

APPUNTI

***Architettura
degli
Elaboratori***

ADE

**PROF.
LUIGI SAURO**

**2020/
2021**

Programma ADE

0.INTRODUZIONE

- I livelli di un architettura
- La regola delle 3Y
- Circuiti digitali / analogici

1.RAPPRESENTAZIONE DELL'INFORMAZIONE

- Codifica, decodifica, ambigua, ridondante, proprietà di una codifica
- Valore massimo rappresentabile, cifre necessarie per rappresentare n
- Codifica ottimale
- codifica binaria/esadecimale
- somma binaria, moltiplicazione
- rappresentazione con segno, complemento a 2, overflow nel complemento a 2
- codice bcd
- rappresentazione di numeri con la virgola in virgola fissa e virgola mobile
- standard ieee 754
- codifiche di caratteri ascii unicode utf-8

2.ALGEBRA DI BOOLE E RETI COMBINATORIE

- porte logiche (not, and, or, xor, nand, nor, xnor, and4, nor3)
- porte logiche dal punto di vista fisico, valori di tensione
- funzioni booleane
- circuiti digitali
- reti combinatorie e sequenziali. differenza
- regole di composizione di reti combinatorie
- tabella di verità ed espressioni booleane
- dalle espressioni alle tabelle e dalle tabelle alle espressioni
- littorali, implicanti, mintermine, maxtermine
- forme SOP e POS, differenze
- Algebra di Boole
- Assiomi dell'algebra di Boole
- Teoremi dell'algebra di Boole, principio di dualità
- Tecniche di dimostrazione: perfect induction, proof teoretico, per casi rilevanti
- Teoremi di De Morgan, Teoremi dell'assorbimento
- De Morgan e forme SOP/POS
- Da forma SOP a POS
- completezza degli operatori
- semplificazione di un'espressione booleana
- teoremi per semplificare espressioni
- schemi circuitali sop, semplificare forme sop
- semplificare forme pos
- circuito a priorità
- valori don't cares (indifferenze)
- vantaggi e limiti delle forme sop/pos
- logiche multilivello
- bubble pushing
- valori illegali

- stato di alta impedenza
- bus condiviso
- mappe di karneugh, minimizzazione di una mappa
- display a 7 segmenti
- multiplexer
- sintetizzare funzioni booleane con mux
- Decoder
- sintetizzare funzioni booleane con decoder

3.CIRCUITI SEQUENZIALI

- logica sequenziale e sistemi sequenziali
- circuiti sequenziali
- state element
- circuito bistabile, analisi
- latch set/reset, analisi
- d latch
- d flip flop
- differenza fra d latch e d flip flop
- i registri: multibit d flip flop
- flip flop enabled
- flip flop resettabili
- flip flop settabili
- criticità logica sequenziale
- logiche sequenziali sincrone. differenza con asincrone
- design di logiche sequenziali sincrone
- finite states machines
- moore fsm, mealy fsm. circuiti relativi
- black box, diagramma di transizione, tabella di transizione, tabella dell'output, schema della logica di transizione, schema della logica di output
- encoding degli stati: binario e one hot
- differenze fra mealy e moore fsm
- fattorizzazione di fsm
- progettare una fsm
- parallelismo: spaziale e temporale
- latenza e throughput

4.CIRCUITI ARITMETICI E MEMORIE

- adders: 1 bit adders, adder, full adder
- multibit adders (carry propagate adders)
- ripple carry adder, ritardo
- carry lookahead, logica di generate e propagate: formule, ritardo
- confronto fra ripple carry e carry lookahead
- sottrattore
- compratore: uguaglianza, comparatore completo a 1 bit, completo a 4 bit, completo a 12bit
- alu, alu con flags di stato, significato dei flags nzcv e dei segnali alucontrol
- shifter: Isl, Isr, asr, rotator
- shifter design

- shifter per moltiplicare e dividere
- counter
- shift registers: seriale-parallelo, parallelo-seriale
- memory arrays: la memoria
- wordline, bitline, come è fatta una cella di memoria
- tipi di memoria: RAM, ROM
- tipi di RAM: SRAM, DRAM
- come è fatta una SRAM, una DRAM
- confronto fra SRAM e DRAM
- DDR SDRAM
- ROM storage, ROM via porte logiche, logica rom, lookup tables

5. MICROARCHITETTURA

- definizione architettura/microarchitettura
- assembly language/machine language
- architettura arm
- i 4 principi di design dell'architettura arm: 1. Regularity supports design simplicity, 2. Make the common case fast, 3. Smaller is faster, 4. Good design demands good compromises
- cos'è un processore
- microarchitetture come automi
- variazioni di stato
- microarchitetture a ciclo singolo, ciclo multiplo, con pipeline. relativi vantaggi e svantaggi
- misura delle prestazioni di un processore
- schema generale microarchitetture: ciclo di vita di una istruzione fetch, decode, execute, store
- progettazione di un'architettura: datapath e unità di controllo
- program counter
- instruction memory
- file register
- data memory
- status
- istruzioni di base: data processing, di memoria, di salto
- istruzioni di memoria: LDR, STR, LDRB, STRB
- organizzazione little endian e big endian
- formato istruzioni di memoria
- i tipi di indicizzazione: offset, preindex, postindex
- campi di funct
- campo sh
- datapath LDR
- datapath STR
- Istruzioni di data processing
- istruzioni logiche AND; ORR, EOR, BIC, MVN.
- istruzioni di shifting: LSL, LSR, ASR, ROR
- istruzioni di moltiplicazione: MUL, UMULL, SMULL
- formato istruzioni di data processing
- formato immediate
- formato register

- formato register shifted register
- Datapath per le istruzioni di data processing ADD, SUB, AND, ORR
- esecuzione condizionata
- istruzione di comparazione CMP
- suffisso S al nome di una istruzione
- istruzioni condizionate
- istruzioni di branching
- datapath istruzioni di branching
- unità di controllo: come è fatta
- logica condizionale
- analisi delle prestazioni
- limiti ciclo singolo
- vantaggi ciclo multiplo
- datapath Idr, str, data processing , branch ciclo multiplo
- unità di controllo ciclo multiplo
- dataflow di una istruzione

6.ARCHITETTURA

- linguaggio assembly
- locazione degli operandi
- generazione di costanti, immediate
- istruzione MOV
- istruzioni condizionali: CMP
- istruzioni iterative for, while..
- istruzioni condizionate B, BL
- array in assembly
- chiamate di funzione in assembly
- argomenti e valori di ritorno
- i registri assembly
- stack delle funzioni
- pop e push
- chiamate a funzioni innestate
- funzioni ricorsive

7. MEMORIE

- memorie vs CPU
- cache memory
- principi di funzionamento cache
- dimensione della cache
- scala e gerarchia delle memorie
- tecniche di gestione della cache
- gestione cache hit
- gestione read miss
- gestione write miss
- prestazioni cache: amat
- terminologia cache
- direct mapped cache
- n.way set associative cache

- fully associative cache
- località spaziale e temporale
- bit di validità
- direct memory access DMA
- gestione del bit di validità
- svuotamento della cache
- coerenza della cache
- algoritmi di sostituzione
- algoritmo lru
- tipi di miss
- funzionamento della memoria in moduli
- cache multilivello
- cache condivise
- memoria virtuale
- pagine di memoria
- indirizzi fisici e virtuali
- memory management unit
- page table

0. INTRODUZIONE

Gestire la complessità, livelli di astrazione di un calcolatore

Un calcolatore può essere visto da diversi livelli di astrazione.

L'**astrazione** è una tecnica fondamentale per controllare la complessità di un sistema e consiste nel nascondere tutti i dettagli non importanti di un determinato sistema e tenere conto solo degli aspetti fondamentali.

Per un calcolatore abbiamo diversi livelli di astrazione:

- al livello più basso c'è la **fisica e il moto degli elettronni**.
- più in alto vi sono i **componenti elettronici fondamentali** dei quali è composto il calcolatore ossia i transistor.
- il livello di astrazione successivo riguarda i **circuiti analogici** del sistema.

I **circuiti analogici** sono dei circuiti che hanno degli ingressi e delle uscite i cui livelli di tensione variano in modo **continuo**.

I **circuiti digitali**, a livello più alto, sono dei circuiti come le porte logiche che limitano i valori di tensione a livelli **discreti**, che sono utilizzati per indicare 0 e 1.

Partendo dai circuiti digitali possiamo costruire le **porte logiche** e da esse possiamo arrivare a **blocchi logici** complessi come i sommatori o le memorie.

- la **microarchitettura** unisce il livello di astrazione della logica e quello dell'architettura. combina elementi logici e circuiti digitali per eseguire le istruzioni definite dall'architettura.
- l'**architettura** è un livello che descrive un calcolatore dal punto di vista del programmatore. è definita attraverso una serie di istruzioni, registri che il programmatore può utilizzare per "programmare" i circuiti.

Disciplina

La disciplina è la tecnica di restringere intenzionalmente le scelte di progetto in modo da lavorare in modo più produttivo ad un livello più alto di astrazione. L'utilizzo di parti intercambiabili è un esempio di applicazione della disciplina.

Differenza tra circuiti digitali e analogici

I **circuiti digitali** utilizzano valori di tensione discreti, mentre i **circuiti analogici** usano valori continui. Quindi i circuiti digitali sono un sottoinsieme dei circuiti analogici con capacità minori. Progettare circuiti digitali però è più semplice e si possono facilmente combinare componenti digitali in un sistema complesso, il quale ha prestazioni migliori rispetto a quelli costruiti con componenti analogici.

Le tre -Y (gerarchia, modularità, regolarità)

Per gestire la complessità di un sistema vengono utilizzati tre "principi" fondamentali: la gerarchia, la modularità e la regolarità

- **gerarchia**: significa che un sistema complesso può essere suddiviso in moduli e ognuno dei moduli può essere suddiviso a sua volta ulteriormente in altri moduli finché non si arriva a blocchi costitutivi facili da comprendere.
- **modularità**: significa che tutti i moduli definiti abbiano funzionalità e interfacce ben definite così da connettersi tra di loro in maniera semplice.

- **regolarità:** significa cercare uniformità tra i moduli. Infatti i moduli più comuni vengono riutilizzati più volte riducendo il numero di moduli diversi che devono essere progettati. Implica il fatto che in un sistema è meglio avere parti intercambiabili.

Astrazione digitale

La maggior parte delle variabili fisiche è continua. I sistemi digitali invece rappresentano informazioni con variabili dal valore discreto, cioè variabili con un numero finito di valori possibili.

I calcolatori utilizzano una codifica, rappresentazione binaria cioè a due valori dal momento che è più facile distinguere tra due sole tensioni che tra dieci. La tensione maggiore indica 1 e la minore indica uno 0.

Una variabile binaria trasporta $\log_2 2 = 1$ bit di informazione.

Boole ha sviluppato una logica che opera su variabili binarie, nota come logica Booleana, nella quale sono definite le variabili booleane. Ogni variabile booleana può assumere solo uno fra i due valori 'VERO' e 'FALSO'. I calcolatori utilizzano una tensione positiva alta (VDD=1,5 V oggi) per rappresentare 1 e quindi VERO. Mentre una tensione positiva bassa zero volt per rappresentare 0 e quindi FALSO.

I componenti digitali operano in codice binario. La rappresentazione in binario non è human friendly perchè genera codici molto lunghi. Usiamo quindi il sistema esadecimale poichè 16 è una potenza della base 2. Ciò facilita la codifica e la decodifica di un numero. E' possibile codificare ogni cifra esadecimale con 4 bit, che nel sistema binario corrispondono ad una cifra hex.

Nel calcolatore le grandezze numeriche sono elaborate mediante sequenze di lunghezza fissa dette "parole" o word. Sono gruppi di bit la cui grandezza dipende dall'architettura del processore. ARM = 32 bit

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

1. RAPPRESENTAZIONE DELL'INFORMAZIONE

informazione: informazione è un qualunque insieme di segnali che condizionano l'evoluzione di un sistema. Problema: occorre definire cosa sia un segnale. restringere il contesto in cui si intende una certa nozione per darne una definizione più precisa

alfabeto: per alfabeto A intenderemo un insieme finito e distinguibile di segni che chiameremo a seconda del contesto cifre, lettere, caratteri, simboli etc.

parola/stringa di un alfabeto A: sequenza finita di simboli dell'alfabeto A

A*: indica tutte le possibili parole generabili a partire dall'alfabeto A

linguaggio L di un alfabeto A: è un qualsiasi sottoinsieme di A^*

codifica: è l'azione di associare gli elementi di un insieme D alle parole di un linguaggio L
è una funzione totale f: D-->L

Se f non è suriettiva allora diremo che è ridondante, la funzione non associa ad ogni elemento di D una parola del linguaggio L

Se f non è iniettiva allora diremo che ambigua, la funzione associa un elemento dell'insieme D a più parole del linguaggio L

Sia ND la cardinalità di D e NL la cardinalità di L

- Se $ND > NL$ qualsiasi codifica $f: D \rightarrow L$ è ambigua
- Se $ND < NL$ qualsiasi codifica $f: D \rightarrow L$ è ridondante

Quindi se f è non ambigua e non ridondante allora f è iniettiva e suriettiva, quindi è una funzione biunivoca

decodifica di D: una funzione $g: L \rightarrow D$

Proprietà delle codifiche

ambigua/non ambigua

ridondante/non ridondante

- Economicità: numero di simboli utilizzati per unità di informazione
- Semplicità nell'operazione di codifica e decodifica
- Semplicità nell'eseguire operazioni sull'informazione codificata

Rappresentazione in base generica B

- 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 $s_{m-1} \dots s_0$

$$\sum_{i=0}^{m-1} s_i \cdot b^i$$

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 un numerale decresce al crescere della base di codifica

Valore massimo rappresentabile

$$v_{max} = 10^m - 1$$

base 10

$$b^m - 1$$

qualsiasi base diversa da 10

Cifre necessarie per rappresentare n

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

$$\begin{aligned} b^m - 1 &\geq n \\ b^m &\geq n + 1 \\ m &\geq \log_b(n + 1) \end{aligned}$$

In particolare cerchiamo il più piccolo m tale che $m \geq \log_b(n + 1)$

$$m = \lceil \log_b(n + 1) \rceil$$

$$\lceil \log_b(n + 1) \rceil = \lfloor \log_b n \rfloor + 1$$

- Esercizio: dimostrare che per ogni n e b $\lceil \log_b(n+1) \rceil = \lfloor \log_b n \rfloor + 1$
- Prima osservazione:

$$\text{Se } x < y \text{ allora } \lfloor x \rfloor + 1 \leq \lceil y \rceil$$

$$\text{Quindi } \lfloor \log_b n \rfloor + 1 \leq \lceil \log_b(n+1) \rceil$$

- Supponiamo per assurdo che $\lfloor \log_b n \rfloor + 1 \leq \lceil \log_b(n+1) \rceil$
↳ chiamiamolo c

- Essendo c un intero allora c deve essere compreso fra

$$\text{Se } \log_b n < c < \log_b(n+1)$$

$$\begin{aligned} \log_b n &< c < \log_b(n+1) \\ b^{\log_b n} &< b^c < b^{\log_b(n+1)} \\ n &< b^c < n+1 \end{aligned}$$

$b \in \mathbb{N}$ $c \in \mathbb{N} \Rightarrow b^c \in \mathbb{N}$

Quindi esisterebbe un numero naturale fra n e $n+1$, assurdo!

Quindi:

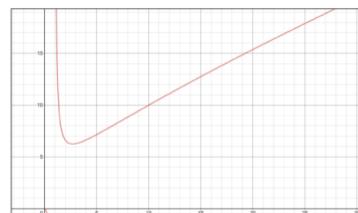
$$\lceil \log_b(n+1) \rceil = \lfloor \log_b n \rfloor + 1$$

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



Il sistema binario è il sistema di codifica meno economico poiché occorrono tante cifre binarie per rappresentare un numero. Ma bisogna dire che la codifica binaria però è quella ottimale perché un calcolatore per operare in una certa base deve poter rappresentare tutte le n cifre di quella base, ossia deve poter avere n stati differenti.

E' stato dimostrato che il valore ottimale per la codifica in un calcolatore è il numero di nepero $e=2.7$. Le codifiche che si avvicinano più a tale numero sono quelle in base 2 e base 3. Per i calcolatori si è scelta la base due perchè si codificano solo due stati. Questo è il motivo per cui i calcolatori operano in base binaria.

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.

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

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

Overflow nella somma binaria

Overflow: al termine della somma ho un riporto di 1

$$\begin{array}{r} 111 \\ 1101 \\ + 0101 \\ \hline 10010 \end{array}$$

Rappresentazione di interi

Nella rappresentazione di numeri interi mediante parole di lunghezza fissata, dobbiamo codificare non solo il valore assoluto del numero intero ma anche il suo segno. Esistono diversi sistemi che possono essere usati per rappresentare i numeri binari negativi.

Le principali rappresentazioni di numeri interi sono:

- **Rappresentazione con modulo e segno**
- **complemento a due**

Rappresentazione con segno

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.

Si utilizza il bit più significativo per esprimere il segno e rimanenti $m-1$ bit per il modulo.

Lo zero ha due possibili rappresentazioni: 0000 e 1000

Intervallo di variabilità: $\{-2^{n-1} + 1 \dots 2^{n-1} - 1\}$

Svantaggio: nelle operazioni di addizione o sottrazione occorre controllare il segno e i valori assoluti dei due operandi per determinare il segno del risultato.

Complemento a 2

In questa rappresentazione il bit più significativo vale **-2^{n-1}**

Lo zero ha un'unica rappresentazione

Calcolo del complemento a 2: consiste nell'invertire tutti i bit all'interno del numero e poi aggiungere 1

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

Il numero più negativo $-2^N - 1$ è anomalo perché il suo complemento a 2 è ancora il numero stesso. Questo numero non ha una controparte positiva.

Overflow nel complemento a 2

Sommare un numero negativo e uno positivo non genera overflow

l'overflow non avviene come per gli unsigned quando ho riporto finale di 1

Nella somma a N bit si può verificare un overflow se il risultato è maggiore di $v_{max} 2^N - 1$ o minore di $v_{min} -2^N - 1$

In particolare l'overflow si verifica quando i due numeri che vengono sommati hanno lo stesso bit di segno e il risultato ha un bit di segno opposto

entrambi gli operandi sono negativi (primo bit=1) o entrambi positivi (primo bit=0) è il risultato ha segno opposto

Estensione del segno: Per estendere un numero in una rappresentazione con più bit basta riprodurre a sinistra il bit più significativo (quello di segno)

Binary Coded Decimal

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.

Poiché le combinazioni da 10 a 15 non si usano, la codifica BCD è ridondante

I numeri vengono rappresentati con le singole cifre a 4 bit, cifra per cifra.

$23 = 0010\ 0011$

i codici 1010, 1011, 1100, 1101, 1110 ,1111 non sono utilizzati (codifica ridondante)

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)

Rappresentazione di Numeri con virgola

Virgola fissa

Nella rappresentazione in virgola fissa si suddividono gli n bit del numero in due sottoparole

- i primi h bit sono dedicati alla codifica della parte intera del numero
- i rimanenti k bit sono dedicati alla codifica della parte frazionaria del numero

Rappresentazione in virgola mobile

In generale, fissare a priori la lunghezza della parte intera h e di quella frazionaria k costituisce una scelta rigida. Alcune applicazioni scientifiche operano con valori ancora più piccoli, altre invece con valori molto grandi

Per ovviare a queste difficoltà è stata introdotta una rappresentazione detta in virgola mobile I numeri in virgola mobile sono equivalenti alla notazione scientifica.

Superano la limitazione di avere un numero costante di bit interi e frazionari, permettendo quindi la rappresentazione di numeri molto piccoli ma anche molto grandi.

La differenza fra virgola fissa e virgola mobile è che se analizziamo la distribuzione dei valori rappresentabili con n bit lungo una retta:

- **in virgola fissa** abbiamo un passo piccolo ma costante e abbiamo un range fissato di rappresentabilità, il che implica la perdita di precisione per numeri molto piccoli o grandi
- **in virgola mobile** il passo si infittisce vicino allo 0 per rappresentare anche numeri molto piccoli, per poi dilatarsi man mano. La precisione vicino allo 0 è maggiore di quella in v. fissa, ma man mano che ci si allontana dallo 0 abbiamo una precisione peggiore di quella in v. fissa perchè il passo si allunga sempre di più

Se abbiamo 32 bit potremmo rappresentare 2^{32} combinazioni di numeri sia in v.fissa che in v.mobile , la differenza sta nella precisione del passo.

I numeri in virgola mobile hanno

- un segno 0 positivo, 1 negativo**
- una mantissa m**
- una base b**
- un esponente e**

Un numero reale è quindi rappresentato da una tripla (s,m,e)

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)

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

Per la codifica dell'esponente si utilizza la “rappresentazione polarizzata” (e-P) $e=e'+P$

Costante di polarizzazione:

$$P = 2^{k-1} - 1$$

- IEEE 754 precisione singola: 8 bit per l'esponente
 - $P = 2^7 - 1 = 127$
- IEEE 754 precisione doppia: 11 bit per l'esponente
 - $P = 2^{10} - 1 = 1023$

Avendo 8 bit a disposizione per l'esponente, $P = 2^8 - 1 = 2^7 - 1 = 127$

Per la mantissa si adotta la rappresentazione scientifica 1,xyz...

e' = si ottiene spostando la virgola vicino al numero più significativo (sempre 1)

Casi speciali IEEE

Lo standard IEEE 754 di rappresentazione dei numeri in virgola mobile prevede codici speciali per rappresentare numeri come lo 0, l'infinito, e i valori impossibili. Per esempio risulta problematica la rappresentazione dello 0 in virgola mobile a causa dell'uno più significativo隐含的. Vengono usati per questi casi particolari esponenti di tutti 0 o tutti 1.

Numero 0

si hanno 2 rappresentazioni, a seconda che il segno sia 1 (negativo) o 0 (positivo)

s = X

e = 00000000 (0)

m = 00000000000000000000000000 (0)

+Infinito

s = 0

e = 11111111 (255)

m = 00000000000000000000000000 (0)

-Infinito

s = 1

e = 11111111 (255)

m = 00000000000000000000000000 (0)

Numeri molto vicini allo 0

s = X

e = 00000000 (0)

m = diversa da 0

NaN not a number

s = X

esponente = 11111111 (255)

m = diversa da 0

Codifica caratteri alfa-numerici

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 (£, &, @, ...)

- Un testo è una sequenza di caratteri. I codici associano un numero intero ad ogni carattere.

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.

2. ALGEBRA DI BOOLE E RETI COMBINATORIE

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

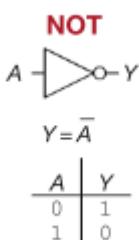
Le **porte logiche** sono semplici circuiti digitali che prendono uno o più input e producono un output, usate per realizzare delle semplici operazioni

La relazione tra ingressi e uscita può essere descritta con una tabella di verità oppure con un'espressione booleana.

Una **tabella di verità** ha come campi gli ingressi in input e l'uscita e ha una riga per ogni possibile combinazione dei valori di ingresso

Un **espressione booleana** è un'espressione matematica che utilizza variabili binarie e operatori dell'algebra di Boole

Porta NOT



La porta NOT restituisce in output il complemento dell'input

Una porta NOT ha un ingresso, A, e un'uscita, Y

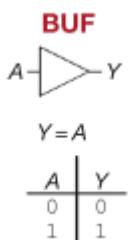
L'uscita della porta NOT è l'esatto contrario del suo ingresso:

se A è FALSO allora Y è VERO

se A è VERO allora Y è FALSO

La porta NOT (negatore) viene anche chiamata porta invertente (**inverter**).

Buffer



L'altra porta logica a un solo ingresso viene chiamata buffer.

Questa porta logica riproduce semplicemente il valore di ingresso in uscita

Porta AND



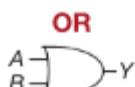
$$Y = AB$$

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

La porta AND restituisce in output la **congiunzione** degli input genera all'uscita Y il valore VERO se e soltanto se sia A sia B hanno valore VERO, altrimenti l'uscita vale FALSO.

Y è uguale a 1 se e solo se A e B sono entrambi uguali a 1

Porta OR



$$Y = A + B$$

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

La porta OR restituisce in output la **disgiunzione** degli input produce all'uscita Y il valore VERO se A, oppure B, oppure entrambi gli ingressi hanno valore VERO.

Y è uguale a 1 se e solo se A o B è uguale a 1 (oppure A e B sono uguali a 1)

Altre porte logiche a due ingressi

XOR



$$Y = A \oplus B$$

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

La porta XOR (OR esclusivo, detto "ex-OR") dà uscita VERO se A oppure B, ma non entrambi, hanno valore VERO.

Y = 1 se e solo se A oppure B è uguale a 1

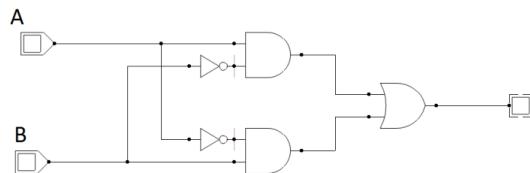
Una porta XOR a N ingressi viene anche chiamata "**porta di parità**" perché produce in uscita VERO se un numero dispari di ingressi è VERO.

la porta XOR può essere ottenuta mediante porte AND, OR e NOT

XOR: $Y = 1$ sse $A \neq B$

$A \neq B$ sse $(A=1 \text{ e } B=0) \circ (\underbrace{A=0 \text{ e } B=1})$

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



NAND

NAND



$$Y = \overline{AB}$$

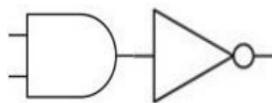
A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

La porta NAND esegue le operazioni NOT e AND.

La sua uscita ha sempre valore VERO tranne quando entrambi gli ingressi sono VERO

Y è uguale a 1 se e solo se A o B non sono uguali a 1

La porta NAND può essere ottenuta complementando una porta AND, ovvero mettendo in serie una porta AND e una NOT



NOR

NOR



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

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

La porta NOR esegue le operazioni NOT e OR.

La sua uscita è VERO se né A né B sono VERO

Y è uguale a 1 se e solo se A e B non sono uguali a 1

La porta NOR si può ottenere mettendo in serie una porta OR e una NOT



XNOR

porta XNOR esegue l'operazione negata rispetto a una porta XOR.

XNOR		A	B	Y
A	-	0	0	1
B	-	0	1	0
		1	0	0
		1	1	1

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

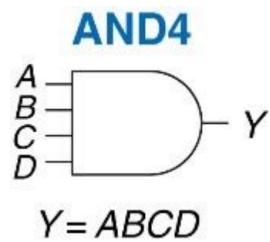
Porte logiche con più linee di input

Le porte logiche AND, OR, NAND,... possono avere anche più di 2 linee di ingresso in input. Una porta AND con un numero N di ingressi produce un valore di uscita VERO quando tutti i valori di ingresso sono VERO.

Invece una porta OR con un numero N di ingressi produce un valore di uscita VERO quando almeno uno dei suoi ingressi è VERO.

AND4

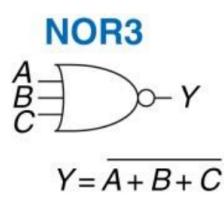
L'uscita vale VERO solo se tutti gli ingressi sono VERO



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

NOR3

L'uscita vale VERO se e solo se nessuno degli ingressi è VERO



A	B	C	Y
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

Porte logiche dal punto di vista fisico

Per realizzare le porte logiche in concreto i valori in ingresso binari devono corrispondere ad una certa grandezza fisica

La grandezza fisica che reifica i valori 0 e 1 è il potenziale elettrico che genera corrente all'interno di un circuito

Il valore logico 0 è rappresentato dal valore di potenziale di 0V (**GRD**)

Il valore logico 1 è rappresentato da un valore di potenziale **VDD** fornito dal generatore.

$VDD \leq 1,5V$ ad oggi

Il **potenziale elettrico** è una grandezza fisica continua con cui rappresentiamo dei valori discreti (0 e 1) mediante le grandezze nominali GRD e VDD

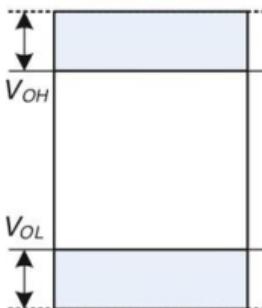
Come ogni sistema fisico reale, un circuito digitale è soggetto a del **rumore** che può alterare i valori nominali su cui operiamo.

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 e andare in zone “proibite” dove si hanno dei valori illegali.

Quindi abbiamo un intervallo di potenziale per cui se stiamo nei termini di tolleranza lo consideriamo GRD o VDD altrimenti se eccediamo questi termini di tolleranza VOL e VOH avremo un rumore così forte che non permette al circuito di operare in un regime digitale e quindi potrebbe malfunzionare o bruciarsi



Funzioni booleane

Le porte logiche esaminate finora costituiscono delle specifiche funzioni booleane

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

Quante funzioni booleane di N variabili esistono?

Data una parola di lunghezza N, abbiamo 2^n combinazioni possibili.

Per ogni combinazione di variabili una generica funzione f booleana può assegnare liberamente due valori 0 o 1 alla combinazione.

Quindi se la parola in ingresso è lunga N, avremo 2^n combinazioni possibili, alle quali si può associare 0 o 1.

Quindi il numero di funzioni booleane di N variabili è pari al numero di parole binarie di lunghezza 2^N , ovvero:

$$2^{2^N}$$

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

Diagramma che associa le 16 funzioni booleane di due variabili a espressioni logiche:

- $y=0$ AND (funtione costante)
- $y=A$
- $y=B$
- $y=\overline{AB}$
- $y=\overline{A}\overline{B}$
- $y=A\overline{B}$
- $y=\overline{A}B$
- $y=XOR$
- $y=OR$
- $y=NOR$
- $y=\overline{XOR}$
- $y=\overline{A}$
- $y=\overline{B}$
- $y=NAND$
- $y=1$ (funtione costante)

Nota: $y = \overline{\overline{A}B} = A + \overline{B}$ ($B \leq A$)

Due funzioni booleane per esprimere queste funzioni (anche negazione di f):

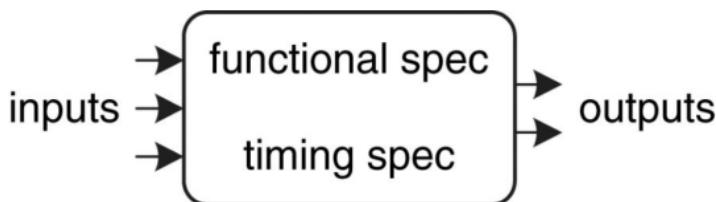
$y = \overline{AB} = \overline{A} + B$ ($A \leq B$)

Circuiti digitali

Un **circuito digitale** è una rete che elabora segnali discreti (rappresentati da variabili booleane).

Un circuito digitale può essere visto come una **black-box** che contiene:

- **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.

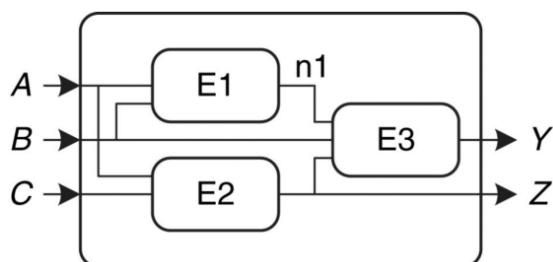


Un circuito al suo interno è composta da **elementi e nodi**

elemento: è esso stesso un circuito digitale

nodo: è una connessione che trasporta il segnale, può essere di input, output o interno

- **nodo di input**: riceve il segnale dal mondo esterno
- **nodo di 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**.

rete combinatoria: i valori degli output dipendono esclusivamente dal valore corrente degli input. Si dicono **memoryless**, ovvero non hanno memoria della “storia” precedente del circuito

rete sequenziale: gli output dipendono non solo dal valore corrente degli input, ma anche dai valori precedenti. Si dice quindi che il circuito ha memoria.

Regole di composizione di reti combinatorie

- 1) Ogni elemento circuitale è esso stesso una rete combinatoria
- 2) Ogni nodo della rete che non sia di input è connesso ad un unico output di un altro elemento della rete

- 3) la rete non contiene percorsi ciclici: ogni cammino interno alla rete visita un nodo al più una volta

Visualizzare una rete come una scatola nera con un'interfaccia e una funzione ben definite è un'applicazione del principio dell'astrazione e della modularità, così come la costruzione di una rete a partire da elementi circuitali più piccoli è un'applicazione del principio della gerarchia. Le regole della composizione combinatoria sono, infine, un'applicazione della regolarità.

La specifica funzionale di un circuito combinatorio è solitamente espressa come una tabella di verità o come un'espressione booleana.

In generale si può ricavare un'espressione booleana data una tabella di verità oppure si può ricavare una tabella di verità data una espressione booleana

Ricavare una tabella di verità data un'espressione booleana

Le espressioni booleane si basano su variabili che possono assumere i due valori VERO o FALSO

Data un'espressione booleana è possibile ricavare la tavola di verità calcolando a ritroso le tabelle di verità delle sotto espressioni dell'espressione booleana principale, partendo dalle singole variabili booleane.

Terminologia

litterale: una variabile booleana A o la sua negata A^*

implicante: è un prodotto di litterali

mintermine: Dato un insieme K di variabili booleane, un mintermine di K è un **implicante** che comprende tutte le variabili in K, siano esse positive o negative.

Quindi il mintermine è il prodotto di tutti gli ingressi di una funzione

maxtermine: Dato un insieme K di variabili booleane, un maxtermine di K è una **somma di litterali** in cui occorrono tutte le variabili in K.

Quindi il maxtermine è la somma di tutti gli ingressi di una funzione.

Forma SOP (Sum of product)

1=variabile in forma dritta

0=variabile in forma negata

Ogni riga di una tabella delle verità è associata a un **mintermine** che è VERO per quella riga.

Si scrive un'espressione booleana a partire dalla tabella di verità tramite la somma di tutti i mintermini in corrispondenza dei quali l'uscita della tabella, Y, vale VERO= 1

Forma POS (Product of sum)

E' una forma duale per rappresentare una funzione booleana

1=variabile in forma negata

0=variabile in forma dritta

Ad ogni riga di una tabella di verità corrisponde un **maxtermine** che è FALSO solo per quella riga

Si scrive un'espressione booleana a partire dalla tabella di verità tramite il prodotto di tutti i maxtermini in corrispondenza dei quali l'uscita della tabella, Y, vale FALSO=0.

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	$\bar{O} + \bar{C}$

$$E = (O + C)(O + \bar{C})(\bar{O} + C)$$

$$= \Pi(0, 1, 3)$$

Forma SOP o 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

Un'espressione booleana ricavata a partire da una tabella delle verità tramite forma POS o SOP non sempre corrisponde all'insieme minimo di porte logiche necessario per realizzare la funzione considerata. Una funzione booleana può essere semplificata per ricavare da essa un'espressione minimizzata più succinta.

Proprio come si utilizza l'algebra per semplificare le espressioni matematiche, è possibile utilizzare l'algebra booleana per semplificare le espressioni booleane

L'algebra booleana si basa su un insieme di **assiomi** che come tali vengono per definizione considerati corretti. A partire da essi è possibile invece dimostrare tutti i teoremi dell'algebra booleana.

I postulati e i teoremi dell'algebra booleana obbediscono al **principio di dualità**:
se i simboli 0 e 1 e gli operatori • (AND) e + (OR) sono scambiati tra loro,
l'affermazione rimane corretta.

Assiomi dell'algebra di Boole

Axiom		Dual		Name
A1	$B = 0$ if $B \neq 1$	A1'	$B = 1$ 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

A2 e A2' definiscono l'operatore NOT (di fatto questi assiomi ripropongono la tabella di verità dell'operatore)

A3, A4 e A5 definiscono l'operatore AND

A3', A4' e A5' definiscono l'operatore OR

Teoremi ad una variabile

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

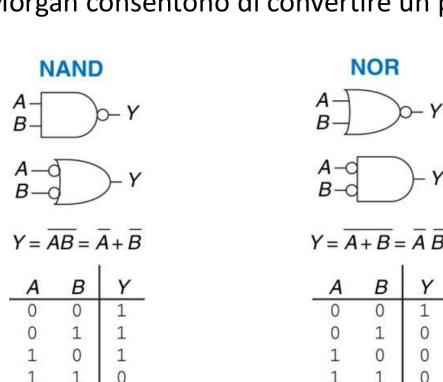
Teoremi più variabili: dualità

#	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

Teoremi di De Morgan

#	Theorem	Dual	Name
T12	$\overline{B_0 \bullet B_1 \bullet B_2 \dots} = \overline{B_0 + B_1 + B_2 \dots}$	$\overline{B_0 + B_1 + B_2 \dots} = \overline{B_0 \bullet B_1 \bullet B_2 \dots}$	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
- I teoremi di De Morgan consentono di convertire un porta AND in una OR e viceversa



Secondo il teorema di De Morgan, una porta NAND è equivalente a una porta OR con gli ingressi negati. Allo stesso modo, una porta NOR è uguale a una porta AND con gli ingressi negati.

Il principio di dualità è una conseguenza dei teoremi di De Morgan

Tecniche di dimostrazione per dimostrare l'equivalenza di due espressioni

Perfect induction: si calcolano le rispettive tabelle di verità delle due espressioni e se coincidono (negli output) allora le due espressioni sono equivalenti

Limiti: Al crescere della lunghezza delle espressioni diventa sempre più laboriosa

Proof teoretico: si usano assiomi e teoremi precedentemente provati per manipolare le espressioni fino ad ottenere espressioni uguali

Per casi rilevanti: si sfrutta la struttura delle espressioni e si verifica che le tabelle di verità coincidono solo nei casi rilevanti

Teoremi di De Morgan e forme SOP/POS

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

- Si applica «De Morgan» per negare la struttura della formula
- Applica esaustivamente la proprietà distributiva dell'AND sull'OR (se si vuole SOP)
- Applica esaustivamente la proprietà distributiva dell'OR sull'AND (se si vuole 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	$A B$

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

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

Completezza degli operatori

La forma SOP usa solo gli operatori NOT, AND, OR per rappresentare qualsiasi espressione booleana

Questo vuol dire che questo insieme di operatori costituisce un insieme completo, per il fatto che si possono ricavare tutti gli altri operatori a partire da questi tre.

Ma 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 {NOT, AND, OR} è un insieme completo ma non minimale essendo {AND, NOT} e {OR, NOT} anche loro completi, ma minimali

Semplificare le espressioni booleane, semplificazione di forma SOP, SOP minima

Avere una forma più succinta di un'espressione booleana significa anche avere un circuito combinatorio con meno porte logiche e quindi più economico e performante.

L'idea è costruire un circuito combinatorio quanto più ottimizzato possibile per la data funzione booleana da calcolare

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 $P = \overline{P}\overline{A} + P\overline{A}$
 $A = A + AP$
- Duplication $A = A + A$
- "Simplification" theorem $\overline{PA} + A = P + A$
 $PA + \overline{A} = P + \overline{A}$

I teoremi dell'algebra booleana sono utili per semplificare le espressioni booleane.

Il principio base per semplificare equazioni in forma SOP è combinare i termini utilizzando la relazione $PA + PA^* = P$, dove P rappresenta un implicante qualsiasi.

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

La proprietà principale che si usa è il combining $(B+C) \bullet (B+\overline{C}) = B$

Quanto si può semplificare un'espressione?

Si definisce un'espressione SOP come **minima** se utilizza il **minor numero possibile di implicanti, quindi se tutti gli implicanti sono implicanti primi**.

Se si confrontano espressioni con lo stesso numero di implicanti, l'espressione minima è quella che usa il **minor numero possibile di letterali**.

Un implicante è detto **implicante primo** se non può essere combinato con nessun altro elemento all'interno dell'espressione per formare un nuovo implicante con un numero minore di letterali.

Schemi circuitali, logica a due e più livelli

- Ad ogni espressione booleana corrisponde in circuito combinatorio, rappresentato da uno schema circuitale

Schemi circuitali SOP: le espressioni booleane in forma SOP hanno degli schemi circuitali molto regolari

- abbiamo una linea di input per ogni variabile che occorre positiva
- abbiamo linee con una porta NOT per ogni variabile che occorre negata
- ogni **mintermine** si costruisce con una porta **AND** che ha in ingresso le linee che corrispondono ai relativi litterali che lo compongono
- tutte le uscite delle porte AND che corrispondono ai mintermine vengono collegate in un unico **OR**, che rappresenta la **somma dei mintermini**.

Per questo la logica in forma SOP viene chiamata logica a due livelli, perché consiste di letterali connessi a un primo livello di porte AND che, a sua volta, sono connesse a un secondo livello di porte OR

Logiche a più livelli: Spesso i progettisti costruiscono reti con più di due livelli di porte logiche, perché può accadere che queste reti combinatorie a più livelli richiedano di utilizzare meno hardware (porte logiche) rispetto alle loro controparti a due livelli. Vedi esempio XOR3

La scelta della logica di realizzazione circuitale dipende da diversi fattori, fra cui la tecnologia di costruzione del circuito utilizzata.

Vantaggi forme SOP/POS logiche a due livelli

La logica a 2 livelli della forma SOP presenta dei vantaggi, ad esempio, nei tempi di propagazione che sono molto stabili e veloci al mutare di qualsiasi valore delle variabili booleane

Logiche non a due livelli possono avere tempi di propagazione più lunghi dovuto al delay delle diverse porte che si accumula di volta in volta, possiamo trovarci in situazioni in cui si verifica un critical path e si deve passare fra tante porte logiche per ottenere il risultato e quindi i tempi di propagazione si allungano.

Limiti SOP

Alcune funzioni booleane, poste in forma SOP, sono estremamente poco succinte e quindi richiedono un numero considerevole di porte per realizzare il circuito

Es. XOR a più variabili

- XOR3 in forma SOP

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

la funzione può essere realizzata + semplicemente con una cascata di XOR a due ingressi, logica multilivello

Circuito a priorità

I circuiti a priorità vengono utilizzati per assegnare una risorsa condivisa secondo un ordine di priorità fra chi ne fa richiesta

Ad esempio posso avere 4 possibili richiedenti con priorità diverse

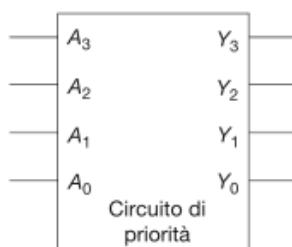
A3 ha la priorità più alta, A2, A1, A0 che ha la priorità più bassa.

- Gli input A0 ,..., A3 rappresentano le richieste della risorsa

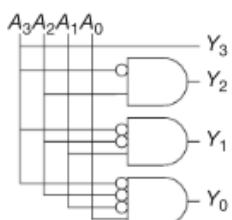
- Gli output Y0 ,..., Y3 rappresentano a chi viene assegnata la risorsa, di volta in volta uno solo di essi sarà uguale a 1, ossia quello che ha la priorità più alta.

Se per esempio A3 e A2 richiedono una risorsa, questa viene assegnata ad A3 perchè ha priorità più alta.

Ad A0 che ha la priorità più bassa ad esempio la risorsa viene assegnata solo se gli altri richiedenti non ne hanno bisogno, quindi se A3,A2,A1=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	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

Figura 2.20

Valori don't cares

Quando l'ingresso A3 è portato a uno nel circuito, le uscite non tengono conto dei valori presenti agli altri ingressi. Tali ingressi, ignorati dalle uscite, vengono contrassegnati con il simbolo X, e sono chiamati **valori don't cares, non conta il loro valore al fine di calcolare l'uscita della funzione**. La tabella di verità si riduce sensibilmente quando si inseriscono i

valori don't cares.. A partire da questa tabella delle verità è più semplice leggere l'espressione booleana in forma somma di prodotti ignorando gli ingressi con una X.

