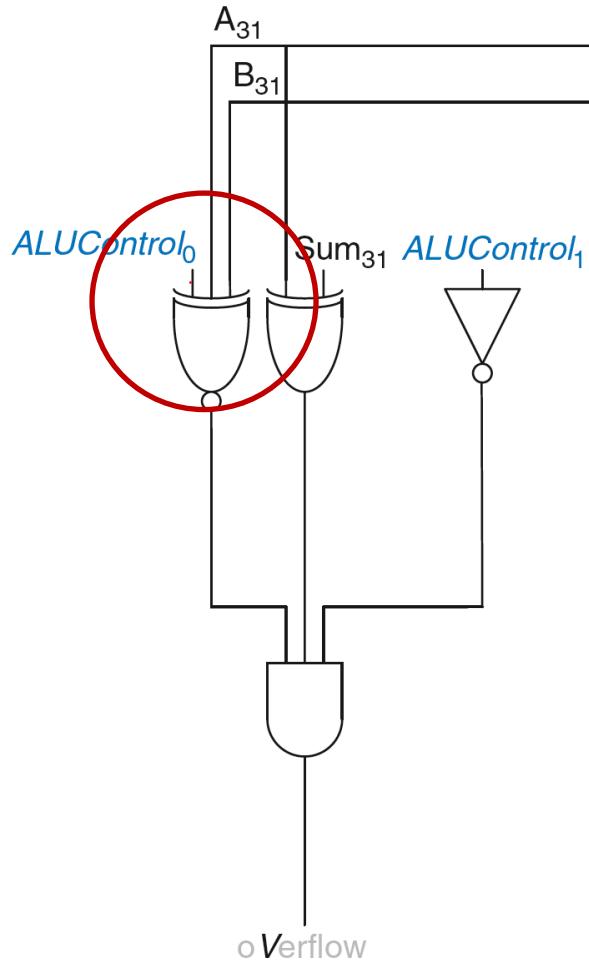
**V = 1** sse:

ALU esegue una addizione o sottrazione  
( $ALUControl_1 = 0$ )

**AND**

A e Sum hanno segno opposto

# ALU con flags di stato: Overflow



**V = 1** sse:

ALU esegue una addizione o sottrazione  
( $ALUControl_1 = 0$ )

**AND**

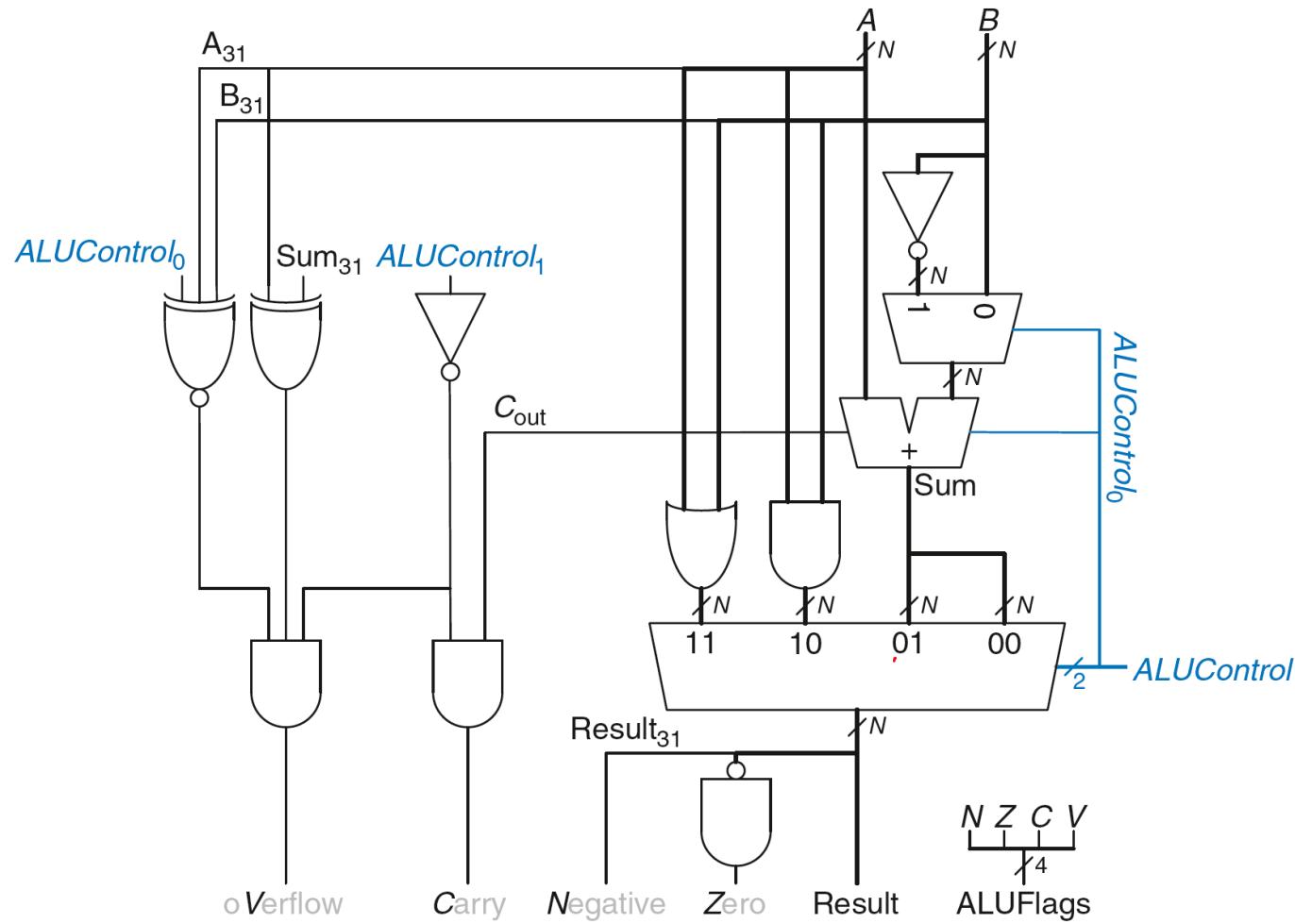
A e Sum hanno segno opposto

**AND**

A e B hanno lo stesso segno sotto addizione  
( $ALUControl_0 = 0$ ) **OR**

A e B hanno segni differenti sotto sottrazione  
( $ALUControl_0 = 1$ )

# ALU con flags di stato



# Shifters

**Logical shifter:** trasla i bit a sinistra o a destra e riempie gli spazi vuoti con degli 0

**Arithmetc shifter:** come il logical shifter a sinistra, nella traslazione a destra invece riempie gli spazi vuoti con il bit più significativo (msb)

**Rotator:** ruota i bits in cerchio, i bits che “escono” da un lato rientrano dall’ “altro”

# Shifters

## Logical shifter:

- Ex: 11001 >> 2 = 00110
- Ex: 11001 << 2 = 00100

.