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)

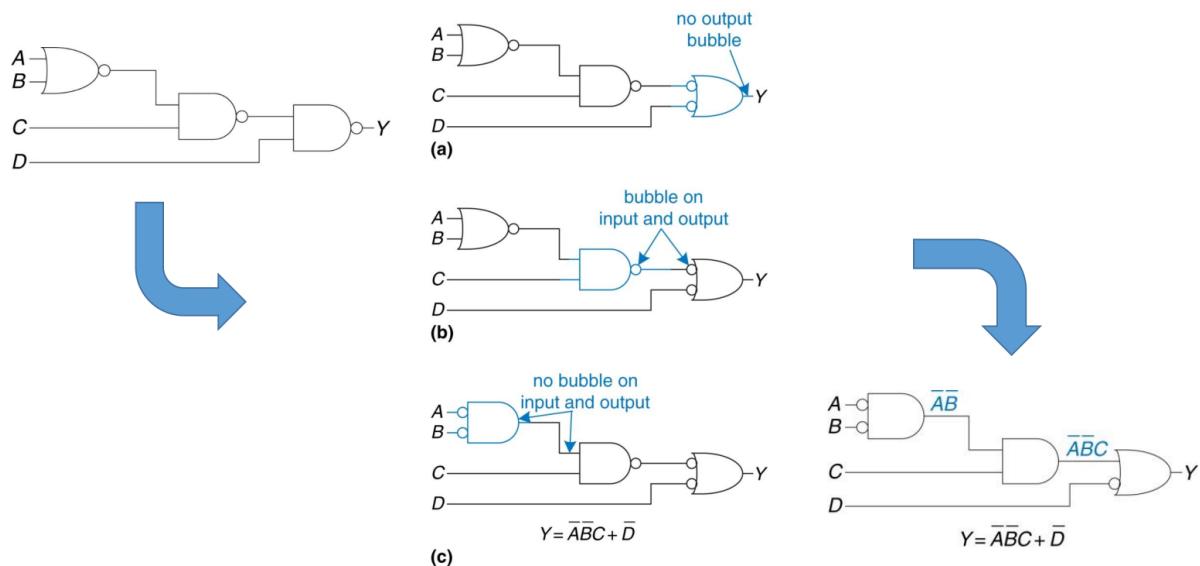
Bubble pushing

Usare logiche multilivello e porte NAND/NOR a volte rende difficile capire quale funzione booleana un circuito realizza a causa delle molte negazioni annidate

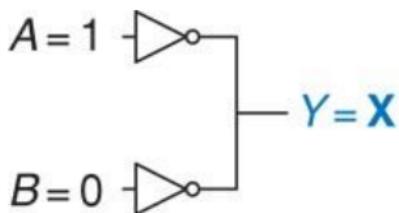
Per avere un'espressione un po' più leggibile si possono applicare le leggi di De Morgan

Il bubble pushing è una tecnica grafica che applica le leggi di De Morgan e la legge della doppia negazione per eliminare le negazioni annidate nei circuiti con tante porte NAND e NOR, per semplificarli e renderli più leggibili

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



Valori illegali



In questo circuito il valore del potenziale è portato ad essere contemporaneamente sia 0 che VDD. Questo, generalmente, accade se al nodo vengono applicati entrambi i valori 0 e 1 allo stesso tempo. **Questo tipo di configurazioni vengono dette illegali**

La tensione elettrica effettiva di un nodo è un valore indefinito compreso tra 0 e VDD e per questo 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 nel quale non c'è passaggio di corrente**

Gli stati di alta impedenza vengono utilizzati per disconnettere una parte di un circuito dal resto. Un circuito che realizza tale funzione è il **tristate buffer**

Tristate buffer

Il buffer tristate, possiede tre possibili stati d'uscita:

ALTO (1), BASSO (0) e alta impedenza (Z).

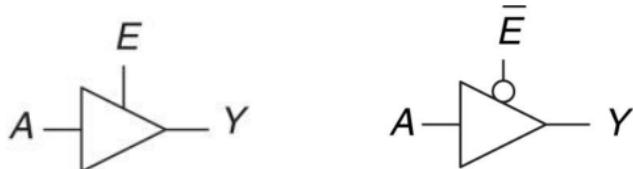
Ha un ingresso A, un'uscita Y, e un'abilitazione **E (Enable)**.

Quando il segnale di abilitazione **E=1**, il buffer lavora come un **buffer normale** e trasferisce il valore dell'ingresso all'uscita.

Quando invece il segnale di abilitazione **E=0**, l'uscita assume il valore di **alta impedenza (Z)**

il segnale di abilitazione E può essere attivo alto oppure attivo basso E*, se è attivo basso quando E=1 avremo lo stato di alta impedenza

Tristate Buffer



E	A	Y	\bar{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

I buffer tristate vengono solitamente utilizzati sui bus che connettono più chip tra loro.

Per esempio, un microprocessore, un controllore video o un controllore Ethernet possono tutti aver bisogno di comunicare con la memoria, a cui sono collegati col medesimo bus.

In questi casi, per evitare stati illegali, solo un componente alla volta può «immettere» segnali sul bus per comunicare con la memoria.

Per questo ogni chip viene collegato al bus di memoria condiviso tramite un buffer tristate. Quando un componente comunica con la memoria gli altri componenti i devono essere temporaneamente disconnessi tramite lo stato di alta impedenza del tristate buffer

Mappe di Karnaugh

Le mappe di Karnaugh (K-map) sono un metodo grafico di semplificazione di espressioni booleane in forma SOP. Esse consentono di rilevare più facilmente implicati che possono essere semplificati

- Quindi alla base delle mappe di Karnaugh c'è il solito principio:

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

Ogni quadrato della mappa di Karnaugh corrisponde a una riga della tabella delle verità e contiene il valore, corrispondente a quella riga, dell'uscita Y

Nella mappa si inseriscono tutte le combinazioni possibili delle variabili booleane presenti nelle espressioni, ogni riquadro della mappa differisce per un singolo cambiamento di una variabile booleana

le combinazioni delle variabili compaiono in un ordine particolare: 00, 01, 11, 10. Questo ordine è chiamato **codice Gray**. È utile proprio perché gli elementi adiacenti differiscono per una sola variabile. Scrivere le combinazioni seguendo l'ordine binario ordinario non avrebbe prodotto come risultato la proprietà utile delle mappe di Karnaugh dei riquadri adiacenti con un'unica variabile diversa.

Le mappe di Karnaugh ci permettono di eseguire la semplificazione graficamente, cerchiando gli 1 nei riquadri adiacenti. Per ogni cerchio si deve poi scrivere l'implicante corrispondente. Bisogna utilizzare il minor numero possibile di cerchi e includendo in ogni cerchio il maggior numero possibile di riquadri

Regole di minimizzazione con K-maps

- Usare il minimo numero di cerchi/bolle necessari per ricoprire tutti gli 1
- Tutte le caselle in una bolla contenere un 1
- Ogni cerchio deve ricoprire un blocco di caselle che è una potenza di 2: 1, 2, 4, ...
- Ogni cerchio deve essere il più largo possibile
- Un cerchio può estendersi oltre i bordi e comprendere le estremità della K-map (struttura toroidale)
- Una casella può essere ricoperta da più cerchi se questo permette un numero inferiore di cerchi

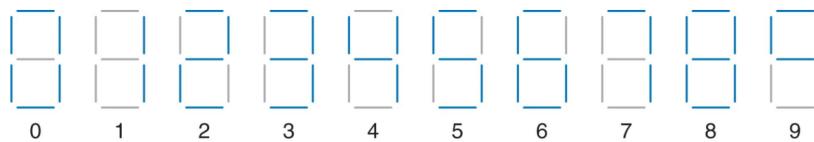
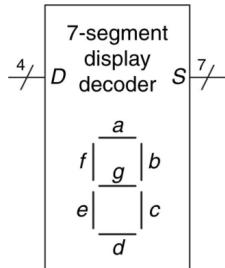
Don't care e K-map

In un circuito valori «don't care» (X) si possono avere anche in output allorché per una data configurazione degli input il valore dell'output è irrilevante

In particolare in una K-map sostituiremo i valori X don't care, se li abbiamo, con degli 1 se questo consente di avere un numero inferiore di cerchi o cerchi più larghi

Display a 7 segmenti

Un display a sette segmenti riceve un dato di ingresso a 4 bit D3:0 e produce sette uscite per controllare 7 diodi emettitori di luce (LED, Light Emitting Diode) che visualizzano una cifra da 0 a 9. Le sette uscite sono spesso chiamate segmenti da a a g, o Sa–Sg



$D_{3:0}$	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

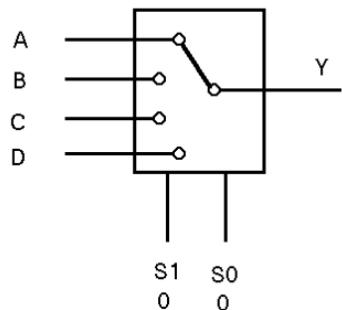
nel display a sette segmenti per gli input «illegali» 10-15 l'output può essere di tipo X, don't care

BLOCCHI COSTITUTIVI COMBINATORI

La logica combinatoria viene spesso raggruppata in blocchi costitutivi più ampi per costruire sistemi più complessi. Questa è chiaramente un'applicazione del principio dell'astrazione, che nasconde i dettagli non necessari a livello delle porte logiche per enfatizzare la funzione del blocco costitutivo.

Multiplexer

Un multiplexer è essenzialmente un selettore di linea, ossia un circuito che è in grado di selezionare un'uscita a partire da un certo numero di ingressi possibili, basandosi sul valore di un segnale di selezione. I multiplexer sono anche chiamati mux.



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

Multiplexer 2:1

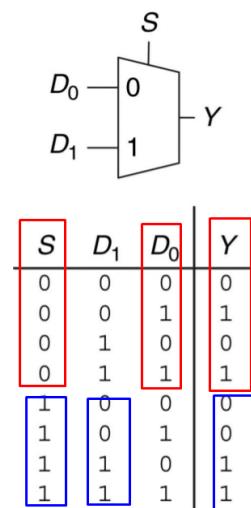
Un mux 2:1 presenta due ingressi D0 e D1, un ingresso per il segnale di abilitazione S e un'uscita Y.

Il multiplexer sceglie tra i due ingressi a seconda del valore assunto da S:

-se $S = 0$, $Y = D0$

-se $S = 1$, $Y = D1$.

S viene anche chiamato **segnale di controllo o di abilitazione** proprio perché controlla la scelta del multiplexer.

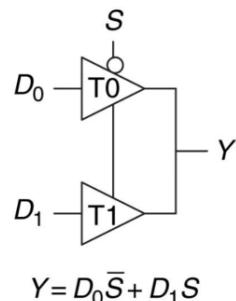
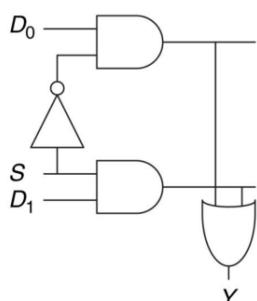


Un multiplexer 2:1 può essere costruito a partire dalla logica SOP costruendo il circuito sulla base dell'espressione SOP minimizzata.

In alternativa, i multiplexer possono essere realizzati con **buffer tristate**

S	00	01	11	10
0	0	1	1	0
1	0	0	1	1

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



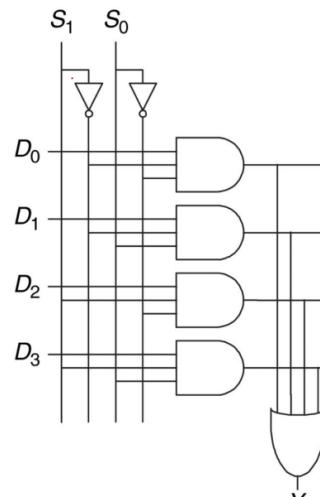
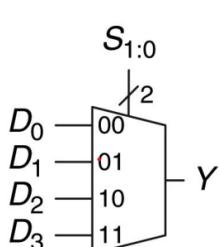
$$Y = D_0 \bar{S} + D_1 S$$

Le abilitazioni dei buffer sono organizzate in maniera tale che, in ogni momento, solo un buffer tristate è attivo. Quando $S = 0$, viene attivato il tristate T_0 , permettendo così a D_0 di passare in Y ; quando invece $S = 1$, viene attivato il tristate T_1 , che permette a D_1 di passare in Y .

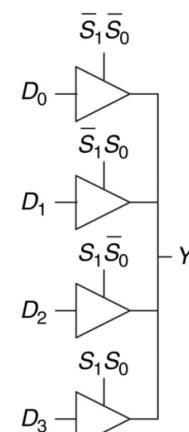
Multiplexer 4:1

Un multiplexer 4:1 possiede quattro ingressi e un'uscita. In questo caso sono necessari due segnali di selezione per scegliere tra i quattro ingressi in input.

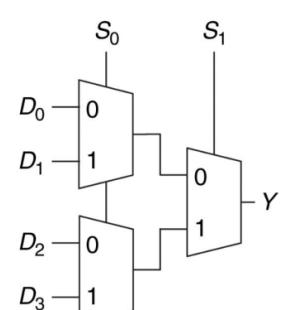
Il multiplexer 4:1 può essere costruito utilizzando la logica sop, i tristate, o alcuni multiplexer 2:1 in cascata



(a)



(b)

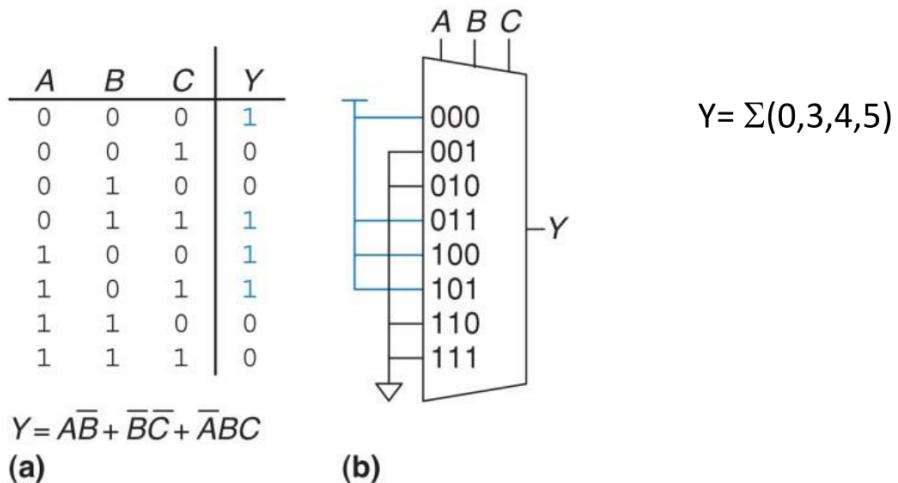


(c)

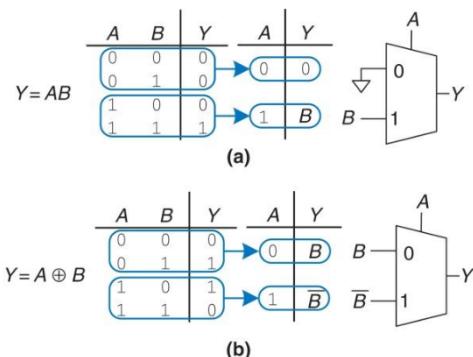
Sintetizzare funzioni booleane con mux

I multiplexer possono essere usati anche per sintetizzare delle funzioni booleane

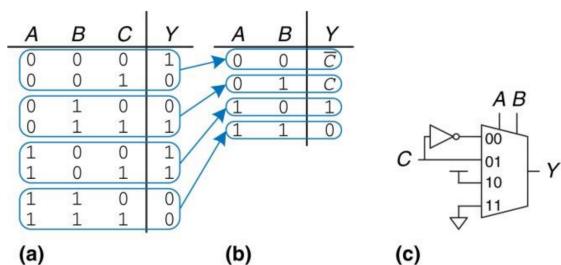
Sintetizzare una funzione di m variabili con un mux a 2^m linee è molto semplice: le variabili saranno linee di selezione. Data una certa configurazione delle variabili, la linea di ingresso corrispondente sarà posta al valore della funzione in quella configurazione, ossia 1(VDD) o GRD (0). Di fatto le linee di ingresso riproducono la tabella di verità della funzione



- 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)



In ingresso possiamo avere anche delle variabili per sintetizzare la funzione booleana e per avere una tavola di verità più succinta

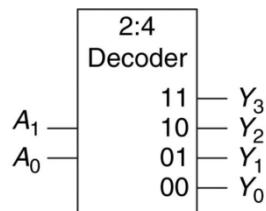


Decoder

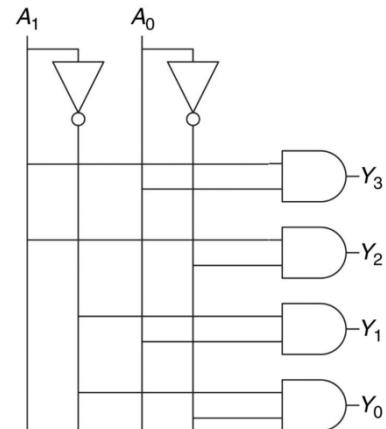
Un decoder ha N linee di ingresso e 2^N linee di uscita e attiva una delle sue uscite a seconda della combinazione dei valori in ingresso.

se n il è numero rappresentato dagli input allora solo l'n-esima linea di uscita è pari a 1 mentre tutte le altre sono a 0

Le uscite sono dette one hot proprio perché solo un'uscita alta in ogni momento

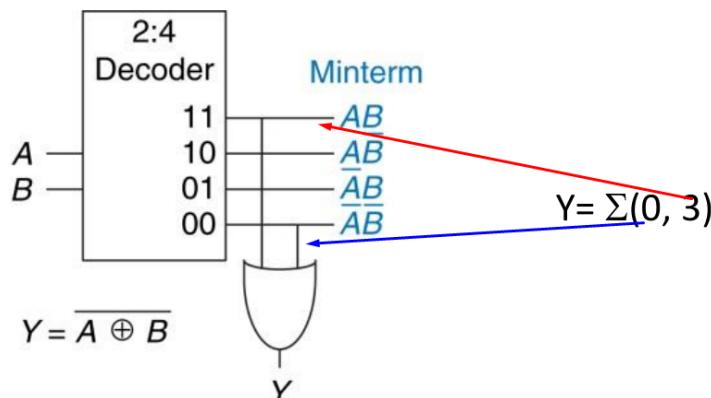


A ₁	A ₀	Y ₃	Y ₂	Y ₁	Y ₀
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0



Anche i decoder possono essere usati per sintetizzare funzioni booleane

Basta mettere in OR tutte le linee di uscita che occorrono nell'espressione della funzione da sintetizzare



3. CIRCUITI SEQUENZIALI

Nei circuiti realizzati con **logica combinatoria** l'output del circuito dipende esclusivamente dai valori presenti in quel momento agli ingressi.

Nei circuiti realizzati con **logica sequenziale** l'output del circuito dipende sia dal valore corrente sia da valori precedenti dell'input. **In tal senso si dice che il sistema ha memoria**

I circuiti sequenziali sono caratterizzati da uno **stato interno** il quale rappresenta l'informazione che mantiene la **storia** di un circuito sequenziale.

La storia del circuito è necessaria per prevedere il suo comportamento futuro.

Lo **stato di un sistema** viene memorizzato in componenti detti "**state elements**" come i **latches e flip-flop**

Com'è realizzato un circuito sequenziale?

Un circuito sequenziale (**sincrono**) è composto da:

- una **logica combinatoria** che definisce l'**evoluzione del sistema**
- **banchi di flip-flop** che servono a memorizzare gli **stati del sistema**

Un aspetto peculiare dei sistemi sequenziali è quello della retroazione (feedback), ovvero i segnali di output vengono riportati in input

State elements

Nella logica sequenziale lo stato di un circuito influisce sull'evoluzione del sistema.

Gli **state elements** sono tutte quelle componenti circuituali fondamentali che vengono adoperate per memorizzare lo stato di un circuito.

- **Circuiti bistabili**
- **SR Latch**
- **D Latch**
- **D Flip-flop**

Circuito bistabile

E' un blocco costitutivo fondamentale di memoria, fa da building block per altri state elements.

E' un circuito che ha due stati stabili ed è composto da una **coppia di inverter (porte NOT)** **retroazionati** collegati ad **anello in modo sequenziale** oppure **in croce**.

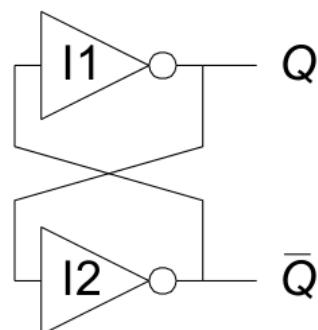
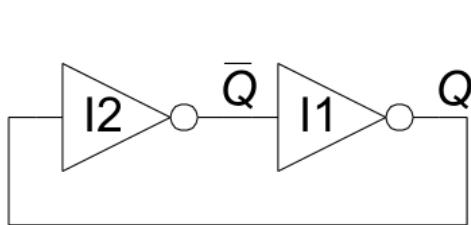
L'ingresso del primo inverter I1 è l'uscita del secondo inverter I2 e viceversa

Il circuito non ha ingressi perchè sono gli output ad essere retroazionati verso l'input.

Ci sono due output: Q e Q*

Q dipende da Q*, e Q* dipende a sua volta da Q.

Per analizzare il circuito si considerano i due casi in cui Q è 0 oppure in cui Q è 1

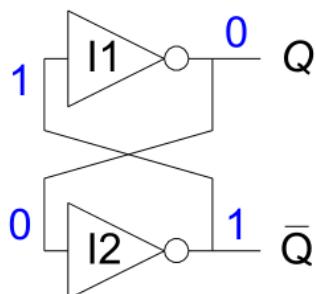


Caso I: Q = 0

I2 riceve in ingresso il valore di Q ossia Q=0 e di conseguenza produce un'uscita su Q*=1.

I1 riceve in ingresso Q*=1, e produce a sua volta un'uscita Q=0.

Dal momento che Q = 0 di nuovo come all'inizio, questo caso viene detto **stabile o consistente**

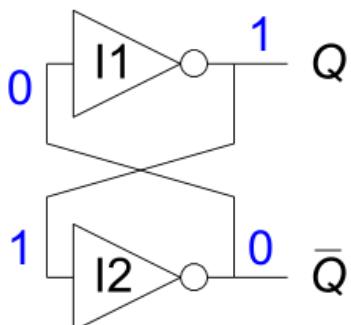


Caso II: $Q = 1$

I2 riceve in ingresso il valore di Q ossia $Q=1$ e di conseguenza produce un'uscita su $Q^*=0$.

I1 riceve in ingresso $Q^*=0$, e produce a sua volta un'uscita $Q=1$.

Dal momento che $Q = 1$ di nuovo come all'inizio, anche questo caso viene detto **stabile o consistente**



Dal momento che gli inverter collegati hanno due stati stabili, $Q = 0$ e $Q = 1$, il circuito viene chiamato **bistabile**.

Esso memorizza 1 bit di informazione nella variabile di stato Q (o Q negato)

Il valore di Q contiene tutte le informazioni sul passato necessarie a spiegare il comportamento futuro della rete. Nello specifico, se $Q = 0$, il valore rimarrà sempre zero, mentre se $Q = 1$, il valore rimarrà sempre 1

Il circuito all'inizio si troverà in uno dei due stati. $Q=0$ o $Q=1$ che dipende dalla configurazione iniziale del sistema. Quando una rete sequenziale viene accesa, lo stato iniziale è sconosciuto e solitamente imprevedibile, e può essere diverso a ogni nuova accensione della rete

L'utente non ha a disposizione un ingresso che gli permetta di controllare lo stato per questo si utilizzano i latch e i flip flop

SR (Set/Reset) Latch

Il latch SR rappresenta uno dei circuiti sequenziali più semplici ed è composto da **2 porte NOR collegate a croce**. Il latch ha due ingressi, **S e R**, e due uscite, **Q e Q***.

Il latch SR è simile al circuito bistabile ma il suo stato può essere controllato mediante gli ingressi, **S e R**, che attivano (set) e disattivano (reset) l'uscita Q.

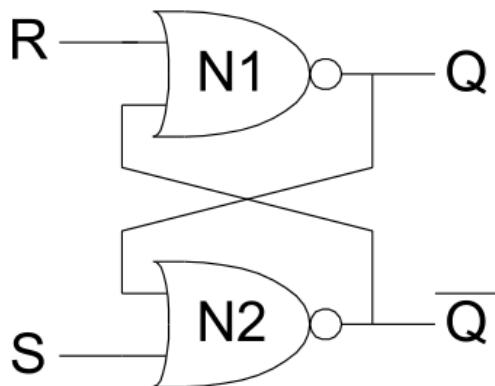
Consideriamo i **4 possibili stati**, combinazione degli ingressi S e R:

S = 1, R = 0 SET

S = 0, R = 1 RESET

S = 0, R = 0 MEMORIA

S = 1, R = 1 NULL STATE



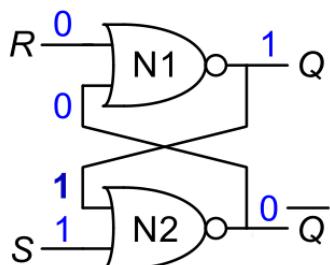
SET (S = 1, R = 0)

La porta NOR N1 riceve come ingressi R=0 e Q*.

Dal momento che il valore di Q* a questo punto è ancora sconosciuto, non è possibile determinare che valore assume Q.

La porta NOR N2 riceve in ingresso S=1, quindi produce un'uscita **Q*=0** (per la tavola di verità del NOR).

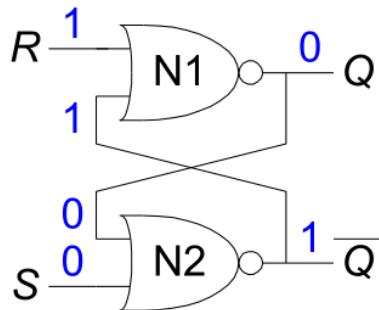
A questo punto è possibile tornare a N1 e, sapendo che entrambi gli ingressi S e R sono 0, il valore dell'uscita **Q= 1**, quindi abbiamo un SET



RESET ($S = 0, R = 1$)

La porta NOR N1 ha come input $R=1$ quindi produce un'uscita $Q=0$.

La porta N2 ha come ingressi quindi $Q=0$ e $S=0$, quindi produce un'uscita $Q^*=1$



MEMORIA ($S = 0, R = 0$)

N1 riceve gli ingressi 0 e Q^* . Dal momento che il valore di Q^* è sconosciuto, non è possibile determinare il valore dell'uscita.

N2 riceve a sua volta gli ingressi 0 e Q , ma visto che anche il valore di Q è sconosciuto, anche in questo caso non è possibile determinare il valore dell'uscita.

Quindi consideriamo l'ipotesi Q abbia assunto un valore precedentemente noto, **Q_{prev}** che può ovviamente essere 0 o 1 e rappresenta lo stato del sistema

- **$Q_{prev} = 0$**

S e Q_{prev} sono entrambi 0, N2 produce un'uscita $Q^*=1$.

A questo punto N1 riceve in ingresso $Q^*=1$ e ha $R=0$ quindi la sua uscita $Q = 0 = Q_{prev}$

- **$Q_{prev} = 1$**

Se $Q_{prev}=1$ e $S=0$ abbiamo che N2 produce un'uscita $Q^*=0$

Ora N1 riceve due ingressi $R=0$ e $Q^*=0$, quindi la sua uscita $Q = 1 = Q_{prev}$

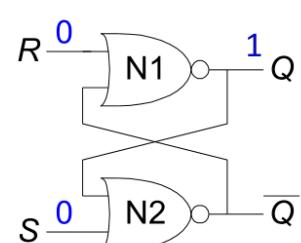
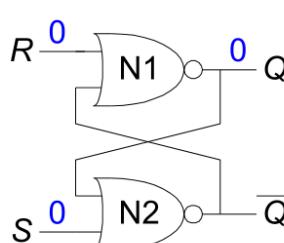
Quando R e S assumono valore 0, Q tiene memoria di tale valore precedente Q_{prev} quindi si dice che il circuito mantiene memoria

$S = 0, R = 0:$

allora $Q = Q_{prev}$
Memoria

$$Q_{prev} = 0$$

$$Q_{prev} = 1$$



NULL STATE (S=1, R=1)

N1 e N2 hanno entrambi almeno un ingresso VERO (R oppure S), quindi entrambi producono un uscita pari a 0. Di conseguenza, sia Q sia \bar{Q} sono 0.

Questo è uno stato NON VALIDO perché non può essere Q e \bar{Q} siano 0 perché devono essere una il complemento dell'altra

S = 1, R = 1:

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

Stato non valido

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

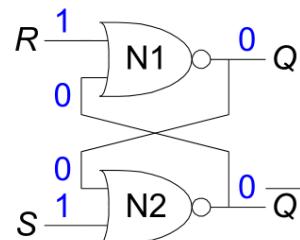


TABELLA DI VERITÀ'

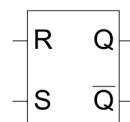
Caso	S	R	Q	\bar{Q}
IV	0	0	Q_{prev}	\bar{Q}_{prev}
I	0	1	0	1
II	1	0	1	0
III	1	1	0	0

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

- 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 $S = R = 1$

Come i negatori collegati a croce, anche il latch SR è un elemento bistabile con un bit di stato immagazzinato in Q . In questo caso però lo stato può essere controllato attraverso gli ingressi S e R. Quando viene attivato R, lo stato viene resettato a 0. Quando viene attivato S, lo stato viene settato a 1. Quando nessuno degli ingressi è attivato, lo stato mantiene il suo valore precedente

D LATCH

E' un'estensione del latch SR

Il D latch ha 2 ingressi:

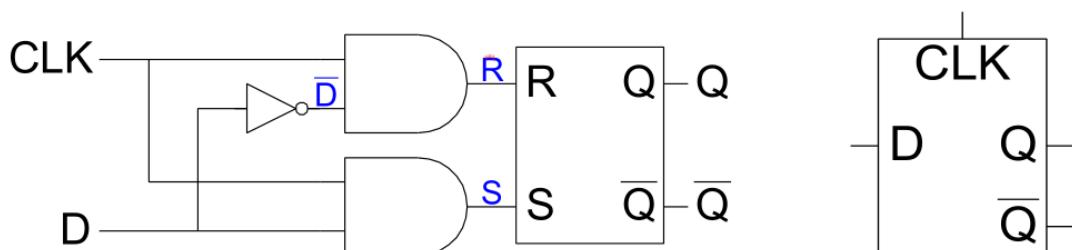
- un data input, D, che controlla in che cosa l'output cambia al prossimo stato
- un ingresso clock, CLK, che controlla invece quando l'output cambia ossia in quale momento avviene il cambio di stato.

In particolare:

Se **CLK = 1**, D passa fino a Q (**stato trasparente**) in output abbiamo quindi il valore di D

Se **CLK = 0**, Q mantiene il suo valore precedente (**stato opaco**) in output abbiamo quindi il valore Q_{prev}

Il clock fa sì che si eviti lo stato non valido



CLK	D	\bar{D}	S	R	Q	\bar{Q}
0	X	<u>X</u>	0	0	Q_{prev}	\bar{Q}_{prev}
1	0	1	0	1	0	1
1	1	0	1	0	1	0

Quando **CLK = 0**, sia S sia R sono 0, indipendentemente dal valore assunto da D.

Q ricorda il suo valore precedente, Q_{prev} . STATO OPACO

Se invece **CLK = 1**, allora una porta AND produce un valore VERO e l'altra un valore FALSO, a seconda del valore di D che viene negato.

In questo caso avremo che i dati scorrono da D verso Q, $D=Q$. STATO TRASPARENTE

D Flip Flop

E' una variante del D-Latch

Presenta sempre 2 input D e CLK

La peculiarità del D Flip Flop è che:

quando il CLK passa da 0 a 1, ossia nel fronte di salita del clock, D passa fino a Q

altrimenti, Q mantiene il suo valore precedente Qprev.

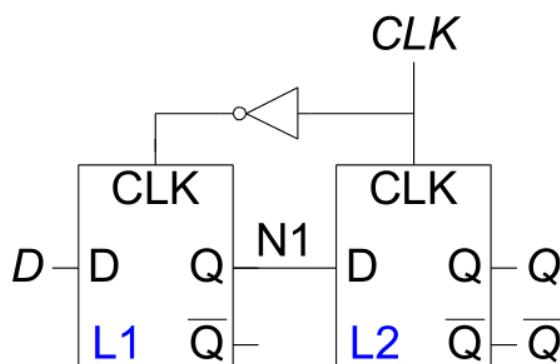
Quindi Q cambia solo durante la transizione di CLK da 0 a 1.

Queste tipologie di componenti sono dette **edge-triggered** perché sono pilotate dalla transizione del clock

E' possibile ottenere un D flip flop mediante la combinazione di 2 D-Latch (L1 e L2) controllati da segnali di clock complementari.

Il primo latch, **L1**, viene detto master, mentre il secondo latch, **L2**, viene detto slave.

Il nodo che unisce i due latch prende il nome N1.



Quando CLK = 0,

il latch **L1 master** è trasparente

il latch **L2 slave** è opaco

Qualsiasi valore di D viene portato a N1

Quando CLK = 1,

il latch **L1 master** è opaco

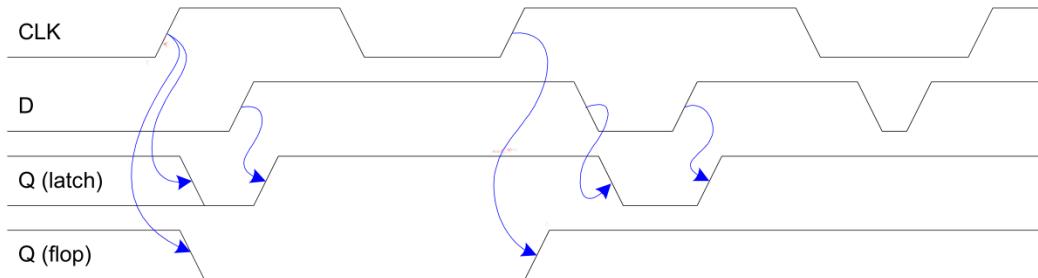
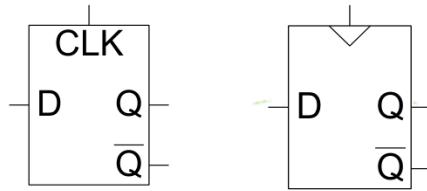
il latch **L2 slave** è trasparente

il valore di N1 viene trasmesso a Q, ma N1 resta isolato da D.

Quindi, D passa fino a Q sulla transizione di CLK da 0 a 1

In tutti gli altri casi, ad esempio se D cambia quando CLK=1 , D non passa a Q e

Q mantiene il suo valore precedente, dal momento che c'è sempre il latch L1 opaco che blocca il passaggio di dati tra D e Q.



Il cambiamento di Q nel D flip flop avviene solo sul fronte di salita del clock, è più sincronizzato con esso

Nel D latch il cambiamento di Q dipende sia dal valore del clock che dal valore di D

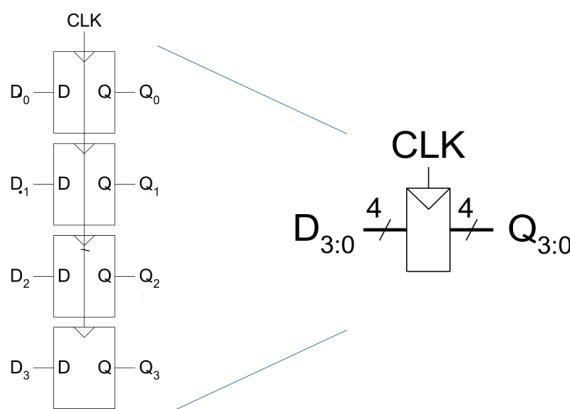
La freccia indica la causa di un cambiamento nell'uscita. Il valore iniziale di Q è sconosciuto e può quindi essere 0 o 1, come indica la coppia di linee orizzontali. Per prima cosa si consideri il latch. Sul primo fronte di salita di CLK, D = 0 quindi Q diventa 0. Ogni volta che D cambia valore, mentre CLK = 1, Q si comporta nello stesso modo. Quando invece D cambia mentre CLK = 0, il suo cambiamento viene ignorato. Si consideri ora il flip-flop: per ogni fronte di salita di CLK, D viene copiato in Q. In tutti gli altri casi, Q mantiene il suo stato precedente.

Registri: Multi-bit Flip-Flop

Un registro a N bit è un banco di N flip-flop D messi in parallelo che condividono un ingresso CLK comune, in modo che tutti i bit vengano aggiornati allo stesso tempo.

I registri costituiscono i blocchi costitutivi chiave per la maggior parte dei circuiti sequenziali.

un registro a quattro possiede un ingresso D_{3:0} e un'uscita Q_{3:0}. Sia D_{3:0} sia Q_{3:0} sono bus a quattro bit.



Flip-Flops “enabled”

Un flip-flop enable ha 3 input:

- CLK che controlla quando lo stato deve cambiare
- D input data
- EN , ENABLE, abilitatore che serve per determinare se memorizzare o no il dato sul fronte alto del clock.

Quando EN=1

il flip-flop si comporta come un normale flip-flop D e memorizza il dato, quindi D passa fino a Q sul fronte alto del clock

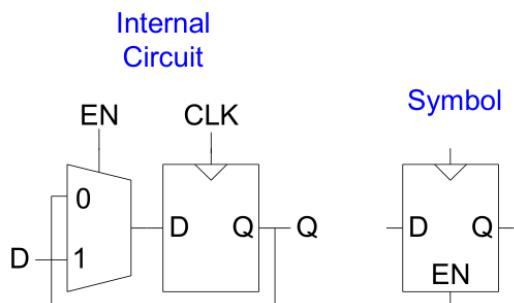
Quando EN=0

il flip-flop indipendentemente dal valore del clock mantiene il suo stato precedente.

I flip-flop con abilitazione sono utili quando si desidera inserire un nuovo valore in un flip-flop esclusivamente in alcuni precisi momenti, piuttosto che a ogni cambio del clock.

Può essere realizzato con un multiplexer che ha come selettore EN

EN=1 trasmette il valore dell'ingresso D e poi viene trasportato a Q sul fronte alto del clock
EN=0, l'output sarà il valore precedente di Q memorizzato, Qprev



Flip-Flops “resettabili”

Un flip-flop resettabile ha 3 input:

- CLK che controlla quando lo stato deve cambiare
- D input data
- RESET, che serve per resettare il flip flop

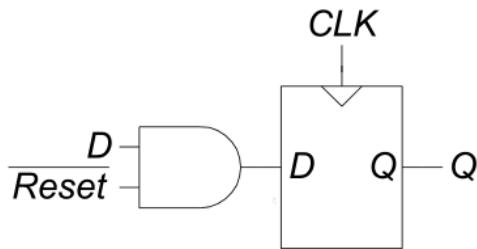
Quando l'ingresso **RESET = 0**, il flip-flop resettabile si comporta come un normale flip-flop D.

Quando invece **RESET = 1**, il flip-flop resettabile ignora D e, appunto, resetta l'uscita **Q= 0**.

Questa tipologia di flip-flop è utile nel caso in cui si desideri forzare uno stato 0 in tutti i flip-flop della rete quando viene accesa.

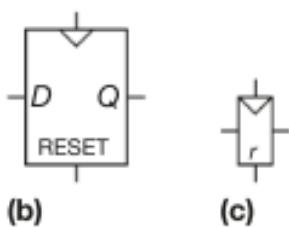
Vi sono due tipi di flip-flop resettabili:

- **Sincroni**: il reset è pilotato dal clock, quindi sul fronte alto del clock, se reset=1 allora Q=0
- **Asincroni**: il reset avviene non appena Reset = 1



RESET è un segnale attivo basso, il che significa che il segnale di reset esegue la propria funzione quando è 0 e non 1, forzando a 0 l'uscita della porta and.

Con l'aggiunta di un negatore, la rete avrebbe invece un segnale di reset attivo alto.



Flip-Flops “settabili”

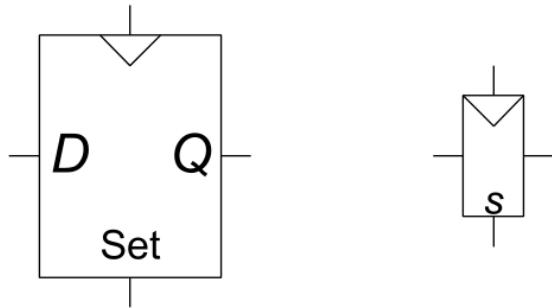
Un flip-flop resettabile ha 3 input:

- CLK che controlla quando lo stato deve cambiare
- D input data
- SET, che serve per settare il flip flop

Quando l'ingresso **SET = 0**, il flip-flop resettabile si comporta come un normale flip-flop D.

Quando invece **SET = 1**, il flip-flop resettabile ignora D e, appunto, setta l'uscita **Q=1**.

Symbols



Criticità nella logica sequenziale

Alcuni circuiti sequenziali sono alquanto problematici perché presentano delle criticità, dovute appunto alla logica sequenziale.

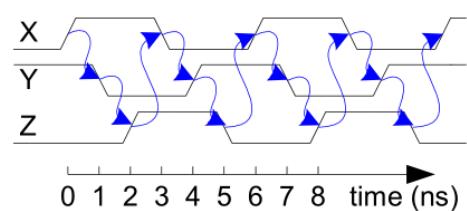
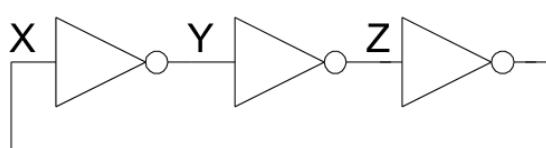
Un esempio è un circuito formato da 3 porte not in serie collegate ad anello.

Innanzitutto notiamo che il circuito è asincrono perchè l'output del terzo inverter è retroazionato in maniera diretta verso l'ingresso del primo inverter.

Se analizziamo il circuito notiamo che esso non ha stati stabili e dunque viene detto circuito **astabile**. Ogni nodo, quindi, oscilla tra 0 e 1 in un periodo (cioè in un tempo di ripetizione) pari a 6 ns. Questa rete è chiamata oscillatore ad anello.

Il periodo dell'oscillatore ad anello dipende dal ritardo di propagazione di ogni singolo negatore

Ogni ingresso prima è 0 e poi dopo è 1, poi di nuovo 0 e poi 1 e così via..nessun stato stabile



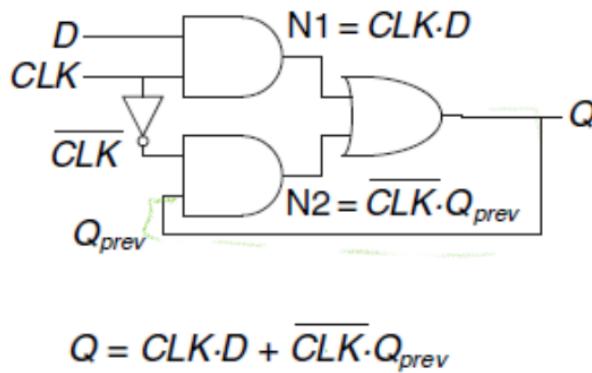
Adesso si prenda in analisi questo **latch d asincrono**, esso presenta dei malfunzionamenti dovuti all'uso di più porte AND, NOT, OR che accumulano ritardi sui singoli cammini.

Si supponga che $CLK = D = 1$: in questo caso il latch è trasparente e trasmette D per portare $Q = 1$. Se a questo punto CLK scende a 0, il latch dovrebbe ricordarsi dell'ultimo valore, mantenendo $Q = 1$.

Si supponga però che ci sia un forte ritardo attraverso il negatore che va da CLK a \overline{CLK} * maggiore di quello delle porte AND e OR: i nodi N1 e Q nel frattempo potrebbero entrambi abbassarsi prima che CLK abbia il tempo di alzarsi.

In questo caso, N2 non si alzerebbe mai, e Q rimarrebbe bloccato al valore 0.

Questo è un esempio di progetto di rete asincrona, nella quale le uscite sono direttamente collegate in retroazione agli ingressi



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

CLK 1→0 Q oscilla

Logiche sequenziali sincrone

I circuiti sequenziali asincroni (nei quali l'output è retroazionato in maniera diretta verso l'input) presentano delle criticità a volte difficilmente analizzabili che possono dipendere anche dalla struttura fisica dei componenti e dai ritardi accumulati, che possono causare dei malfunzionamenti nel circuito e portati a circuiti astabili oppure in stati di oscillamento

Per evitare questi problemi, si cerca di evitare di retroazionare l'output in maniera diretta e si interpone un registro nel ciclo di retroazione.

Questa operazione trasforma il circuito in un insieme di logica combinatoria e registri. I registri contengono lo stato del sistema, che cambia solo in corrispondenza dei fronti di salita del clock, motivo per cui lo stato viene detto sincronizzato con il clock.

Il clock è abbastanza lento da far sì che tutti gli ingressi dei registri abbiano il tempo di adeguare il proprio valore prima del fronte di clock successivo

Quindi i circuiti sequenziali sincroni sono regolati dal clock

In generale un circuito sequenziale sincrono ha un insieme finito di stati $\{S_0, \dots, S_{k-1}\}$. L'output del sistema è funzione sia dell'input che dello stato corrente del sistema. Il prossimo stato è funzione sia dell'input che dello stato corrente del sistema.

- Logica sequenziale sincrona:

$$\begin{aligned} \text{out} &= f(\text{in}, s_c) \\ s_n &= g(\text{in}, s_c) \end{aligned}$$

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 sono

Finite State Machines (FSMs)

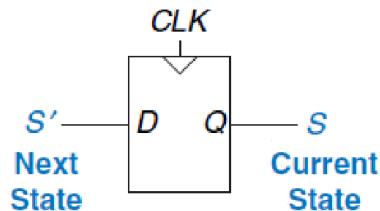
Pipelines

Un flip-flop D è il più semplice circuito sequenziale sincrono

possiede un ingresso, D, un clock, CLK, un'uscita, Q, e due stati, {0, 1}.

La specifica funzionale di un flip-flop definisce che il next state è D e che l'uscita, Q, è il current state

- $Q = s_c$
- $D = s_n$



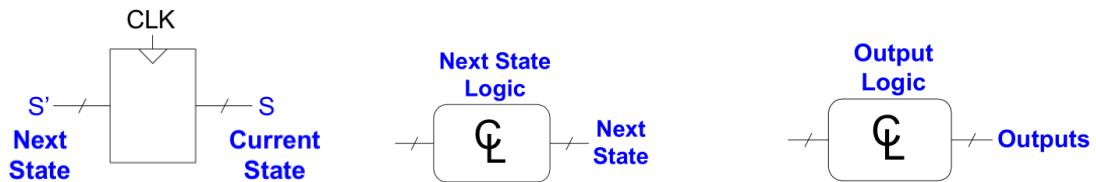
Finite State Machines

Una FSM è una macchina a stati finiti, il nome deriva dal fatto che **un circuito con k registri può trovarsi in uno di un numero finito (2 alla k) di stati diversi.**

Una FSM possiede M ingressi, N uscite e k bit di stato. Inoltre, riceve un segnale di clock e, a volte, anche un segnale di reset.

Una FSM è composta da:

- almeno un registro, detto **STATE REGISTER**, che memorizza lo stato corrente e carica il prossimo stato al battere del clock
- da due blocchi di **logica combinatoria**: **la logica di next state e la logica di output**
La logica di next state "computa" il prossimo stato del sistema in funzione degli input e dello stato corrente $s_n = g(\text{in}, \text{sc})$
La logica di output "computa" l'output del sistema (f)



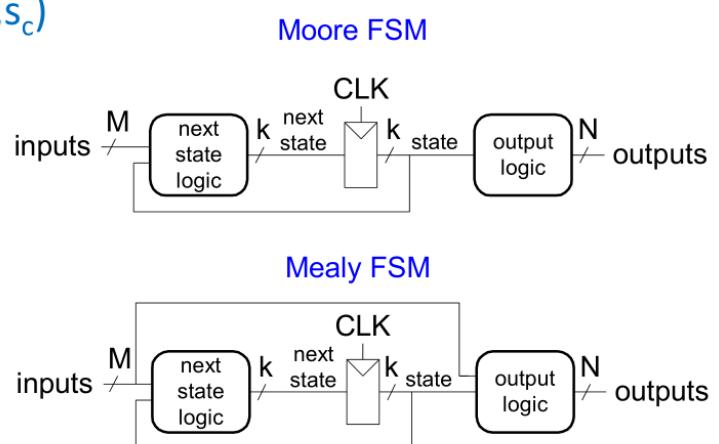
Ci sono 2 tipologie di FSM a seconda della logica di output:

- **MOORE FSM**: l'output è funzione del solo stato corrente $\text{out} = f(\text{sc})$
- **MEALY FSM**: l'output è funzione sia degli input che dello stato corrente $\text{out} = f(\text{in}, \text{sc})$

In entrambe la logica di next state è funzione sia dell'input che dello stato corrente
Mealy è più generale.

- 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:** $out=f(in, s_c)$



Come si progetta una FSM?

1-Si identificano **gli input e output** del sistema

2-Si **disegna il diagramma degli stati**, che indica tutti i possibili stati del sistema e le transizioni tra di essi. Nel diagramma degli stati, i cerchi rappresentano gli stati e gli archi rappresentano le transizioni tra di essi. Le transizioni avvengono al fronte di salita del clock

3-Si riporta il diagramma degli stati nella **tavella di transizione degli stati** che indica, per ogni stato presente e valori di ingresso, il next state S' del sistema.

4-Selezionare un **encoding** degli stati: one hot o binario

5-Macchina di Moore/Mealy: **si riscrive la tavella di transizione degli stati con l'encoding** degli stati

6-Si scrive in maniera analoga la **tavella degli output** che indica, per ogni stato, quale deve essere il valore assunto dall'uscita.

7-Scrivere le **espressioni booleane** relative alla logica di prossimo stato e alla logica di output in forma SOP

8- **Minimizzare** le espressioni ottenute

9- Realizzare **schema circuitale** mediante le espressioni minimizzate di logica next state e logica di output

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

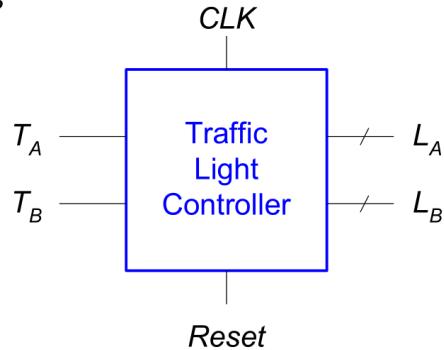


Diagramma di transizione: Moore FSM

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

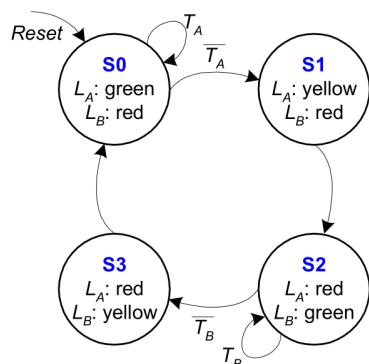


Tabella di transizione

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

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

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

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

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

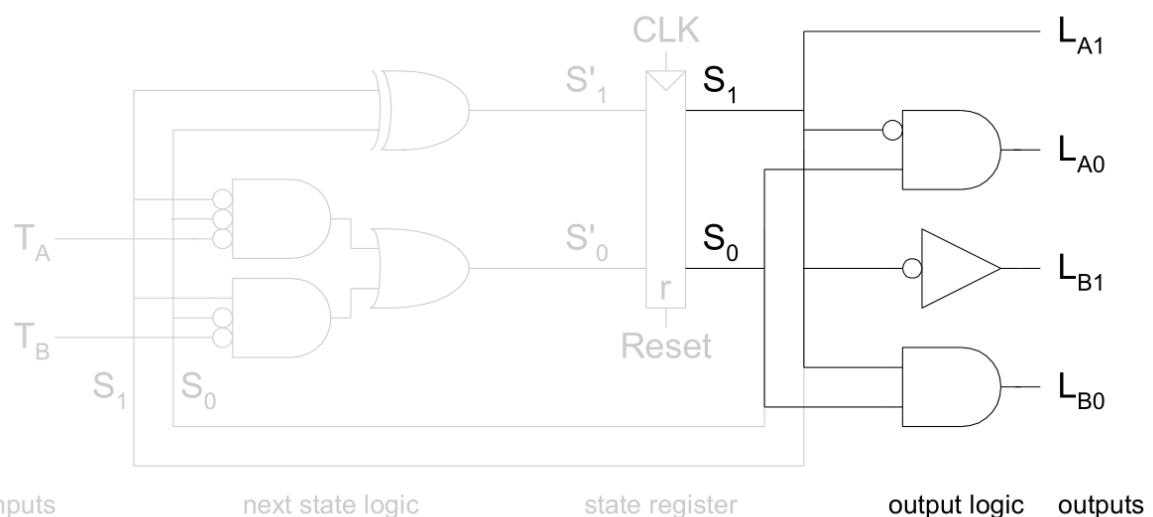
Output	Encoding
green	00
yellow	01
red	10

$$L_{A1} = S_1$$

$$L_{A0} = \overline{S_1} S_0$$

$$L_{B1} = \overline{S_1}$$

$$L_{B0} = S_1 S_0$$



Encoding degli stati

Binario: ogni stato viene rappresentato da un numero binario.

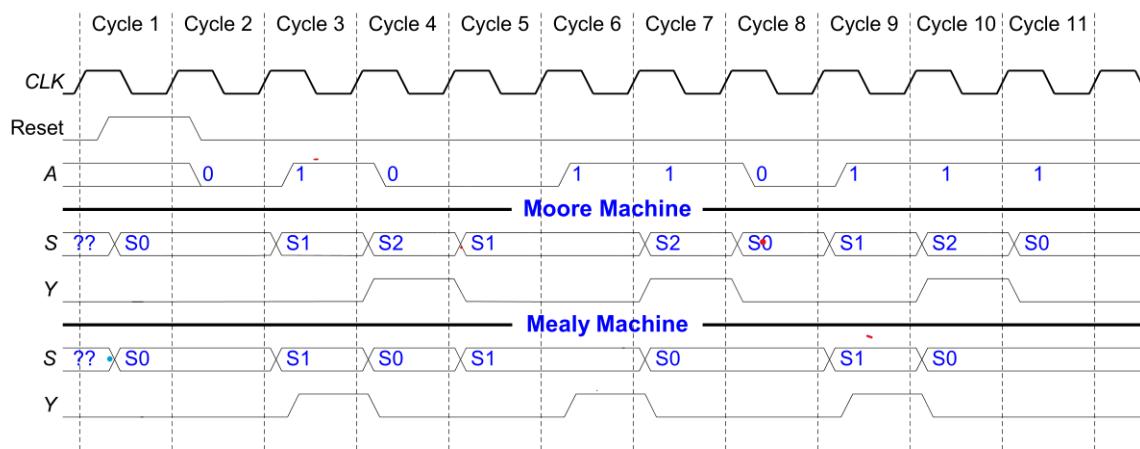
Dal momento che K numeri binari possono essere rappresentati da $\log_2 K$ bit, un sistema con K stati avrà bisogno solo di $\log_2 K$ bit di stato.

One hot: viene utilizzato un bit di stato per ognuno degli stati.

ogni bit è associato ad uno stato quindi solo un bit alla volta sarà alto
es per 4 stati, 0001, 0010, 0100, 1000

Richiede più flip-flops perché ogni bit di stato viene immagazzinato in un flip-flop, ma spesso la logica combinatoria associata è più semplice

Moore vs Mealy FSM. Timing diagram. Chi è più sincrono?



Le due macchine seguono due diverse sequenze di stati.

Nella macchina alla Mealy l'uscita passa a 1 in anticipo di un ciclo perché risponde all'ingresso invece di attendere il cambiamento dello stato. Se l'uscita della macchina alla Mealy fosse ritardata da un flip-flop, la sua temporizzazione corrisponderebbe a quella della macchina alla Moore. Nella scelta dello stile di progettazione di una FSM, serve quindi decidere quando si vuole che le uscite rispondano.

Nell' automa di Moore l'output è quasi sincronizzato rispetto al battere del clock, perché esso dipende dal solo stato corrente e lo stato, essendo memorizzato nei registri è sincronizzato col clock e cambia sul fronte di salita.

Moore è più sincrono perché è in funzione del solo stato corrente che è sincronizzato col clock

Negli automi di Mealy l'output dipende sia dal registro che memorizza lo stato corrente che è sincronizzato col clock, ma anche dall'input e se l'input non è sincronizzato col clock il valore dell'output può cambiare in maniera non sincrona rispetto al clock. L'output viene influenzato da un certo ritardo rispetto al battere del clock dovuto al fatto che esso dipende anche dagli input. Il tempo per leggere l'output è ridotto in prossimità del battere del nuovo clock

Fattorizzazione di FSM

Spesso è più semplice progettare FSM complesse se queste possono essere decomposte in diverse macchine a stati più semplici che interagiscono tra loro, facendo sì che le uscite di alcune macchine siano gli ingressi di altre. Questa applicazione dei principi di gerarchia e modularità alle macchine viene chiamata fattorizzazione delle macchine a stati

Fattorizzare consiste quindi nel suddividere una FSM complessa in FSM più piccole che interagiscono fra loro

Parallelismo

Parallelismo vuol dire eseguire in contemporanea più task/comitti

La velocità di un sistema è caratterizzata dalla **latenza** e dal **throughput**
il parallelismo incrementa il throughput

Token: Gruppo di input da processare per ottenere un output significativo.

Pacchetto di informazioni

latenza: Tempo che occorre ad un token per essere processato e produrre un output

throughput: Numero di output prodotti per unità di tempo

2 tipi di parallelismo:

- **Spaziale:** duplicare l'hardware per eseguire più task contemporaneamente es. processori che lavorano in parallelo
- **Temporiale:** l'hardware è sempre 1 ma il task da eseguire viene suddiviso in più fasi e queste fasi sono eseguite in **pipelining** ossia in sequenza . Quando si finisce la fase di un task già inizia un'altra fase di un task diverso

il parallelismo temporale migliora il throughput a scapito della latenza.

4. CIRCUITI ARITMETICI E MEMORIE

I circuiti aritmetici sono i blocchi costruttivi centrali dei calcolatori. I calcolatori e la logica digitale eseguono molte funzioni aritmetiche: addizioni, sottrazioni, confronti, traslazioni, moltiplicazioni e divisioni. L'addizione è una delle operazioni più comuni nei sistemi digitali.

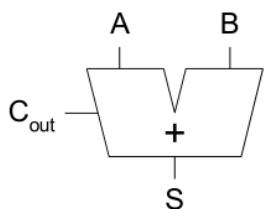
1-Bit Adders

Half Adder (1 bit)

ha due input, A e B, e due uscite, S e Cout.

S rappresenta la somma di A e B. Se sia A sia B hanno valore 1, S è uguale a 2, un valore che non può essere rappresentato con una sola cifra binaria. Di conseguenza, il valore 2 viene rappresentato con un riporto (carry) Cout

Riporta la somma dei due bit dati in input e l'eventuale riporto.



A	B	C _{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$S = A \oplus B$$

$$C_{out} = AB$$

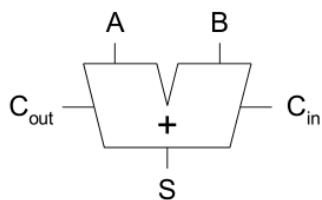
Full adder

E' utilizzato per creare sommatori a più bit, prende in input i due bit di ingresso A e B e l'eventuale riporto Cin che è il riporto che proviene dalle cifre meno significative precedenti. Ritorna il valore della somma tra i due bit e un eventuale riporto Cout.

somma quindi il riporto eventualmente ottenuto dai due bit di ordine immediatamente inferiore.

Cout=1 se almeno due dei tre ingressi è uguale a 1

Full Adder



C_{in}	A	B	C_{out}	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

$$S = A \oplus B \oplus C_{in}$$

$$C_{out} = AB + AC_{in} + BC_{in}$$

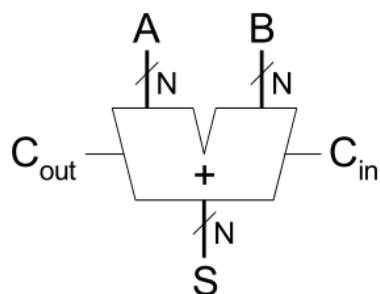
Multibit Adders (CPAs)

Sono sommatori a n bit che si utilizzano quando vogliamo sommare parole di più bit ad esempio quando abbiamo due ingressi A e B a n bit

Hanno in input :

- due parole a n bit A e B
- un riporto Cin che si propaga nei bit successivi

In output abbiamo l'eventuale riporto Cout e la somma a n bit S degli ingressi A e B



Abbiamo 2 tipologie di multibit adders:

- ripple carry (lento e più ottimizzato per sommare pochi bit)
- carry-lookahead (più veloce e ottimizzato per sommare più bit)

Ripple-Carry Adder

E' formato da una concatenazione di full adder a 1 bit messi in serie dalla cifra meno significativa alla cifra più significativa

Il Cin iniziale è posto a 0. Le prime due cifre binarie meno significative di A e B vengono sommate e nel caso avessimo un Cout esso sarà il Cin delle cifre seguenti, quindi si propaga. Le cifre seguenti saranno sommate con in aggiunta il Cout delle cifre precedenti
Il riporto si propaga lungo la catena

Il principale svantaggio legato a questo sommatore è il progressivo rallentamento all'aumentare di N. Infatti, S₃₁ dipende da C₃₀, che dipende da C₂₉, che dipende a sua volta da C₂₈ e così via fino a risalire a Cin

Si dice quindi che il riporto si propaga a onda attraverso la catena.

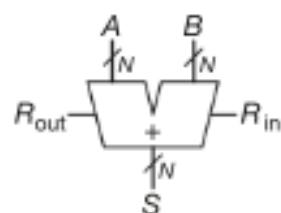
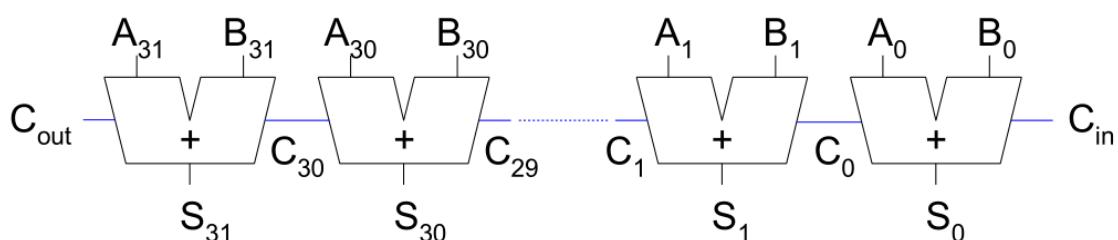
Il ritardo di propagazione nel sommatore, triplice, aumenta all'aumentare del numero di bit coinvolti e quindi dei full adder inseriti.

t_{FA} rappresenta il ritardo di un full adder.

$$t_{\text{ripple}} = N t_{\text{FA}}$$

t_{FA} è il ritardo di un singolo 1-bit full adder

N è il numero di 1-bit full adders



Carry lookahead

Questa tipologia di multibit adder minimizza i tempi di ritardo rispetto al ripple carry perché divide il sommatore in blocchi di k-bit ai quali viene aggiunto un circuito per determinare velocemente il riporto di uscita Cout di ciascun blocco appena è noto il riporto di ingresso Cin

In questo modo i riporti si propagheranno più velocemente

Per esempio, un sommatore a 32 bit può essere diviso in otto blocchi da 4 bit ciascuno.

I riporti vengono calcolati attraverso le funzioni **Generate G** e **Propagate P**

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

La configurazione AiBi (11) viene detta **GENERATE** perchè genera un carry uguale a 1 a prescindere dai valori del Cin

Le configurazioni AiBi (01,10) vengono chiamate **PROPAGATE** perchè propagano il valore del Cin sul Cout

La configurazione AiBi (00) invece riporta sempre un Cout uguale a 0 a prescindere del Cin

GENERATE

La colonna i di un sommatore genera sicuramente riporto se A_i e B_i sono entrambi uguali a 1. Di conseguenza G_i , cioè il riporto generato dalla colonna i , viene calcolato come
$$G_i = A_i * B_i$$

In generale, per k bit:

$$G_{i:j} = G_i + P_i (G_{i-1} + P_{i-1} (G_{i-2} + P_{i-2} (\dots G_j) \dots))$$

Un blocco di k bit genera un riporto se la colonna più significativa genera un riporto, oppure se la colonna più significativa propaga un riporto e quella precedente ne genera uno, e così via.

PROPAGATE

Si dice che la colonna i propaga un riporto se produce un riporto di uscita ognqualvolta ci sia un riporto di ingresso. La colonna i propaga un Cin al Cout iesimo, **se o A_i o B_i sono uguali a 1**. Di conseguenza,

$$P_i = A_i + B_i$$

In generale, per k bit:

$$P_{i:j} = P_i P_{i-1} P_{i-2} P_j$$

Un blocco propaga un riporto se tutte le colonne del blocco propagano un riporto.

LOGICA DI CARRY

logica di riporto di una specifica colonna del sommatore:

la colonna i del sommatore produce un riporto di uscita C_i se c'è un generate, G_i , o se propaga il riporto di ingresso, $P_i C_{i-1}$.

- Carry out: il carry i (C_i) è dato da:

$$C_i = A_i B_i + (A_i + B_i) C_{i-1} = G_i + P_i C_{i-1}$$

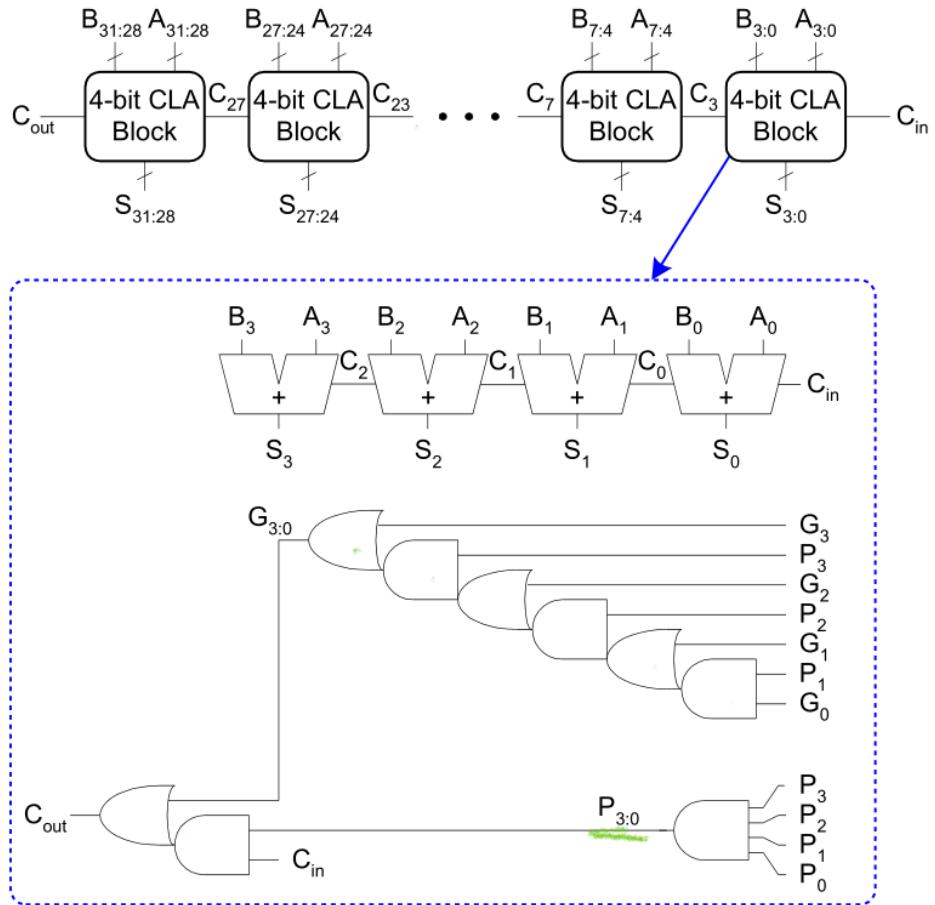
Abbiamo carry out se c'è una configurazione generate o un propagate che si è propagato dal riporto precedente C_{i-1}

$$C_i = G_{i:j} + P_{i:j} C_{j-1}$$

Abbiamo un riporto al blocco se abbiamo configurazioni di Generate o Propagate che propagano il riporto del blocco precedente, oppure configurazioni Generate e Propagate insieme

Com'è fatto un Carry lookahead adder

32-bit CLA with 4-bit Blocks



Step 1: calcola tutti i Gi and Pi

Step 2: calcola G and P per blocchi di k-bit

Step 3: Cin si propaga mediante la logica propagate/generate dei vari blocchi di k-bit

Avremo dei circuiti combinatori che consentono di calcolare in parallelo ognuno dei valori generate e propagate

Si suddividono in questo caso i full adder in blocchi da 4. Ogni blocco è composto da 4 sommatori e poi da una logica che consente di trasportare il Cin in ingresso ad un blocco direttamente al Cout del blocco in questione. Poi i riporti si propagano da blocco a blocco. I valori di G e P sono calcolati in parallelo su tutti i blocchi.

Tutti i blocchi del CLA calcolano i segnali di generazione e di propagazione sia di colonna sia di blocco simultaneamente.

Ritardo del Carry Lookahead Adder

Per un N -bit CLA con blocchi di k bit:

$$t_{CLA} = t_{pg} + t_{pg_block} + (N/k - 1)t_{AND_OR} + kt_{FA}$$

tpg : ritardo per generare P_i , G_i di ogni i esima colonna

tpg_block: ritardo per calcolare i segnali di generazione e di propagazione $P_{i:j}$ e $G_{i:j}$ per ogni blocco a **k bit**,

tAND_OR: ritardo delle porte AND/OR a monte della logica propagate/generate, questo ritardo si propaga da Cin a Cout che si propaga lungo i tutti i blocchi (N num bit totali / k – 1 dimensioni blocco)

ktFA: ritardo di un full adder per il numero di bit di cui è costituito un blocco

Per $N > 16$ il sommatore ad anticipazione di riporto è generalmente molto più veloce rispetto

Sottrattore

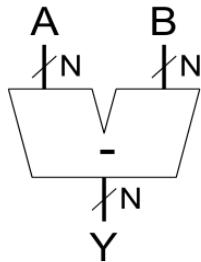
I sommatori sono in grado di sommare numeri sia positivi sia negativi utilizzando la rappresentazione dei numeri in complemento a due. La sottrazione è quindi facile quanto l'addizione: si inverte il segno del secondo numero B e poi si esegue la somma.

Il cambio di segno di un numero in complemento a due si esegue negando tutti i bit e aggiungendo un 1.

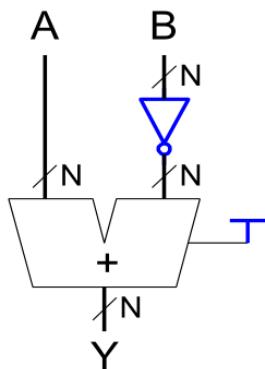
Per calcolare $Y = A - B$, per prima cosa si crea il numero in complemento a due di B: si negano tutti i bit di B per ottenere B^* e si aggiunge 1 per ottenere $-B = B^* + 1$. Questo valore viene aggiunto ad A per ottenere $Y = A + B^* + 1 = A - B$.

Il +1 aggiunto al complemento di B è dato dal Cin che è settabile ed è impostato inizialmente a 1 proprio per effettuare il complemento a 2 del numero B.

Symbol



Implementation



Comparatore

Un comparatore determina se due numeri binari sono uguali o se uno dei due è maggiore o minore dell'altro.

Un comparatore riceve due numeri binari a N bit in ingresso, A e B.

Esistono due tipi comuni di comparatori:

comparatori di uguaglianza: produce una singola uscita che indica se A è uguale a B o no ($A == B$) oppure no.

comparatore completo: produce una o più uscite che indicano tutti i valori relativi di A e di B quindi se $A > B$, $A < B$ o $A = B$

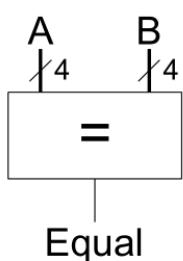
Comparatore di uguaglianza a 4 bit

Confronta 2 ingressi a 4 bit e ritorna 1 se e solo se i due ingressi sono uguali.

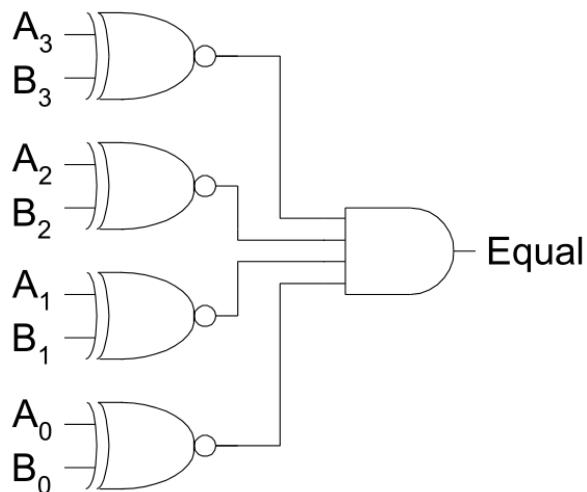
A e B sono uguali quando bit per bit di ogni colonna sono uguali

il comparatore determina se i bit corrispondenti a ogni colonna (bit a bit) sono uguali utilizzando delle porte XNOR. I due numeri sono uguali se tutte le colonne sono uguali, quindi alla fine avremo un AND a 4 ingressi di tutti gli output delle porte XNOR

Symbol

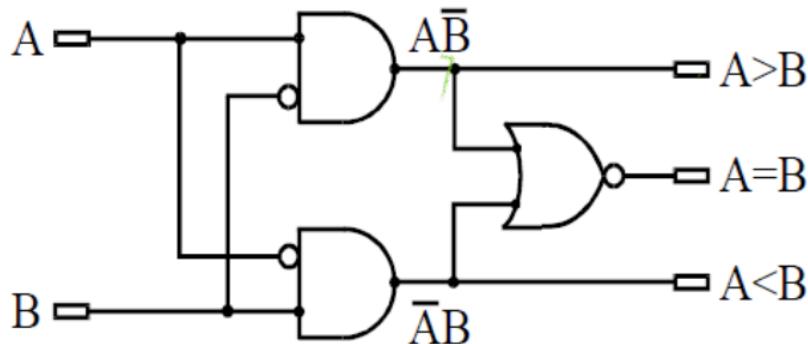


Implementation



Comparatore completo a 2 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



Ha 3 uscite che indicano

- se $A = B$ (ossia $A=0$ e $B=0$ oppure $A=1$ e $B=1$ quindi AB^* NOR A^*B)
- se $A > B$ (ossia quando $A=1$ AND $B=0$, AB^*)
- se $A < B$ (ossia quando $A=0$ AND $B=1$, A^*B)

Per ogni valore di A e B una sola delle 3 uscite sarà uguale a 1

Un comparatore completo a 4 bit o 12 bit suddivide le parole in blocchi e ottiene i risultati del confronto per ogni blocco. I valori di un blocco di uscita diventano i valori di ingresso del blocco successivo. I valori si propagano dall'inizio alla fine di un blocco a meno che i bit non siano uguali e si passa al blocco successivo

La comparazione di valore dei numeri con segno viene solitamente effettuata calcolando $A - B$ e guardando il segno (cioè il bit più significativo) del risultato dell'operazione.

Se il risultato è negativo (cioè se il bit del segno è uguale a 1) allora A è minore di B. Al contrario, se il risultato è positivo, A è maggiore o uguale a B. Tuttavia, questo comparatore non lavora correttamente in caso di traboccamiento (overflow).

ALU

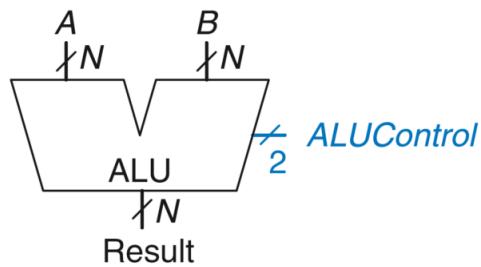
E' l'unità logico aritmetica che esegue i calcoli principali del calcolatore

- **Addizione**
- **Sottrazione**
- **AND (bit a bit)**
- **OR (bit a bit)**

Una ALU a N bit ha:

- 2 ingressi a N bit A e B
- un output (Result) a N bit
- un segnale di controllo a 2 bit **ALUControl** che specifica quale funzione debba eseguire l'ALU

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



Schema circuitale

La ALU contiene un sommatore a N bit per la somma e un numero N di porte AND o OR a due ingressi per le operazioni logiche di AND e OR

Contiene un inverter a N bit e un multiplexer per invertire l'ingresso B quando il segnale di controllo **ALUControl0 = 1** ossia quando dobbiamo eseguire una sottrazione per complementare il numero

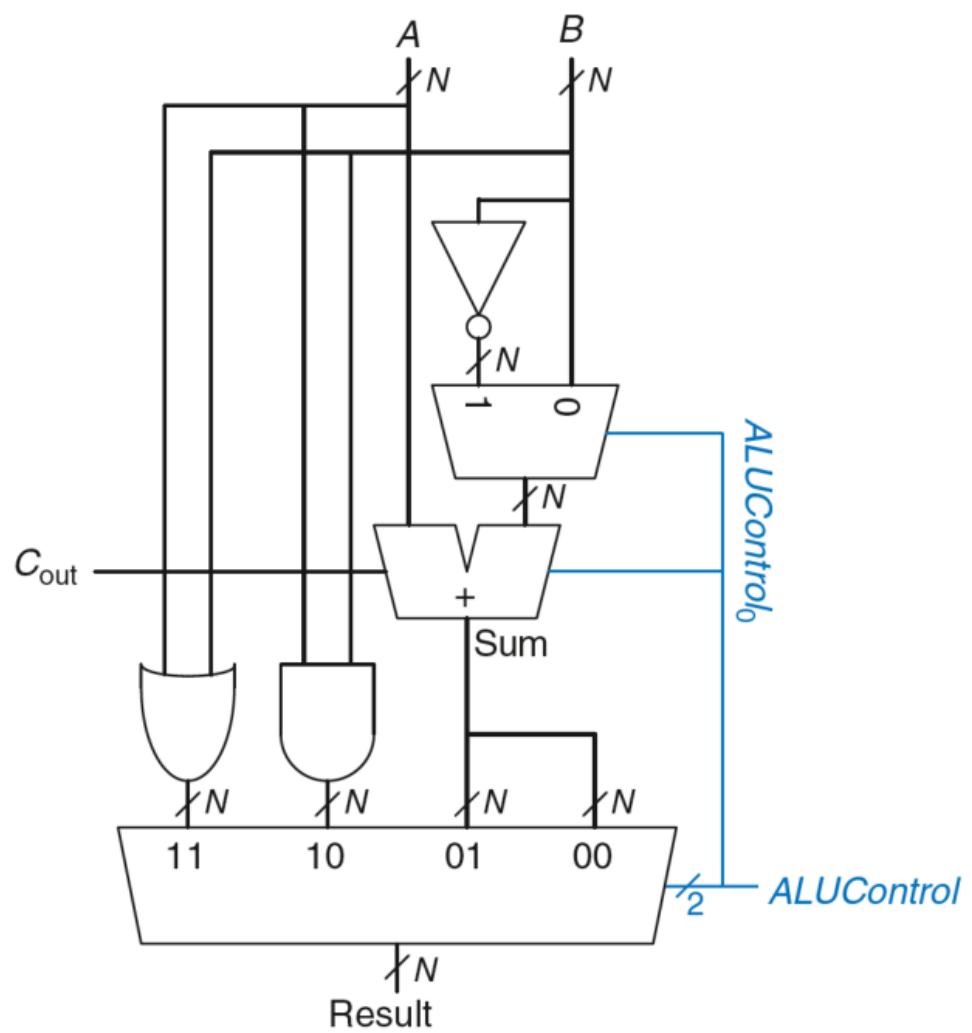
Infine un multiplexer 4:1 sceglie l'operazione desiderata in output sulla base del segnale ALUControl.

se ALUControl = 00, il multiplexer di uscita sceglie **A + B**.

se ALUControl = 01, la ALU calcola $A - B$ (dal momento che ALUControl0 è uguale a 1, il sommatore riceve gli ingressi A e B^* e un riporto di ingresso a 1, il che fa sì che il sommatore esegua la sottrazione col complemento a 2: $A + B^* + 1 = A - B$).

se ALUControl = 10 l'ALU esegue A AND B

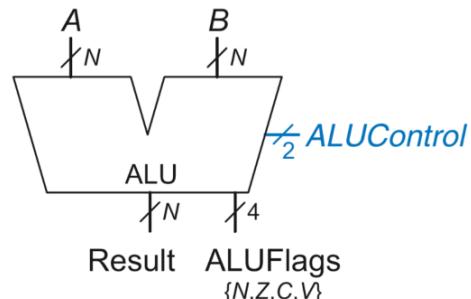
se ALUControl = 11, l'ALU esegue A OR B.



ALU con flags di stato

E' un alu che come output ha anche delle uscite chiamate flag, che danno informazioni aggiuntive sul risultato dell'operazione effettuata dall'ALU.

Flag	Description
<i>N</i>	<i>Result</i> is Negative
<i>Z</i>	<i>Result</i> is Zero
<i>C</i>	Adder produces Carry out
<i>V</i>	Adder oVerflowed



L'uscita ALUFlags è a 4 bit ed è composta dalle flag

N (negative) : indica che il risultato dell'ALU è negativo, ossia il bit più significativo del risultato $\text{Result}_{31}=1$

Z (zero): indica che il risultato dell'ALU è uguale a zero quindi se tutti i bit di Result sono uguali a 0. Si prendono tutti i bit di $\text{result}_{31:0}$, si commutano (porta NOT) e si mettono in AND. La porta AND restituisce 1 se tutti i 32 ingressi commutati sono 1 e quindi sono tutti 0.

C (carry): indica che il sommatore ha prodotto un riporto CarryOut

La flag C =1 quando il sommatore produce un riporto e l'ALU sta eseguendo una somma o una sottrazione ($\text{ALUControl}_1 = 0$). ALUControl_1 viene commutato (quindi se è 0 diventa 1 e messo in AND con il Cout = 1) se abbiamo 1 dalla porta AND allora il Carry flag si attiva

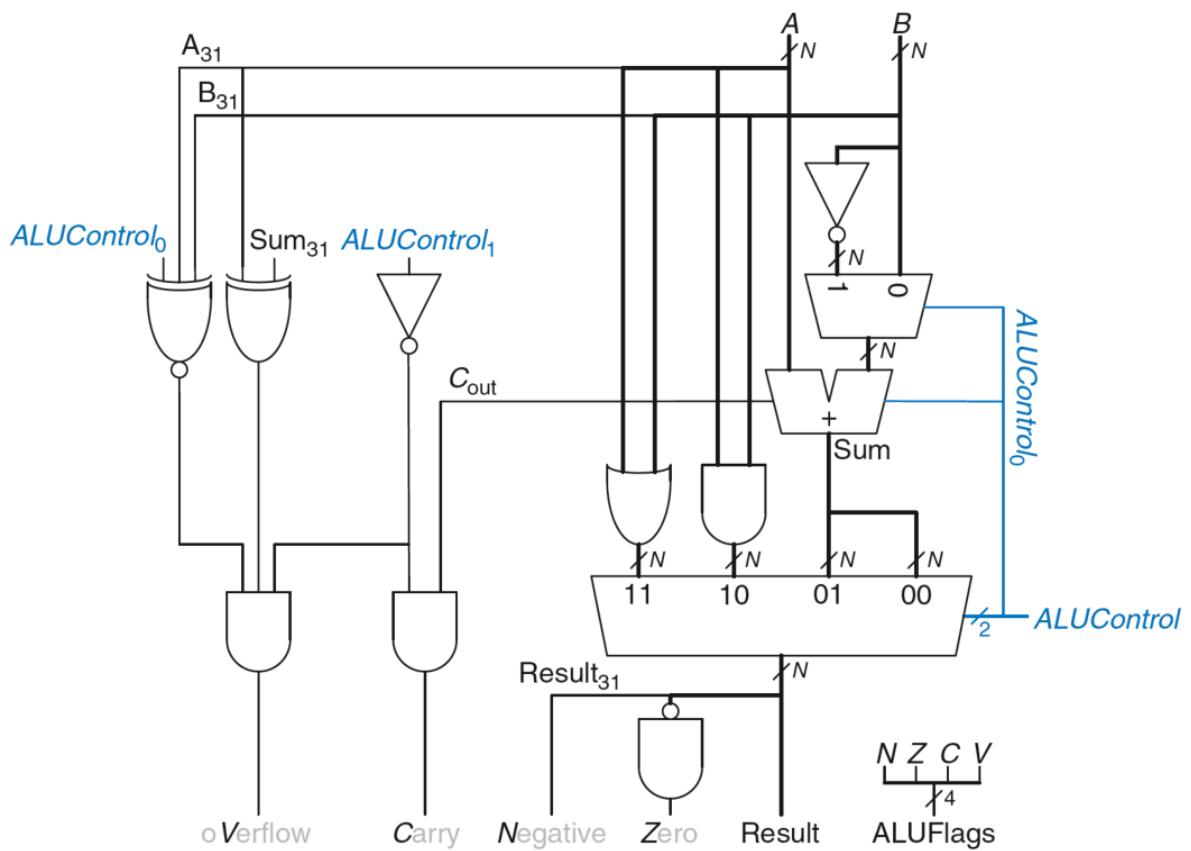
V (overflow): indica che il sommatore ha prodotto un Overflow. esso si verifica quando la somma di due numeri di egual segno produce come risultato un numero di segno opposto.

V è attiva nel caso in cui si verifichino le 3 seguenti condizioni:

(1) la ALU sta eseguendo una **somma o una sottrazione ($\text{ALUControl}_1 = 0$ commutato 1)**
 (2) **il bit A31 e quello della somma Sum31 hanno segni opposti**, come identificato dalla porta **XOR**

(3) come identificato dalla porta XNOR i bit A31 e B31 sono uguali, hanno lo stesso segno e il sommatore sta seguendo un'addizione ($\text{ControlloALU}_0 = 0$), oppure A31 e B31 hanno segno opposto e il sommatore sta eseguendo una sottrazione ($\text{ControlloALU}_0 = 1$).

La porta AND a tre ingressi riconosce quando tutte e tre le condizioni si avverano e quindi attiva V



Shifters

Sono circuiti che traslano o ruotano i bit ed eseguono la moltiplicazione o la divisione per potenze di 2.

Logical shifter: trasla i bit a sinistra a (LSL, Logical Shift Left) o a destra (LSR, Logical Shift Right) e riempie gli spazi vuoti con degli 0

Arithmetic shifter: come il logical shifter a sinistra, nella traslazione a destra (ARS, Arithmetic Shift Right) invece riempie gli spazi vuoti con il bit più significativo (msb)

Rotator: ruota i bits circolarmente verso sinistra (ROL, Rotate Left) o destra (ROR, Rotate Right)

i bits che “escono” da un lato rientrano dall’ “altro”

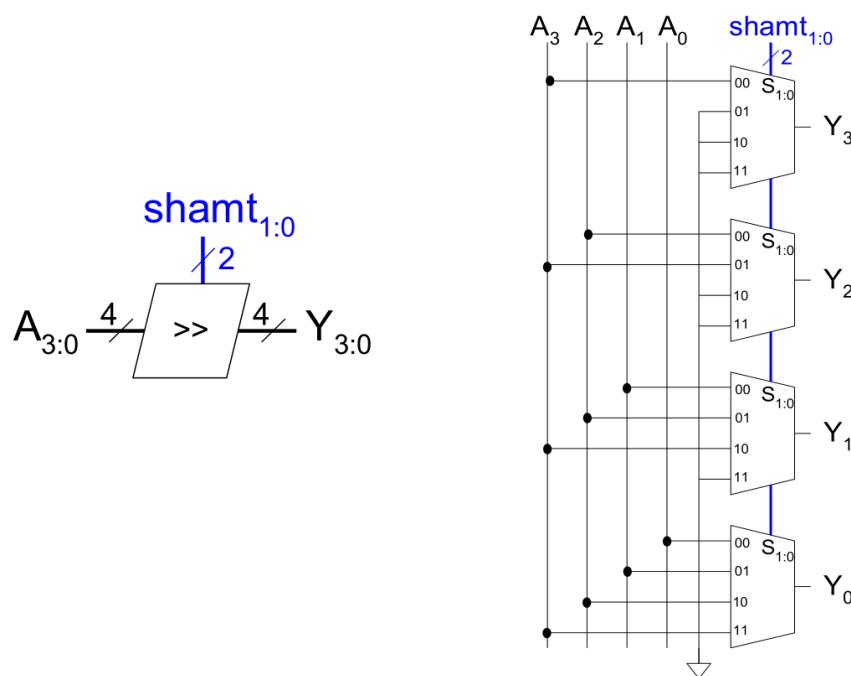
Shifter Design

Uno shifter a N bit può essere costruito con un numero N di multiplexer N:1.