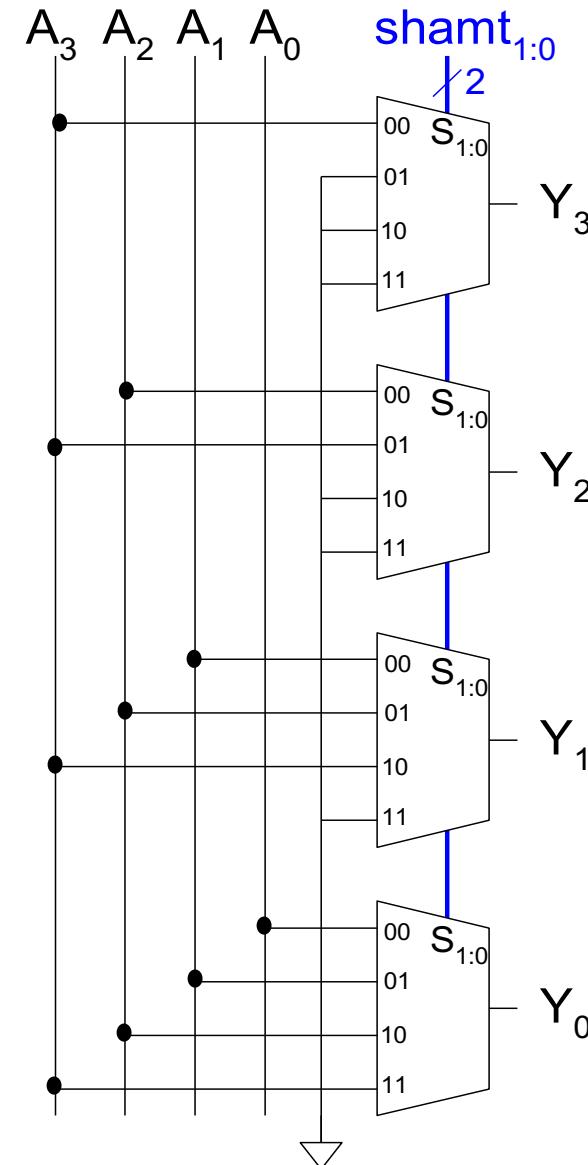
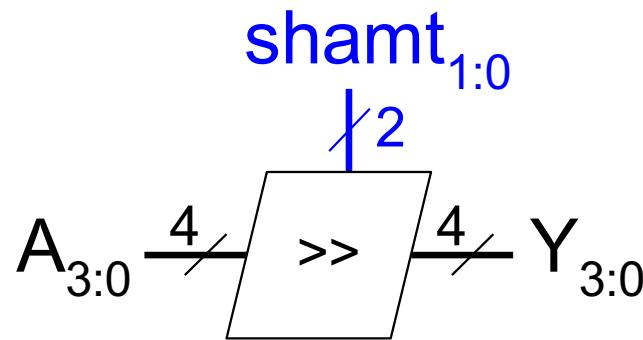
## Arithmetic shifter:

- Ex: 11001 >>> 2 = 11110
- Ex: 11001 <<< 2 = 00100

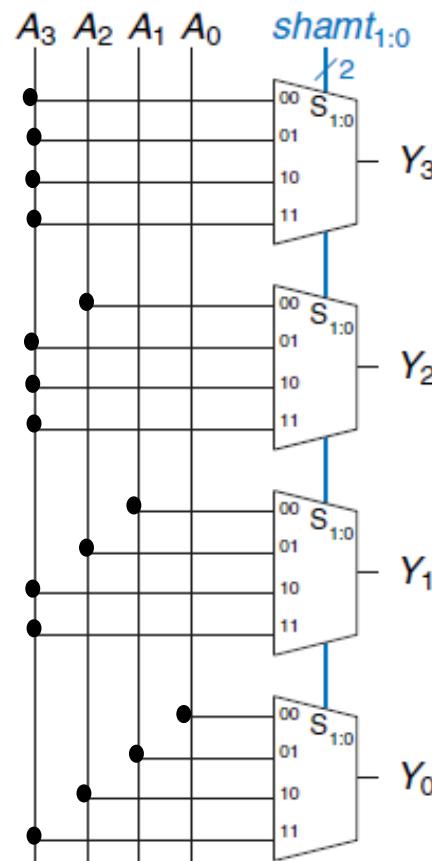
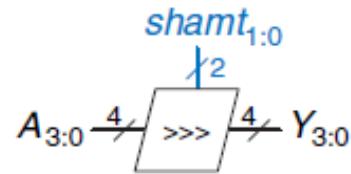
## Rotator:

- Ex: 11001 ROR 2 = 01110
- Ex: 11001 ROL 2 = 00111

# Shifter Design



# Shifter Design



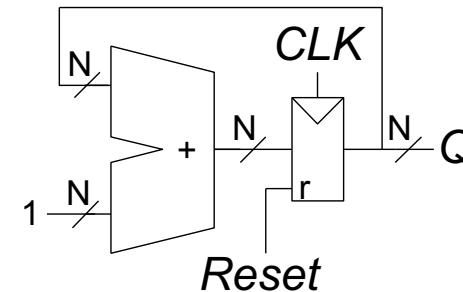
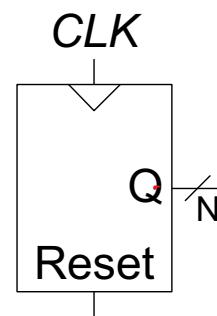
# Shifters as Multipliers, Dividers

- $A << N = A \times 2^N$ 
  - Example:  $00001 << 2 = 00100$  ( $1 \times 2^2 = 4$ )
  - Example:  $11101 << 2 = 10100$  ( $-3 \times 2^2 = -12$ )
- $A >> N = A \div 2^N$ 
  - Example:  $01000 >> 2 = 00010$  ( $8 \div 2^2 = 2$ )
  - Example:  $10000 >> 2 = 11100$  ( $-16 \div 2^2 = -4$ )

# Counters

- Incrementa ad ogni clock
- Usato per eseguire cicli su numeri:  
000, 001, 010, 011, 100, 101, 110, 111, 000, 001...
- Si usa:
  - Come orologio digitale
  - Program counter: tiene traccia della istruzione corrente da eseguire

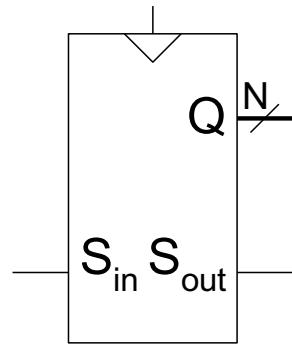
## Symbol      Implementation



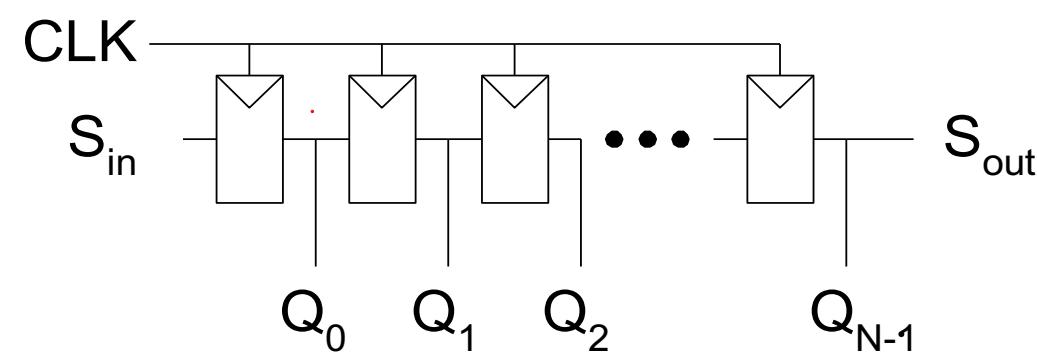
# Shift Registers

- Trasla di un bit ad ogni clock
- Ritorna un bit in uscita ( $S_{out}$ ) ad ogni ~~clock~~
- *Serial-to-parallel converter*: converte un input seriale ( $S_{in}$ ) in un output parallelo ( $Q_{0:N-1}$ )