L’ingresso viene traslato da 0 a N – 1 posizioni, a seconda dei valori presenti sulle log2N linee di selezione

Es. Logical Shift Right

In output Y abbiamo la parola di 4 bit shiftata verso dx di una quantità che viene indicata dai due selettori che prendono il nome di sham1:0 (shift amount) ed indicano la quantità di cui deve shiftare la parola. Essendo la parola in ingresso A di 4 bit è possibile shiftare fino a 4 posizioni, quindi avrebo bisogno di log2 4 = 2 selettori per lo sham1



Shifters as Multipliers, Dividers

Gli arithmetic shifter sono detti aritmetici perchè le operazioni di asr e asl corrispondono alla moltiplicazione di un numero per 2^N dove N è la quantità della traslazione (shift a sx lsl o asl)

oppure alla divisione di un numero per 2^N dove N è la quantità della traslazione (shift a dx asr)

- $A \lll N = A \times 2^N$

- **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

Il contatore è un dispositivo che conta, quindi incrementa il suo valore di output di 1 ad ogni ciclo di clock sul fronte alto. Viene utilizzato in particolare dal program counter per tenere traccia dell'istruzione corrente da eseguire o come orologio digitale.

Un contatore binario a N bit, è circuito sequenziale aritmetico

Ha come ingresso il segnale di clock che pilota il contatore

Il segnale RESET permette di resettare l'output, inizialmente è resettato a 0

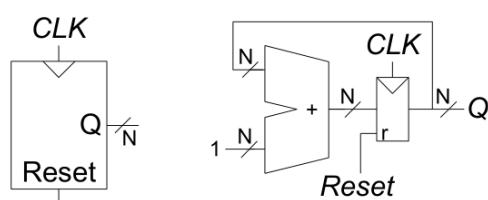
In output abbiamo N uscite, Q che rappresenta di volta in volta il numero corrente

C'è un flip flop D resettabile che tiene memoria del valore corrente del contatore, è resettabile quindi è possibile resettare il valore del contatore.

Il valore corrente in output Q viene retroazionato in un sommatore che prende in input come argomenti il valore corrente dell'output Q e il valore 1. Quindi il sommatore di volta in volta somma al valore corrente 1. **Per ogni clock $Q = Q_{prev} + 1$**

Genera 2^N valori

Symbol	Implementation



Shift Registers

Lo shift register esegue una traslazione di un **ingresso seriale Sin** di 1 bit per ogni battito del clock.

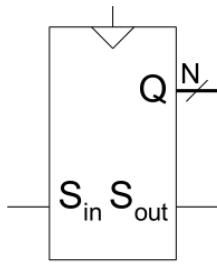
Ha **un ingresso seriale Sin cioè 1 alla volta**, **un'uscita seriale Sout**, e **N uscite parallele QN-1:0**

A ogni fronte di salita del clock, un nuovo bit viene inserito dall'ingresso Sin e tutti i bit seguenti vengono traslati in avanti. L'ultimo bit nel registro diventa quindi disponibile all'uscita Sout.

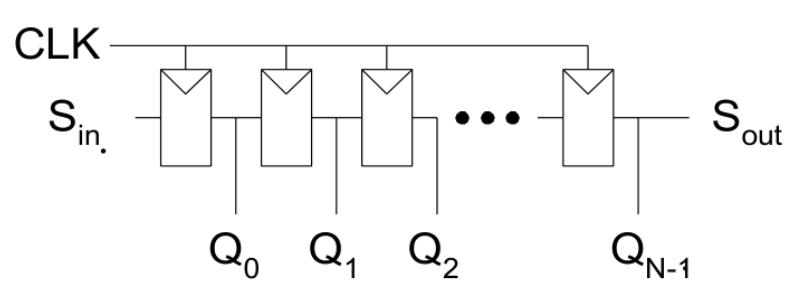
E' implementato da una serie di flip flop d cioè registri che memorizzano 1 bit in sequenza tra loro. Alla fine di ogni flip flop abbiamo l'uscita Q₀, Q₁, Q_{n-1} alla fine tutte le uscite compongono l'uscita finale Sout. Tutti i flip flop sono pilotati dallo stesso clock

Viene detto **convertitore seriale parallelo** : l'ingresso viene infatti ricevuto in serie (un bit alla volta) da Sin. Dopo N cicli, gli N ingressi ricevuti sono disponibili in parallelo su Q. Quindi in parallelo si legge quella che è la storia di Sin negli scorsi n clock, a partire dal meno recente (N clock fa) fino all'ultimo valore che è il più recente dell'ultimo clock

Symbol:



Implementation:



$$Q_0 = 1$$

$$Q_1 = 0$$

$$Q_2 = 1$$

...

$$Q_{N-1} = 1$$

$$Q = 1011$$

Sout mostra l'output uno alla volta

Shift Register con load parallelo

E' una versione più complessa dello shift register.

Esegue sia la conversione seriale-parallelo sia quella parallelo-seriale, aggiungendo un ingresso parallelo D a N bit e un segnale di controllo LOAD che pilota un mux di scelta

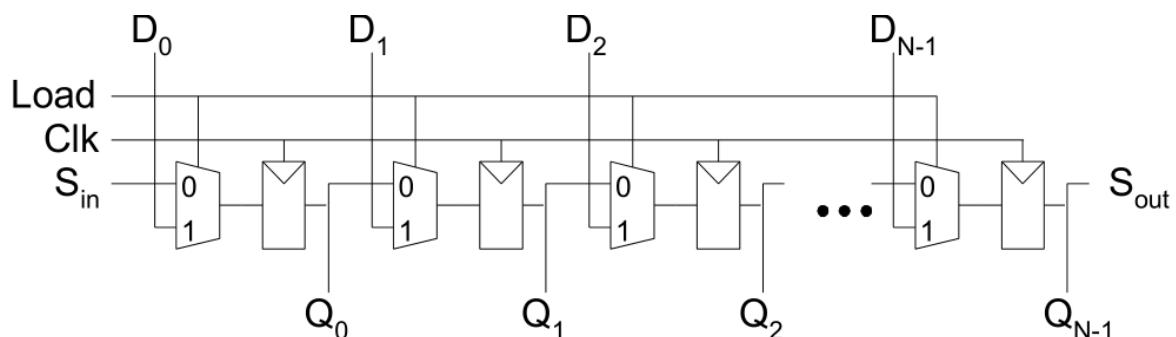
Quando LOAD=1 -> parallelo seriale

i flip-flop vengono caricati in parallelo a partire dall'ingresso D₀ fino a D_{N-1}. funziona come un usuale registro a N-bit, leggendo poi le uscite in serie una alla volta su Sout
Sei disabilitato grazie al mux

Quando LOAD=0 -> seriale parallelo

il registro effettua la normale traslazione seriale-parallelo

l'input viene letto un bit alla volta e poi alla fine le uscite vengono mostrate in parallelo Q_{0:N-1}



Memory arrays

Lo scopo delle memorie è quello di memorizzare efficacemente una grossa quantità di dati
Esistono 3 tipologie di memorie che immagazzinano grandi quantità di dati:

- **Dynamic random access memory (DRAM)**
- **Static random access memory (SRAM)**
- **Read only memory (ROM)**

Ognuna di queste memorie è differente nella modalità di immagazzinamento dei dati

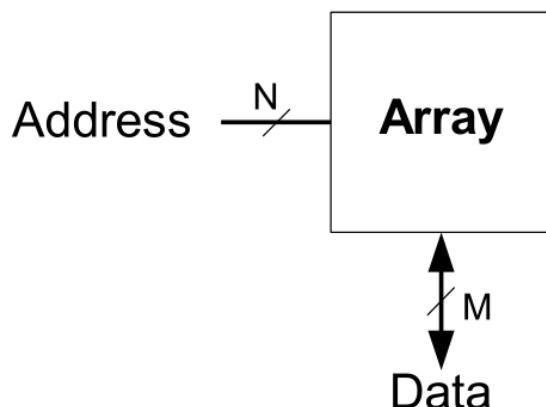
La memoria è organizzata come array bidimensionale di celle di memoria

In ingresso abbiamo n bit che identificano un certo **indirizzo** di una locazione di memoria nella quale si può leggere o scrivere il contenuto di una riga dell'array bidimensionale di memoria detta parola (tipicamente di 8 bit=1 byte).

Il valore letto o scritto nella memoria viene chiamato data ed è una parola di m bit.

Tipicamente m = 8 ossia 1 byte.

N (indirizzo) = 32

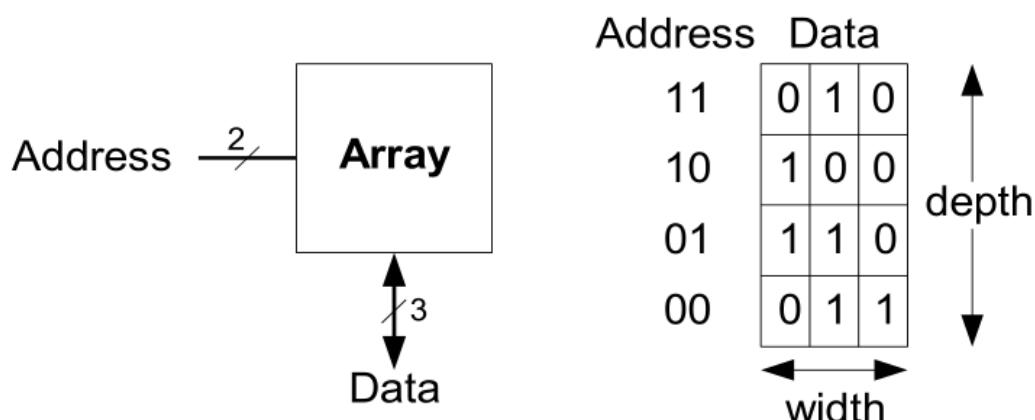


Ogni cella memorizza un bit

Le memorie sono byte addressable perchè ogni byte ha un suo specifico indirizzo

Con N bit di indirizzo e M bits data: 2^N righe e M colonne

- **Depth:** numero di righe (ossia numero di parole)
- **Width:** numero di colonne (lunghezza di una parola)
- **Dimensione Array :** depth × width = $2^N \times M$



- $2^2 \times 3$ -bit array
- Numero parole: 4
- Lunghezza parola: 3-bits
- All'indirizzo 10 corrisponde la parola 100

Bitline e Wordline

Le memorie vengono realizzate come matrici, ogni elemento della matrice è una cella di memoria che può contenere un bit di dato.

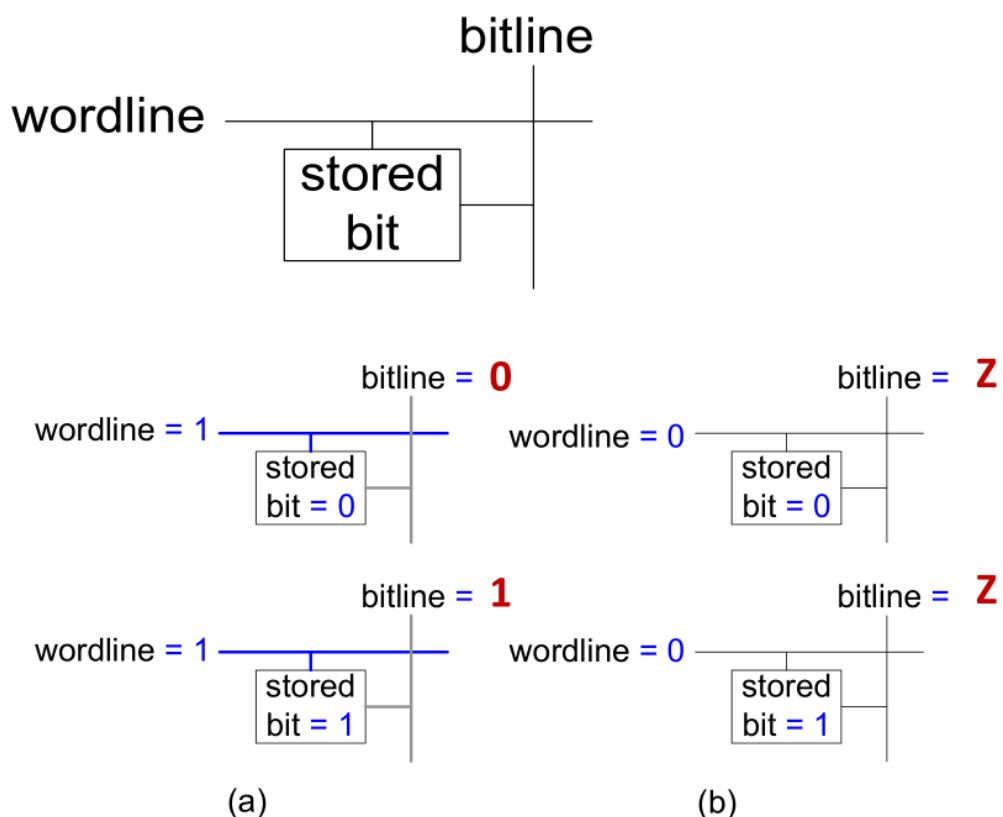
La cella di memoria è un elemento circuitale realizzato in modo diverso a seconda delle memorie utilizzate. Ogni cella di memoria è connessa ad una bitline (il cui insieme costituisce i valori di "data") ed è collegata ad una wordline che consente l'attivazione di una data parola. Ad ogni parola corrisponde una wordline

Per ogni configurazione dei bit di indirizzo, la memoria attiva una wordline che, a sua volta, attiva le bitline presenti nella riga corrispondente.

Quando la wordline = 1, il bit memorizzato viene inviato alla bitline o prelevato dalla stessa. Altrimenti, la bitline è disconnessa dalla cella in cui è memorizzato il bit.

Quando la wordline=1 e il bit memorizzato è 0 nella cella, leggeremo sulla bitline 0, altrimenti se il bitline memorizzato è 1 leggeremo 1 sulla bitline

Quando la wordline=0 la wordline non è attivata quindi è come se il bit fosse scollegato dalla sua bitline che si trova relativamente a questo bit in uno stato di alta impedenza

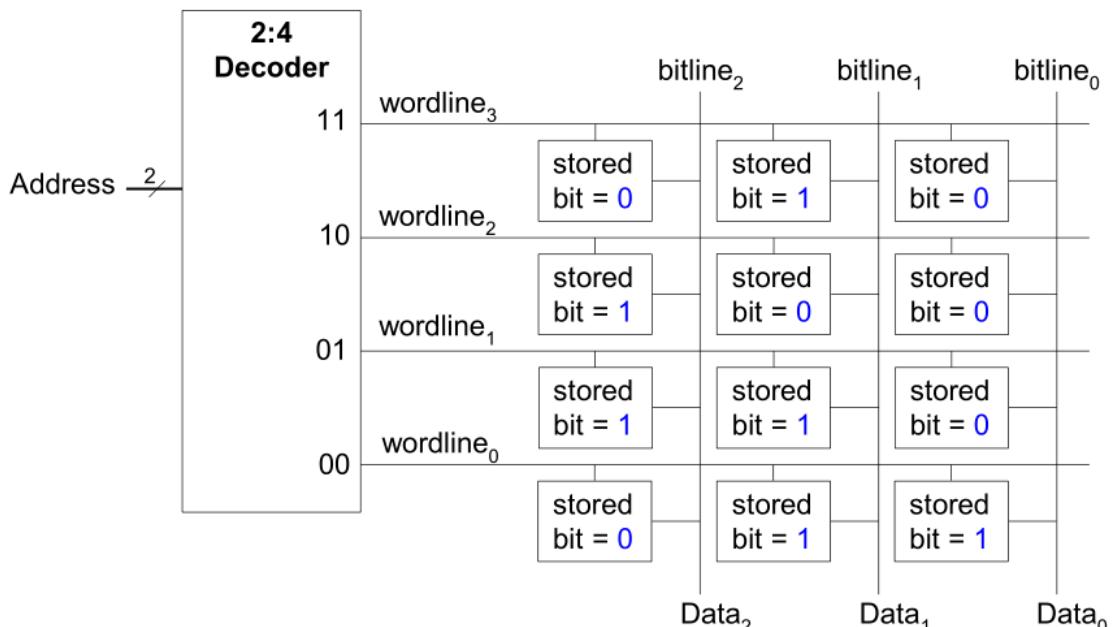


Organizzazione della memoria

Un array di memoria dal punto di vista circuitale è formato da una serie di N ingressi relativi all'indirizzo, poi vi è un decoder che ad ogni valore dell'indirizzo in input attiva la wordline corrispondente, che ci consente di leggere/scrivere alla wordline selezionata con le relative bitline. Le altre wordline sono scollegate temporaneamente

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): non volatile**

RAM

La RAM è una memoria volatile nel senso che i dati sono persi allo spegnimento del computer. Quindi perderemo i valori delle parole all'interno della memoria.

Caratteristica della RAM è che le operazioni di lettura e scrittura sono più veloci rispetto alle ROM. Di fatto la memoria RAM è la memoria principale di un computer.

Per eseguire un programma si carica dalla memoria di massa il programma nella memoria ram, insieme ai dati relativi all'esecuzione e poi il programma viene eseguito

Viene chiamata random access memory perchè il tempo per accedere ad una parola ad un certo indirizzo è uguale al tempo di accesso di altre parole locate ad indirizzi diversi. Il tempo di accesso ad ogni singola parola è costante indipendentemente dall'indirizzo

Le operazioni di lettura e scrittura sono più veloci (rispetto alle ROM)

ROM

Per ROM si intende una memoria non volatile, cioè una memoria che mantiene i dati contenuti al suo interno anche quando il pc è spento.

Inizialmente venivano chiamate di sola lettura perchè scritte attraverso dei fusibili cioè elementi circuitali che una volta bruciati non era più possibile modificare i contenuti della memoria

La lettura è relativamente veloce (meno della ram), la scrittura meno

Tipi di RAM: DRAM e SRAM

Le ram le dividiamo in 2 grandi categorie che si differenziano per le componenti usate per realizzare le singole celle di memoria e quindi memorizzare i dati:

- **DRAM (Dynamic random access memory) usano dei CONDENSATORI/CAPACITÀ**
- **SRAM (Static random access memory) usano INVERTITORI**

DRAM

I bit di una cella di memoria sono memorizzati in delle capacità/condensatori

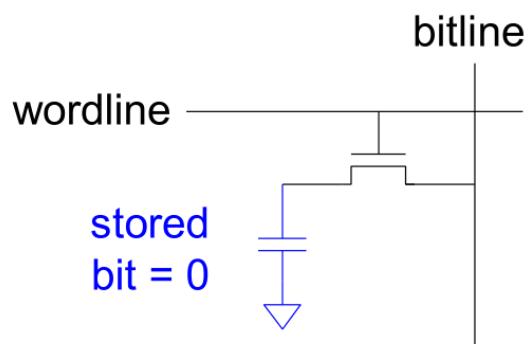
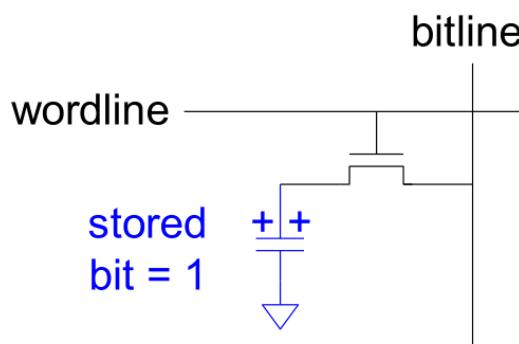
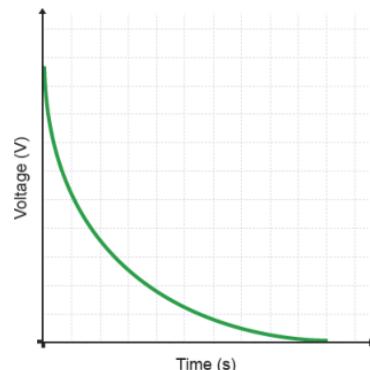
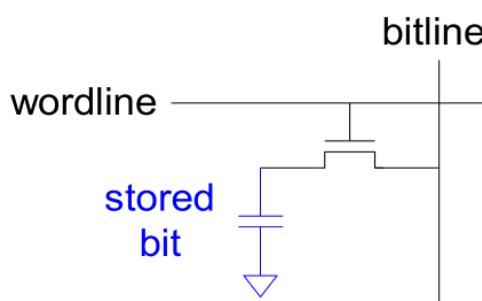
I bit sono quindi memorizzati come presenza o assenza di carica nel condensatore

Il valore del bit viene memorizzato in un condensatore.

C'è un transistor che si comporta come un interruttore che connette o disconnette il condensatore dalla bitline. Quando wordline è attiva, il transistor si accende e il valore del bit immagazzinato viene trasferito alla o dalla bitline.

quando il condensatore viene caricato a VDD, il bit immagazzinato è 1; quando invece viene scaricato fino a GND ,il bit immagazzinato è 0.

Dynamic perché il valore di un bit deve essere “refreshed” (riscritto) periodicamente e anche dopo che è stato letto: ciò è dovuto al fatto che la perdita di carica di una capacità degrada il valore memorizzato e che la lettura distrugge il valore letto



Un condensatore possiamo paragonarlo ad una vasca cioè un elemento che ha la capacità di immagazzinare le cariche elettriche. La carica elettrica è il fluido all'interno della vasca.

Il livello del fluido corrisponde al potenziale (alto o basso). Quando abbiamo un valore di potenziale alto VDD è perchè vi è una differenza di potenziale che genera corrente quindi un movimento di elettroni e cariche elettriche che riempiono il condensatore “la vasca” fino ad un certo livello di potenziale VDD.

Quando il livello è uguale a VDD non vi è più flusso di elettroni e la corrente cessa.

Il condensatore si trova in uno stato logico = 1.

Se abbiamo uno stato logico = 0 non vi è alcun flusso di corrente e il condensatore rimane scarico.

I condensatori hanno delle "perdite" nel senso che il valore del potenziale tende a degradarsi (ossia ritornare a 0) e non si mantiene costante a VDD . Quindi se voglio mantenere un bit pari a 1 devo di tanto in tanto refreshare la memoria , cioè ribadirei il valore della cella di memoria in particolare per i bit che hanno un valore di memoria alto.

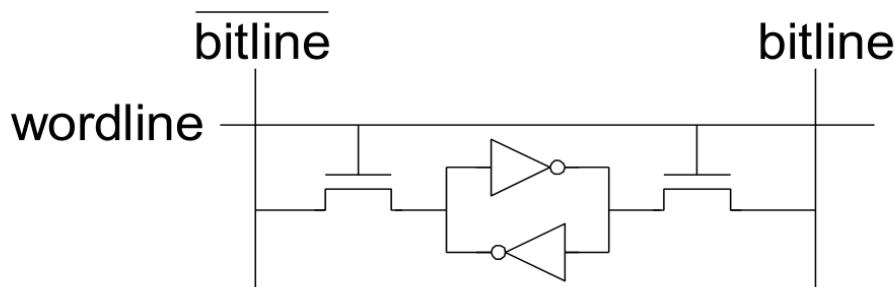
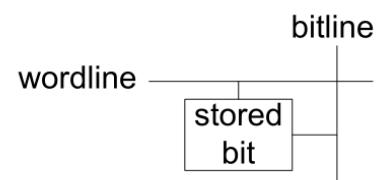
Altrimenti dopo pochi ms tutti i bit tornerebbero a 0.

Transistor

L'operazione di collegare/scollegare la stored bit (cella) alla bitline dipende da una componente circuitale detta transistor cmos
è un interruttore che quando la wordline è uguale a 1 è chiuso e quindi il condensatore è collegato alla rispettiva bitline
quando il il valore della wordline è 0 il transistor corrisponde ad un interruttore aperto e quindi il condensatore è scollegato dalla bitline
Serve allora per controllare l'accesso alla cella

SRAM

Una RAM statica (SRAM, Static RAM) viene chiamata in questo modo perché i bit immagazzinati nella memoria non hanno bisogno di essere refreshati
Viene realizzata attraverso un circuito bistabile costituito da due porte NOT invertite
La bitline viene sdoppiata in due linee bitline e *bitline che hanno sempre valori complementari
Quando bitline=1 , *bitline=0 e viceversa
Queste due linee sono collegate alla memoria attraverso 2 transistor che attivano o disattivano la cella di memoria
Quando la wordline viene attivata, entrambi i transistors si accendono e i bit di dato vengono trasferiti da o verso le linee di bit.
Gli invertitori ribadiscono il valore del potenziale in uno dei due punti è come se il dato fosse intrappolato



DRAM vs SRAM confronto

Le DRAM sono più lente perchè il condensatore ci mette un po' di tempo per riempirsi fino ad un certo livello logico o per scaricarsi per passare da un livello logico 1 a 0. Questi tempi di latenza di carica e scarica rendono questo tipo di memorie più lente rispetto alla SRAM realizzata con gli invertitori che sono più veloci e reattivi e quindi le sram sono più reattive e veloci

Le dram devono essere refreshate e riscritte e quindi bisogna fare un refresh ogni pochi ms ciò induce un maggior consumo di energia ma esse sono più economiche delle SRAM perchè necessitano di un solo transistor per costruirle mentre le sram 6 transistor

I flip flop hanno una 20ina di transistor e sono ancora più costosi delle dram e sram. Sono i più veloci come tempi di risposta e latenza poi abbiamo le SRAM e le DRAM che sono le più lente
I flip flop sono utilizzati per costruire i registri che usa la CPU

La SRAM viene utilizzata per costruire la memoria cache

La DRAM viene utilizzata per costruire la memoria RAM del PC la main memory

Le DRAM si dicono DDR (double data rate) Synchronous RAM perchè le operazioni di scrittura e lettura sono sincronizzate da un clock e avvengono sia sul fronte alto che sul fronte basso del clock. La Data bandwidth è raddoppiata rispetto alla frequenza di clock = double pumping

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

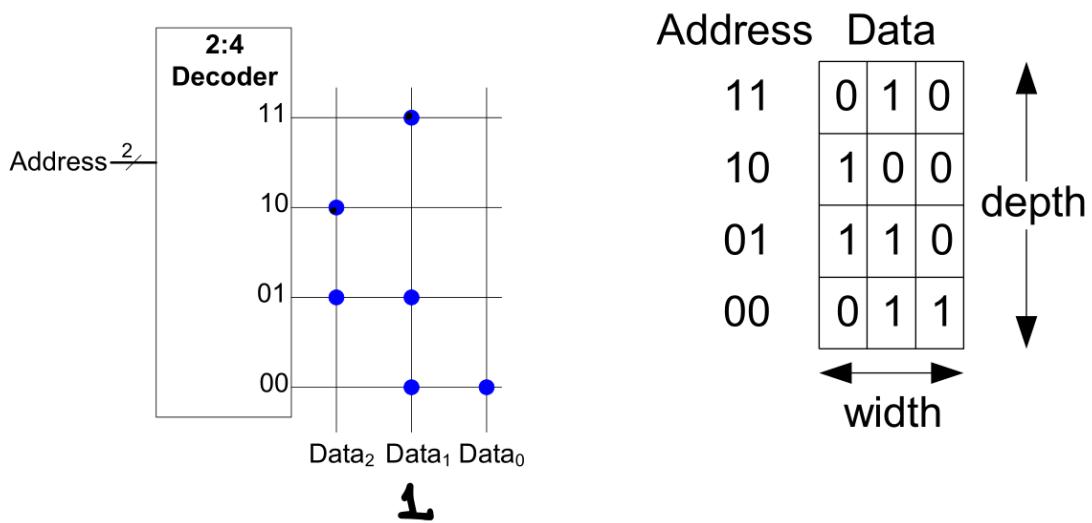
DDR SDRAM

- (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 che falling-edge ($1 \rightarrow 0$)
- In questo modo il data-bandwidth doppio rispetto alla frequenza di clock (double pumping)

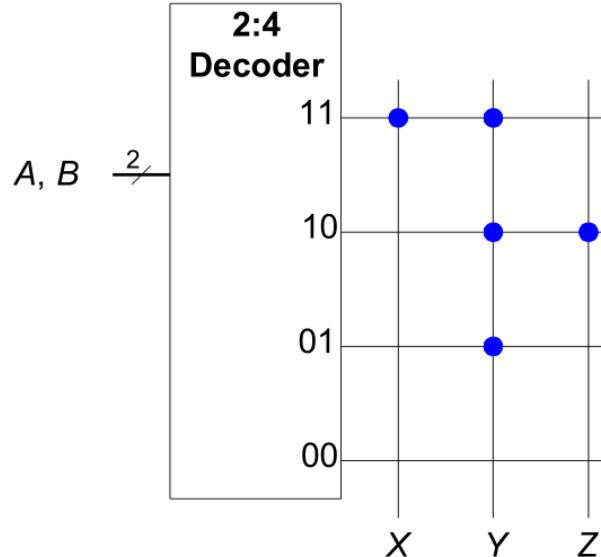
ROM

Il contenuto di una ROM può essere indicato con la notazione a punti.. Un punto posto all'intersezione tra una riga (linea di parola) e una colonna (linea di bit) indica che il bit di dato è uguale a 1. Per esempio, la prima linea di parola ha un unico punto su Dato1, quindi la parola memorizzata all'Indirizzo 11 è 010.

Concettualmente, le ROM possono essere costruite utilizzando la logica a due livelli, con una serie di porte AND seguite da una serie di porte OR. Le porte AND producono tutti i mintermini possibili, quindi vanno a formare un decoder. Ogni punto nella corrisponde a una linea di ingresso a una porta OR . Per linee di bit con un solo punto, in questo caso Dato0, non è necessaria una porta OR. Questa rappresentazione di una ROM è particolarmente interessante perché mostra come la ROM sia in grado di realizzare una qualsiasi funzione logica a due livelli.



- $X = AB$
- $Y = A + B$
- $Z = A \bar{B}$



5. MICROARCHITETTURA

L'architettura rappresenta la struttura di un calcolatore dal punto di vista di un programmatore. È definita dal set di istruzioni e dagli operandi che vengono codificati in linguaggio macchina. Al linguaggio macchina corrisponde un linguaggio assembly ossia un formato human-readable delle istruzioni, rispetto a quello macchina che è un formato computer-readable (1 e 0). Esistono molte architetture differenti tra cui ARM e X86

La **MICROARCHITETTURA** rappresenta la congiunzione fra i circuiti logici e l'architettura: **definisce la disposizione specifica di registri, ALU, macchine a stati finiti, le memorie e altri blocchi logici necessari per implementare una architettura**

Due microarchitetture possono realizzare la medesima architettura ma con soluzioni tecnologiche e prestazioni molto differenti. Ad esempio Intel e AMD realizzano diversi modelli (microarchitetture) della medesima architettura x86.

Anche l'architettura ARM può avere diverse microarchitetture, caratterizzate da diversi rapporti prestazioni/costo/complessità. Tutte devono essere in grado di eseguire gli stessi programmi, ma la loro struttura interna può essere anche molto diversa.

L'architettura ARM è stata sviluppata negli anni 80 dalla Advanced RISC Machines ed è impiegata soprattutto nel settore degli smartphone e tablet.

Principi di design dell'architettura ARM

L'architettura ARM si basa su 4 principi di progettazione fondamentali RMSG:

1. Regularity supports design simplicity (la regolarità favorisce la semplicità)
2. Make the common case fast (rendere veloci le cose frequenti)
3. Smaller is faster (più piccolo è più veloce)
4. Good design demands good compromises (un buon progetto richiede buoni compromessi).

Applicazione del 1° principio ARM: Regularity supports design simplicity

Il **formato costante delle istruzioni arm** è un'applicazione del primo principio Regularity supports design simplicity.

Ogni **istruzione ARM** 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).

In ogni istruzione inoltre è presente un **numero costante di operandi: di solito due operandi sorgenti e un operando destinazione**.

Questo rende le cose più facili da gestire in hardware, perché **i formati delle istruzioni sono consistenti, e l'encoding in hardware è dunque facilitato**.

Applicazione del 2° principio ARM: Make the common case fast

L'uso di **più istruzioni semplici assembly per eseguire attività complesse** è un esempio di applicazione del secondo principio: rendere veloci le cose frequenti.

Infatti l'architettura **ARM** rende veloci le cose frequenti perché **comprende solo istruzioni semplici e di uso frequente**. Il numero di istruzioni diverse è tenuto basso in modo tale che il circuito **hardware richiesto per decodificare ed eseguire le istruzioni sia semplice e veloce**.

Elaborazioni più complesse eseguite più raramente sono realizzate con **sequenze di molteplici istruzioni ARM semplici**. Dunque istruzioni di alto livello più complesse vengono tradotte in più sequenze di semplici istruzioni arm (*ad es. se ho una somma e una sottrazione in sequenza effettuo prima la somma e poi sul risultato ottenuto effettuo la sottrazione ottenendo il risultato finale*).

ARM quindi si basa su un'architettura RISC che ha un insieme limitato di istruzioni semplici. Architetture con molte istruzioni complesse si basano su un architettura CISC (complex instruction set computer) esempio x86. CISC comporta hardware aggiuntivo e sovraccarichi che rallentano anche le istruzioni semplici. **L'architettura RISC minimizza la complessità hardware e la codifica necessaria per le istruzioni mantenendo piccolo l'insieme di diverse istruzioni.**

Applicazione del 3° principio ARM: Smaller is faster

I dati nell' architettura ARM possono essere memorizzati:

- Direttamente all'interno di una istruzione:** si definiscono costanti (**immediates**)
- In registri:** ARM ha solo **16 registri, ognuno costituito da 32 bit**, che sono più veloci della memoria
- In memoria:** Più lenta dei registri ma più capiente, 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.

Infatti le istruzioni hanno bisogno di raggiungere rapidamente gli operandi per poter essere eseguite velocemente, ma gli operandi salvati nella memoria richiedono tempi lunghi di accesso in memoria per poter essere recuperati. Per questo vengono definiti un numero limitato di **registri per memorizzare gli operandi più usati**. ARM ha 16 registri globalmente indicati come **register file**. **Meno sono i registri e più rapidamente sono accessibili, questa è un'applicazione del 3° principio più piccolo è più veloce. Infatti leggere un dato da pochi registri è molto più rapido che leggerlo da una memoria grande.** Il register file di solito è costituito da un array di memoria SRAM.

Applicazione del 4° principio ARM: Good design demands good compromises

ARM usa istruzioni da 32 bit. Siccome regolarità garantisce semplicità, la scelta che da la massima regolarità è quella di destinare **una parola di memoria a ciascuna istruzione**, anche se non tutte le istruzioni hanno bisogno di 32 bit per essere codificate. Usare istruzioni di lunghezza variabile vorrebbe dire aumentare inutilmente la complessità circuitale.

La semplicità suggerirebbe di avere un solo formato per le istruzioni, ma ciò sarebbe troppo restrittivo non si riesce in un unico formato a descrivere le particolarità di ogni istruzione: questo consente di introdurre il quarto principio ossia un buon progetto richiede buoni compromessi.

ARM sceglie il compromesso di avere 3 possibili formati di istruzione:

- **Istruzioni di Data-processing;** processano i dati memorizzati nei registri
- **Istruzioni di Memory;** che si dividono in istruzioni di LOAD o STORE
- **Istruzioni di Branch.** istruzioni di salto da un punto all'altro del programma

Il numero limitato di formati consente di avere una certa regolarità tra le varie istruzioni, quindi di semplificare i circuiti di decodifica pur soddisfacendo le esigenze di diverse istruzioni.

Cos'è un processore general purpose

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

Lo **stato (architetturale)** 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.

A partire dallo stato corrente il processore esegue una particolare istruzione su un particolare insieme di dati per produrre un nuovo stato architetturale.

Una microarchitettura quindi può essere vista come un automa

L'automa si trova inizialmente in un certo stato dato dal valore del clock, dai valori memorizzati nelle locazioni di memoria, nei registri, del program counter.

Quando viene eseguita una nuova istruzione che cambia il contenuto della memoria, dei registri ecc, si passa ad un nuovo stato dell'automa e così via per ogni istruzione.

Il numero di stati in cui può trovarsi una microarchitettura è esorbitante.

Variazioni di stato

I registri, la memoria dati e la memoria istruzioni operano mediante logica combinatoria.

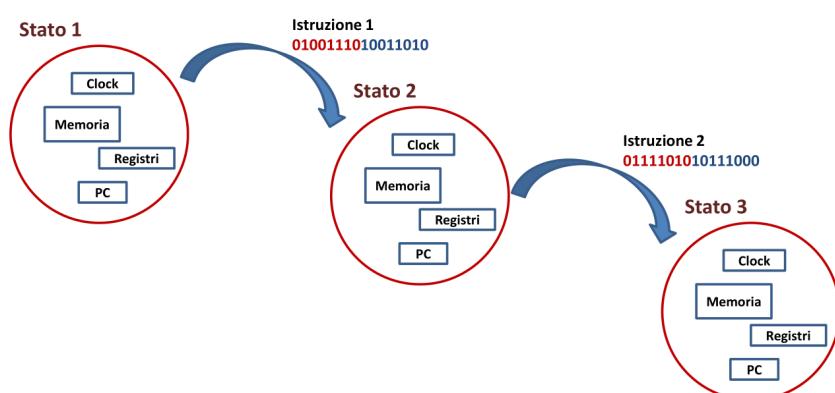
Le transizioni degli stati sono regolate da un clock che da sincronicità a tutto il sistema.

In particolare tutti i componenti effettuano la **scrittura sul fronte alto del clock**, cosicché **lo stato del sistema cambia solo su un fronte del clock (quello alto)**.

Per scrivere sui registri , memorie ecc. ci sono degli **abilitatori** che consentono di scrivere/leggere dai registri/memoria. Prima di eseguire una istruzione gli abilitatori devono essere impostati in maniera corretta.

Quindi gli indirizzi, i dati ed il segnale di write enable devono essere impostati prima del fronte alto del clock e mantenuti immutati per un tempo superiore al ritardo di propagazione.

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.



Tipologie di microarchitettture

Esistono 3 tipologie di microarchitettture per l'architettura ARM:

- **microarchitettura a ciclo singolo**
- **microarchitettura a ciclo multiplo (multiciclo)**
- **microarchitettura pipeline**

Esse differiscono su come vengono eseguite le istruzioni e per il modo in cui i vari elementi di stato sono connessi tra loro.

Microarchitetture a ciclo singolo

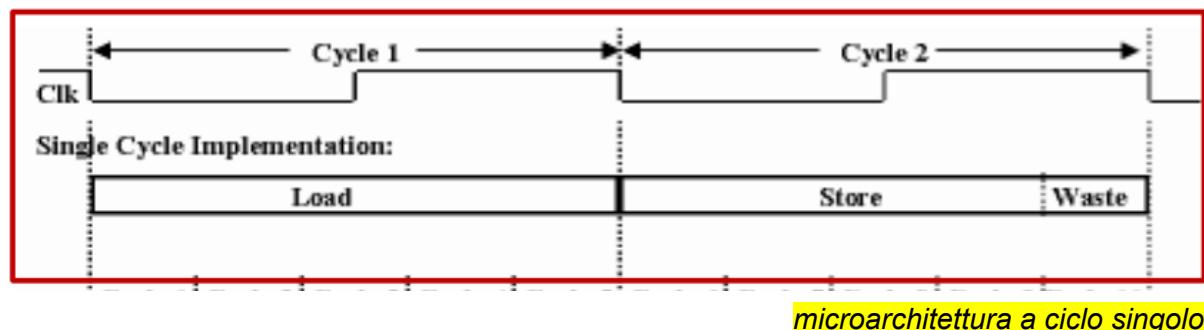
Sono le più semplici. **Un'intera istruzione è eseguita in un singolo ciclo di clock.**

Vantaggi:

- semplice da comprendere;
- unità di controllo molto semplice;
- non richiede stati non architettonici (gli unici stati del sistema sono quelli dati dai valori correnti della memoria, registri ecc.)

Svantaggi:

- **periodo di clock pari a quella dell'istruzione più lenta;** (la frequenza o il periodo di clock del clock deve essere almeno pari a quello dell'istruzione più lenta)
- **memoria dati e memoria istruzioni separate;**



Microarchitetture a ciclo multiplo

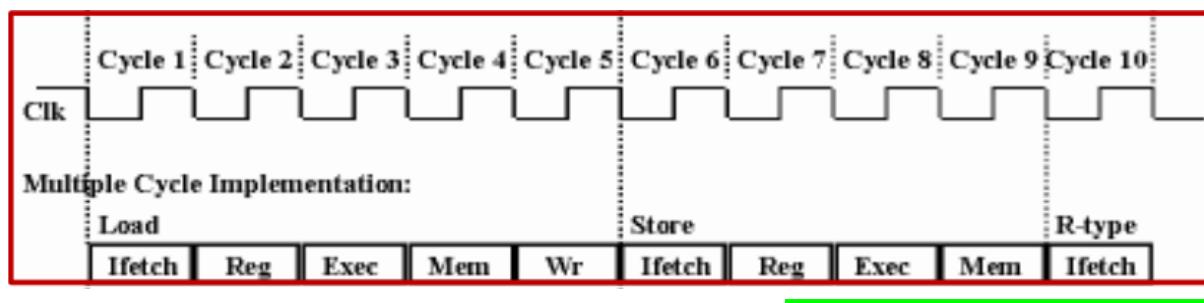
Un'istruzione è eseguita in più cicli di breve durata.

Vantaggi:

- riuso dei componenti (riduce il costo hardware riutilizzando blocchi circuitali costosi come i sommatori e le memorie);
- durata variabile delle istruzioni (le istruzioni più semplici vengono eseguite in meno cicli di quelle più complesse);
- non richiede la separazione delle memorie

Svantaggi:

- richiede stati non architetturali
- esegue una sola istruzione alla volta ma ogni istruzione richiede più cicli di clock per essere completata



microarchitettura a ciclo multiplo

Microarchitetture con pipeline

Sono le più avanzate. **Eseguono più istruzioni contemporaneamente migliorando sensibilmente le prestazioni.**

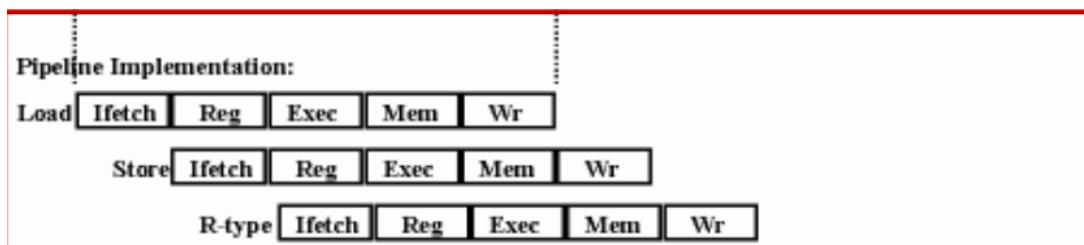
Si applica il concetto di pipelining al processore a ciclo singolo, migliorandone le performance. Ogni istruzione viene suddivisa in varie fasi e nel ciclo, mentre si esegue una fase di un'istruzione inizia anche una fase dell'istruzione successiva.

Vantaggi:

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

Svantaggi:

- logica di controllo più complessa
- richiede registri di pipeline



microarchitettura con pipeline

Misura delle prestazioni

Il processore può avere diverse microarchitetture con diversi rapporti costo/prestazioni.

Il costo dipende dalla quantità di hardware necessaria e dalla tecnologia di realizzazione.

Più ci sono porte logiche e memoria e più i costi sono maggiori.

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

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

In generale il tempo di esecuzione di un programma (in secondi) si calcola secondo la seguente formula:

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

Il numero di istruzioni in un programma dipende dall'architettura del processore, ma anche dall'abilità del programmatore.

Il numero di cicli per istruzione è detto CPI (Cycles Per Instruction) ed è il numero di cicli di clock richiesti in media per eseguire un'istruzione.

Il numero di secondi per ciclo è il periodo del clock (clock period) ed è determinato dal percorso critico attraverso i circuiti del processore. La **frequenza di clock** è l'inverso del periodo di clock ed è data dal numero di cicli per secondo (**cycle/seconds**).

L'inverso del CPI è la potenza di elaborazione (**IPC Instruction Per Cycle**) (instructions/cycle)

Microarchitetture: schema generale

Un PROCESSORE è costituito da:

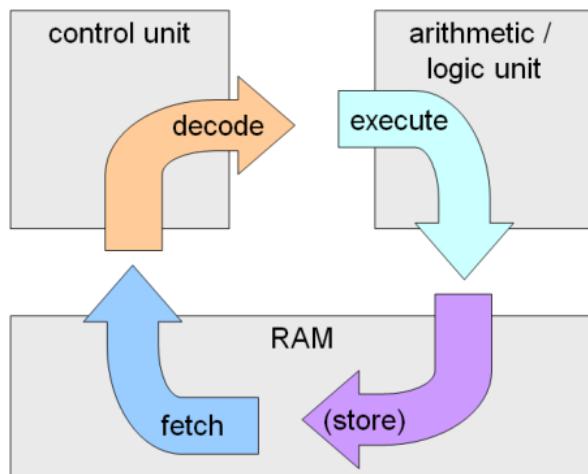
- un **datapath** ossia un circuito che realizza il flusso dei dati e l'esecuzione materiale dell'istruzione
- dalla **control unit** che istruisce come il datapath deve operare a seconda della tipologia d'istruzione

Una **istruzione** ha un **ciclo di vita** che consta di **4 fasi** principali: (fdes)

- 1) **fetch**
- 2) **decode**
- 3) **execute**
- 4) **store**

- 1) Nella fase di **FETCH** dell'istruzione, partendo da un registro che contiene l'indirizzo dell'istruzione corrente (cioè quella da eseguire) - PC - si va a ricercare in memoria tale istruzione e la si carica in un registro apposito (32 bit) dove ci sarà il contenuto di tale istruzione
- 2) Nella fase di **DECODE** dell'istruzione, eseguita da un'unità del processore chiamata **control unit CU**, essa decodifica l'istruzione e istruisce il circuito datapath su come deve operare a seconda della tipologia di istruzione da eseguire

- 3) Nella fase di **EXECUTE** dell'istruzione essa viene eseguita. Bisogna ricercare nei registri appositi gli operandi di tale istruzione , poi essa viene eseguita , calcolando il risultato mediante l'**ALU** che si occupa dei calcoli aritmetici e logici dell'istruzione
- 4) Nella fase di **STORE** dell'istruzione il risultato finale dell'istruzione viene memorizzato in memoria (ad esempio all'interno dei registri del file register che sono i registri presenti all'interno della cpu)



Progettazione di una microarchitettura ARM

Una microarchitettura consta di due componenti che **interagiscono** fra loro:

- **il datapath:** è un circuito che 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. E' a 32 bit.
- **l'unità di controllo:** riceve l'istruzione corrente dal datapath ed "istruisce" il datapath su come eseguire tale istruzione. "Istruisce" significa che definisce dei valori di selezione dei vari multiplexer del datapath e poi stabilisce i segnali di abilitazione di varie strutture quali le memorie, il file register.

L'unità di controllo riceve l'istruzione corrente dal datapath e comunica ad esso come eseguirla, attivando opportunamente gli ingressi di selezione dei multiplexer, le abilitazioni dei registri e i segnali di lettura e scrittura in memoria per controllare le operazioni del datapath.

I 5 componenti fondamentali di un processore (elementi di stato del datapath)

La memoria è suddivisa in 5 ELEMENTI DI STATO (pc, rf, sr, im, dm):

1-program counter

2-i registri (register file) ossia l'insieme di registri a 32 bit che corrispondono alla memoria operativa su cui l'alu opera

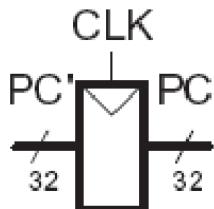
3-il registro di stato (status register) che è un registro a 4 bit

4-la memoria istruzioni (instruction memory)

5-la memoria dati (data memory)

A questi elementi di stato si aggiungono blocchi di logica combinatoria per generare il nuovo stato dell'automa a partire dallo stato corrente.

1. PROGRAM COUNTER



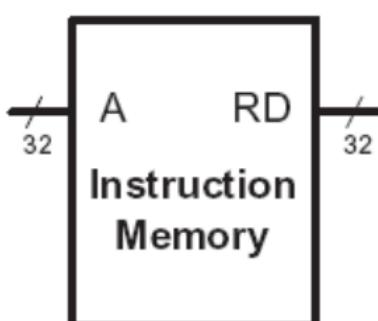
Il PC è un registro a 32 bit sincronizzato dal clock il cui scopo è quello di memorizzare di volta in volta l'istruzione corrente da eseguire. Ad ogni ciclo di clock conterrà l'istruzione che deve essere eseguita e poi si aggiornerà in modo tale che al prossimo clock conterrà l'istruzione successiva da eseguire.

Esso fa logicamente parte del register file, che contiene tutti i registri, ma viene letto e scritto ad ogni ciclo indipendentemente dagli altri registri ed è quindi implementato come un registro autonomo 32-bit.

La sua **uscita a 32 bit, PC, indica l'indirizzo dell'istruzione corrente.**

Il suo **ingresso, PC', indica l'indirizzo della successiva istruzione da eseguire.**

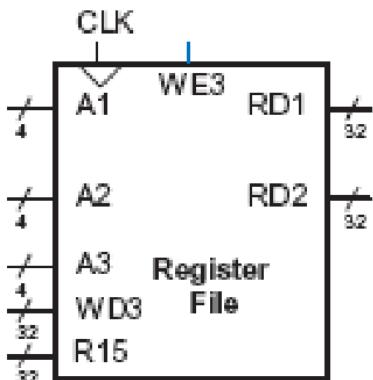
2. INSTRUCTION MEMORY



L' **instruction memory** è la memoria istruzioni. Essa ha una sola porta di lettura. Riceve in ingresso l'**indirizzo di un'istruzione a 32 bit, A**, ed emette sull'**uscita di lettura RD ReadData** (ossia un registro) il dato, appunto l'istruzione, contenuta nella parola di indirizzo A.

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? 2 alla 32 istruzioni cioè circa 4 miliardi di istruzioni

3. FILE REGISTER



Il register file consiste di 16 registri (da R0 a R15) di 32 bit ciascuno

Dal punto di vista FISICO i registri sono equivalenti. Dal punto di vista LOGICO non sono equivalenti, svolgono al livello architetturale durante l'esecuzione di un programma funzioni particolari.

R13: registro SP STACK POINTER

R14: registro LR LINK REGISTER

R15: registro PC PROGRAM COUNTER (proveniente dal program counter, contiene l'indirizzo dell'istruzione successiva a quella contenuta nel PC vero e proprio)

L'ingresso di R15 è separato perché è collegato direttamente al PC da cui riceve l'indirizzo corrispondente alla prossima istruzione da eseguire e con il quale è sincronizzato

Vi sono diversi ingressi e uscite.

Ci sono due porte di lettura A1 e A2 e una porta di scrittura A3.

Le porte di lettura A1 e A2 ricevono in ingresso un input di 4 bit (2 alla 4=16), che specificano l'indirizzo di uno dei 16 registri come operando sorgente.

Il valore a 32 bit dei registri indirizzati viene letto sulle uscite di lettura dati RD1 e RD2.

Si utilizza evidentemente un MUX per selezionare il registro da cui leggere il valore a 32 bit, dati gli ingressi A1 e A2

Gli ingressi A3 e WD3 servono per memorizzare quindi "scrivere" un dato in un registro.

Costituiscono la porta di scrittura. Essa ha:

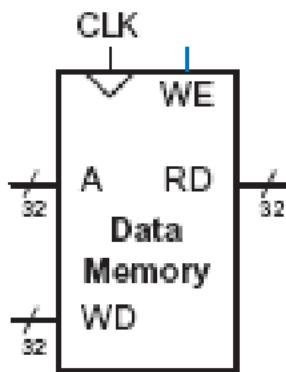
L'ingresso A3 che riceve un indirizzo di 4 bit che specifica il registro nel quale scrivere il dato.

L'ingresso WD3 che riceve il dato vero e proprio da memorizzare, a 32 bit.

Un segnale di Write Enable, di abilitazione alla scrittura che sarebbe WE3

Se il segnale di abilitazione alla scrittura **WE3 = 1** e quindi è attivo il dato specificato da WD3 viene memorizzato (scritto) nel registro specificato da A3 in corrispondenza del fronte di salita del clock.

4. DATA MEMORY



Il **Data Memory** ha una singola **porta di lettura/scrittura, A, a 32 bit**.

Il contenuto da scrivere in memoria viene dato all'ingresso **WD**.

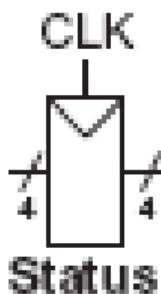
Quando il segnale **WE = 1** (Write Enable) è attivo, essa scrive la parola contenuta all'ingresso WD nella cella puntata dall'indirizzo A durante il fronte alto del clock.

Si prende l'indirizzo A, e si scrive nella cella di memoria corrispondente a tale indirizzo la parola che si trova in WD.

Quando il segnale **WE = 0** (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 in lettura.

Si prende l'indirizzo A, lo si decodifica, si legge il contenuto della cella di memoria e lo si riporta in RD.

5. STATUS REGISTER



Ha 4 ingressi in input usati per memorizzare lo stato dei flags **N,Z,C,V** forniti dall'ALU al fine di eseguire istruzioni condizionate.

Non sempre i flags in output dell'alu vengono memorizzati nello status register. solo per alcune istruzioni.

Questo registro permette di definire le **istruzioni condizionali** cioè istruzioni assembly che vengono eseguite solo se vengono soddisfatte alcune condizioni.

N=Negative

Z=Zero

C=Carry

V=Overflow

Istruzioni di base datapath

Studiamo il datapath di alcune istruzioni principali dell'architettura ARM, considerando un insieme limitato del set di istruzioni.

-**Istruzioni di data-processing:** ADD, SUB, AND, ORR (con registri e modalità di indirizzamento diretto, senza shift)

-**Istruzioni di memoria:** LDR, STR (con positive immediate offset)

-**Istruzioni di salto:** B

Indirizzi di memoria

La memoria è organizzata come un vettore di parole. L'architettura arm usa 32 bit per gli indirizzi di memoria e 32 bit per le parole (dati). La memoria si dice **byte-addressable** perché ad ogni byte di memoria (8 bit) corrisponde un unico indirizzo univoco.

Nell'architettura ARM operiamo con parole di 32 bit, che sono 4 byte (8 bit ciascuno).

Questo vuol dire che gli indirizzi delle parole incrementano di 4 in 4.

Il byte più significativo è a sinistra MSB (Most significant byte)

Il byte meno significativo è a destra LSB (Least significant byte)

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

Istruzioni di memoria: LOAD (LDR)

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]

In questo caso calcola un indirizzo a partire dal contenuto del registro **R1 (detto BASE ADDRESS)** sommato ad uno **spiazzamento (offset)** che è **dato da un immediate** (costante) ossia #12. Calcolato questo indirizzo preleva il contenuto locato in main memory a tale indirizzo e lo carica nel registro **R0, registro di destinazione**

R1 e l'immediate #12 sono operandi sorgente.

I dati locati in RAM devono essere caricati nei registri della cpu per eseguire le operazioni

Vi sono anche altre segnature di LDR, ad esempio:

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

R0 \leftarrow mem32[R1 + R2] R0 \leftarrow mem32[R1 + (R2*4)]

Nel primo caso vi è un offset da registro, non è indicata una costante come offset bensì un registr. In questo caso al contenuto del registro R1 che è il base address viene sommato un offset che è dato dal contenuto del registro R2.

Nel secondo caso abbiamo un offset da un registro shiftato.

Al contenuto del registro R1 che è il base address viene sommato un offset che è dato dal contenuto del registro R2 shiftato di 2 posizioni verso sinistra, il che vuol dire moltiplicare il contenuto del registro R2 per 4. ($R2^2$)

Quindi abbiamo 3 tipi di offset: da immediate, da registro, da registro shiftato

Istruzioni di memoria: STORE (STR)

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

Segniture analoghe

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

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

Prende il contenuto di un certo registro sorgente e lo salva in memoria ad un certo indirizzo
Il primo argomento è il registro che contiene la sorgente del dato che deve essere salvato in memoria ad un certo indirizzo.

Istruzioni di memoria: Varianti LDRB, STRB

LDRB/STRB eseguono rispettivamente il load e lo store di un **singolo byte** e non della parola a 32 bit (4 byte)

Organizzazione della memoria: come sono indirizzati i byte all'interno di una parola. Disposizione LITTLE ENDIAN e BIG ENDIAN

Le memoria byte addressable sono organizzate in modalità big-endian o little-endian, che definiscono la numerazione dei byte in una parola.

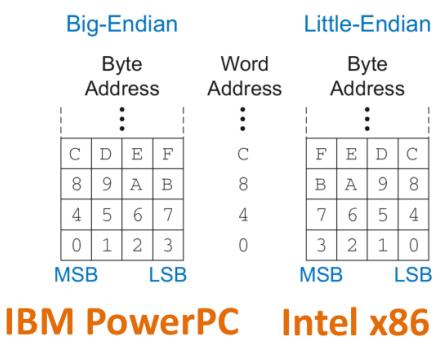
In entrambe il byte più significativo della parola MSB è a sinistra.

Il byte meno significativo della parola LSB è a destra.

little-endian: la numerazione dei byte inizia dal byte meno significativo LSB

big-endian: la numerazione dei byte inizia dal byte più significativo MSB

ARM non stabilisce se il tipo di indirizzamento dei byte in una parola deve essere little o big endian. Scieglie la casa produttrice del processore. Di solito si sceglie LITTLE ENDIAN.



IBM PowerPC Intel x86

Indicizzazione (tipi di indirizzamento)

ARM fornisce tre tipi di indicizzazione: **OFFSET**, **PREINDEX**, **POSTINDEX**

Offset: L'indirizzo è calcolato sommando o sottraendo un offset al contenuto del registro di base. Il registro di base non viene aggiornato ad un nuovo valore. **PW: 10**

Preindex: L'indirizzo viene calcolato come nel caso dell'offset, sommando o sottraendo un offset al contenuto del registro di base. La differenza è che il registro di base viene aggiornato col nuovo indirizzo calcolato. **PW: 00**

Postindex: L'indirizzo corrisponde al contenuto del registro di base.

Dopodiché l'offset viene aggiunto o sottratto al contenuto del registro di base che viene aggiornato col nuovo indirizzo appena calcolato con l'offset. **PW: 11**

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

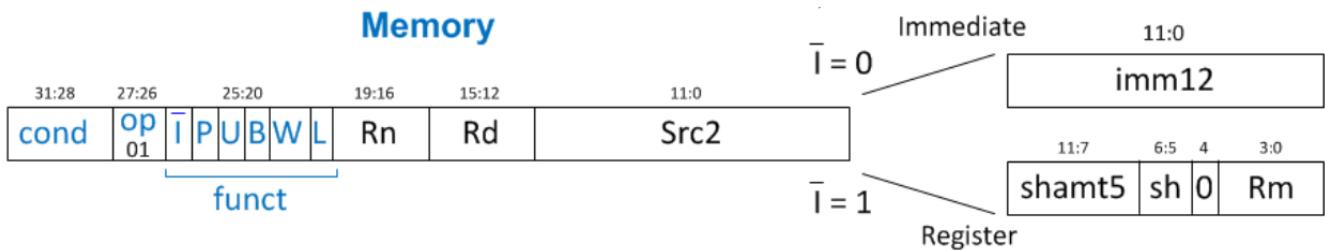
Offset: LDR R1, [R2, #4] ; R1 = mem[R2+4]

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

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

Codifiche istruzioni di memoria: LDR, STR, LDRB, STRB

Vengono codificate in parole a 32 bit.



cond: sono i **4 bit più significativi** dell'istruzione (28:31) e sono i bit di condizione, servono a codificare un'esecuzione condizionata ossia eventuali condizioni che determinano o meno l'esecuzione di questa istruzione.

Le istruzioni di memoria di solito sono **non condizionate** **cond = 1110 (14)**

op: sono **2 bit di operazione**. Siccome esistono 3 categorie di istruzione che hanno formati diversi, questi due bit specificano il formato di istruzione a cui ci si riferisce (istruzione di memoria, data processing o branch).

L'operation code per le istruzioni di memoria è **op = 01 (1)**

funct: sono **6 bit di controllo** ossia { I negato, P, U, B, W, L } corrispondono a significati diversi.

L : è il bit che ci dice se l'istruzione è di tipo LOAD o STORE.

Quando **L=0 abbiamo uno STORE**.

Quando **L=1 abbiamo un LOAD**.

B : è il bit che specifica se l'istruzione deve riferirsi ad una parola o un byte

Quando **B=0 si riferisce ad una PAROLA di memoria**

Quando **B=1 si riferisce ad un solo BYTE di memoria**

P e W : sono 2 bit che specificano il tipo di **INDIRIZZAMENTO** fra quelli possibili (**Postindex, Offset, Preindex**). La combinazione **01** è ridondante.

I negato : è 1 bit che indica se il **tipo di offset è da un immediate o un registro**. Indica come interpretare i 12 bit **Src2**

Quando **I negato=0** abbiamo un offset con **immediate (imm12)**

Quando **I negato=1** abbiamo un offset da **registro (Rm)**

U : è un bit che ci dice se l'offset va sommato o sottratto dal base address

Quando **U=0 l'offset viene sottratto** dal base address

Quando **U=1 l'offset viene sommato** al base address

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

Rn: sono 4 bit che specificano l'indirizzo del BASE ADDRESS, ossia il primo operando dell'istruzione di memoria, che è uno dei 16 registri (per questo 4 bit)

Rd: sono 4 bit che specificano l'indirizzo del primo argomento delle istruzioni di memoria.

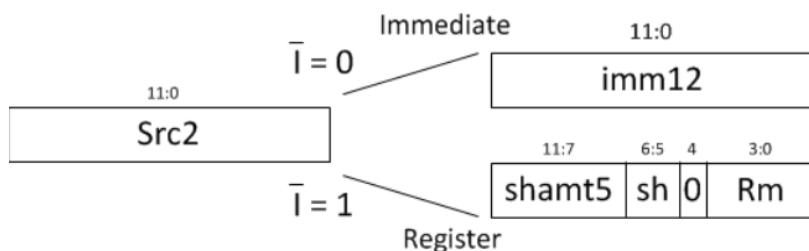
Se l'istruzione è di tipo **LDR (L=1)** questo è il **registro di destinazione**.

Se l'istruzione è di tipo **STR (L=0)** questo è il **registro sorgente** il cui contenuto viene memorizzato in memoria.

Src2: sono 12 bit che specificano il secondo operando delle istruzioni di memoria ossia l'offset. Questi 12 bit hanno due codifiche differenti a seconda del valore del bit di controllo **I negato**

Se **I negato=0** si interpretano i 12 bit di offset come un **immediate (costante) a 12 bit unsigned imm12**

Se **I negato=1** si interpretano i 12 bit di offset come un **offset preso da registro, eventualmente ruotato o shiftato di una certa quantità oppure shiftato di una quantità contenuta in un altro registro**.



Nel caso che **I negato = 1** l'offset è contenuto in un registro e il formato del campo Src2 cambia.

Abbiamo 4 campi: Rm, 0, sh, sham5

Rm: sono 4 bit che specificano l'indirizzo del registro che contiene l'offset

Il bit 4 è sempre 0.

Sh: sono due bit che codificano il tipo di shift che si può avere, nel caso di un registro shiftato. **LSL 00 - LSR 01 - ASR 10 - ROR 11**

Non c'è un ASL perchè è sostituito da LSL equivalente.

ROL non c'è perchè possiamo ottenerlo sfruttando ROR. Il ROL sarà uguale al ROR 32-n

Infatti ruotare di n posizioni verso Sx significa ruotare verso destra di 32-n posizioni.

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

Shamt5: sono 5 bit che costituiscono lo "shift amount" cioè di quante posizioni deve shiftare il contenuto del registro Rm, fino ad un massimo di 32.

Datapath dell'istruzione LDR

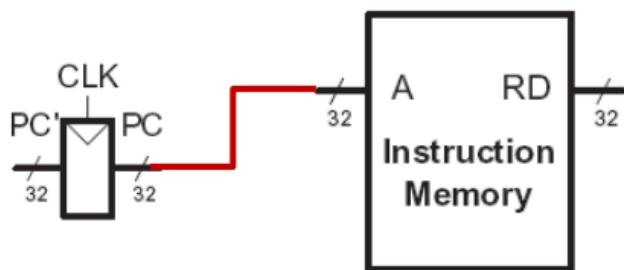
Il tipo dell'istruzione che consideriamo è con offset di tipo immediate (costante).

LDR Rd, [Rn, imm12]

Innanzitutto si parte dal **PC** che contiene l'indirizzo dell'istruzione da eseguire.

La prima fase è quella di **fetch dell'istruzione**, cioè si deve prelevare e leggere l'istruzione da eseguire che è contenuta nell' **instruction memory** all'indirizzo fornito dal PC.

Dunque il PC è collegato all'indirizzo di ingresso a 32 bit della memoria di istruzioni, **A**.

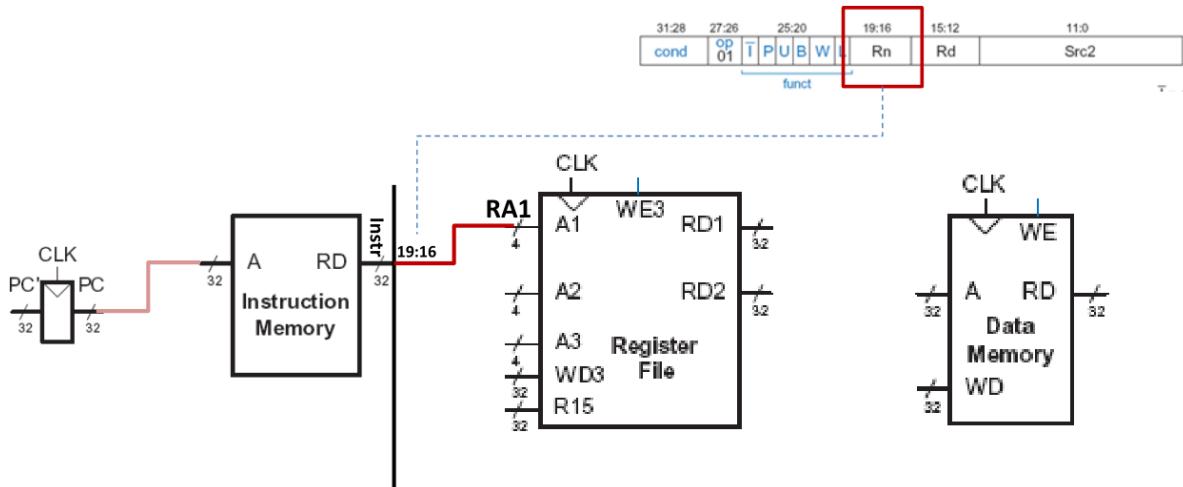


L'istruzione a 32 bit viene letta e riportata sull'uscita **RD** ed è rappresentata dall'etichetta **Instr.**

Per eseguire l'istruzione di LDR il passo successivo è quello di calcolare l'indirizzo da cui andare a prendere nella memoria dati (data memory) il dato da caricare all'indirizzo Rd. Tale indirizzo è calcolato sommando o sottraendo al valore Rn l'offset imm12 che è la costante.

Si inizia col prelevare il valore di **Rn** quindi si legge il **registro contenente l'indirizzo di base**. Il registro è specificato nel campo Rn della codifica dell'istruzione LDR, **Instr19:16 (4 bit)**

Questi 4 bit che specificano l'indirizzo del registro Rn vengono collegati all'ingresso di una delle porte del file register, **A1**. Il register file legge il contenuto del registro specificato ossia una parola a 32 bit e lo pone all'uscita **RD1..**



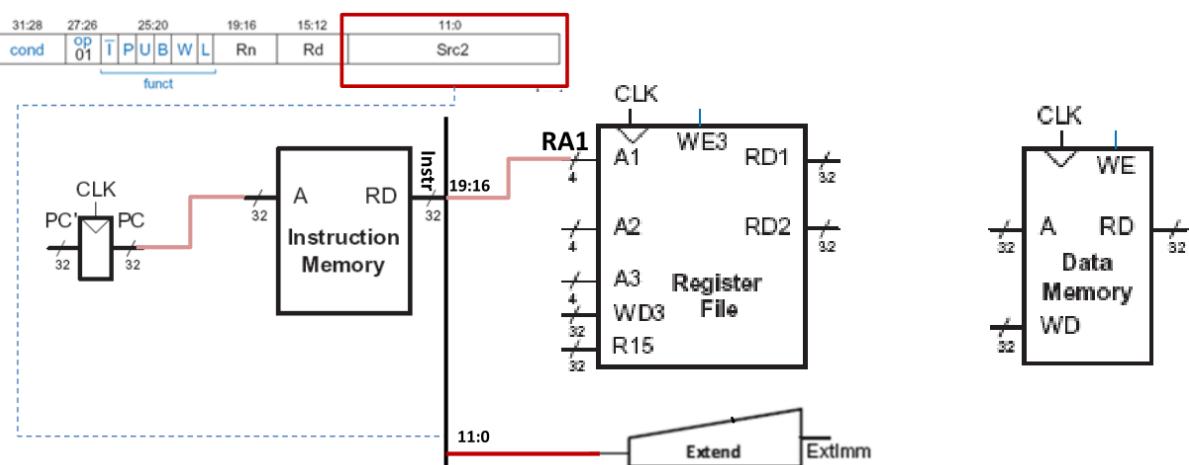
A questo punto ci occupiamo dell'**offset** che in questa istruzione di LDR è un immediate (costante) memorizzato nel campo **imm12** che sarebbero i primi 12 bit dell'istruzione

Istr11:0

L'offset è un valore unsigned a 12, che viene sommato o sottratto al valore Rn a seconda del bit U del campo di funct. Siccome Rn che è il base address ha un valore a 32 bit , per sommare l'offest dobbiamo **estendere il valore imm12 da 12 a 32 bit**.

I 12 bit imm12 vengono quindi prelevati dall'istruzione ed estesi a 32 bit attraverso un extender che produce il **valore esteso a 32 bit ExtImm**.

L'extender non fa altro che aggiungere degli zeri ai bit imm12 (Instr11:0) per farlo diventare a 32 bit. Quindi **ExtImm31:12=0 e ImmExt11:0=Instr11:0**



Adesso il processore deve sommare l'indirizzo di base Rn all'offset per ottenere l'indirizzo di memoria dati in cui andare a prelevare il dato da caricare nel registro di destinazione Rd. Per eseguire la somma si utilizza un ALU.

L'ALU riceve due operandi a 32 bit: SrcA e SrcB

- srcA proviene dal register file ossia dall'uscita RD1 e sarebbe il base address Rn
- srcB proviene da ExtImm e sarebbe il valore dell'offset imm12 esteso a 32 bit

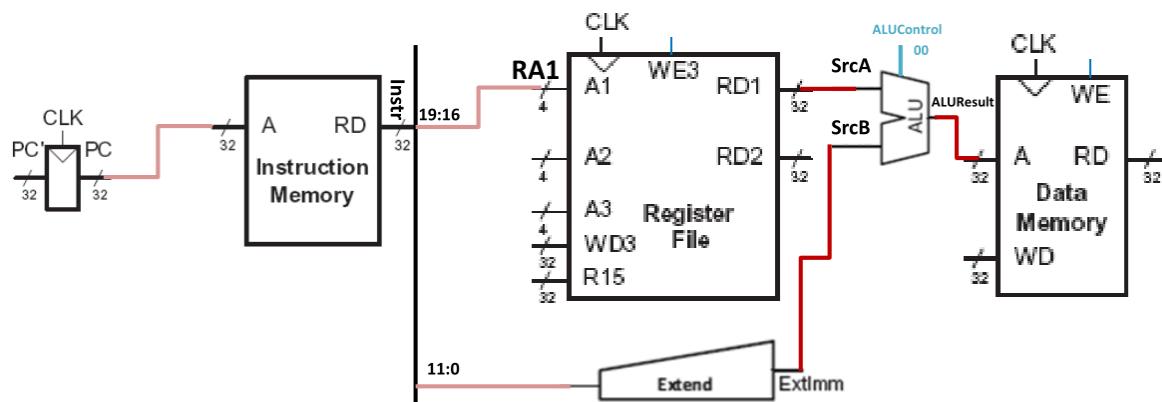
Sappiamo che l'ALU può effettuare varie operazioni aritmetiche/logiche. L'operazione da eseguire viene specificata dal segnale di controllo a 2 bit **ALUControl**.

ALUControl specifica quindi l'operazione da effettuare:

- una somma (ALUControl=00), se il valore del bit U = 1
- una sottrazione (ALUControl=01), se il valore del bit U = 0

In questo caso siccome dobbiamo fare una somma tra Rn e offset (e quindi U=1) **ALUControl vale 00** ed è impostato dalla CU.

La ALU dopo aver sommato i due operandi SrcA e SrcB genera un valore a 32 bit **ALUResult**. Questo risultato viene inviato alla memoria dati come indirizzo di lettura sulla porta A. Questo indirizzo è l'indirizzo in cui la memoria dati andrà a prendere il valore da caricare nel registro Rd.



Il dato viene prelevato dalla memoria dati e posto sull'uscita **RD** del data memory.

Esso poi passa sul bus **ReadData** e poi viene scritto nel registro destinazione Rd alla fine del ciclo, sul fronte alto del clock.

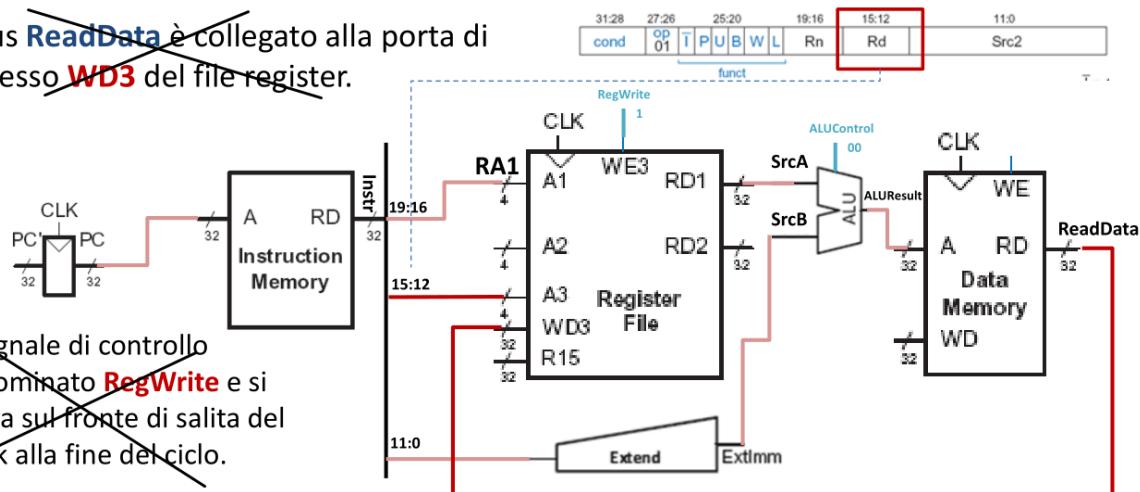
Il registro di destinazione per l'istruzione LDR è specificato nel campo **Rd** dell'istruzione **Instr15:12**. Questi 4 bit sono collegati all'ingresso **A3** del register file che sarebbe la porta di scrittura e questo ingresso A3 indica dove andare a scrivere il dato, appunto il registro Rd specificato dai 4 bit.

Il bus **ReadData** sul quale è presente il dato viene collegato alla porta di ingresso **WD3** del file register, che indica il contenuto del registro Rd specificato in A3.

Il segnale di controllo denominato **RegWrite** è collegato all'abilitazione alla scrittura della porta A3, **WE3** e si attiva sul fronte di salita del clock alla fine del ciclo per scrivere il dato letto dalla memoria e contenuto in WD3 nel registro di destinazione Rd contenuto in A3.

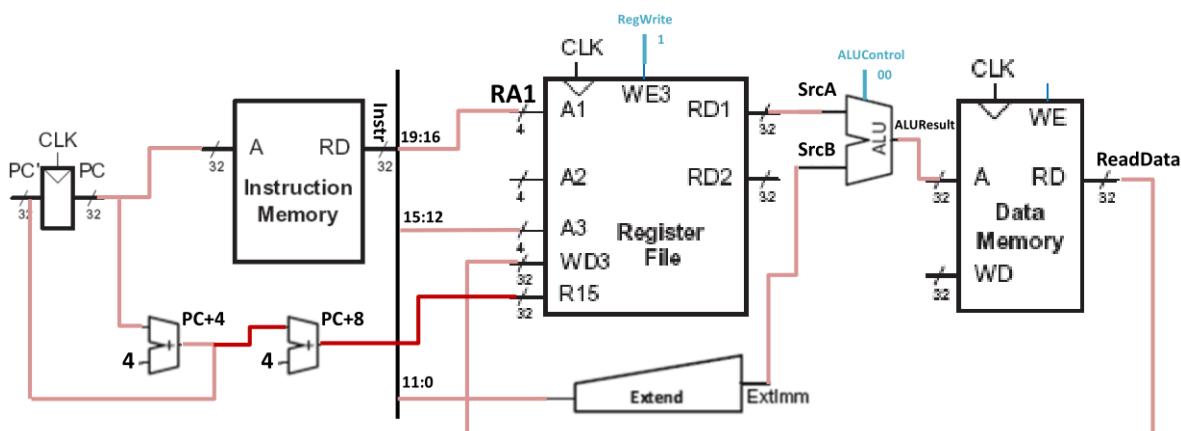
Il bus **ReadData** è collegato alla porta di ingresso **WD3** del file register.

Il segnale di controllo denominato **RegWrite** e si attiva sul fronte di salita del clock alla fine del ciclo.



Infine mentre l'istruzione viene eseguita , il processore deve calcolare l'indirizzo dell'istruzione successiva **PC'**. Siccome le istruzioni sono a 32 bit (4 byte), l'istruzione successiva è a **PC + 4**. Si utilizza un sommatore per incrementare il **PC** di 4. Il nuovo indirizzo viene scritto nel **PC** al successivo fronte di salita del **clock**.

Siccome nelle architetture ARM il registro **R15** contiene il valore **PC+8**, ossia contiene l'istruzione successiva a quella contenuta nel **PC** , è necessario un ulteriore **sommatore (+4)**, quindi **PC+4+4 = PC+8** la cui uscita è collegata all'ingresso di **R15**.



infine, se il registro di base o di destinazione è proprio R15 esso viene modificato e sovrascritto, e siccome deve contenere PC+8, deve mantenere una coerenza col PC, che quindi viene modificato .Quindi il valore del PC può provenire dal risultato dell'istruzione ReadData oppure da PC+4 PCPlus4 , si introduce un mux per scegliere fra le due possibilità il **multiplexer, permette di selezionare fra:**

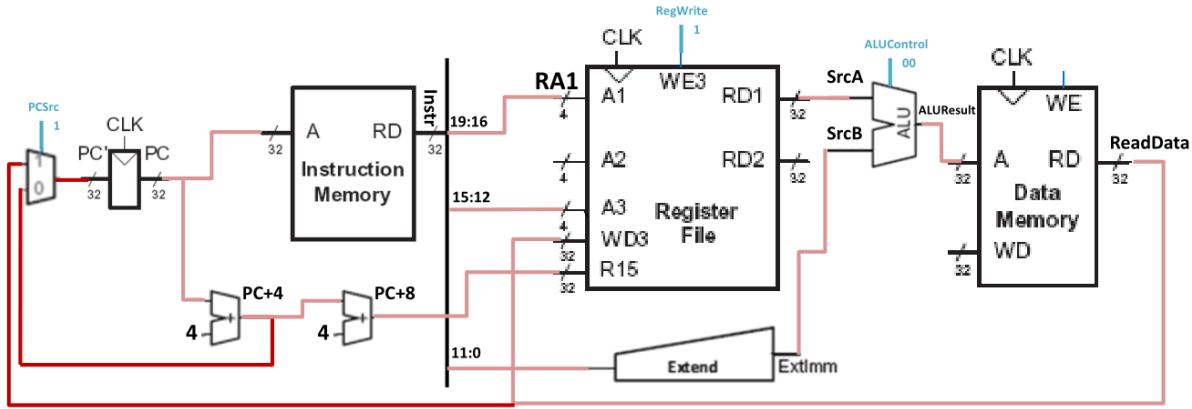
0 – PCPlus4

1 – ReadData.

Il segnale di controllo associato al multiplexer è PCSrc.

Se PCSrc=0 seleziona PCPlus4

Se PCSrc=1 seleziona ReadData



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

STR Rd, [Rn, imm12]

Si estende il datapath in modo da poter gestire anche le istruzioni di STR oltre che quelle di LDR viste in precedenza.

Per prima cosa si calcola l'indirizzo in cui dobbiamo andare a scrivere il dato contenuto in Rd. Questo calcolo è analogo a quello del LDR. Si prende l'indirizzo di base Rn e si somma all'offset imm12 che viene esteso a 32 bit.

L'indirizzo di base viene letto sempre dalla porta A1 del register file.

L'ALU aggiunge o sottrae a seconda del bit U l'indirizzo di base alla costante per trovare l'indirizzo di memoria nel quale memorizzare il dato.

Il dato che vogliamo salvare nel **data memory** tramite operazione di store si trova nel registro **Rd** specificato dai **4 bit Instr15:12 dell'istruzione di store**.

I 4 bit che specificano il registro Rd vengono collegati alla porta **A2** del register file.

Il **valore contenuto nel registro Rd** viene letto sulla porta di uscita del register file **RD2**.

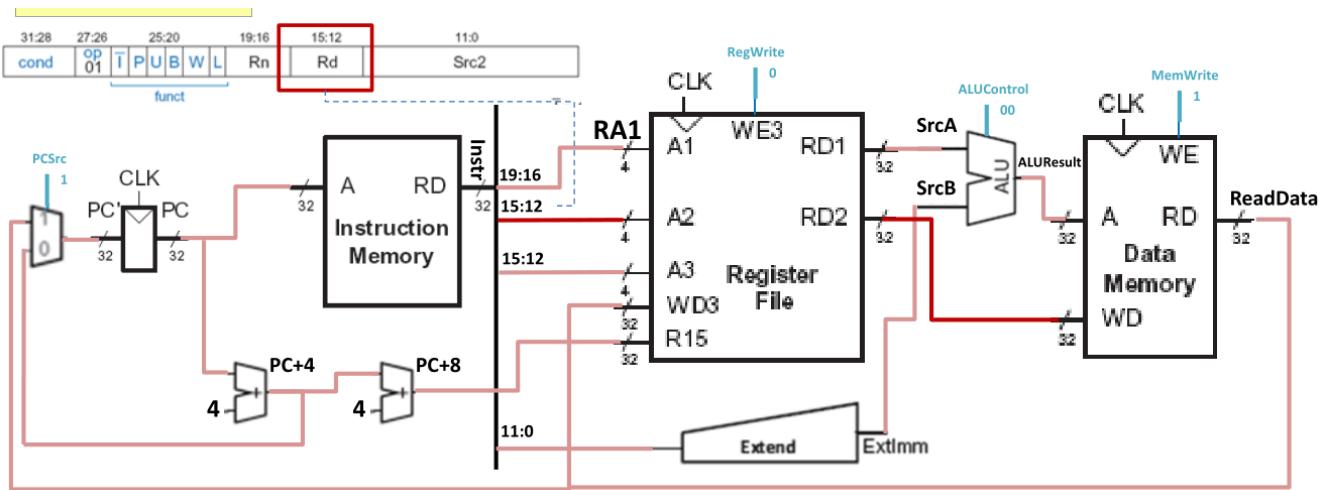
La porta di uscita **RD2** del register file (che contiene il dato da salvare nella memoria dati) è collegata alla **porta di scrittura dati WD** del Data Memory.

L'abilitazione alla scrittura in memoria dati **WE** è controllata dal segnale **MemWrite**
Quando **MemWrite=1** il dato viene memorizzato in memoria

RegWrite=0 perchè non c'è nessuna operazione di scrittura nel registro.

Pur essendo Rd associato anche a A3 e ReadData a WD3, questo non produce effetti sul File register, essendo RegWrite impostato a 0

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



Confronto RegWrite e MemWrite in operazioni di STR e LDR

Per quanto riguarda i segnali "RegWrite" e "MemWrite" nell'operazione di LDR

RegWrite=1 perchè dobbiamo scrivere nel registro

MemWrite=0 perchè l'accesso in memoria è solo in lettura per prelevare il dato

Nell'operazione di STR invece questi valori sono speculari perché dobbiamo salvare il dato in memoria e quindi scrivere in memoria

MemWrite=1

RegWrite=0 perchè non ci serve scrivere nel register file

il PCsource in questo caso sarà uguale a 0, l'istruzione successiva PC' ogni volta è calcolata come PC+4

Istruzioni di data processing

Sono istruzioni di elaborazione dei dati che eseguono appunto operazioni su dati memorizzati in registri e poi salvano i risultati delle operazioni in un registro di destinazione

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

N. Mohinian

MOV sposta nel registro di destinazione Rd un valore (costante o il contenuto di un registro),
ADD esegue una somma a 32 bit e salva il risultato nel registro di destinazione Rd
SUB sottrazione

Istruzioni per i branch

CMP esegue la sottrazione del primo operando col secondo e aggiorna i flags di stato NZCV. Memorizza nel registro current status program register i valori di uscita dei flags dell'operazione

CMN come CMP ma fa la somma anziché la sottrazione

TST fa l'and fra Rd e il secondo argomento e poi cambia i flags

TEQ fa lo xor fra Rd ed il secondo argomento e poi cambia i flags

Istruzioni di moltiplicazione

MUL moltiplica 32x32 bit Rs e Rm e mette il risultato a 32 bit in Rd

MLA moltiplicatore con accumulo fa la moltiplicazione tra Rm e Rs e poi somma un altro argomento Rn

UMULL moltiplicazione unsigned 32x32 bit, risultato a 64 bit salvato in 2 registri destinazione

SMULL moltiplicazione signed 32x32 bit risultato a 64 bit salvato in 2 registri destinazione

Istruzioni di data processing : istruzioni logiche

Operano bit a bit e scrivono il risultato in un registro di destinazione. La prima sorgente è sempre un registro e la seconda può essere un immediate o un altro registro

- **AND**: esegue l'and bit a bit
- **ORR**: esegue l'or bit a bit
- **EOR (XOR)**: exclusive OR ossia XOR bit a bit
- **BIC (Bit Clear)**: ogni volta che c'è un 1 nel secondo operando il valore del risultato è 0 maschera i bit
- **MVN (MoVe and NOT)**: commuta il valore del secondo operando e lo sposta in Rd

Le istruzioni AND or BIC sono utili per mascherare bit .

L'istruzione ORR: è utile per combinare bit fields

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	1111 0001
MVN	R7, R2	R7	0000 0000	0000 0000	1111 1111

Istruzioni di data processing : istruzioni di shifting

Le istruzioni di shift traslano a sinistra o a destra il valore contenuto in un registro eliminando i bit a una delle due estremità. L'istruzione di rotazione invece fa ruotare il valore di un registro fino a un massimo di 31 posizioni.

Sono 4 perchè la arithmetic shift left ASL è analogo al LSL e perchè il ROL viene implementato attraverso un ROR di 32-n posizioni.

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** .

Sono:

- LSL logical shift left 00 => inseriscono zeri nei bit meno significativi a dx lasciati liberi
- LSR logical shift right 01 => zeri inseriti nei bit più significativi
- ASR arithmetic shift right 10 => si inseriscono bit pari al bit più significativo
- ROR rotate right 11

L'ampiezza della traslazione può essere definita da una costante o dal contenuto in un registro

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
R6	0000 0000	0000 0000	0001 0100
1110 0111			

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

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**, eventualmente **traslato o ruotato**.

La **destinazione** è un **registro**.

L'istruzione è a 32 bit. Formato simile alle istruzioni di memoria: 2° principio regularity supports design simplicity

Data-processing

cond	op	funct	Rn	Rd	Src2
4 bits	2 bits	6 bits	4 bits	4 bits	12 bits

Abbiamo 6 campi: **cond**, **op**, **funct**, **Rn**, **Rd**, **Src2**

Gli **operandi** dell'istruzione sono codificati in 3 campi:

- **Rn**: campo a 4 bit che specifica il registro che contiene il valore del primo operando sorgente

- **Src2**: campo a 12 bit che specifica il registro (Shiftato, ruotato) o l'immediate che indicano il secondo operando sorgente

- **Rd**: campo a 4 bit che specifica il registro destinazione in cui viene salvato il risultato dell'istruzione di data processing

Campi di controllo:

cond: sono i **4 bit più significativi** dell'istruzione (28:31) e sono i bit di condizione, servono a codificare un'esecuzione condizionata ossia eventuali condizioni che determinano o meno l'esecuzione di questa istruzione. **non condizionate cond = 1110 (14)**
cmd = 11012 (13) codice usato per tutti i tipi di shift (LSL, LSR, ASR, ROR)

op: **operation code**, sono **2 bit** che specificano l'istruzione da eseguire. Per le istruzioni di data processing il suo valore è **00**

funct: sono **6 bit** che specificano il tipo di funzione da eseguire. Questo campo è formato da 3 sottocampi: **I, cmd, S**

I : è **1 bit** che serve a interpretare i 12 bit del campo Src2.