## Symbol:

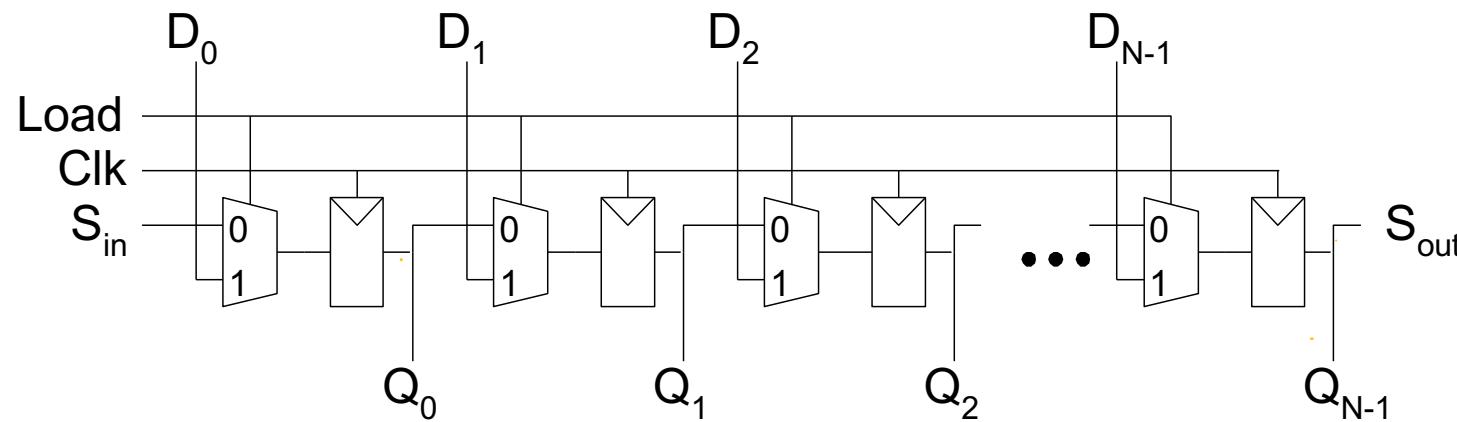


## Implementation:



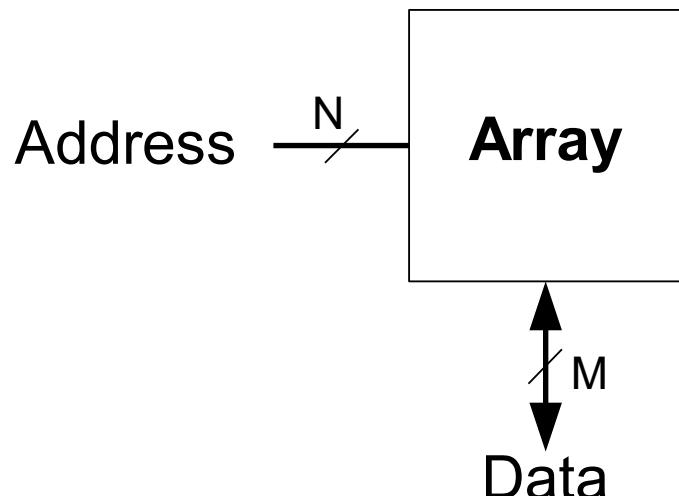
# Shift Register con load parallelo

- Quando  $Load = 1$ , funziona come un usuale registro a  $N$ -bit
- Quando  $Load = 0$ , agisce da shift register
- Quindi può agire da convertitore *seriale-parallelo* (da  $S_{in}$  a  $Q_{0:N-1}$ ) o da convertitore *parallelo-seriale* ( $D_{0:N-1}$  to  $S_{out}$ )



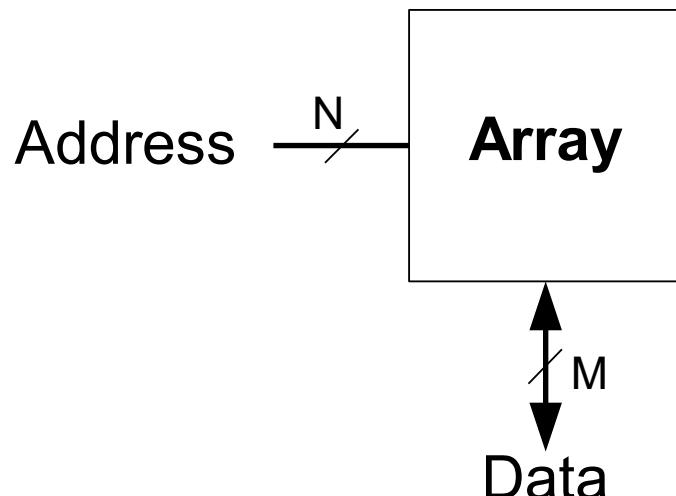
# Memory Arrays

- Memorizzare efficacemente una grossa quantità di dati
- 3 tipologie:
  - Dynamic random access memory (DRAM)
  - Static random access memory (SRAM)
  - Read only memory (ROM)
- dato a  $M$ -bit letto/scritto su di un unico indirizzo a N-bit



# Memory Arrays

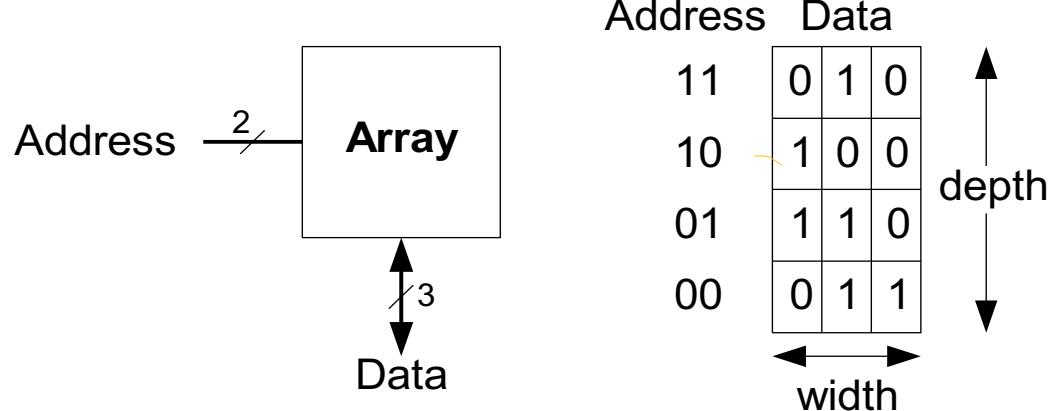
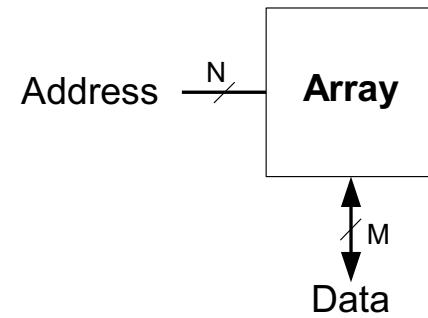
- Memorizzare efficacemente una grossa quantità di dati
- 3 tipologie:
  - Dynamic random access memory (DRAM)
  - Static random access memory (SRAM)
  - Read only memory (ROM)
- dato a  $M$ -bit letto/scritto su di un unico indirizzo a  $N$ -bit



Tipicamente  
 $M=8$  (un byte) e  
 $N=32$  (o 64)

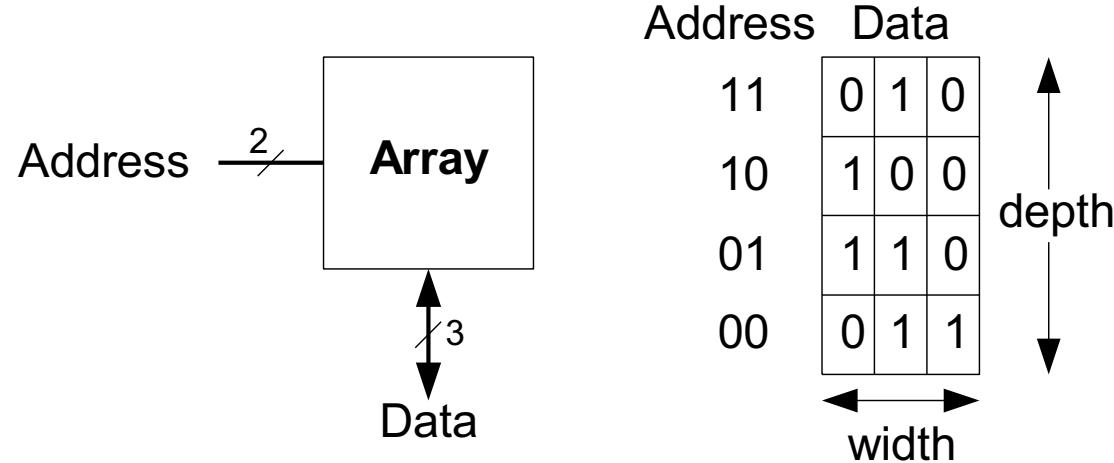
# Memory Arrays

- Array bidimensionali di celle di memoria
- Ogni cella memorizza un bit
- Con  $N$  bit di indirizzo e  $M$  bits data:
  - $2^N$  righe e a  $M$  colonne
  - Depth: numero di righe (parole)
  - Width: numero di colonne (lunghezza di una parola)
  - Dimensione Array : depth  $\times$  width =  $2^N \times M$