Se **I=0** , Src2 è un registro

Se **I=1** , Src2 è un immediate

cmd : sono **4 bit** che specificano l'**istruzione di data processing specifica da svolgere**.

Ad esempio

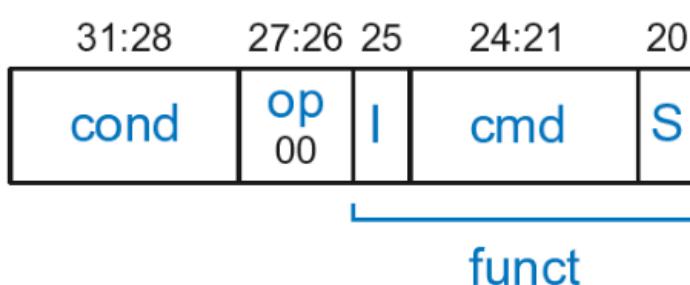
cmd=0100=ADD

cmd=0010=SUB

S-bit: è **1 bit** che serve per impostare ed aggiornare le **conditions flags NZCV** dell'alu al termine dell'istruzione.

Quando **S=1** vengono aggiornate le condition flags

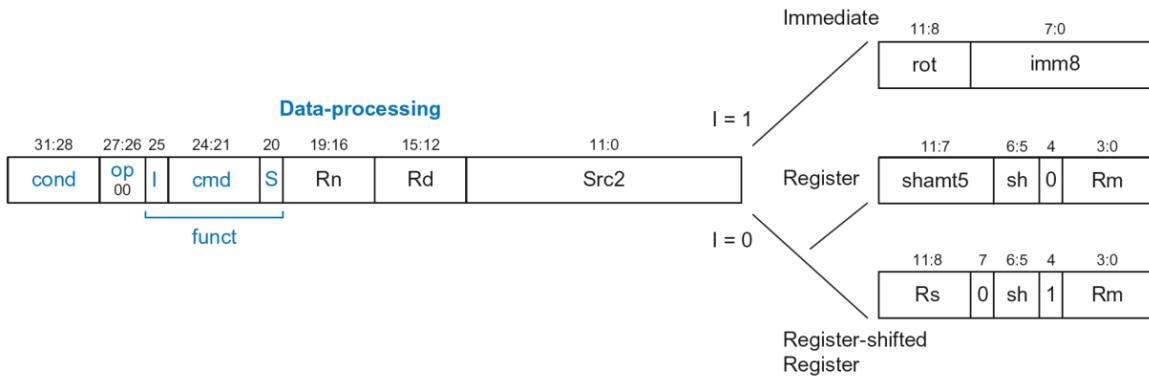
Quando **S=0** NON vengono aggiornate le condition flags



Campo Src2

Per quanto riguarda il campo **Src2** esso può essere a seconda del **bit I**:

- **un Immediate**
- **un Registro eventualmente traslato di una costante shift amount, oppure può essere anche un Registro “shiftato” dal contenuto di un altro registro**



Nel caso **I=1** abbiamo che il campo a 12 bit **Src2** è un **immediate**.

L'immediate è codificato con **12 bit** e ha 2 campi: **rot e imm8**

rot: sono **4 bit** che specificano il **rotation value**, ossia sono dei bit che ci dicono di quanto deve **ruotare la costante imm8 verso destra**, questo ci consente di ottenere una costante dal valore molto alto quando essa viene estesa a **32 bit** (infatti $\text{rot} \times 2$, se $\text{rot}=1111=16$, $\text{rot} \times 2=32$) **imm8 ROR (rot × 2)**

Es. per produrre 0xFF0 imm8 deve essere ruotato di 4 bit a sinistra, ovvero 28 bit a destra. Quindi, rot = 14 , dato che rot * 2 = 28

imm8: sono 8 bit che specificano un unsigned immediate

Nel caso **I=0** abbiamo che il campo a 12 bit **Src2** è un **registro**.

Rm: sono **4 bit** che indicano il registro che fa da secondo operando

sh: **2 bit** che indicano il tipo **tipo di shift LSL 00 - LSR 01 - ASR 10 - ROR 11**

bit 4 : è quello che ci dice come interpretare i 5 bit da 11:7 per capire se il **valore di shift** è una **costante** oppure è contenuto in un **registro**.

Se il bit 4=0 : gli ultimi **5 bit** codificano il valore di shift il quale è indicato da una **costante shamt5**, shift amount che indica di quanto il valore in Rm è shiftato

Se il bit 4=1: il bit 7 rimane a 0 e i successivi 4 bit rimanenti indicano il registro **Rs** ossia l'indirizzo del registro nel quale è contenuto un valore che indica di quanto shiftare Rm. il numero di posizioni di cui Rm deve shiftare non è una costante ma è indicato dal registro **Rs**.

Datapath per le istruzioni di data processing di tipo logico: ADD, SUB, AND, ORR

Estendiamo il datapath per gestire le istruzioni di data processing ADD, SUB, AND e ORR.

Si considera inizialmente un'istruzione il cui secondo operando è un immediate, quindi un'istruzione con indirizzamento immediato. es. **SUB R2, R3, #0xFF0**

In tal caso, le istruzioni hanno come operandi un **registro Rn** ed una **costante imm8** contenuta nei bit dell'istruzione stessa. E' l'ALU esegue l'operazione di data processing logica specificata dall'istruzione e il risultato viene scritto in un registro di destinazione **Rd**

Queste istruzioni di data processing differiscono solo nella specifica operazione eseguita dall'ALU. Quindi, possono essere implementate tutte con lo stesso hardware utilizzando diversi segnali **ALUControl** che specificano l'istruzione logica da eseguire. La **CU si occupa di settare opportunamente il segnale ALUControl** a seconda dell'istruzione logica da eseguire.

I valori per **ALUControl** sono:

ADD – 00;
SUB – 01;
AND – 10;
ORR – 11.

L'ALU al termine dei calcoli da effettuare genera anche **4 flags, ALUFlags3:0 (1 bit per flag, Negative Zero Carry oVerflow) che vengono inviati alla CU.**

A differenza delle istruzioni di memoria LDR e STR dove il campo Src2 dell'istruzione è a 12 bit se è l'indirizzamento è di tipo immediate e che viene esteso tramite l'extender a 32 bit, **nelle istruzioni di data processing il campo Src2 se è di tipo immediate è di 8 bit**

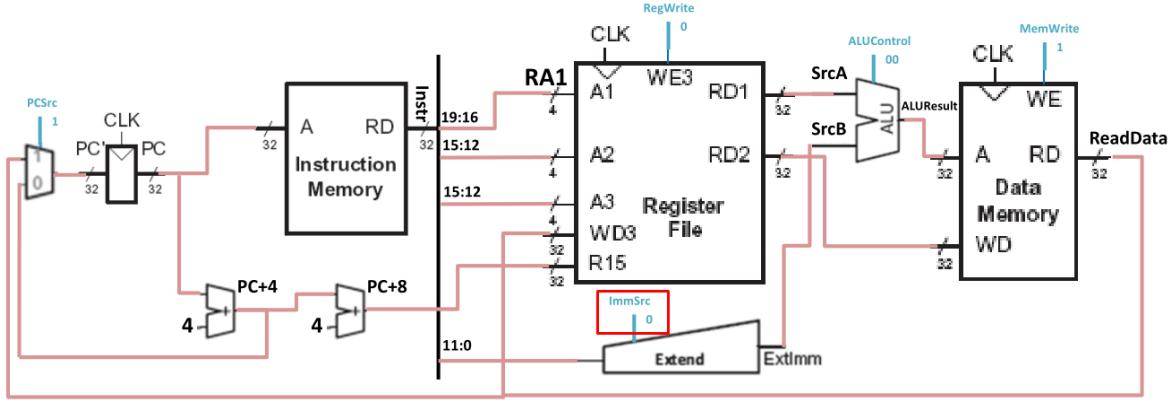
Quindi le istruzioni di data processing utilizzano costanti di 8 bit (non 12 bit). Questo vuol dire che l'extender deve funzionare in maniera diversa a seconda che l'istruzione sia di data processing o memoria.

Per cui il blocco circuitale **Extend** riceve in input un **segnale di controllo ImmSrc**:

Se ImmSrc = 0 → ExtImm cioè l'immediate è esteso da Instr7:0
(a partire dall'8 bit vengono aggiunti gli 0, quindi si tratta di un immediate riferito a istruzioni di data processing)

Se ImmSrc = 1 → ExtImm cioè l'immediate è esteso da Instr11:0
(a partire dal 12esimo bit vengono aggiunti gli 0, quindi si tratta di un immediate riferito a istruzioni di memoria LDR STR)

Il segnale ImmSrc è pilotato sempre della CU a seconda dell'istruzione che si sta eseguendo.



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

L'ALU dopo aver effettuato l'operazione tra SrcA (registro) e SrcB (immediate) genera un risultato: **ALUResult** che deve essere memorizzato nel registro destinazione corrispondente Rd che si trova nel Register File.

Il register file quindi riceve il risultato dell'operazione in input sulla porta WD3:

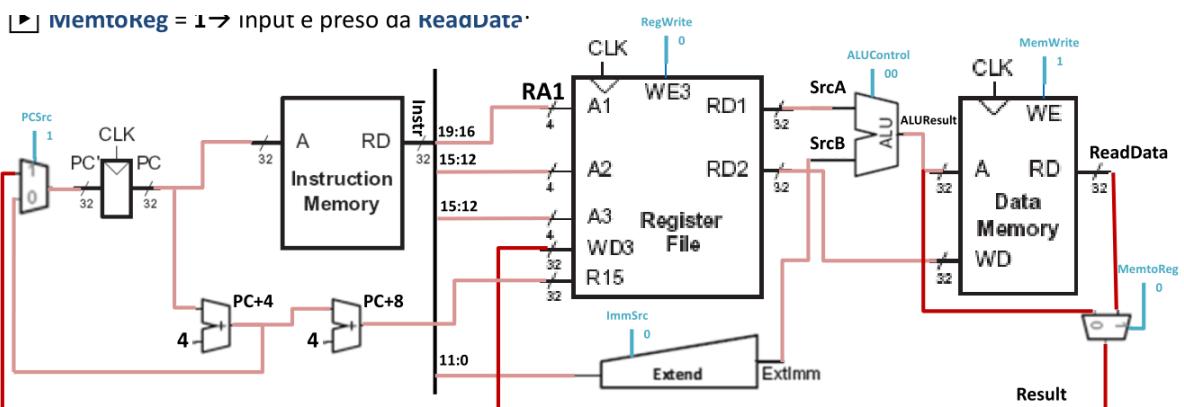
- sia dal data memory attraverso il bus **ReadData** in caso di istruzione di LDR
- sia direttamente dall'ALU nel caso di operazioni aritmetiche tramite **ALUResult**

Serve quindi un ulteriore multiplexer per selezionare con un selezionatore il risultato dell'operazione tra ReadData e ALUResult a seconda dell'istruzione. L'uscita di tale mux è denominata **Result** ed è collegata all'ingresso WD3 della porta di scrittura del RegisterFile.

Il multiplexer richiede un segnale di controllo per selezionare l'input corretto, ovvero **MemtoReg**.

Se MemtoReg = 0 → input è preso da ALUResult e trasferito al mux
(l'istruzione è di data processing);

Se MemtoReg = 1 → input è preso da ReadData e trasferito al mux
(l'istruzione è di memoria di tipo LDR, per quella STR il valore non interessa dato che non scrive in memoria);



Consideriamo ora le istruzioni di data processing con indirizzamento da registro es. EOR R8, R9, R10, ROR R12

esse ricevono il secondo operando sorgente da Rm, ossia il registro specificato dai primi 4 bit meno significativi dell'istruzione Instr3:0 quando I=0

Rm indica l'indirizzo del registro in cui andare a prendere nel register file il valore del secondo operando e quindi si deve poter associare in questo caso i 4 bit di Rm all'ingresso A2 del register file.

L'ingresso A2 del register file è associato però anche ai bit 15:12 che specificano Rd nel caso di istruzione di memoria di tipo STR.

Si aggiunge allora un ulteriore mux sull'ingresso A2 del file register.

Esso è pilotato dal segnale di controllo RegSrc . In base al suo valore stabilito dalla CU

=> A2 riceve in ingresso Rd (Instr15:12) (istruzione di STR)

=> A2 riceve in ingresso Rm (Instr3:0) per istruzioni di data processing con indirizzamento da registro

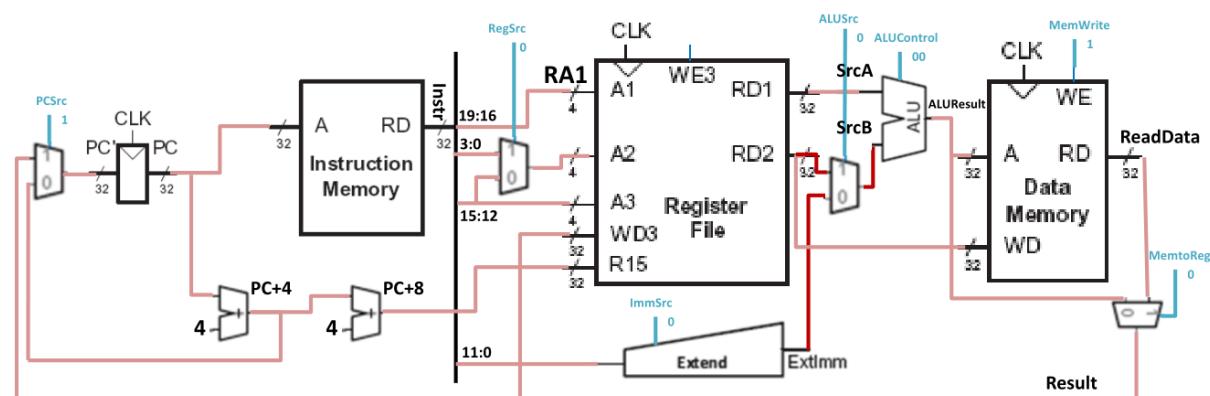
Infine nelle istruzioni di data processing il secondo operando SrcB in ingresso all'ALU può essere quindi un immediate a 8 bit che viene esteso a 32 bit e portato in ingresso all'ALU oppure un valore contenuto nel registro Rm che viene prelevato dal register file.

Quindi aggiungiamo un ulteriore multiplexer sugli ingressi dell'ALU per selezionare questo secondo operando sorgente.

E' pilotato dal segnale di controllo ALUSrc , che seleziona la seconda sorgente della ALU SrcB.

A seconda del segnale di controllo ALUSrc , SrcB viene selezionato da:

- ExtImm ossia un immediate esteso a 32 bit per istruzioni, che utilizzano costanti;
- dal register file sull'uscita RD2 per istruzioni di data processing con indirizzamento da registro.



Esecuzione condizionata di un'istruzione

Come abbiamo visto nei formati delle istruzioni precedenti i primi 4 bit più significativi costituiscono i bit di condizione. Un'esecuzione non condizionata ha il codice 14 (1110). ciò indica che l'istruzione è eseguita sempre e in modo incondizionato dai valori del CURRENT PROGRAM STATUS REGISTER (cpsr) . **A volte si vuole che l'esecuzione di una istruzione dipenda dal verificarsi o meno di una certa condizione. Sono istruzioni che vengono eseguite solo se certe condizioni dei 4 flags N Z C V memorizzati nel CPSR vengono soddisfatte. Per questo si dicono istruzioni condizionate.** Queste istruzioni di basso livello assembly consentono a linguaggi di programmazione di alto livello di realizzare costrutti di selezione quali if..else oppure iterazione while, for.

Il suffisso -S alle istruzioni di data processing

I flag N Z C V sono dette **CONDITION FLAGS** e sono memorizzate nel CPSR.

Questi flag sono settati da istruzioni di data processing a cui viene aggiunta una S finale (ad esempio ADDS) . **La S finale indica che una volta eseguita l'istruzione i valori dei flag N Z C V di output dell'alu devono essere memorizzati nel CPSR . Può essere aggiunto a tutte le istruzioni di data processing**

Esempio ADDS R1, R2, R3

Esegue: R2 + R3

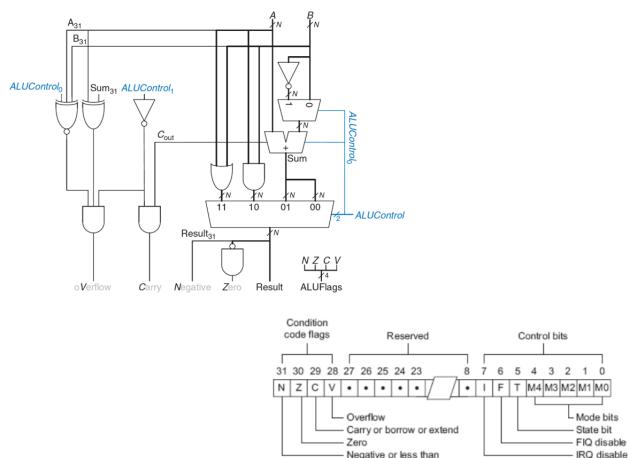
- Salva il risultato in R1
- Aggiorna i flag nel Current Program Status Register.

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

Il CPSR

I valori dei flag N Z C V vengono dall'ALU (non sempre vengono memorizzati nel CPSR) . Sono memorizzati nei 4 bit più significativi del CURRENT PROGRAM STATUS REGISTER, che è un registro a 32 bit che presenta altri bit oltre questi 4 per gestire funzioni del s.o. come context switch o interrupt.



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

Istruzioni condizionate

Le istruzioni condizionate in assembly sono un particolare tipo di istruzioni che sono definite attraverso lo mnemonico che indica il tipo di operazione da eseguire (tipo ADD) al quale viene aggiunto un suffisso che indica la condizione che deve essere soddisfatta affinché l'istruzione e quindi l'operazione venga eseguita.

L'istruzione sarà eseguita condizionalmente sulla base dei valori dei condition flags.

Il suffisso dell'istruzione condizionata è detto **condition mnemonic**

Ogni condition mnemonic corrisponde ad una particolare condizione e ad un particolare stato in cui devono trovarsi i flag N Z C V per far sì che l'istruzione venga eseguita

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.

Es.

CMP R1, R2

SUB**NE** R3, R5, R8

- **NE**: SUB verrà eseguito solo se $R1 \neq R2$, ovvero $Z = 0$. Se Z fosse uguale a 1 vuol dire che i valori contenuti in R1 e R2 sono uguali , la sottrazione col CMP ritorna 0 e imposta il flag $Z=1$ e quindi not equal non è soddisfatta quindi non esegue l'istruzione

Tabella dei condition mnemonic

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

$A = 1001_2$ $B = 0010_2$	$A = 9$ $B = 2$	$A = -7$ $B = 2$
$A - B: \begin{array}{r} 1001 \\ + 1110 \\ \hline 10111 \end{array}$	$NZCV = 0011_2$ $HS: \text{TRUE}$ $GE: \text{FALSE}$	

$A = 0101_2$ $B = 1101_2$	$A = 5$ $B = 13$	$A = 5$ $B = -3$
$A - B: \begin{array}{r} 0101 \\ + 0011 \\ \hline 1000 \end{array}$	$NZCV = 1001_2$ $HS: \text{FALSE}$ $GE: \text{TRUE}$	

HS controlla bit C = true per essere eseguita

GE controlla la condizione *(N xor V) = true per essere eseguita

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 le istruzioni in sequenza, incrementando il Program Counter (PC) di 4 byte (32 bit) dopo ciascuna istruzione, in modo da puntare alla successiva istruzione da eseguire.

Ci possono però essere delle eccezioni, ad esempio nelle istruzioni di data processing quando il registro di destinazione è proprio R15, che contiene l'istruzione successiva a quella contenuta nel PC e questo vuol dire che dobbiamo modificare di conseguenza anche il PC.

Le istruzioni di branching permettono di modificare il flusso e l'esecuzione delle istruzioni, non rendendole più sequenziali. Esse permettono di compiere dei salti da un punto all'altro del programma. **Modificano quindi attraverso i salti (branch) il flusso dell'esecuzione del programma**

Le istruzioni branching permettono di cambiare il valore del PC.

ARM include 2 tipi di branch:

- simple branch (B)
- branch and link (BL).

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

spesso sono condizionati cioè si verificano ad una certa condizione espressa da un suffisso e permettono di implementare a basso livello cicli ecc.

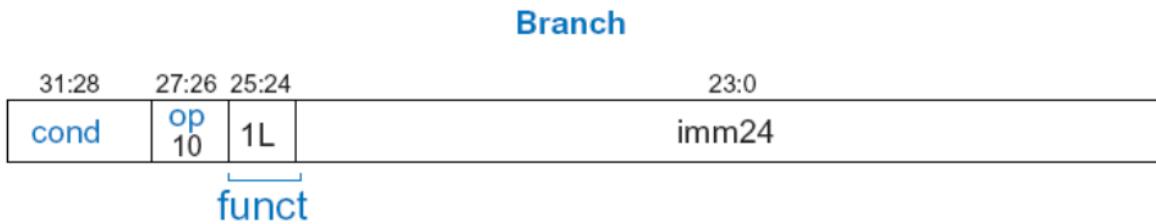
Il branch semplice (B) è utilizzato per realizzare istruzioni di controllo

Il branch and link (BL) viene utilizzato per le chiamate a funzione

Quando c'è una chiamata a funzione c'è un salto da un punto all'altro del programma, un cambio del flusso di controllo. dobbiamo salvarci l'indirizzo dell'istruzione seguente a quella di branch per ritornare nella funzione chiamante e ripartire col blocco di istruzioni successive al branch dopo aver eseguito quelle della subroutine.

Il codice assembly utilizza le etichette per indicare i blocchi di istruzione nel programma , blocchi che si possono saltare o ripetere. Quando il codice assembly è tradotto in linguaggio macchina queste etichette vengono tradotte in indirizzi di istruzione

Formato delle istruzioni di branching



Le istruzioni di branching utilizzano un **unico operando costante di 24 bit, imm24.**
Questa costante è rappresentata in complemento a due (per avere salti maggiori) e specifica l'indirizzo dell'istruzione alla quale saltare, partendo dal valore PC+8

cond: sono i primi 4 bit significativi che indicano la condizione

op : campo di 2 bit, che specifica l'operazione il cui valore è **10. (istruzione di branch)**

funct: il campo di funct è formato da 2 bit.

Il primo bit è sempre **1**

Il secondo bit **L** assume il valore:

- 0 se il branch è semplice (B)

- 1 se è un branch and link (BL)

La costante a 24-bit viene moltiplicata per 4 per sapere di quante parole saltare ed estesa con segno a 32 bit.

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 (memorizzato in R5) e scrive il risultato di nuovo nel PC.

Extender deve effettuare 3 forme di estensione diversa a seconda di ImmSrc

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

00=>8 bit estesi a 32 per istruzioni di data processing

01=>12 bit estesi a 32 per istruzioni di memoria

10=>24 bit estesi a 32 (con segno) moltiplicati per 4 per istruzioni di branch

Istruzione BL

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

Essa salva l'indirizzo di ritorno, cioè l'indirizzo che è memorizzato nel registro R15 (PC), nel registro R14 (LR). Poi salta alla prima istruzione della subroutine ed il ritorno dalla subroutine una volta eseguita avviene ripristinando (cioè copiando) l'indirizzo contenuto nel registro R14 in R15 quindi facendo un MOV R15, R14 ossia **MOV PC, LR**

- 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).

Datapath istruzioni di branching

Nelle istruzioni B e BL si deve calcolare l'indirizzo a cui saltare alla prossima istruzione, quindi si deve calcolare questo indirizzo a partire dal registro R15, che contiene PC+8 a cui deve essere sommata l'immediate a 24 bit imm24 e poi si deve scrivere il risultato nel PC.

L'immagine viene moltiplicato per 4 ed esteso a 32 bit con segno. Quindi il segnale che pilota l'extender **ImmSrc viene esteso a 2 bit, nella codifica 10 il segnale estende la costante a 24 bit per l'istruzione di branch.**

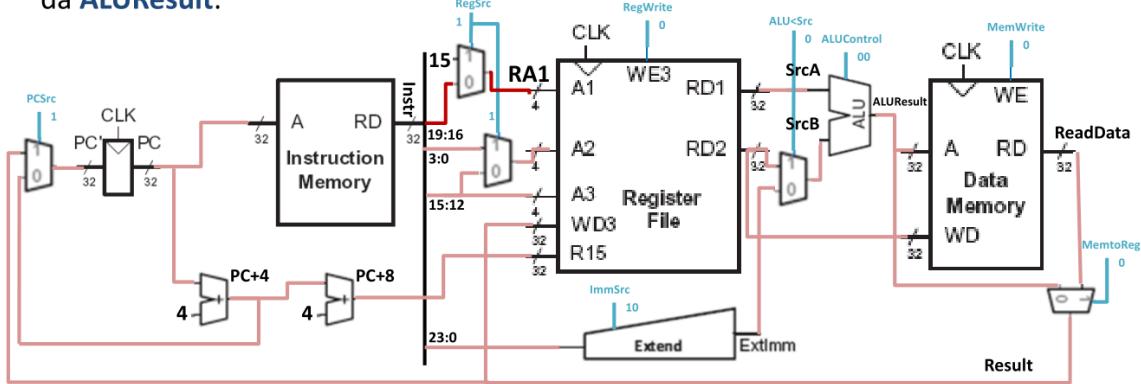
L'indirizzo PC+8 che è contenuto in R15 viene letto dalla prima porta del file register A1 e il valore viene dato poi in ingresso a RD1. Dunque un mux deve poter selezionare R15 come registro in ingresso alla porta A1, e non più Rn come di solito.

Dunque il mux è pilotato da un altro bit di RegSrc che seleziona i bit Rn Instr19:16 per le altre istruzioni e 15 per l'istruzione B, perchè è il registro che contiene PC+8 dato in input che deve essere sommato all'alu nelle istruzioni di branching. il risultato ottenuto deve essere inviato al pc che viene modificato

Dopo aver sommato l'indirizzo pc+8 all'immagine l'alu (alucontrol=00 perchè somma) produce il risultato ALUResult che passa al mux, il cui segnale MemtoReg vale 0 perchè il risultato dell'alu viene riportato direttamente al program counter, che ha come segnale PCSrc=1 dato che l'indirizzo della prossima istruzione da eseguire proviene direttamente dall'ALU

RegWrite e MemWrite vengono settati a 0 perchè non c'è un'operazione di scrittura in memoria

MemtoReg è impostato a 1, **ImmSrc** è impostato a 1 per selezionare il numero da **ALUResult**.



Tutti i segnali di controllo del datapath:

- PCSrc
- MemtoReg
- MemWrite
- ALUControl
- ALUSrc
- ImmSrc
- RegWrite

Control Unit

La control unit è un'unità di controllo che genera i segnali di controllo del datapath, memorizza e aggiorna opportunamente le flag di stato.

E' costituita da una logica di combinatoria, nell'architettura a ciclo singolo. Nell'architettura a ciclo multiplo è un automa di mealy.

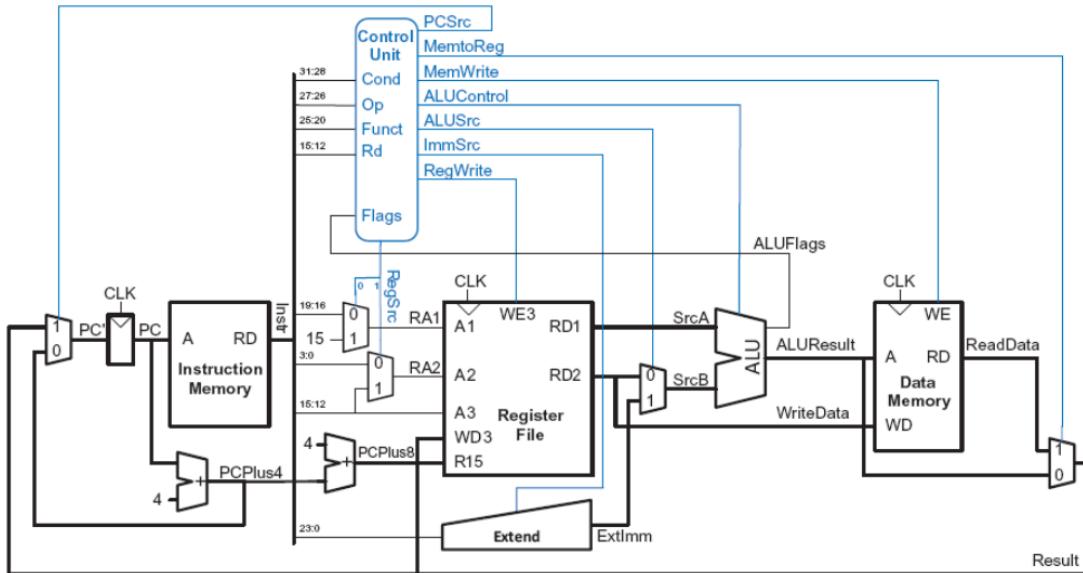
La CU genera i segnali di controllo prendendo in input:

- il campo **cond** dell'istruzione: gli serve per verificare se l'istruzione corrente deve essere verificata o meno sulla base dei flags
- il campo **op** dell'istruzione: che gli indica il tipo di istruzione da eseguire e in base ad essa la CU cambia i selettori
- il campo **funct** dell'istruzione
- il registro di destinazione **Rd**: per verificare se esso sia o meno R15(PC)
- i **flags** direttamente dall'ALU (**ALUFlags**) : perchè la condizione viene verificata sulla base dei valori correnti dei flags di stato

La CU deve memorizzare anche i flag di stato attuali nel Current Program Status Register e li deve aggiornare in modo appropriato.

In output genera tutti i segnali di controllo che vanno a finire sul datapath e che consentono di eseguire le istruzioni di diverso tipo.

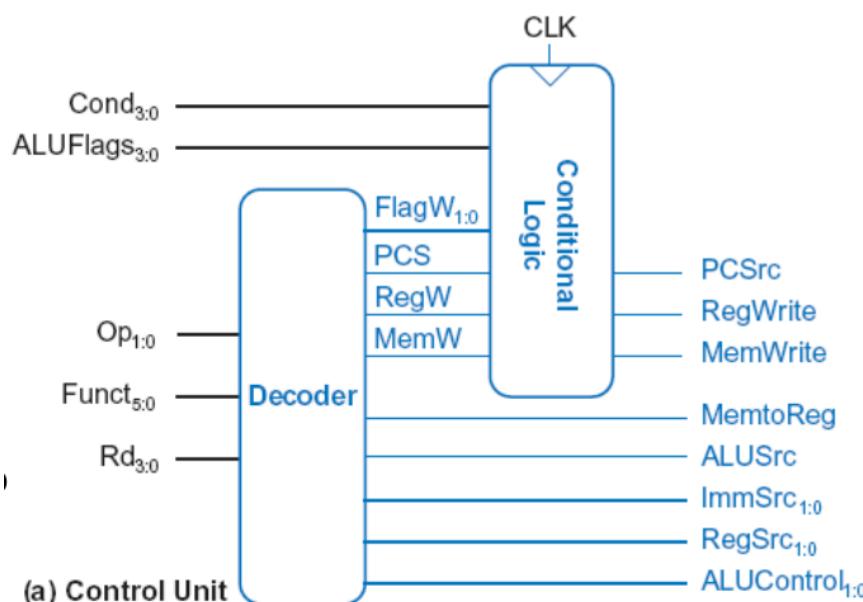
Schema completo datapath



Com'è fatta la CU all'interno

L'unità di controllo è divisa in 2 unità principali:

- **decoder:** che decodifica i valori di **op**, **funct**, **Rd** e genera i segnali di controllo
- **logica condizionale:** prende in input i **bit di cond** dell'istruzione e i **flag** che vengono dall'alu e verifica se rispetto ai valori memorizzati nel cpsr la condizione è soddisfatta o meno e poi **aggiorna i valori dei flag nel cpsr** nel caso in cui l'istruzione è tipo CMP, ADDS ecc. Quindi gestisce i flag di stato e li aggiorna quando l'istruzione deve essere eseguita su condizione.



Decoder della CU

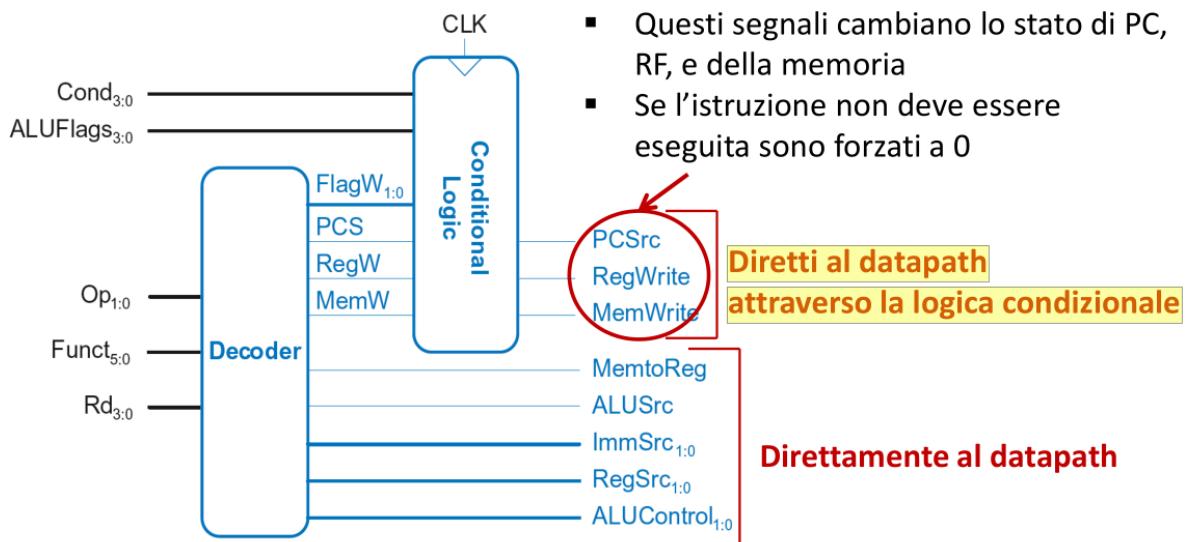
Dal decoder vengono inviati in output i vari segnali che vanno **direttamente sul datapath**, i quali sono *MemtoReg ALUSrc ImmSrc1:0 RegSrc1:0 ALUControl1:0*

Poi abbiamo altri output ossia FlagW1:0, PCS, RegW e MemW.

PCS, RegW e MemW diventeranno i veri output PCSrc, RegWrite e MemWrite solo se la condizione definita dai bit di cond e dagli aluflags e gestita dalla logica condizionale viene soddisfatta. Se la condizione non è soddisfatta questi segnali vengono sempre azzerati e quindi saranno uguali a 0 in output. **Diretti al datapath attraverso la logica condizionale**

Il Decoder della CU è inoltre 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 e quindi l'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.



I 2 bit FlagW(1:0)

Sono i bit di **FlagWrite Signal** che indicano quando le **ALUFlags** devono essere **aggiornate**, consentendo anche l'aggiornamento del CPSR con i flag di stato, ovviamente quando il valore del bit S delle istruzioni di data processing è = 1. Quindi quando S=1 le ALUFlags devono essere aggiornate.

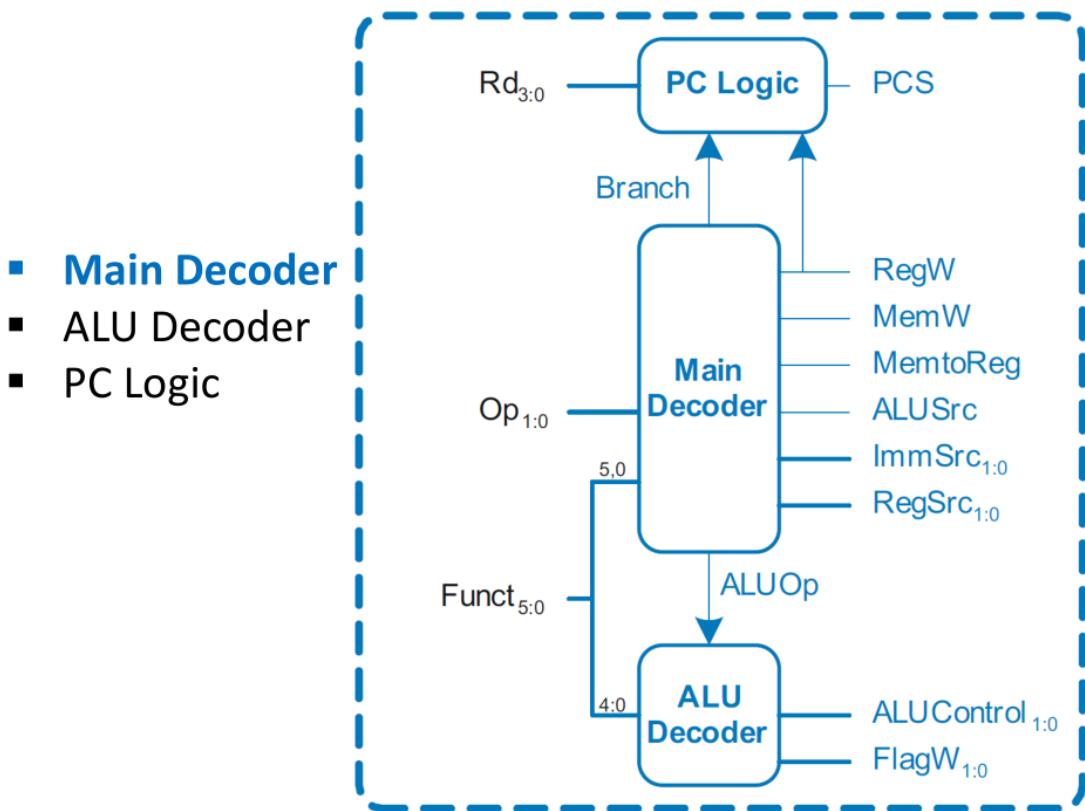
Sono 2 bit perchè sappiamo che le operazioni aritmetiche come ADD,SUB aggiornano tutti i flags NZCV, mentre le operazioni logiche dell'ALU come AND e ORR non hanno necessità di aggiornare i bit C, V di Carry e Overflow, ma aggiornano solo i bit N e Z

Quindi sono necessari due bit

- FlagW1 = 1: NZ (ALUFlags3:2 saved)
- FlagW0 = 1: CV (ALUFlags1:0 saved)

Es. istruzione ADDS, aggiorna tutti i flags => FlagW1=1 FlagW0=0
istruzione ANDS, aggiorna solo i flag NZ => FlagW1=1 FlagW0=0

I compiti del decoder ALU



Il decoder ha i seguenti compiti:

- determinare il tipo di istruzione e se il secondo operando è un registro o un immediate: 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, ImmSrc_{1:0}, e RegSrc_{1:0}.
- 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.

Il **Main Decoder** genera in particolare i valori ALUOp e Branch che indicano il tipo di istruzione, se è di branch o data processing. Ha in ingresso Op e Funct

L' **ALU Decoder** indica il tipo di operazione da eseguire (ADD, SUB ..) e genera il segnale ALUControl e i FlagW che indicano i flag da aggiornare

Il **PCLogic** genera il segnale PCS che deve passare per la logica condizionale prima di diventare PCSource (indica se il valore successivo del PC deve essere l'istruzione successiva PC+4 oppure se proviene dall'ALU nel caso di branch perché fa saltare la prossima istruzione in un punto diverso). Ha in ingresso Rd, e si attiva se Rd=15 (registro R15=PC).

La PC logic controlla se l'istruzione comporta una scrittura in R15 (quindi devo aggiornare anche PC) e RegW deve essere settato a 1 perché dobbiamo scrivere nel register file oppure è un branch secondo la condizione

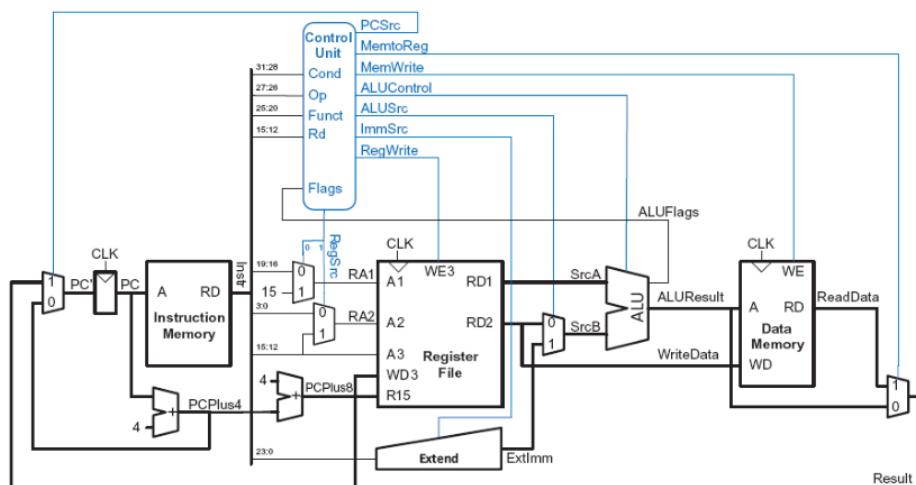
PCS = ((Rd == 15) AND RegW) OR Branch

In uscita abbiamo PCS

PCS=0=>PCSrc=0 => il PC si aggiorna in modo standard a PC+4

PCS=1=>PCSsrc=PCS=> l'istruzione è eseguita e dobbiamo eseguire un branch
Importante

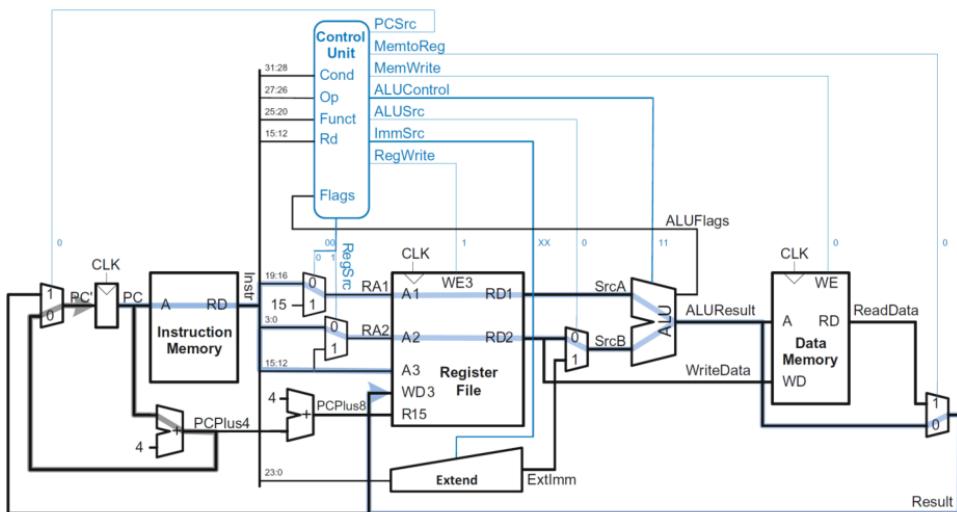
Op	Funct ₅	Funct ₀	Type	Branch	MemoReg	MemW	ALUSrc	ImmSrc	RegW	RegSrc	ALUOp
00	0	X	DP Reg	0	0	0	0	XX	1	00	1
00	1	X	DP Imm	0	0	0	1	00	1	X0	1
01	X	0	STR	0	X	1	1	01	0	10	0
01	X	1	LDR	0	1	0	1	01	1	X0	0
11	X	X	B	1	0	0	1	10	0	X1	0



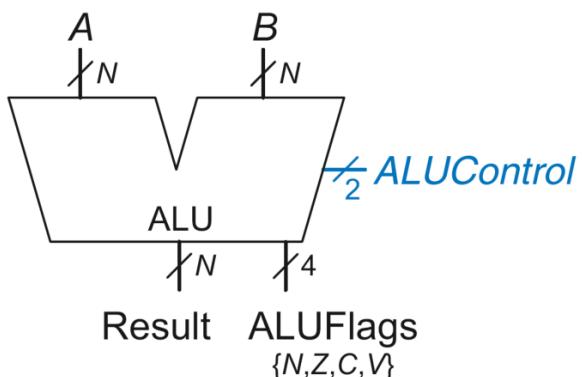
Dataflow

Esempio: DReg

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



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

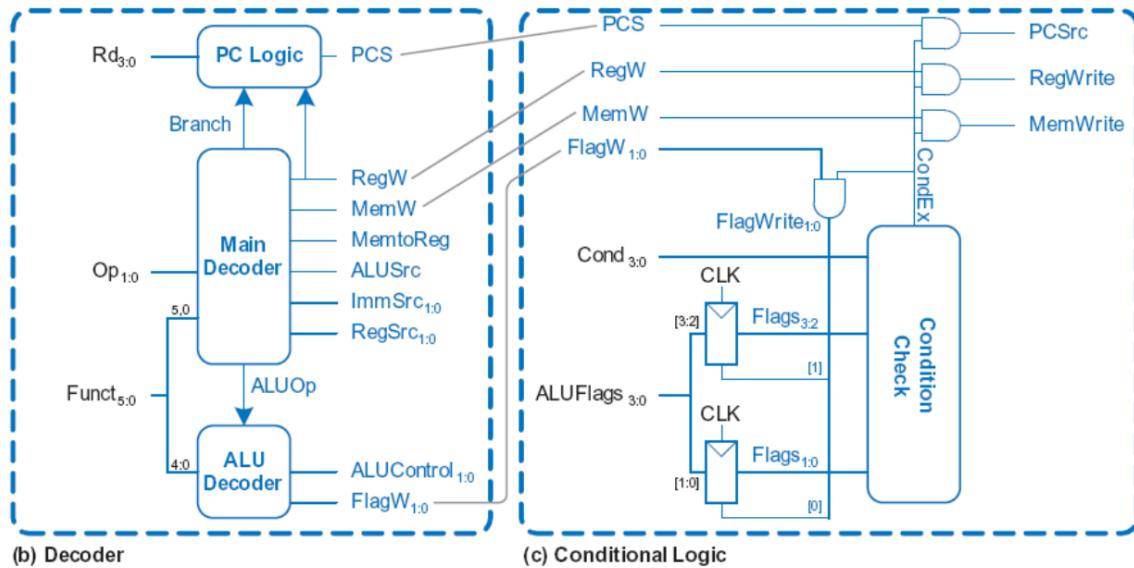
Logica condizionale della CU

La logica condizionale è quella parte dell'unità di controllo che si occupa di verificare che la condizione codificata nei 4 bit più significativi di un'istruzione (cond) sia effettivamente soddisfatta e che in tal caso abilita la scrittura e l'aggiornamento dei flag rendendoli operativi e con i rispettivi segnali sul datapath.