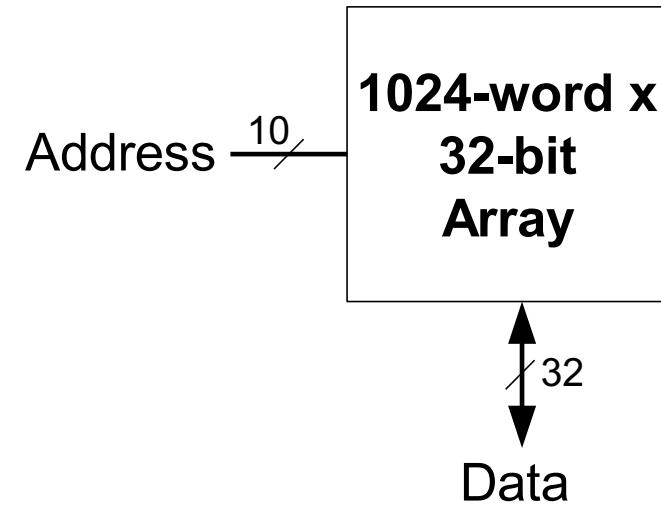


# Esempio

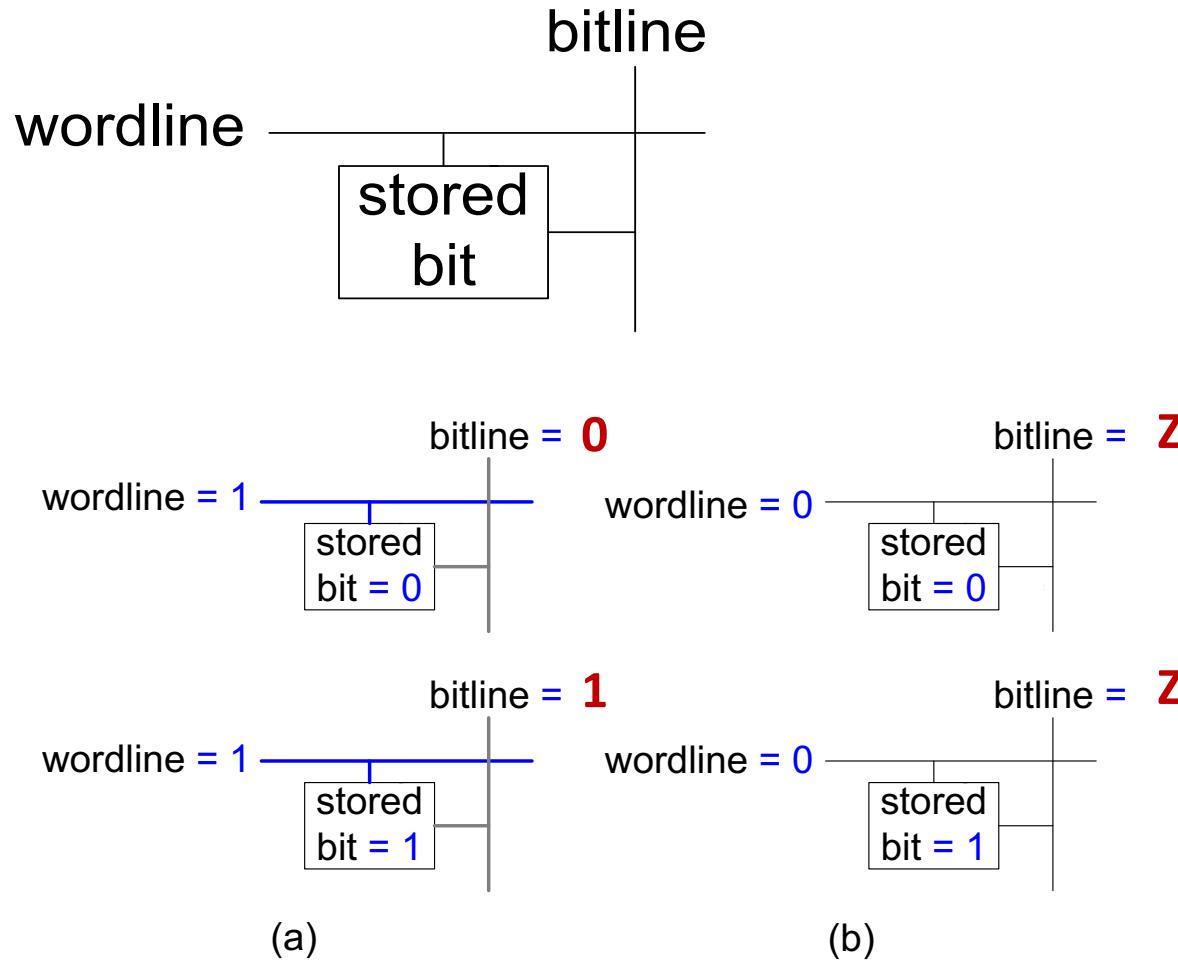
- $2^2 \times 3$ -bit array
- Numero parole: 4
- Lunghezza parole: 3-bits
- All'indirizzo 10 corrisponde la parola 100



# Memory Arrays



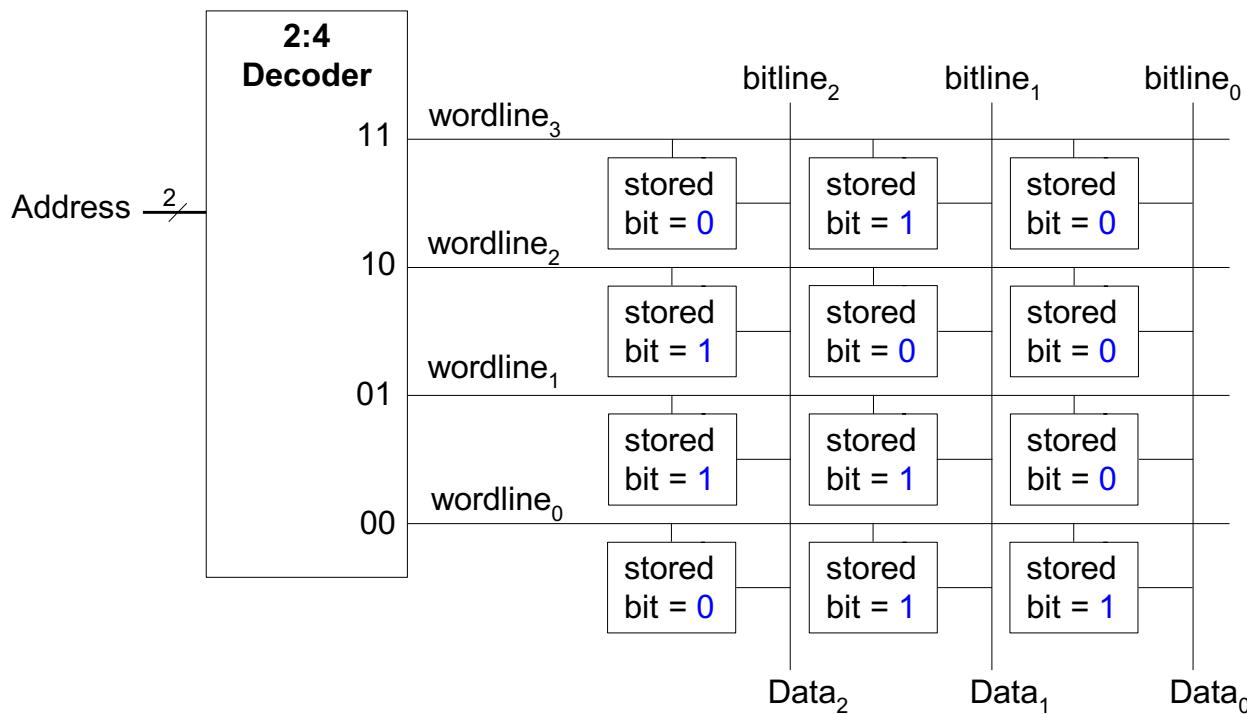
# Memory Array Bit Cells



# Memory Array

- **Wordline:**

- Agisce come un enable
- Seleziona una riga nella memoria
- Corrisponde ad un unico indirizzo
- Solo una wordline per volta è attivata