Quindi determina se l'istruzione deve essere eseguita in base al campo cond dell'istruzione e ai valori attuali delle flag NZCV (i bit flags3:0). Se l'istruzione non deve essere eseguita le abilitazioni alla scrittura e il segnale PCSrc sono forzati a 0. La logica condizionale aggiorna anche alcune o tutte le flag ai valori ALUFlags quando FlagW è attivato dal decoder dell'alu e la condizione di esecuzione dell'istruzione si è verificata.

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 PCSrcWrite).

Tali segnali possono essere azzerati dalla logica condizionale, se la condizione non è soddisfatta.



Condizioni su flags di stato

Le condizioni sono stabilite partendo dall'istruzione CMP che fa la sottrazione del primo argomento col secondo argomento. Per ogni mnemonico di condizione è associata un'espressione booleana nei flags di stato NZCV

Dopo aver eseguito l'istruzione CMP i flags vengono aggiornati nel CPSR che serve alla logica condizionale per prelevare i valori NZCV per calcolare le espressioni delle condizioni ciò avviene anche aggiungendo il suffisso -S all'istruzione, solo che il risultato dell'istruzione poi viene salvato in un registro

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

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	$Z \text{ OR } (N \oplus V)$

La differenza A-B si fa col CMP

Dimostrazione delle CondEX degli mnemonici

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

Mnemonic	Name	CondEx
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	$Z \text{ OR } (N \oplus V)$

Per dimostrare questi mnemonici si parte dall'operatore **LT** (signed less than)

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

Questa condizione può generare overflow oppure no

-Quando $A - B$ non genera overflow allora il risultato $A - B$ deve essere negativo per avere che $A < B$. Quindi faccio $A - B$, verifico che non ci sia overflow (bit $V = 0$ dei flags) e se il risultato è $<$ di 0 (ossia il bit $N = 1$) allora la condizione è rispettata. Questo accade sempre quando A e B hanno lo stesso segno. **Quindi se $V=0$, deve esserci $N=1$**

-Può accadere che $A - B$ generi overflow: in tal caso avremo che $V=1$. Ciò significa che A e $(-B)$ hanno segni diversi. In questo caso $A < B$ è vera sse A negativo e B positivo, cioè A negativo e $(-B)$ negativo. Essendoci overflow il risultato sarà per forza positivo, discorde dai due segni **quindi se $V=1$, deve esserci $N=0$.**

Ecco che abbiamo N xor V

Ricaviamo ora **LE** (signed less than or equal)

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

Un operando A è minore uguale di un altro B se e solo se **$A - B < 0$** ossia è rispettata la condizione di **LT** (N xor V) oppure **(or, +)** se **$A - B == 0$** cioè **A e B sono uguali, in questo caso si aggiunge il bit di flag $Z=1$** .

Le altre due condizioni **GE** e **GT** si ottengono per negazione delle precedenti

Dimostrazione degli mnemonici che valgono per interi senza segno

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 per l'istruzione CMP, A-B che semantica ha nella rappresentazione senza segno?

-Sappiamo che $A-B$ è in complemento a due $A+(B+1)$, dove B è la negazione bit a bit di B . Questo vale per numeri con segno.

-Nella rappresentazione senza segno invece per fare la sottrazione, complementare bit a bit corrisponde a fare **1-bit corrente, quindi 2^n-1-B**

$$\sim(X_{N-1}\dots X_0) = (1-X_{N-1})\dots(1-X_0) = 1\dots 1 - (X_{N-1}\dots X_0) = 2^N - 1 - B$$

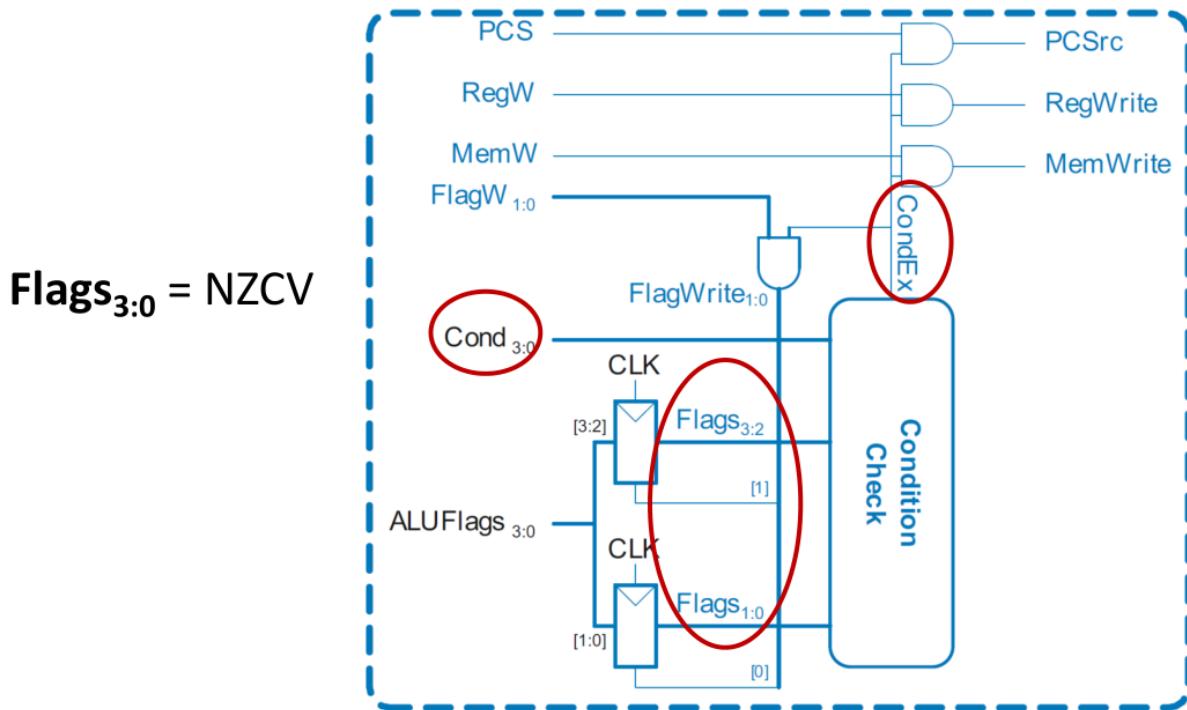
- Quindi $A+(\sim B+1)$ corrisponde nella rappresentazione senza segno a 2^N+A-B

Perchè **-1 e +1 si semplificano**

- 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 $\bar{Z}C$

$$A \geq B = !(A < B) \Rightarrow \text{NOT}(\text{NOT } C) = C$$

Logica condizionale dal punto di vista architettonico



Prende dal decoder i valori in input PCS, RegW, MemW, FlagW1:0 , i 4 bit di Cond e ALUFlags. Ha come output PCSrc, RegWrite, MemWrite.

E' composta da una logica che verifica la condizione "condition check" che prende in input i bit di cond dell'istruzione perchè per ogni codice diverso verificheremo una condizione diversa su flag di stato. Poi prende i 4 flags ALUFlags del CPSR che sono CV (3:2) e NZ (1:0). in base ai flag verifica se la condizione è soddisfatta o meno.

Es. Se abbiamo il codice MI prende il valore dei flag NZ, in particolare valuta N e se esso è uguale a 1 il valore di uscita CondEx sarà uguale a 1, altrimenti condex=0.

Il valore di uscita del condition check Condex va in and con tutte e tre le uscite di output PCSrc, RegWrite e MemWrite e quindi se vale 0 gli output vengono forzati a 0 e non dovremo aggiornare i flag di stato, quindi nessun registro o memoria.

Se il valore di uscita Condex=1 e quindi la condizione è rispettata i segnali PCS RegW e MemW diventano operativi e quindi si tramutano nei vari segnali finali di output uguali 1 e diventano operativi, essendo riportati nel datapath. Nel caso S=1 nelle istruzioni di data processing si aggiornano anche i flags nel CPSR

I flagw valgono 1 nel caso di operazioni aritmetiche (aggiornano tutti i flags). Valgono 10 per operazioni logiche che fanno sì che si aggiornano solo NZ. In questo caso l'abilitatore flags1:0 va a 0

Analisi delle prestazioni

Ogni istruzione nel processore a ciclo singolo impiega un ciclo di clock, quindi il **CPI è 1**. Ogni porzione del datapath e la control unit producono dei ritardi.

Ad esempio c'è un ritardo dovuto al caricamento del nuovo indirizzo nel PC nel fronte di salita del clock.

C'è un ritardo di lettura dell'istruzione in memoria, dovuto al decoder che deve impostare i segnali e gli abilitatori delle memorie e dei vari mux in modo opportuno per far sì che il datapath dell'istruzione sia corretto.

Inoltre lungo il datapath ci sono 2 ritardi che avvengono in contemporanea ad esempio quando abbiamo un'istruzione di tipo LDR da una parte dobbiamo ricavare il base address da registro e dall'altra l'offset dell'immediate. Queste istruzioni sono eseguite in parallelo e producono ritardi dovuti al mux, al tempo di risposta in lettura dal file register, ritardo dovuto all'extender per l'estensione dell'immediate a 32 bit.

Abbiamo un ritardo dovuto all'ALU che deve eseguire l'operazione. Ritardo in scrittura sul register file.

Tutti i ritardi tra loro quando si verificano costituiscono diversi **critical path per l'istruzione LDR**

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.

Sommendo tutti questi tempi parziali otteniamo il tempo di ritardo totale che il processore a ciclo singolo impiega per eseguire l'istruzione di LDR

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

In realtà se analizziamo quanto valgono i ritardi notiamo che il ritardo in lettura dal register file Trfread è molto maggiore del ritardo che impiega l'extender ad estendere una parola. questo vuol dire che nella maggior parte delle implementazioni l'alu, la memoria ed il register file sono sostanzialmente più lenti di tutti gli altri blocchi combinatori, per tanto il tempo di ciclo può essere semplificato come:

Table 7.5 Delay of circuit elements

$$T_{c1} =$$

$$t_{pcq_PC} + 2t_{mem} + t_{dec} + t_{RFread} \\ + t_{ALU} + 2t_{mux} + t_{RFsetup};$$

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

2 perchè facciamo 2 accessi in memoria
e passiamo in 2 mux

Limiti architetture a ciclo singolo

Le architetture a ciclo singolo hanno **3** 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 architetture a 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 processore MULTICICLO

considereremo un limitato set di istruzioni:

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

LDR

Si riuniscono istruzioni e dati in un'unica memoria.

Si suddivide l'istruzione di LOAD in più fasi.

La prima fase è quella di FETCH.

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

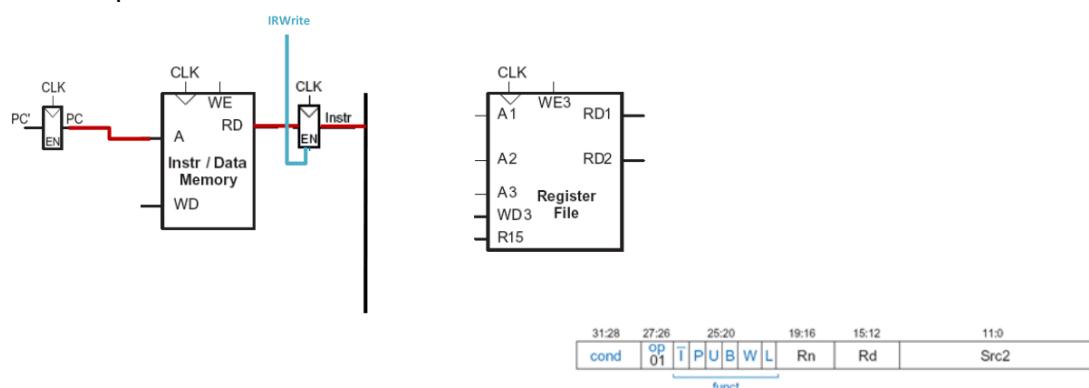
Esso è associato alla memoria dati/istruzioni, quindi il suo output PC è l'input della porta A della memoria, che riceve l'indirizzo dell'istruzione da eseguire.

La memoria dato l'indirizzo A ritorna in output il contenuto sulla porta RD1, cioè l'istruzione.

La porta RD1 è collegata ad un registro IR (Instruction Register).

L'istruzione a 32 bit viene letta e memorizzata nel registro IR.

Il registro IR riceve un segnale IRWrite che indica quando caricare una istruzione. In questo caso, nella fase di fetch, è uguale a 1. Mettiamo un registro perché vogliamo suddividere le varie fasi in più clock.

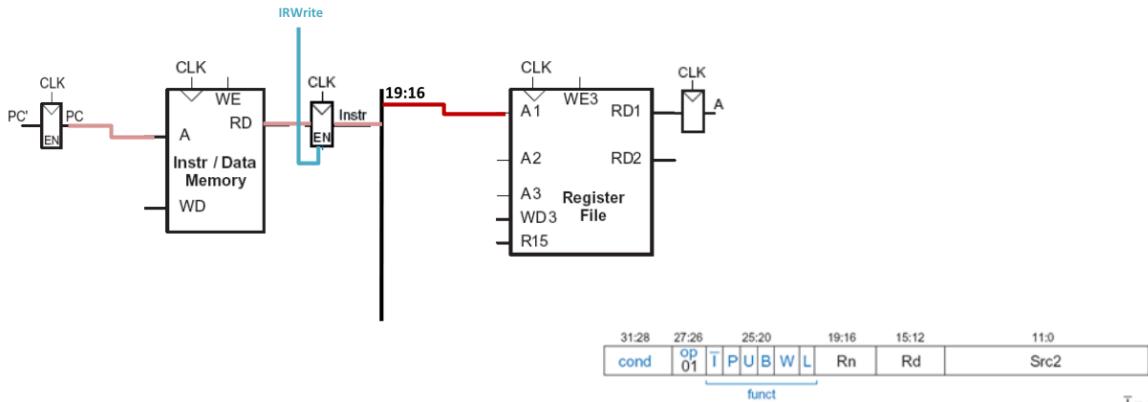


Il passo successivo è quello di leggere il registro sorgente contenente l'indirizzo di base. Questo registro è specificato nel campo Rn dell'istruzione, Instr19:16

I 4 bit Rn vengono collegati all'ingresso della porta indirizzo del file register, (A1).

La porta A1 del file register riporta in output RD1 il valore del base address.

Il valore del base address viene memorizzato in un registro A dal file register.

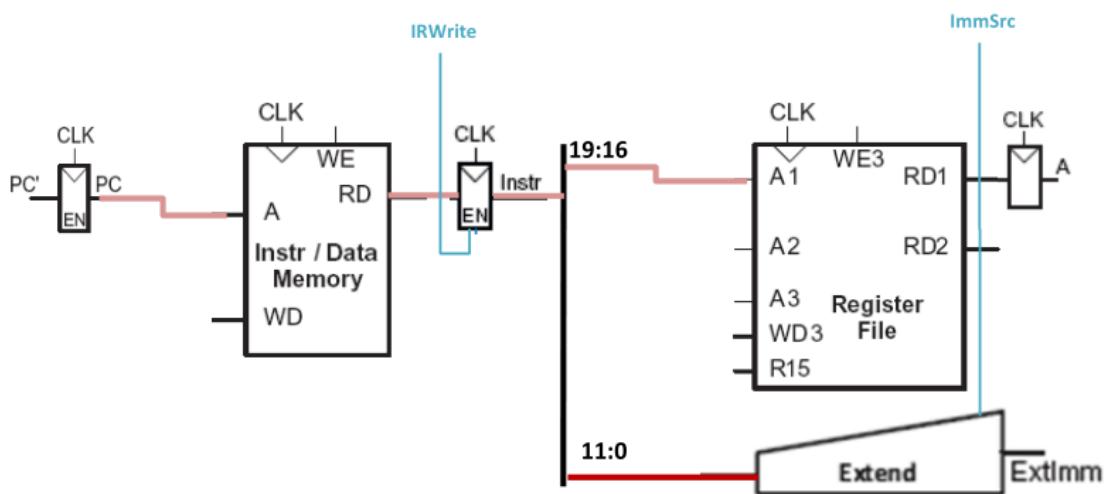


Contemporaneamente, sappiamo che l'istruzione LDR richiede anche un offset, il quale è memorizzato nell'istruzione stessa e corrisponde ai bit Instr11:0, ossia l'Immediate a 12 bit. Si prendono i 12 bit di immediate dell'istruzione e si estendono a 32 bit (unsigned) tramite la logica Extend per ottenere l'offset.

Il valore a 32 bit (ExtImm) è tale che ExtImm31:12 = 0 e ExtImm11:0 = Instr11:0.

ExtImm estende a 32 bit costanti a 8, 12 e 24 bit, quindi usiamo il selettore ImmSrc per scegliere la tipologia di offset da estendere, per istruzioni di memoria l'offset da estendere è a 12 bit unsigned.

Il valore esteso ExtImm non viene memorizzato in un registro, poiché dipende solo da Instr, che non cambia durante l'esecuzione dell'istruzione.

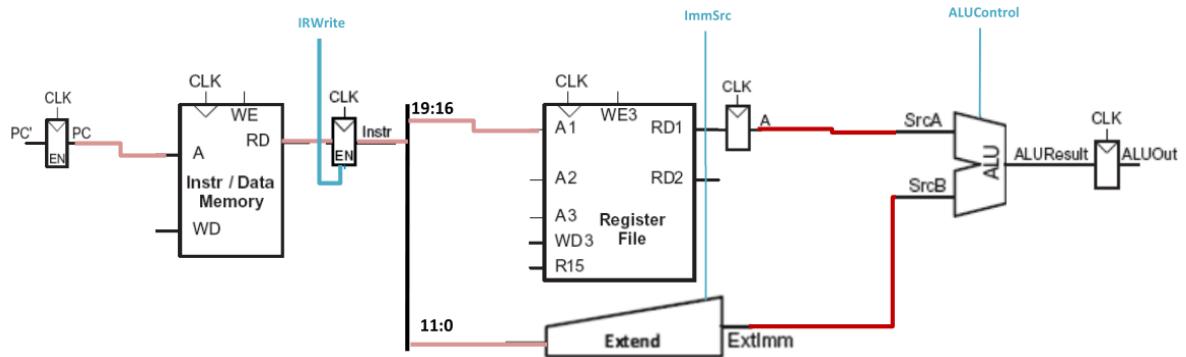


Nella fase successiva il processore deve sommare l'indirizzo di base all'offset per trovare l'indirizzo di memoria a cui leggere il dato da caricare nel registro di destinazione Rd.

La somma è effettuata dall'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).

L'alu genera un risultato a 32 bit ALUResult ossia l'indirizzo di memoria da cui fare il load del dato, il quale viene salvato in un registro ALUOut



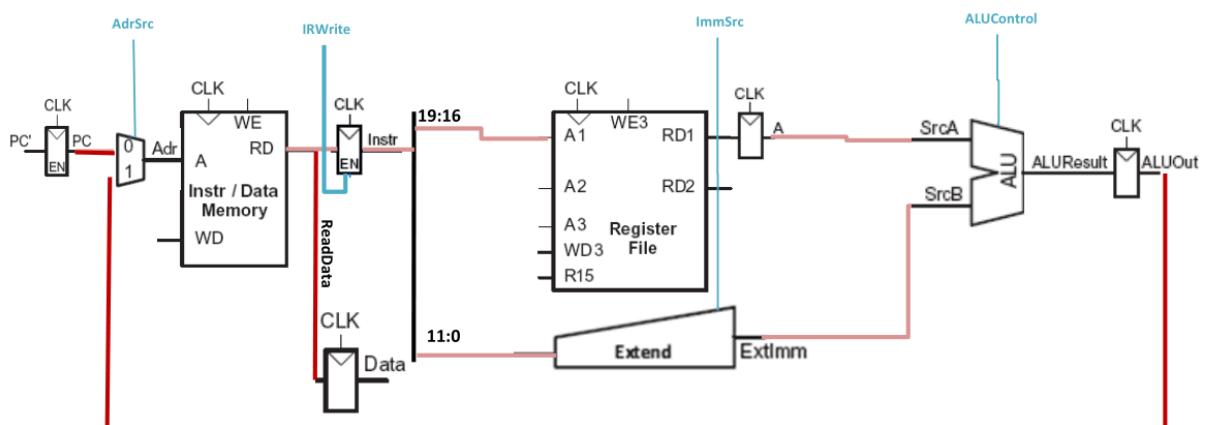
A questo punto si prende l'indirizzo generato dall'ALU e contenuto nel registro ALUOut e lo si invia alla porta A della memoria, che deve perlevare il dato a quell'indirizzo. In questo caso poiché la memoria si utilizza sia per caricare l'istruzione che per andare a prendere il dato, abbiamo bisogno di un mux che disambigua l'accesso alla memoria con ALUOut o PC. Il multiplexer è controllato dal segnale AdrSrc.

Nella fase di fetch il mux sarà 0 => viene caricata l'istruzione in memoria tramite il pc

Nella fase di caricamento del dato il mux sarà 1 poiché deve essere caricato in memoria l'indirizzo ALUOut.

Nel ciclo multiplo gli abilitatori assumono diversi valori in fasi diverse, non sono sempre stabili ad un valore.

Il contenuto del dato viene letto dalla memoria dati e inviato sul bus ReadData, e poi viene memorizzato in un registro chiamato Data.



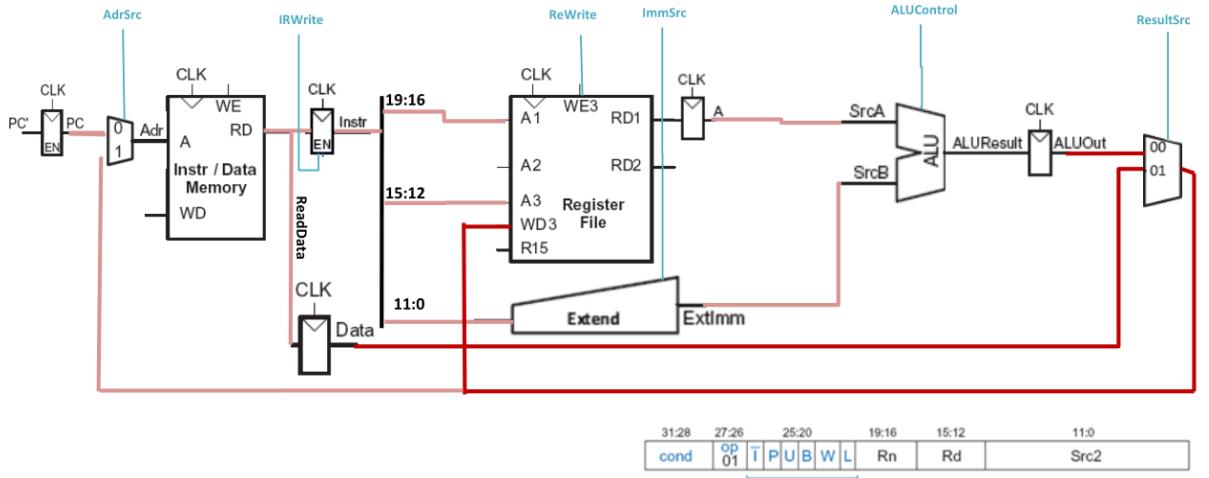
Ora il dato deve essere scritto e salvato nel file register nel registro indicato all'indirizzo contenuto nel campo Rd dell'istruzione, che è il registro di destinazione. Il registro Rd in cui scrivere tale dato è specificato dai bit 15:12 dell'istruzione Instr.

Piuttosto che collegare direttamente il Data alla porta di scrittura WD3 del file register , facciamo passare Data in un mux perchè WD3 può ricevere sia in ingresso dal registro Data ma anche direttamente dall'ALU, come per l'istruzione di data processing ADD . Quindi aggiungiamo un mux che seleziona fra ALUOut e Data.

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

Se il mux ha il segnale ResultSrc = 00 collega ALUOut a WD3.

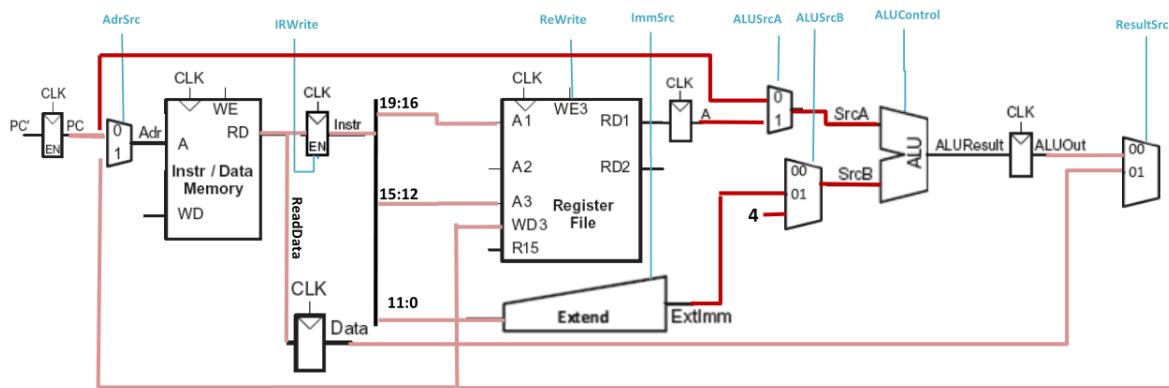
Se il mux ha il segnale ResultSrc = 01 collega Data a WD3



Un ulteriore compito a carico dell'ALU è l'incremento del PC, operazione che prima era svolta da una ALU diversa. Il valore del PC deve essere riportato verso l'alu aggiungendo un mux per sceglierlo in input il suo ingresso. Il contenuto del PC deve essere poi sommato a +4, che deve essere l'ingresso SrcB dell'alu per aggiornare il valore del PC.

Quindi aggiungiamo un mux sul primo ingresso dell'alu che permette di scegliere fra il contenuto del registro A e il PC. Sul secondo ingresso SrcB aggiungiamo un ulteriore multiplexer che permetta di selezionare fra ExtImm e la costante 4.

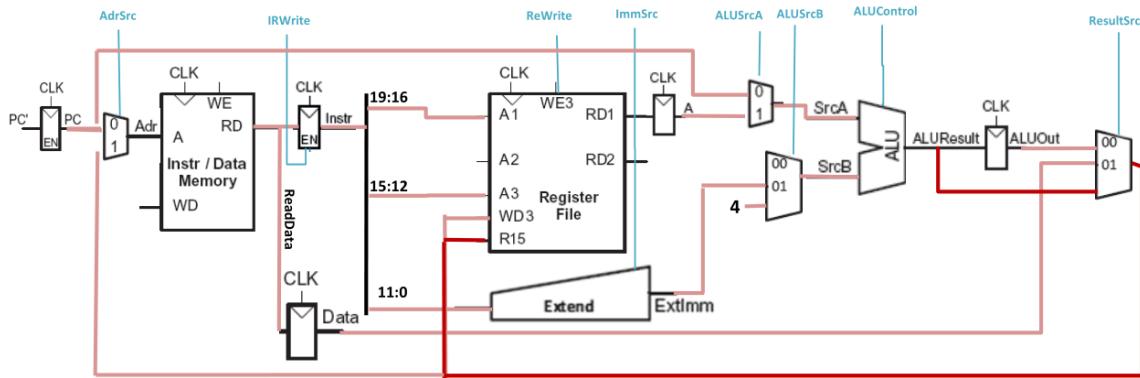
L'aggiornamento del PC avviene durante la fase di fetch e decode perché l'alu non è ancora impegnata per la somma base address+offset.



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, collegiamo ALUResult con uno dei tre ingressi del multiplexer.



Datapath STR

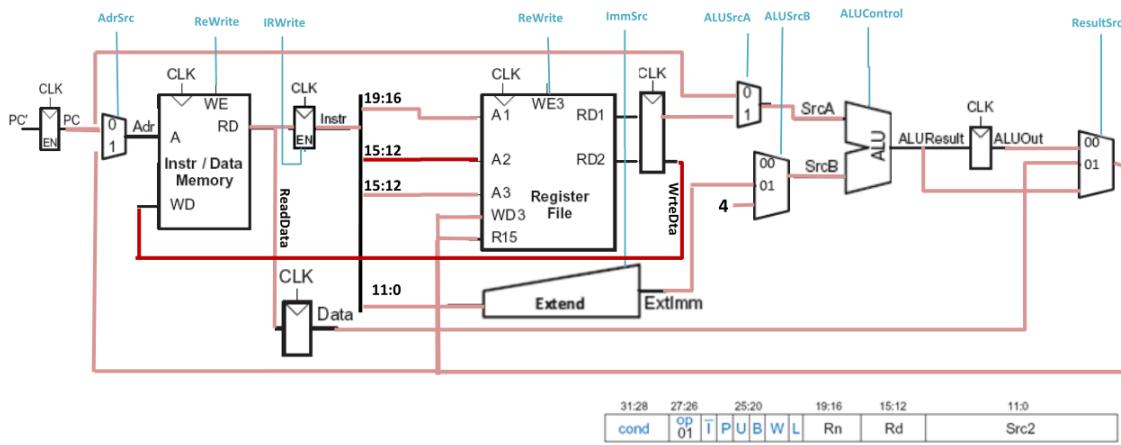
Analogamente all'istruzione di caricamento, STR legge l'indirizzo di base dalla porta RD1 del register file, estende la costante e l'ALU somma i due valori per calcolare l'indirizzo di memoria. Tutte queste operazioni sono già supportate.

In aggiunta, STR legge il registro Rd da cui scrivere, che è specificato nei bit Instr15:12.

I bit corrispondenti a Rd finiscono all'ingresso A2 del register file da cui si prende il contenuto da memorizzare in memoria sull'uscita RD2.

Il contenuto di RD2 è inserito in un registro temporaneo WriteData e al passo successivo è inviato alla memoria, con segnale MemWrite attivo, sull'ingresso WD.

Ciclo in meno rispetto a LDR



Datapath data processing

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

Per le istruzioni di data processing con registro (ADD, SUB, AND, OR), il datapath legge il secondo operando specificato da Rm, indicato nei bit Instr3:0.

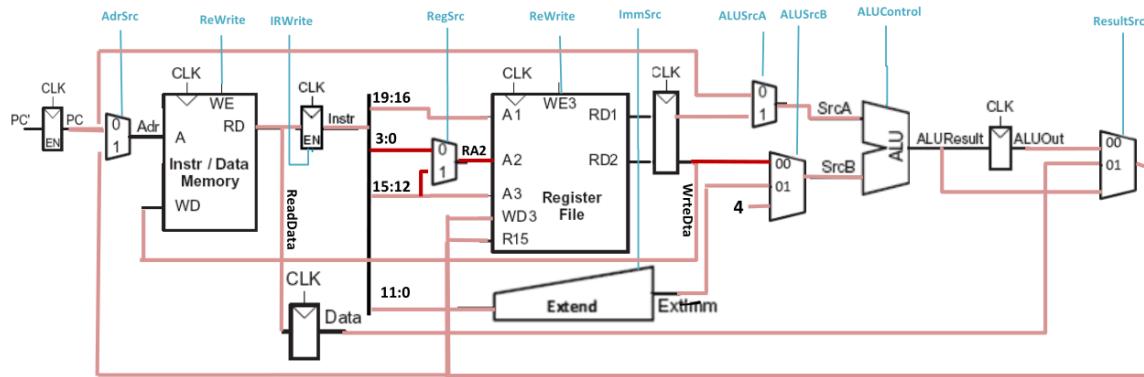
Inseriamo un multiplexer per selezionare tale campo sulla porta A2 del register file. Il mutiplexer è controllato dal segnale RegSrc. Inoltre, estendiamo il multiplexer in SrcB in modo da considerare questo caso.

ALUSrcB:

00 - SrcB da registro

01 Src da extender

4 - aggiorna pc

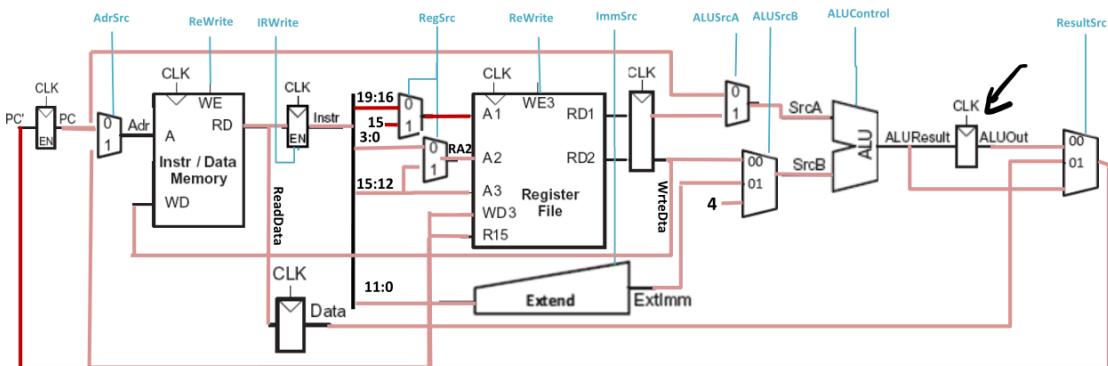


Datapath branch

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 ciclo multiplo

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 (Instr31:28, Instr27:26, e Instr25:20), 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.

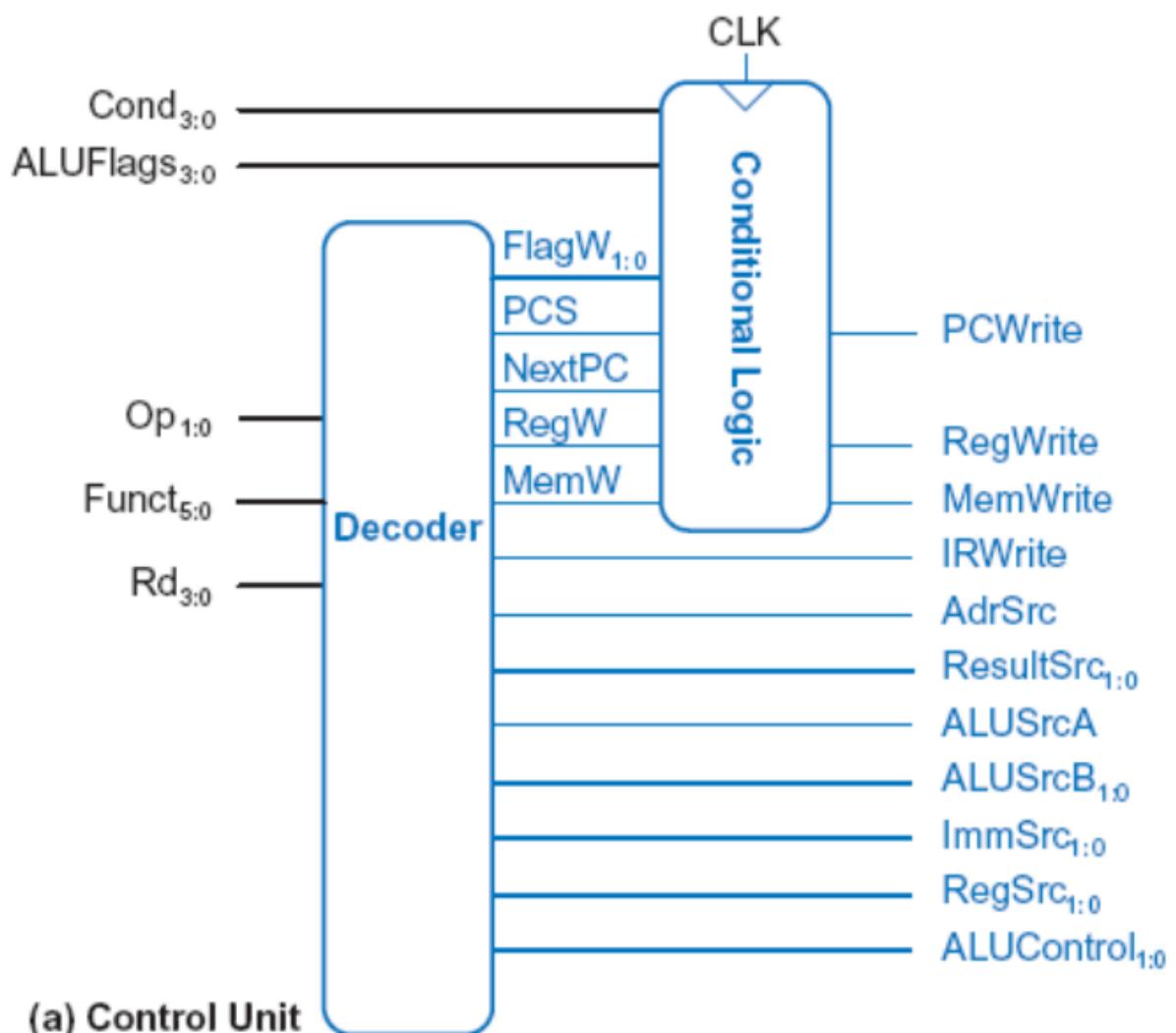
Come nel processore a ciclo singolo, l'unità di controllo è suddivisa in decodificatore e logica condizionale. Il decodificatore è progettato come una FSM, che produce i segnali appropriati per i diversi cicli, sulla base del proprio stato.

Il decodificatore è realizzato con una macchina di Moore, in tal modo le uscite dipendono solo dello stato attuale.

I valori dei selettori cambiano fase dopo fase , il decoder deve essere una macchina a stati finiti che stato per stato , fase per fase, aggiorna i valori di output dei segnali di controllo in modo differente

Il decoder principale viene quindi sostituito nel processore multiciclo da una FSM principale, che deve produrre la sequenza di segnali di controllo nei passi opportuni. Si progetta questa macchina di moore in modo che le sue uscite siano in funzione del solo stato presente.

Il decoder dell'alu e la logica del pc sono identiche a quelli del processore a ciclo singolo. La logica condizionale è quasi uguale a quella del processore a ciclo singolo: serve solo il segnale aggiuntivo NextPc per forzare una scrittura nel PC quando si calcola PC+4



Dataflow

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.

La FSM deve generare i segnali di selezione dei multiplexer, le abilitazioni dei registri e i segnali di scrittura in memoria del datapath

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.

I segnali di abilitazione RegW, MemW, IRWrite e NextPC sono elencati solo quando devono essere attivati, cioè portati a 1. Se non sono elencati si assume che valgono 0.

Dataflow LDR-STR

Il primo passo di ogni istruzione è il fetch dalla memoria dell'istruzione all'indirizzo presente nel PC, e l'incremento del PC per puntare all'istruzione successiva.

La FSM si porta in questo stato denominato Fetch al reset.

Per leggere dalla memoria l'istruzione, AdrSrc=0 in modo che l'indirizzo sia preso dal PC
IRWrite è attivato per salvare l'istruzione nel registro istruzioni IR

In contemporanea il PC deve essere incrementato di 4 per puntare all'istruzione successiva.

Il processore può adoperare l'ALU perchè inutilizzata, la usa in parallelo alla fase di fetch per calcolare PC+4.

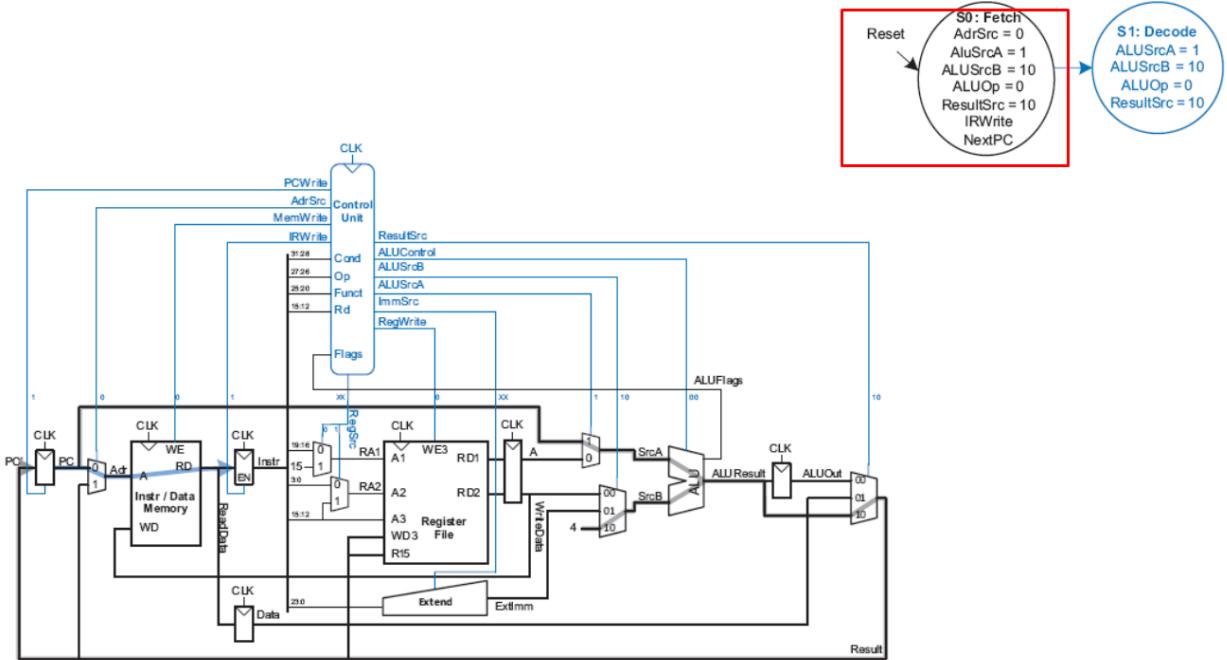
Gli ingressi dell'alu ALUSrcA=1 in modo che provenga dal PC l'indirizzo;
ALUSrcB=10, quindi si seleziona la costante 4 da sommare all'indirizzo del PC.

ALUOp=0, ALUControl=00 quindi si effettua la somma di ALUSrcA e ALUSrcB.

Per aggiornare il PC col nuovo valore PC+4, il segnale ResultSrc deve essere 10 per selezionare direttamente il risultato dell'alu ALUResult.