# Tipi di memoria

- Random access memory (RAM): **volatile**
- Read only memory (ROM): **nonvolatile**

# RAM: Random Access Memory

- **Volatile:** i dati sono persi quando il computer è spento
- Le operazioni di lettura e scrittura sono più veloci (rispetto alle ROM)
- E' la memoria primaria di un computer (DRAM)

E' chiamata *random access memory* perché il tempo di accesso ad una parola è lo stesso per ogni indirizzo (a differenza delle memorie ad accesso sequenziale come i nastri che si usavano ai tempi del C64)

# ROM: Read Only Memory

- **Non volatile:** mantiene i dati anche quando il computer è spento
- La lettura è relativamente veloce ma la scrittura è lenta o semplicemente non è possibile scrivere
- Drives, flash memory, Bios etc. sono ROMs

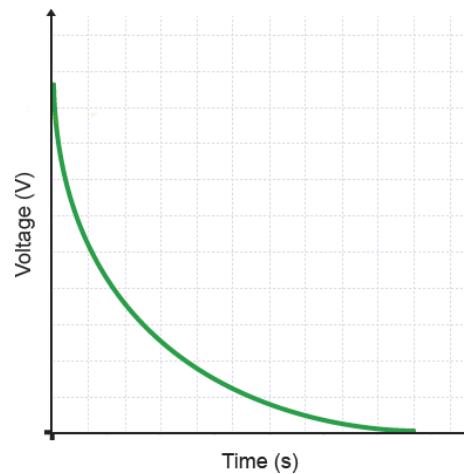
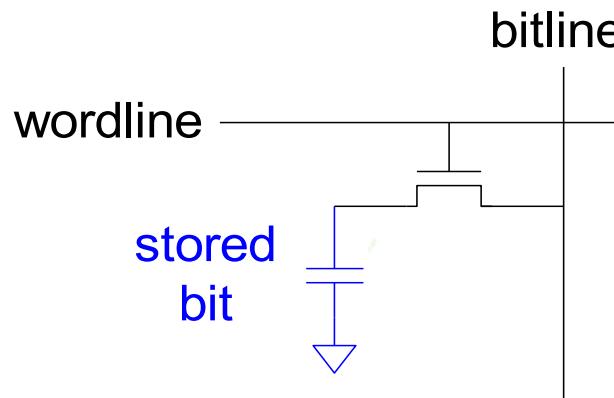
ROM sta per *read only* memory perché inizialmente questo tipo di memorie venivano “scritte” bruciando dei fusibili. Quindi, una volta configurate, non erano più manipolabili. Chiaramente questo non è più il caso per Flash memory e altri tipi di ROMs.

# Tipi di RAM

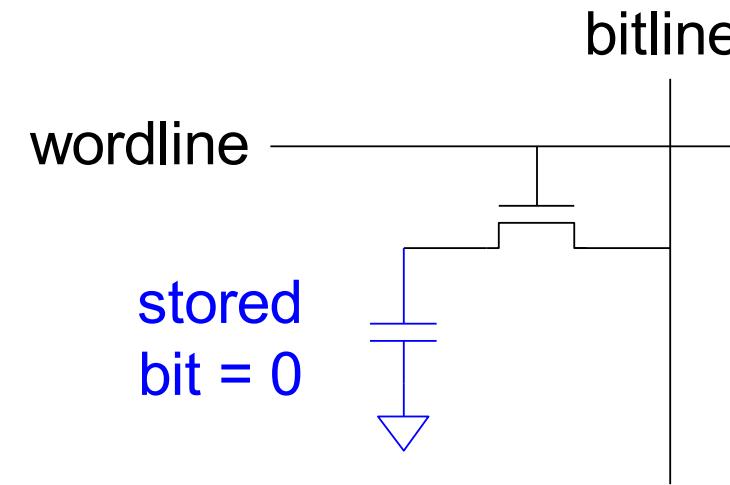
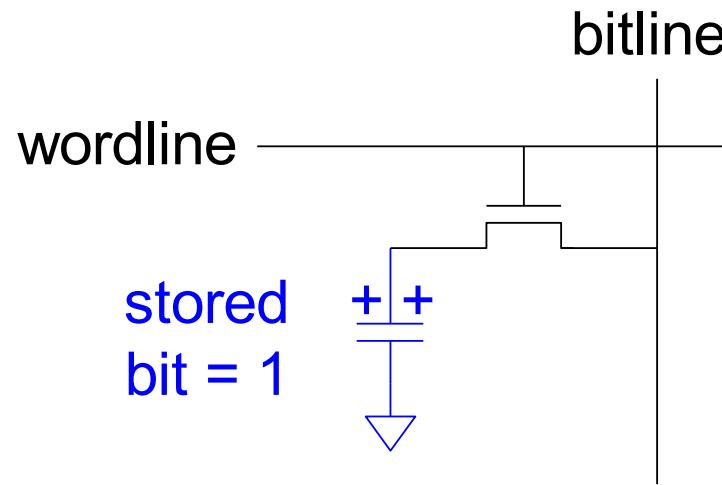
- DRAM (Dynamic random access memory)
- SRAM (Static random access memory)
- Si differenziano per le componenti usate per memorizzare i dati:
  - DRAM usa delle capacità
  - SRAM usa invertitori

# DRAM

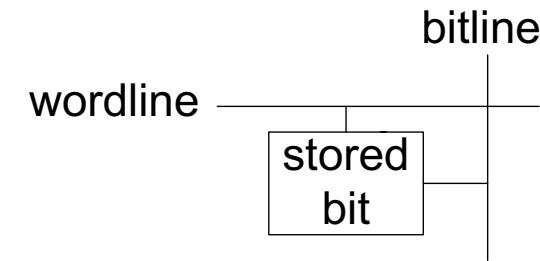
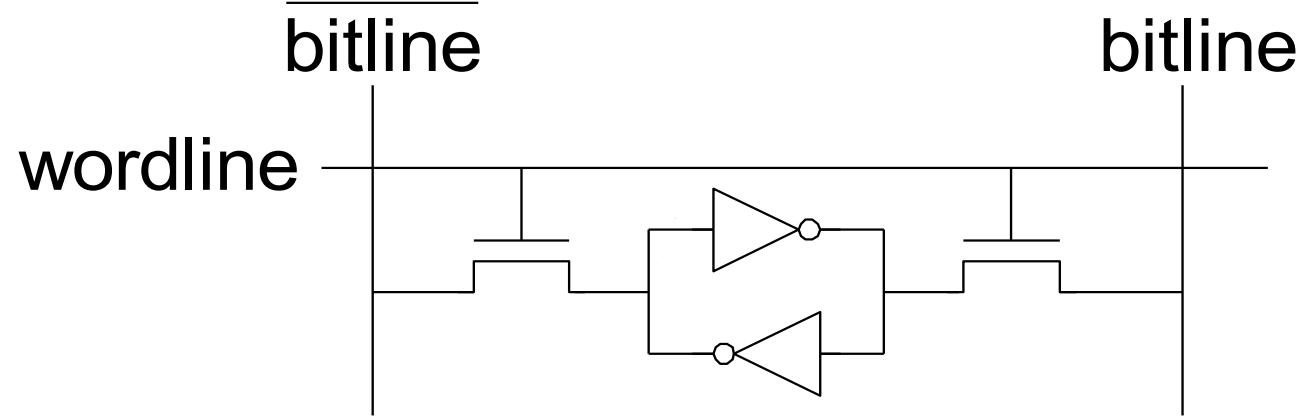
- i bit data sono immagazzinati in delle capacità
- *Dynamic* perché il valore di un bit deve essere “refreshed” (riscritto) periodicamente e anche dopo che è stato letto:
  - La perdita di carica di una capacità degrada il valore memorizzato
  - La lettura distrugge il valore letto



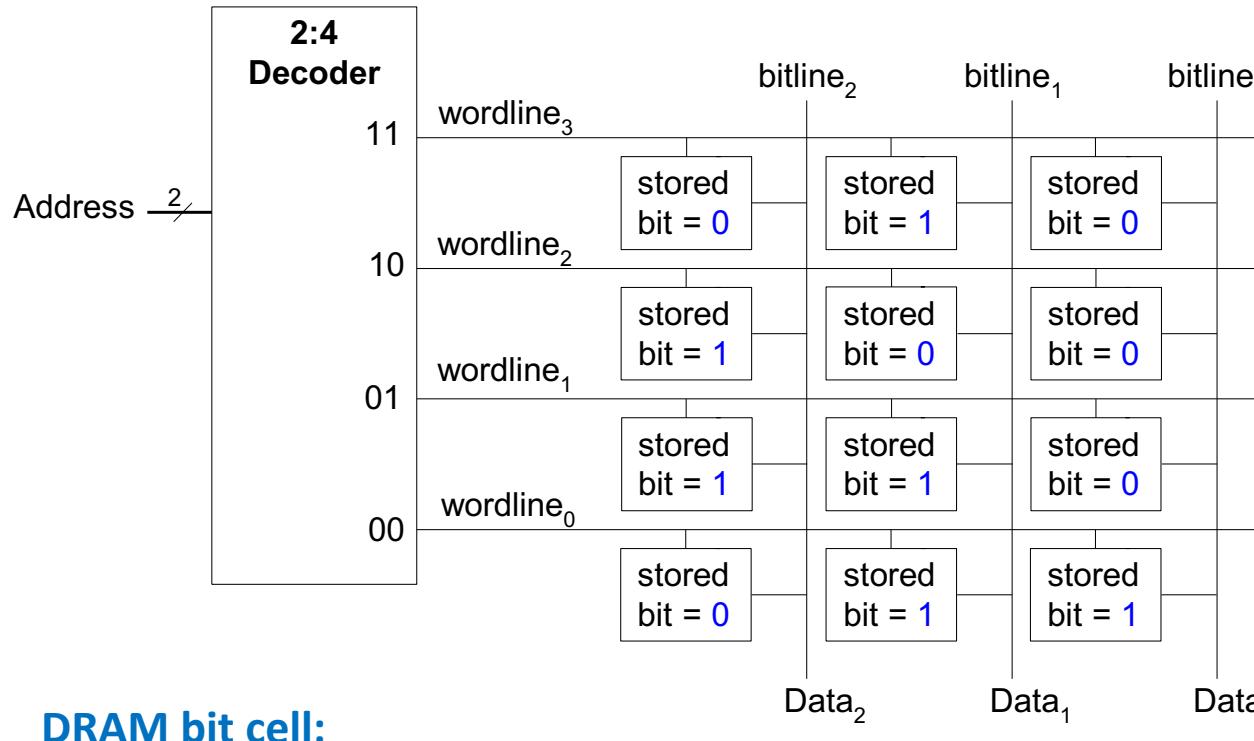
# DRAM



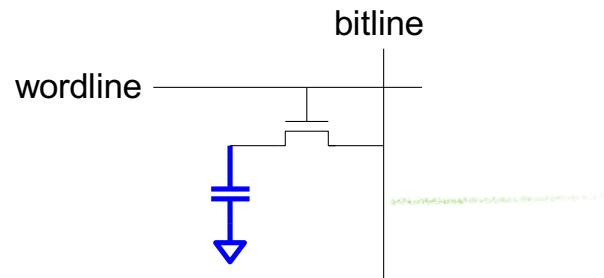
# SRAM



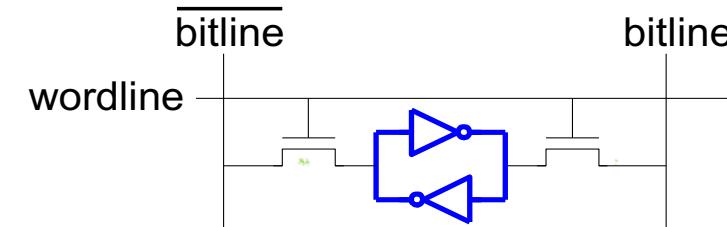
# Memory Arrays Review



**DRAM bit cell:**



**SRAM bit cell:**



# DRAM vs SRAM

- DRAM è più lenta
- Scrittura e refresh (millisecondi) inducono un consumo maggiore di energia
- DRAM, più economiche

| Memory Type | Transistors per Bit Cell | Latency |
|-------------|--------------------------|---------|
| flip-flop   | ~20                      | fast    |
| SRAM        | 6                        | medium  |
| DRAM        | 1                        | slow    |

# DRAM vs SRAM

- DRAM è più lenta
- Scrittura e refresh (millisecondi) inducono un consumo maggiore di energia
- DRAM, più economiche

| Memory Type | Transistors per Bit Cell | Latency |
|-------------|--------------------------|---------|
| flip-flop   | ~20                      | fast    |
| SRAM        | 6                        | medium  |
| DRAM        | 1                        | slow    |



The diagram features a callout bubble with a blue arrow pointing from the 'SRAM' row in the table towards the word 'registri' located to the right of the table.

# DRAM vs SRAM

- DRAM è più lenta
- Scrittura e refresh (millisecondi) inducono un consumo maggiore di energia
- DRAM, più economiche

| Memory Type | Transistors per Bit Cell | Latency |
|-------------|--------------------------|---------|
| flip-flop   | ~20                      | fast    |
| SRAM        | 6                        | medium  |
| DRAM        | 1                        | slow    |



cache

# DRAM vs SRAM

- DRAM è più lenta
- Scrittura e refresh (millisecondi) inducono un consumo maggiore di energia
- DRAM, più economiche

| Memory Type | Transistors per Bit Cell | Latency |
|-------------|--------------------------|---------|
| flip-flop   | ~20                      | fast    |
| SRAM        | 6                        | medium  |
| DRAM        | 1                        | slow    |

memoria centrale

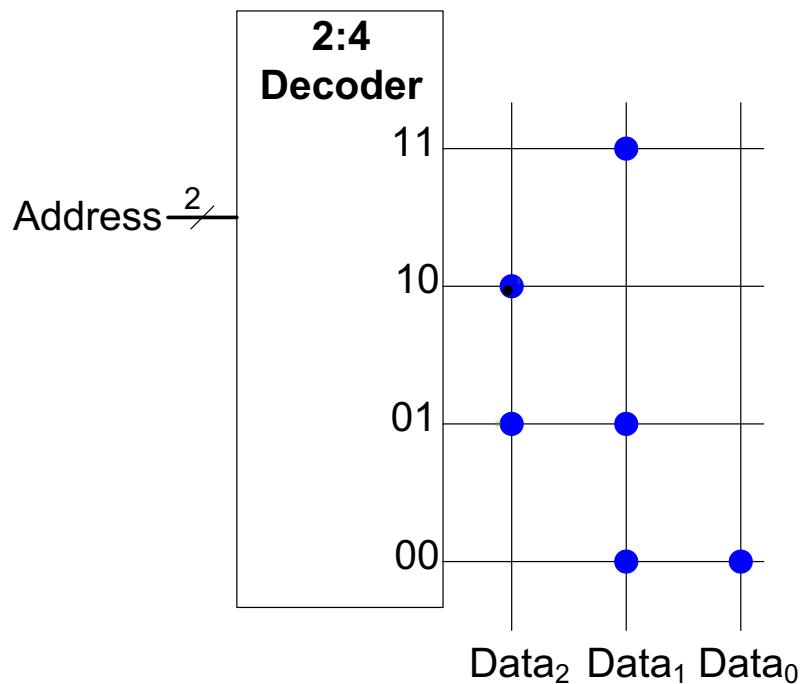
# DDR SDRAM

- (DDR) Double data rate (S) synchronous (DRAM) dynamic random access memory
- Le operazioni di lettura e scrittura sono sincronizzate da un clock
- Le trasmissioni avvengono sia nel rising-edge del clock ( $0 \rightarrow 1$ ) che nel falling-edge ( $1 \rightarrow 0$ )
- In questo modo il data-bandwidth doppio rispetto alla frequenza di clock (double pumping)

# DDR SDRAM

| Standard name | Memory clock<br>(MHz) | Cycle time <sup>[4]</sup><br>(ns) | Data rate<br>(MT/s) | Peak transfer rate<br>(MB/s) |
|---------------|-----------------------|-----------------------------------|---------------------|------------------------------|
| DDR-200       | 100                   | 10                                | 200                 | 1600                         |
| DDR-266       | 133.33                | 7.5                               | 266.67              | 2133.33                      |
| DDR-333       | 166.67                | 6                                 | 333.33              | 2666.67                      |
| DDR-400       | 200                   | 5                                 | 400                 | 3200                         |

# ROM Storage

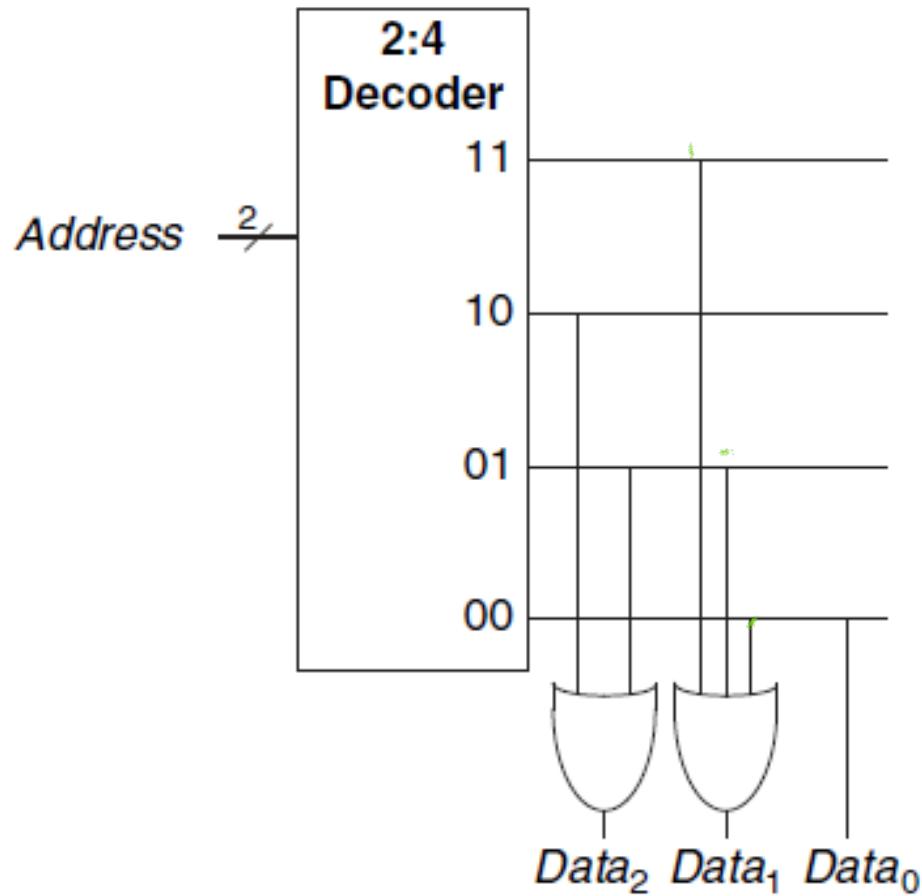


| Address | Data |   |   |
|---------|------|---|---|
| 11      | 0    | 1 | 0 |
| 10      | 1    | 0 | 0 |
| 01      | 1    | 1 | 0 |
| 00      | 0    | 1 | 1 |

depth

width

# ROM via porte logiche



Address      Data

| Address | Data |   |   |
|---------|------|---|---|
| 11      | 0    | 1 | 0 |
| 10      | 1    | 0 | 0 |
| 01      | 1    | 1 | 0 |
| 00      | 0    | 1 | 1 |

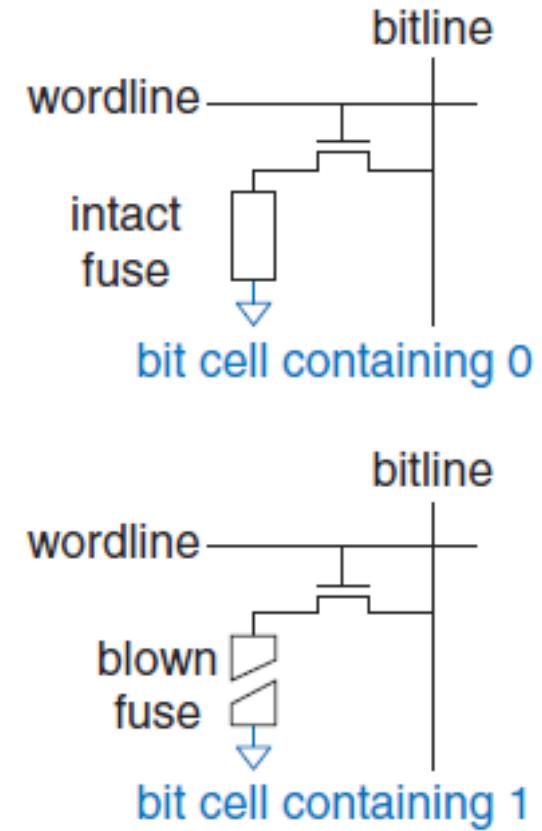
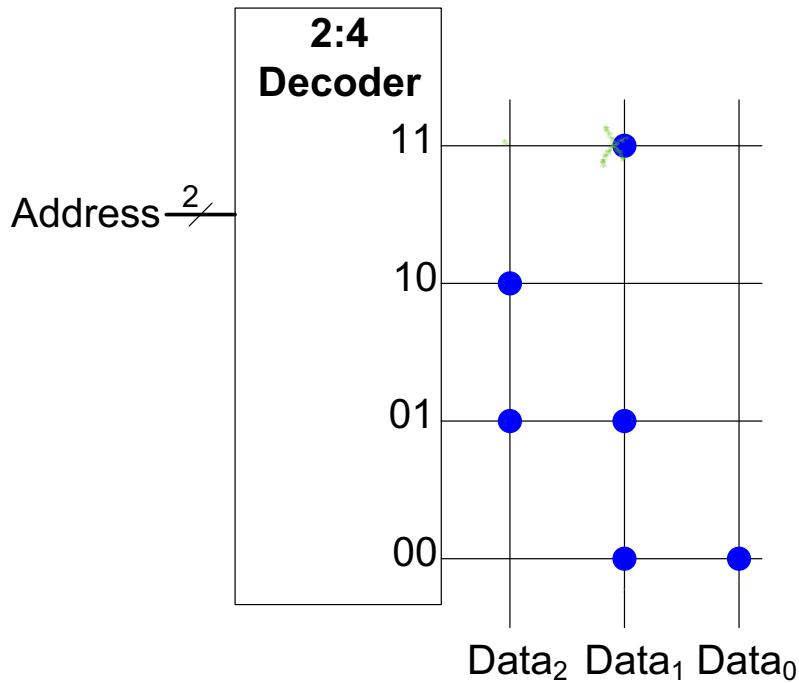
depth

width

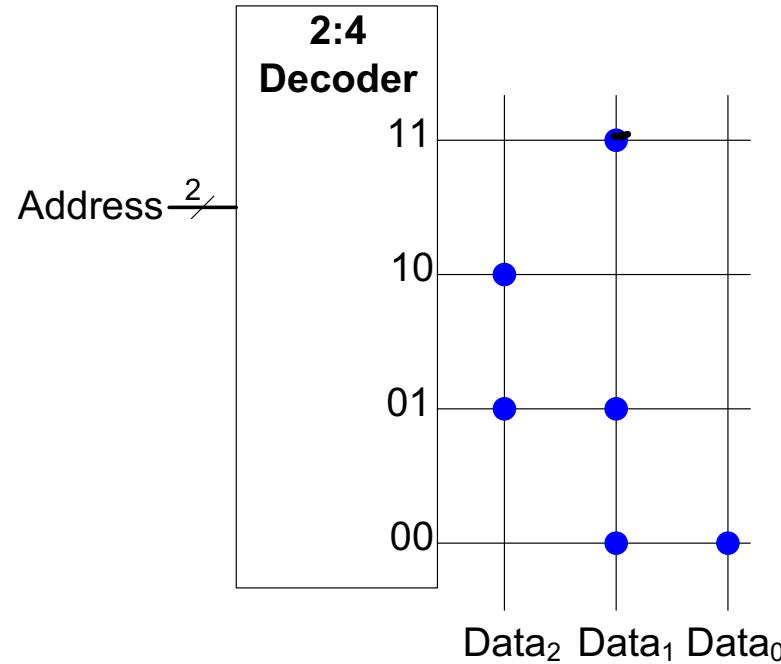
depth

width

# Fuse Programmable ROM



# ROM Logic



$$Data_2 = A_1 \oplus A_0$$

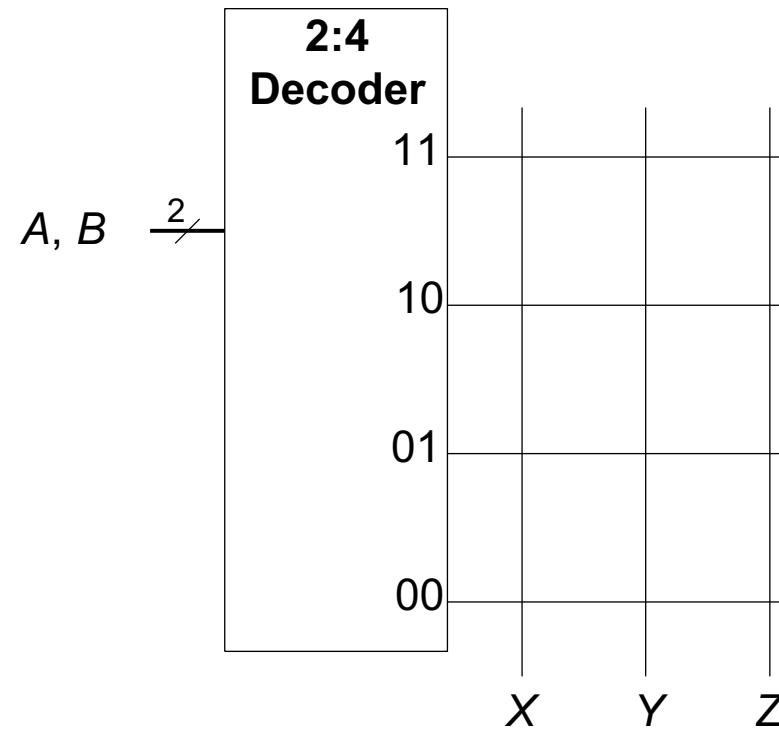
$$Data_1 = \overline{A_1} + A_0$$

$$Data_0 = \overline{A_1} \overline{A_0}$$

# Esempio

Usare un  $2^2 \times 3$ -bit ROM per implementare funzioni booleane:

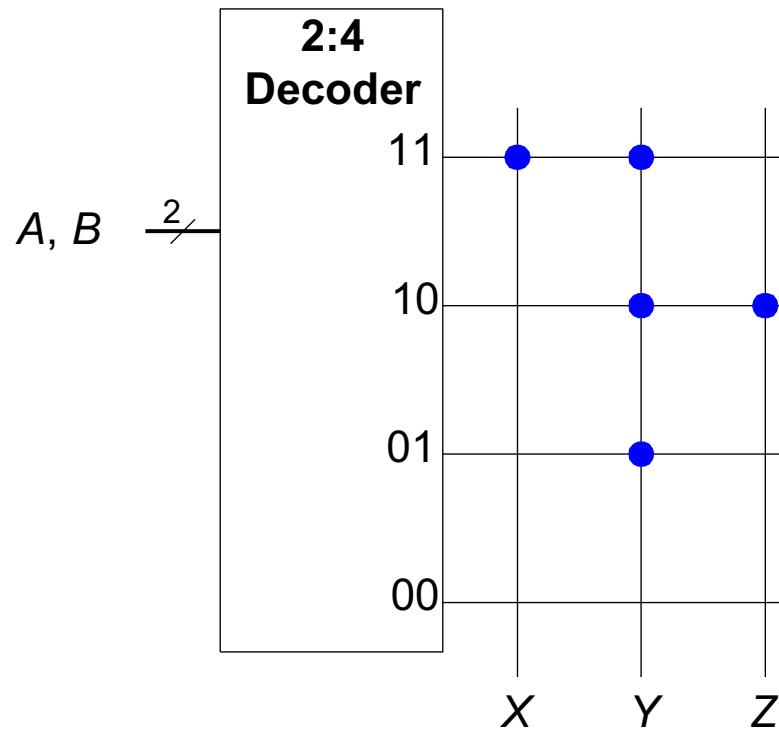
- $X = AB$
- $Y = A + B$
- $Z = A \bar{B}$



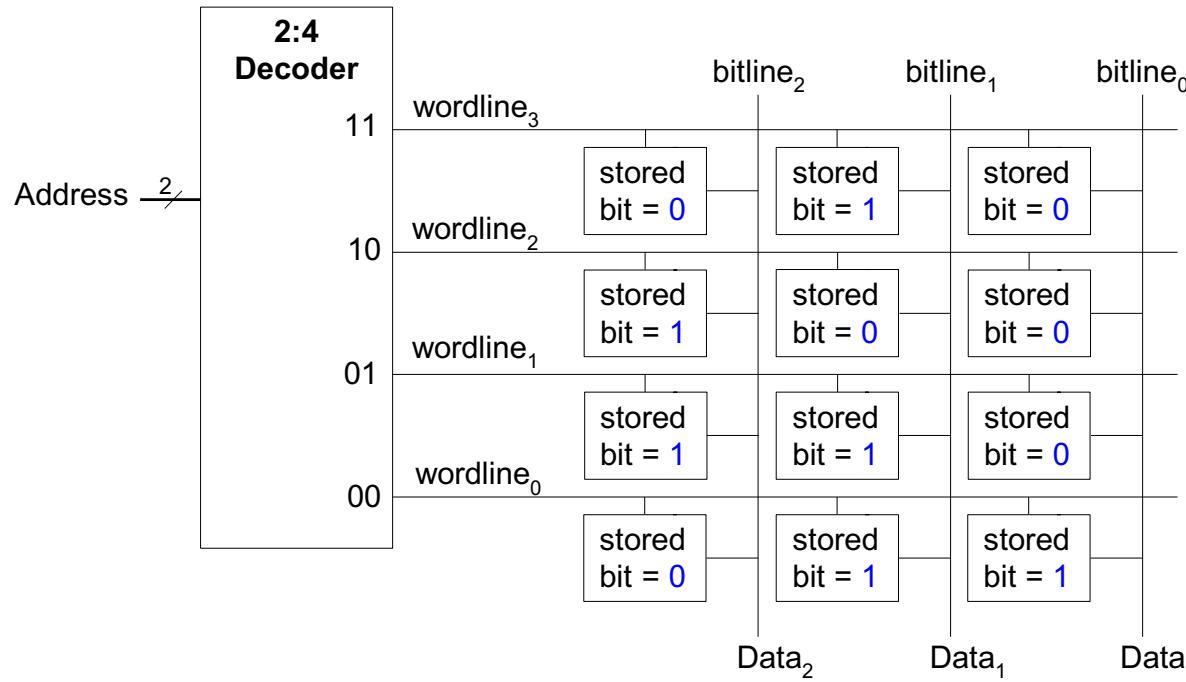
# Esempio

Usare un  $2^2 \times 3$ -bit ROM per implementare funzioni booleane:

- $X = AB$
- $Y = A + B$
- $Z = A \bar{B}$



# *Lookup tables (LUTs)*



$$Data_2 = A_1 \oplus A_0$$

$$Data_1 = \overline{A}_1 + A_0$$

$$Data_0 = \overline{A}_1 \overline{A}_0$$

# Logiche con memory array

Sono chiamate *lookup tables* (LUTs): si “guardano” gli output per ogni combinazione di input (indirizzo)