Inoltre NextPC=1 perchè si deve abilitare la scrittura nel PC del nuovo indirizzo, PCWrite=1.

Quindi si aggiorna il PC e si effettua il fetch dell'istruzione.



Si passa ora alla fase di Decode.

Si deve leggere il secondo operando dal register file oppure come immediato e si deve decodificare l'istruzione.

I registri e l'immediate sono selezionati da RegSrc e ImmSrc, che sono generati dal Decoder dell'ALU opportunamente sulla base dei bit Instr dell'istruzione.

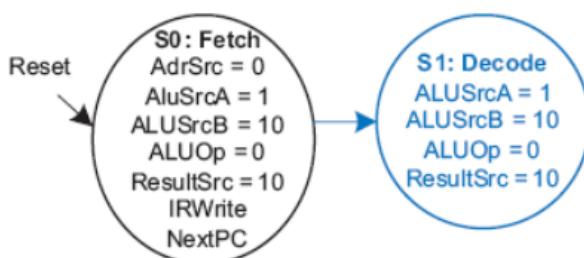
RegSrc0 vale 1 per i salti, per leggere 15 e quindi il registro R15 che contiene $Pc+8$ come SrcA da inviare all'alu. RegSrc1 è 1 per le istruzioni di scrittura in memoria STR, per leggere come SrcB il valore Rn da memorizzare

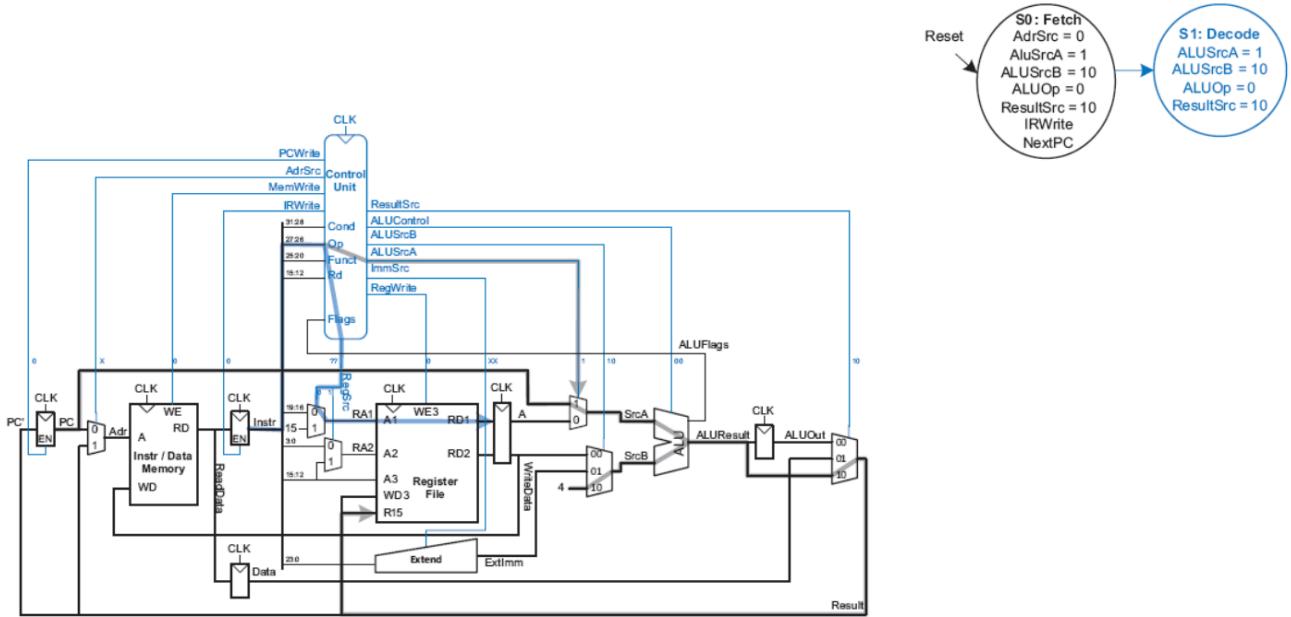
ImmSrc = 00 per istruzioni di data processing, 01 istruzioni di memoria, 10 salti

Nel mentre l'alu viene usata per calcolare il valore di R15 da aggiornare, $Pc+8$, comando ancora 4 al PC già incrementato nello stato di Fetch.

I segnali di controllo sono generati in modo tale da selezionare il valore PC come primo ingresso dell'ALU ($ALUSrcA=1$) e 4 come secondo ingresso dell'alu ($ALUSrcB=10$) e viene attivata l'operazione di somma ($ALUControl=00$).

La somma viene selezionata direttamente dall'ALU e inviata all'ingresso di R15 del register file tramite Result, per cui $ResultSrc=10$.





Ora si passa allo stato di execute.

La FSM procede ad uno dei diversi possibili stati in base ai campi op e funct dell'istruzione esaminati durante il passo Decode.

Se l'istruzione è di memoria (STR o LDR, con op=01) il processore multiciclo deve calcolare l'indirizzo di Rd sommando all'indirizzo di base lo spiazzamento esteso dall'extender.

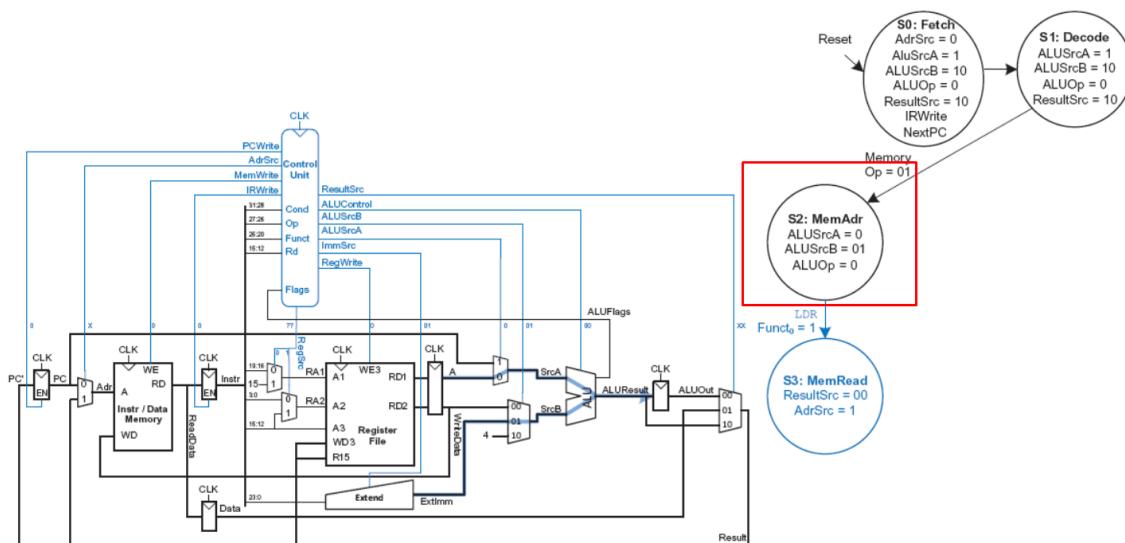
Questo richiede ALUSrcA=0 per selezionare l'indirizzo di base del register, dal file register, che poi è memorizzato temporaneamente nel register A.

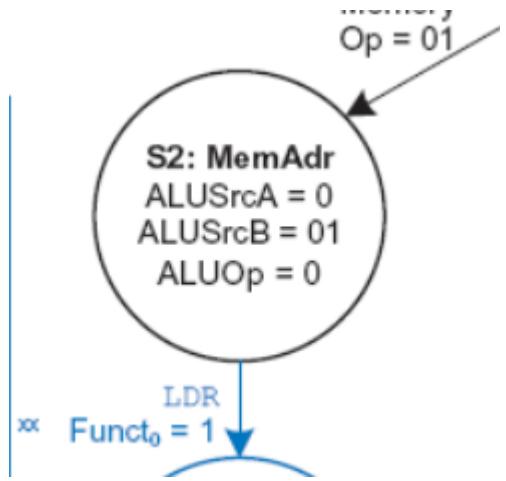
ALUSrcB=01 per selezionare l'immagine estesa da sommare al base address, ExtImm.

ALUControl=00 per effettuare la somma.

L'indirizzo calcolato viene memorizzato nel registro ALUOut per i passi successivi.

Operazione: Execute (memory address computation)





A questo punto la FSM ha due possibilità in base al tipo di istruzione:

Se l'istruzione è LDR (funct L=1) il processore multiciclo deve leggere un dato dalla memoria e scriverlo nel register file.

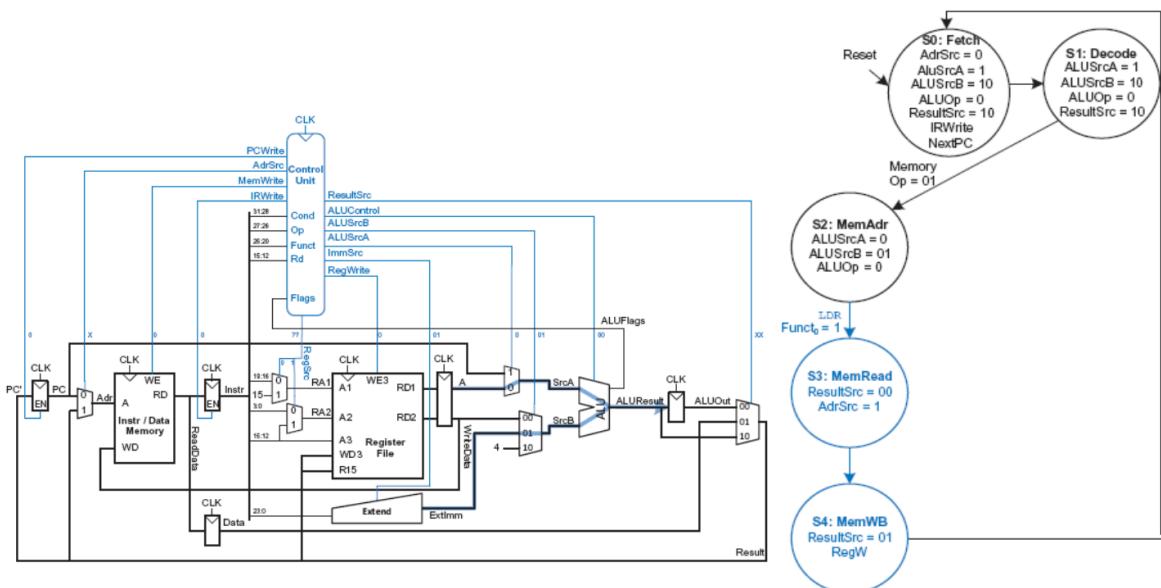
Per leggere dalla memoria ResultSrc=00 per selezionare il risultato dal registro ALUOut che contiene l'indirizzo da passare alla memoria da cui poi si va a leggere.

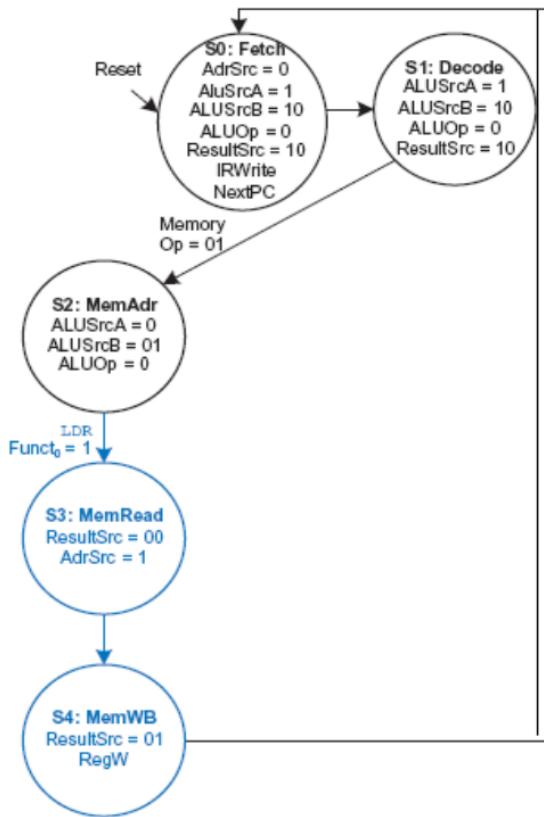
AdrSrc=1 per selezionare l'indirizzo di memoria Result appena calcolato e salvato in ALUOut. Il contenuto della parola indirizzata viene letto dalla memoria e salvato nel registro Data durante il passo MemRead.

Nel passo di scrittura finale MemWB , il contenuto di Data viene scritto nel register file all'indirizzo di Rd sulla porta WD3.

ResultSrc=01 per selezionare Result da Data e viene attivato RegW per scrivere nel register file, completando l'esecuzione dell'istruzione di LDR.

Infine la FSM torna allo stato di fetch per iniziare l'istruzione successiva



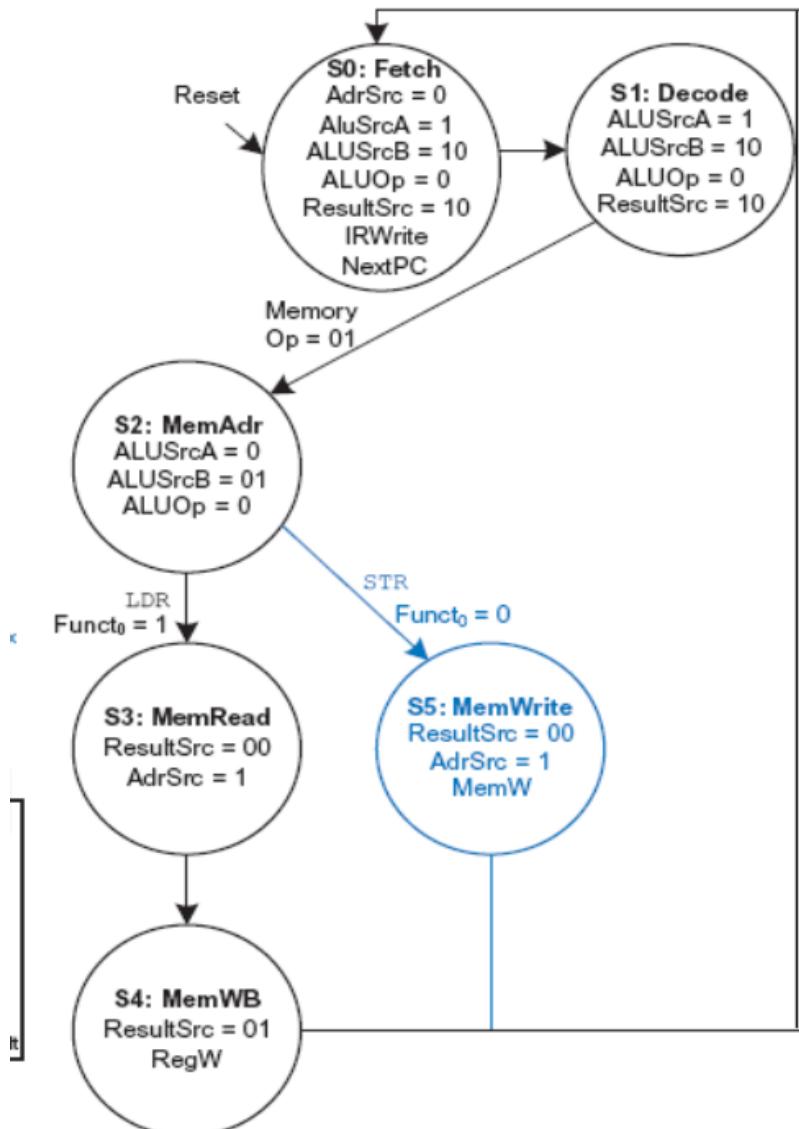


Dallo stato MemAdr, se l'istruzione è di tipo STR (funct L=0) il dato letto dalla seconda porta del register file deve essere semplicemente scritto in memoria.

Si va nello stato MemWrite.

ResultSrc=00 per selezionare l'indirizzo proveniente dall'alu e AdrSrc=1 per selezionare l'indirizzo calcolato nello stato MemAdr e salvato in registro ALUout

MemW viene attivato per scrivere in memoria e la FSM torna nello stato di fetch.



Dataflow data processing

Per le istruzioni di elaborazione dati (con Op=00) il processore multiciclo deve calcolare il risultato usando l'ALU e scriverlo nel register file.

Il primo operando sorgente viene sempre da un registro (ALUSrcA=0)

ALUOp=1 in modo che il decoder dell'alu possa selezionare il valore appropriato di ALUControl per la specifica istruzione usando il campo cmd.

Il secondo operando sorgente proviene dal register file per istruzioni con modo di indirizzamento a registro (ALUSrcB=00) oppure da ExtImm per istruzioni con modo di indirizzamento immediato (ALUSrcB=01)

La FSM ha bisogno dei due stati ExecutR e Executel per gestire le due diverse situazioni. In entrambi casi poi l'istruzione avanza allo stato ALUWB di scrittura del risultato dell'ALU, nel quale il risultato del calcolo viene selezionato da ALUOut (ResultSrc=00) e scritto nel register file (RegW=1)

Dataflow branch

Per l'istruzione di salto il processore deve calcolare l'indirizzo di destinazione ($PC+8+offset$) e scriverlo nel PC. Durante la fase di Decode $PC+8$ è già stato calcolato e portato al register file. Quindi nello stato Branch l'unità di controllo setta $ALUSrcA=0$ per selezionare R15

($PC+8$), $ALUSrcB=01$ per selezionare ExtImm

$ALUOp=0$ per la somma con $ALUControl=00$

Il mux che produce Result seleziona $ALUResult$ (quindi $ResultSrc=10$).

Branch è attivato per scrivere il risultato nel pc

6. ARCHITETTURA

L'**architettura** di un calcolatore è come viene visto il calcolatore a basso livello dal programmatore. Essa è definita da un **set di istruzioni (linguaggio)** e dagli **operandi** di tali istruzioni che sono memorizzati nei registri.

Le parole che costituiscono il linguaggio sono chiamate istruzioni. Tutte le istruzioni definiscono un insieme ossia un set di istruzioni del calcolatore. Tutti i programmi su un calcolatore utilizzano il medesimo set di istruzioni.

Le istruzioni indicano sia l'operazione che il calcolatore deve effettuare sia gli operandi da usare, che si possono trovare in memoria, nei registri del processore o direttamente definiti nelle istruzioni.

Le singole istruzioni vengono poi codificate come stringhe binarie in quello che si definisce **linguaggio macchina**, per poi essere eseguite dal processore. L'architettura ARM rappresenta ogni istruzione con una parola di 32bit.

I processori sono dunque sistemi digitali che leggono ed eseguono istruzioni scritte in linguaggio macchina. Siccome per l'uomo è difficile leggere ed interpretare le istruzioni direttamente in linguaggio macchina si preferisce rappresentare le istruzioni in una forma simbolica: **il linguaggio assembly**.

Tutte le architetture definiscono nel proprio set di istruzioni delle istruzioni di base, che quindi sono uguali per tutte le architetture ad esempio come addizione, salto ecc.

E' bene ricordare che l'architettura non definisce la sottostante struttura circuitale di un calcolatore; infatti esistono spesso differenti realizzazioni circuitali della medesima architettura (come Intel e AMD con l'architettura x86) che possono eseguire gli stessi programmi ma con strutture circuitali diverse; sono magari ottimizzate per particolari applicazioni.

L'organizzazione specifica dei registri del processore, della memoria, dell'ALU e degli altri blocchi circuitali costruttivi del calcolatore viene chiamata microarchitettura.

Possono esistere diverse implementazioni di una microarchitettura per una stessa architettura come menzionato in precedenza.

Noi studiamo l'architettura **ARM acronimo di Advanced Risc Machines**, sviluppata nel 1980.

Si basa su un set di istruzioni RISC: reduced instruction set computer.

Il set RISC si basa su un'idea di progettazione di architetture per microprocessori che predilige lo sviluppo di un'architettura semplice e lineare. Questa semplicità di progettazione permette di realizzare microprocessori in grado di eseguire il set di istruzioni in tempi minori rispetto a una architettura CISC.

L'architettura ARM si basa su **4 principi di progettazione fondamentali:**

1. **Regularity supports design simplicity** (la regolarità favorisce la semplicità)
2. **Make the common case fast** (rendere veloci le cose frequenti)
3. **Smaller is faster** (più piccolo è più veloce)
4. **Good design demands good compromises** (un buon progetto richiede buoni compromessi).

Il linguaggio assembly è la forma human-readable del linguaggio macchina, nativo del calcolatore. Ogni istruzione in linguaggio assembly specifica sia l'operazione da svolgere che gli operandi da elaborare.

L'operazione più comune dei calcolatori è l'addizione.

Addizione: ADD a, b, c

La prima parte di un istruzione in linguaggio ARM Assembly (in questo caso ADD) è chiamata **mnemonico** e indica l'operazione da eseguire. Tale operazione di addizione deve essere eseguita sugli **operandi sorgente** (b,c) e il risultato ottenuto dall'operazione di addizione su **b** e **c** deve essere scritto in **a**, l'**operando destinazione**.

Sottrazione: SUB a, b, c

Applicazione del 1° principio ARM: Regularity supports design simplicity

Il formato costante delle istruzioni arm è un'applicazione del primo principio Regularity supports design simplicity infatti istruzioni con un numero costante di operandi (due sorgenti e una destinazione) sono più facili da gestire in hardware. Istruzioni di alto livello più complesse vengono tradotte in più sequenze di semplici istruzioni arm (ad es. se ho una somma e una sottrazione in sequenza effettuo prima la somma e poi sul risultato ottenuto effettuo la sottrazione ottenendo il risultato finale).

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, e questo facilita l'encoding in hardware.**

Applicazione del 2° principio ARM: Make the common case fast

L'uso di più istruzioni assembly per eseguire attività complesse è un esempio di applicazione del secondo principio: rendere veloci le cose frequenti. Infatti l'architettura ARM rende veloci le cose frequenti perché **comprende solo istruzioni semplici e di uso frequente**. Il numero di istruzioni diverse è tenuto basso in modo tale che il circuito richiesto per decodificare ed eseguire le istruzioni sia semplice e veloce. **Elaborazioni più complesse eseguite più raramente sono realizzate con sequenze di molteplici istruzioni ARM semplici. ARM quindi si basa su un'architettura RISC che ha un insieme limitato di istruzioni.** Architetture con molte istruzioni complesse si basano su un'architettura CISC (complex instruction set computer). CISC comporta hardware aggiuntivo e sovraccarichi che rallentano anche le istruzioni semplici. **L'architettura RISC minimizza la complessità hardware e la codifica necessaria per le istruzioni mantenendo piccolo l'insieme di diverse istruzioni.**

Esempio

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

Operandi

Un'istruzione lavora su **operandi i quali hanno un nome simbolico e afferiscono a locazioni di memoria fisiche**. Infatti le istruzioni richiedono locazioni fisiche dalle quali prelevare i dati binari. **Gli operandi e quindi i dati possono essere memorizzati nei registri del processore, o in memoria oppure essere costanti (immediates) memorizzate nelle istruzioni stesse.**

I calcolatori usano locazioni diverse per gli operandi al fine di ottimizzare velocità e capacità: gli operandi memorizzati come costanti o nei registri sono veloci da raggiungere ma possono contenere solo piccole quantità di dati. Dati aggiuntivi devono essere prelevati dalla memoria,

capiente ma lenta. **ARM opera su dati a 32 bit**. Non esistono istruzioni che operano direttamente sulla memoria centrale. Ogni volta che si deve eseguire un'istruzione su un dato che si trova in memoria si deve fare prima un load del dato dalla memoria in un registro e poi si può operare su quel registro contenente il dato che ci interessa. Il risultato dell'operazione sarà salvato in un altro registro e quindi se si vuole salvare il dato in memoria centrale bisogna effettuare un'operazione di STORE.

Immediate (costanti)

Un immediate è un operando costante memorizzato direttamente all'interno di un'istruzione.

I valori degli immediates sono immediatamente disponibili nell'istruzione stessa e non richiedono accessi a registri o memoria. Un immediate è preceduto dal carattere **#** e può essere scritto in decimale o esadecimale. In assembly ARM le costanti esadecimali cominciano con **0x**. Gli immediates sono numeri senza segno (**unsigned**) a **8 o 12 bit**.

Registri: applicazione del 3° principio, smaller is faster

Per quanto riguarda i registri del processore, ARM ha solo **16 registri**, più veloci della memoria e ogni registro dell'architettura è costituito da **32 bit**.

Le istruzioni ARM operano esclusivamente sui registri, quindi i dati residenti nella memoria devono essere spostati in un registro prima di poter essere elaborati.

Infatti le istruzioni hanno bisogno di raggiungere rapidamente gli operandi per poter essere eseguite velocemente, ma gli operandi salvati nella memoria richiedono tempi lunghi d'accesso in memoria per poter essere recuperati. Per questo vengono definiti un numero limitato di **registri per memorizzare gli operandi più usati**.

ARM ha 16 registri globalmente indicati come **register file**. Meno sono i registri e più rapidamente sono accessibili, questa è un'applicazione del 3° principio più piccolo è più veloce. Infatti leggere un dato da pochi registri è molto più rapido che leggerlo da una memoria grande. Il register file di solito è costituito da un array di memoria SRAM.

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, soprattutto per risolvere il problema delle chiamate a funzione. Per questo sono state introdotte delle convenzioni.

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

argomento (argument): [registri R0 a R3] il registro può essere utilizzato per memorizzare un argomento (o parametro) che la funzione chiamante passa alla funzione chiamata durante le chiamate a funzione.

valore di ritorno (return value): [registro R0] è un valore che viene memorizzato solo dal registro R0. Nelle chiamate a funzione, una volta che la funzione chiamante ha chiamato una subroutine, questa funzione chiamata viene eseguita e deve ritornare un valore di ritorno al suo

termine, che viene memorizzato sempre nel registro R0. Questo è il motivo per cui possiamo ritornare solo un valore per ogni funzione in C ad esempio, visto che abbiamo un unico registro a disposizione per memorizzare il valore di ritorno. La funzione chiamante, al termine della subroutine, va a vedere nel registro R0 il valore di ritorno della funzione chiamata.

variabile temporanea (temporary variable): [registri R0 a R3] le variabili si dividono in temporanee e saved variables. Le temporary variables sono quelle variabili che una funzione utilizza in maniera temporanea e che possono essere modificate da altre funzioni. Al termine dell'esecuzione i valori nei registri che contengono la variabile possono cambiare.

saved variables: [registri R4 a R11] sono delle variabili contenute nei registri che non possono essere modificate; quando la funzione chiamante chiama una subroutine, quando essa ritorna il valore in R0 deve lasciare i valori contenuti nei registri su cui ha operato così come li ha trovati al momento della chiamata a funzione. La funzione chiamata se vuole operare su questi registri deve prima memorizzare i valori iniziali di questi registri, salvandoli in memoria con un operazione di STORE, dopodiché può operare su questi registri ma prima di ritornare alla funzione chiamante deve effettuare un operazione di LOAD dalla memoria per ripristinare i valori iniziali dei registri saved variables che ha utilizzato, prima di ritornare il valore di ritorno alla funzione chiamante. In questo modo la funzione chiamante non si "accorge" delle operazioni effettuate su quei registri.

I registri R13, R14 e R15 gestiscono il processo di chiamata e di ritorno di una funzione

R15 (PC - program counter): fra questo registro e il program counter effettivo c'è sempre una differenza di 4, vuol dire che R15 punta sempre all'istruzione successiva a quella che sta per essere eseguita. Il PC contiene l'indirizzo dell'istruzione corrente che si sta eseguendo, non si può utilizzare come operando.

R14 (LR - link register): serve per effettuare il ritorno dalla funzione chiamata alla funzione chiamante

R13 (SP - stack pointer): contiene l'indirizzo che punta alla testa dello stack utilizzato per le chiamate a funzione. Infatti si deve organizzare la memoria in modo tale che due funzioni non vadano ad occupare entrambe le stesse locazioni di memoria; non si deve fare in modo che per esempio la funzione chiamante ha salvato un valore in memoria ad un certo indirizzo e la funzione chiamata utilizzi proprio l'indirizzo di quel valore salvato per salvare altri dati. Quindi deve esserci un'organizzazione dello spazio degli indirizzi in memoria in modo che nessuna funzione occupi lo spazio di un'altra funzione: ciò è realizzato mediante uno stack o pila.

Istruzione MOV

L'istruzione mov è utile per inizializzare i valori dei registri e quindi delle variabili. In particolare inizializza variabili con immediates, spostando il contenuto della costante in un registro (es. MOV R4, #0 oppure MOV R5, #0xFF0) . Oppure l'istruzione mov può usare anche registri come operandi sorgente copiando il contenuto di un registro in un altro registro (es. MOV R7, R1). Quindi MOV viene usato per spostare il contenuto di un registro in un altro registro o per generare delle costanti.

Memoria

I dati possono essere salvati anche in memoria che al contrario dei registri che sono pochi e veloci, essa è grande e più lenta. Per questo motivo le variabili utilizzate spesso vengono tenute nei registri. Siccome in ARM le istruzioni operano solo sui registri , i dati presenti in memoria devono essere copiati nei registri prima di essere elaborati.

Istruzioni condizionali ed iterative

Costrutto if in assembly

Il codice assembly del costrutto if valuta la condizione opposta rispetto a quella presente nel codice di alto livello. Dal momento che ogni istruzione ARM può essere condizionata, il codice assembly arm può essere sostituito in forme più compatte usando gli mnemonici tipo EQ, NE.. detti Mnemonici di condizione (tabella 6.3 pag. 227)

La versione con istruzioni condizionate è più corta e anche più veloce perché richiede di eseguire meno istruzioni. Poi i salti a volte introducono ritardi mentre l'esecuzione condizionata di istruzioni è sempre veloce. In generale quando un blocco di codice è costituito da una sola istruzione conviene usare l'esecuzione condizionata invece dei salti, che diventano utili quando il blocco di codice è più lungo perché evitano la fase di fetch di istruzioni che poi non saranno eseguite.

CMP: effettua la sottrazione tra i due registri e aggiorna i flag di stato NZCV

if/else: valuta la condizione opposta rispetto a quella presente nel codice di alto livello. se la condizione opposta è vera, l'istruzione di salto salta il blocco if e va eseguire il blocco else. altrimenti si esegue il blocco if che termina con un salto incondizionato per saltare il blocco else, anche in questo caso ogni istruzione può essere condizionata usando una forma compatta con mnemonici condizionali.

costrutto while: come per il csotrutto if/else il codice assembly del ciclo while valuta la condizione opposta rispetto a quella presente nel codice di alto livello: se la condizione opposta è vera allora si esce dal ciclo, altrimenti il salto condizionato non viene effettuato e si esegue il corpo del ciclo.

costrutto for: ha una semantica del seguente tipo

**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

In ARM, i loop decrescenti fino a 0 sono più efficienti.

Si risparmiano 2 istruzioni per ogni iterazione:

- Si accorpano decremento e comparazione: **SUBS R0, R0, #1**
- Solo un branch invece di due

Arrays

Per facilitare memorizzazione e accessi dati simili possono essere raggruppati in una struttura chiamata array. L'array consente l'accesso ad una quantità di dati simili. Esso memorizza i suoi dati (elementi) in indirizzi sequenziali di memoria. Ogni elemento dell'array è identificato da un numero detto Index che da accesso al singolo elemento. Il Size dell'array definisce il numero di elementi dell'array.

In un array (parole di 32 bit) l'indirizzo del primo elemento è detto Base address e rappresenta l'array. Tutti gli altri elementi dell'array sono raggiungibili a partire dal base address.

Siccome l'array è costituito da parole di 32 bit, gli elementi dell'array avranno indirizzi che vanno di 4 in 4.

```
#0 = array[0]  
#4 = array[1]  
#8 = array[2]
```

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
```

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

Le funzioni ricevono valori di ingresso denominati parametri e forniscono un valore di uscita denominato valore di ritorno. Quando una funzione ne chiama un'altra, la prima funzione detta caller (chiamante) e la seconda funzione detta callee (chiamato) devono accordarsi stipulando una specie di “contratto” su dove mettere i parametri e il valore di ritorno.

In ARM convenzionalmente il caller mette fino a 4 parametri nei registri R0-R3 (argument) prima di eseguire la chiamata a sottoprogramma e il callee mette il valore di ritorno nel registro R0 prima di terminare. Con questa convenzione entrambe le funzioni sanno dove trovare parametri e valore di ritorno. Il callee non deve interferire con le attività del caller e non deve invadere il suo spazio di memoria. Questo significa che deve sapere dove mettere il valore di ritorno ma anche che non deve modificare nessuno dei registri o delle celle di memoria necessari al chiamante.

Per contratto (convenzione):

- **il caller** deve passare gli argomenti alla funzione chiamata (callee) e poi deve eseguire un jump alla prima istruzione sempre della funzione chiamata (callee).
- **il callee** ossia la funzione chiamata esegue le istruzioni contenute al suo interno e poi deve ritornare il risultato finale al caller, ritornando nel punto in cui è stata chiamata dal caller. Poi non deve sovrascrivere i registri saved variables (R4-R11) e la memoria che è occupata dal caller.

Convenzioni ARM per chiamata a funzione

Il passaggio dal caller al callee viene effettuato attraverso un'istruzione di BL (branch and link), compito del caller. **L'istruzione di BL memorizza l'indirizzo di ritorno (cioè l'indirizzo dell'istruzione successiva a BL del caller) nel registro LR (link register) cioè R14 e poi salta al callee che esegue le sue istruzioni.**

Dopo che il callee ha eseguito le istruzioni al suo interno deve passare il valore di ritorno e tornare al caller. Ciò viene eseguito attraverso un'istruzione di MOV che prima salva il valore di ritorno nel registro R0 e poi con un'altra istruzione di MOV si copia il valore del LR (cioè l'indirizzo di ritorno al caller) nel PC (ossia il registro R15, program counter) così si ritorna al punto in cui il programma chiamante ha chiamato il callee.

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

Parametri di ingresso e valori di ritorno. Le saved variables e lo stack delle funzioni.

Per convenzione ARM le funzioni usano i registri R0-R3 per i parametri di ingresso e il registro R0 per il valore di ritorno. Se è necessario chiamare una funzione con più di 4 parametri, i parametri aggiuntivi vanno messi nello stack e non è possibile sovrascrivere indebitamente i registri R4-R11 dedicati alle saved variables.

Quindi una funzione usa lo stack per memorizzare temporaneamente i valori di questi registri prima di operare su di essi.

Infatti se una funzione chiamata ha bisogno di utilizzare questi registri “saved variables” può salvare i valori dati dal programma chiamante subito in memoria con operazioni di store STR, operare con questi registri, e poi prima di ritornare al chiamante ripristinare i valori di questi registri mediante operazioni di load LDR a partire dagli indirizzi in cui si erano salvati il contenuto di questi registri inizialmente.

Quindi per salvare i valori di questi registri prima di operare su di essi viene utilizzato lo stack, nel quale si salvano questi valori. Se salvassimo questi valori in indirizzi della memoria a caso si potrebbero sporcare delle aree di memoria riservate ad altri programmi o aree di memoria che la

funzione chiamante ha utilizzato. Lo stack fa sì che non ci sia confusione negli indirizzi di memoria che si vanno ad utilizzare, cioè che ogni funzione utilizzi porzioni di memoria diverse.

Lo stack delle funzioni

Lo stack costituisce un modo semplice di organizzare la memoria per salvare temporaneamente il valore dei registri “saved variables” che servono temporaneamente ad una funzione chiamata (callee). Ogni chiamata a funzione ha una sua porzione ben definita dello stack. Esso è organizzato come una pila, che funziona secondo una tecnica detta LIFO (last in first out). Le operazioni sullo stack sono di Expands e Contracts.

Expands: lo stack si espande, cioè usa più memoria, quando ha bisogno (operazione di push)

Contracts: lo stack si contrae, cioè libera memoria, quando non si hanno più bisogno delle variabili temporanee memorizzate in precedenza (operazione di pop)

Lo stack è di tipo LIFO perché l'ultimo elemento messo sullo stack cioè l'elemento pushed, è il primo che potrà essere estratto (cioè popped).

Una funzione può allocare spazio sullo stack per memorizzare le variabili locali, ma deve deallocare tale spazio prima di ritornare alla funzione chiamante.

Lo stack di ARM cresce in memoria verso il basso, cioè l'espansione dello stack avviene in senso decrescente, dagli indirizzi maggiori verso indirizzi minori quando il programma ha bisogno di spazio ulteriore.

Il registro R13 (SP - stack pointer) è un registro che contiene un indirizzo che “punta” alla cima dello stack, cioè all'ultimo elemento inserito. Lo stack pointer SP inizia a un indirizzo di memoria alto e si decrementa se serve spazio aggiuntivo. Ad esempio se deve contenere ulteriori parole di memoria temporanea l'indirizzo si decrementa di 4,8,16 byte ecc espandendo lo stack.

Uno degli usi più importanti dello stack è il salvataggio e ripristino dei valori dei registri “saved variables” usati da una funzione, senza modificare alcun registro tranne R0 che contiene il valore di ritorno.

La funzione chiamata deve salvare i registri sullo stack prima di modificarli e ripristinare i valori originari prima di terminare. In particolare deve svolgere i seguenti passi:

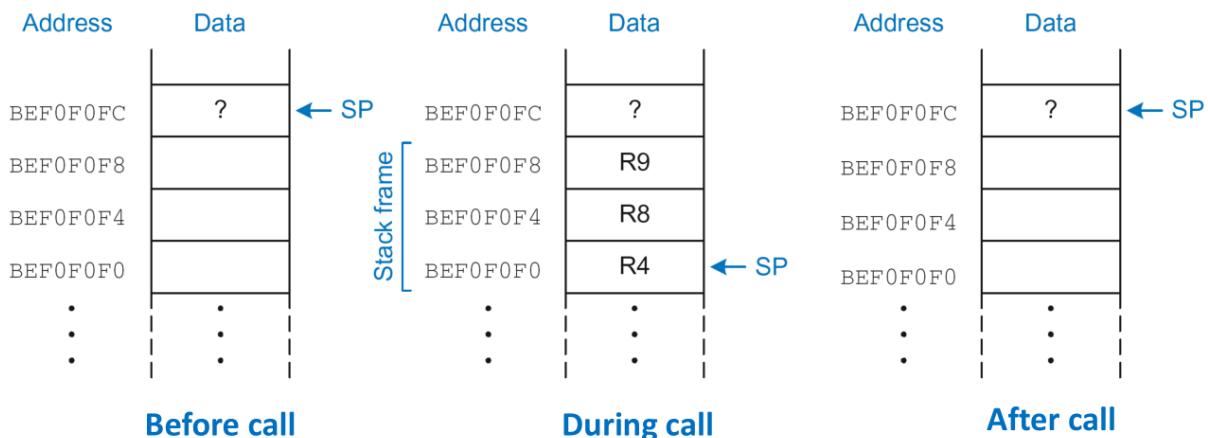
- allocare inizialmente spazio sullo stack per memorizzare i valori contenuti in uno o più registri
- salvare i valori dei registri nello stack
- eseguire le proprie attività utilizzando i registri salvati
- ripristinare i valori originali dei registri prelevandoli dallo stack
- deallocare spazio nello 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

```



Registri preservati e non preservati

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

ARM divide i registri in preservati e non preservati.

Ogni funzione deve salvare e ripristinare tutti i registri preservati che intende usare, ma può modificare liberamente i registri non preservati.

Visto che il callee può modificare qualsiasi registro non preservato, è responsabilità del

chiamante caller salvare il contenuto di questo tipo di registri prima di effettuare la chiamata, se gli servono, e ripristinarlo dopo il ritorno del callee.

La parte di stack più in alto dello stack pointer è automaticamente preservato dato che il callee evita qualsiasi modifica di parole di memoria a indirizzi maggiori di SP: in questo modo si evita di accedere agli stack frame di altre funzioni. Lo stack pointer è esso stesso preservato, perché il callee dealloca lo spazio allocato all'inizio prima di terminare, sommando a SP lo stesso valore che aveva sottratto.

Il callee quindi ha il compito di preservare i valori dei registri "saved" quindi deve effettuare dei STR e LDR dei registri R4-R11 prima di usarli (str) e dopo averli usati (ldr). Non deve sporcare il valore del LR e deve fare in modo di ripristinare il valore iniziale dello SP. Non deve sporcare nessun indirizzo che sta al di sopra dello SP.

Il caller deve salvarsi i valori dei registri temporanei R0-R3 , R12 se gli servono poiché temporanei. Deve salvarsi i valori del CPSR se gli servono. Non deve salvare qualcosa negli indirizzi al di sotto dello SP.

Funzioni pop e push in assembly

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 e di evitare la gestione a basso livello dello stack con i LDR e STR dei registri "saved".

- **Push** consente di salvare in memoria più registri aggiornare consistentemente lo stack:

PUSH {R4, R8, R9} //decrementa di 12 il valore iniziale dello SP

- **Pop** ripristina uno o più registri e incrementa consistentemente il valore dello stack:

PUSH {R4, R8, R9} //incrementa di 12 il valore iniziale dello SP

Chiamate a funzioni innestate

Tutte le funzioni che richiamano altre funzioni innestate in esse devono fare un push del LR corrente all'inizio nello stack per non sporcare e perdere il LR quando loro termineranno e dovranno ritornare il valore di ritorno alla funzione che le ha chiamate (caller), dato che il LR viene sovrascritto dalla funzione innestata

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          F2  
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 ricorsive

Si comporta sia da chiamante che da chiamato quindi deve salvare registri preservati e non preservati

Fattoriale

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

7. MEMORIE

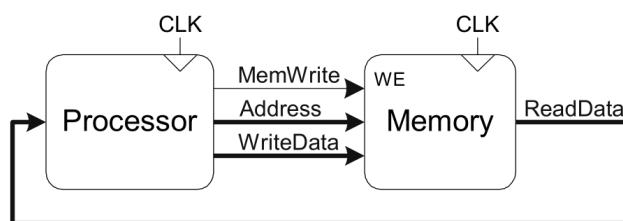
CPU vs Memorie

Nell'architettura Von Neuman la comunicazione tra la CPU (processore) e la memoria è il punto critico (collo di bottiglia) del sistema, ossia uno dei punti in cui il sistema perde più tempo durante l'esecuzione del programma. Nel tempo sono state realizzate CPU sempre più veloci e memorie sempre più grandi ma sostanzialmente le memorie hanno avuto sempre una velocità di accesso inferiore rispetto alla velocità della CPU. I due componenti viaggiano a velocità diverse.

Il processore ha di per sé una memoria interna, ossia i 16 registri contenuti nel file register, che però costituiscono solo una memoria di supporto per l'esecuzione delle singole istruzioni una dopo l'altra e dunque non è sufficiente avere registri di 32 bit; c'è bisogno di una memoria esterna, le cui operazioni principali sono operazioni di LOAD e STORE.

Il processore richiede di caricare una certa locazione di memoria o scrivere su una certa locazione di memoria fornendo un certo indirizzo. In particolare il processore comunica con la memoria attraverso un'interfaccia inviando un indirizzo alla memoria attraverso il bus indirizzi (*Address*). In caso di lettura il segnale *MemWrite* vale 0 e la memoria restituisce il dato sul bus di lettura *ReadData*. In caso di scrittura il segnale *MemWrite* vale 1 e il processore invia il dato alla memoria sul bus di scrittura *WriteData*.

Nell'architettura di Von Neumann questo processo è uguale sia per i dati che per le istruzioni dato che entrambi risiedono in memoria principale. Gli indirizzi passati dal processore attraverso il bus *Address* o sono contenuti nel program counter e si riferiscono alla prossima istruzione da eseguire o sono indirizzi che riguardano il *fetch* di dati dalla memoria.



Storicamente quindi 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 dal fatto che:

- queste memorie hanno un costo elevatissimo: ad esempio avere una memoria realizzata solo attraverso SRAM avrebbe un costo elevatissimo. Sarebbe veloce ma antieconomica.
- le memorie dovrebbero essere piazzate in gran parte sugli stessi chip delle CPU, il che non è possibile, in quanto di capacità grandi.

Gerarchia delle memorie

Si è pensato di introdurre quindi una “**gerarchia di memoria**”: cioè la memoria è organizzata in livelli in cui abbiamo:

- una quantità molto piccola di memoria (qualche Mb) estremamente veloce, la **cache (SRAM)**
- una quantità più grande (Gb) di memoria più lenta, la **main memory (DRAM)**
- una quantità di memoria molto grande (Tb) ma estremamente lenta, la **virtual memory (HDD-SSD)**

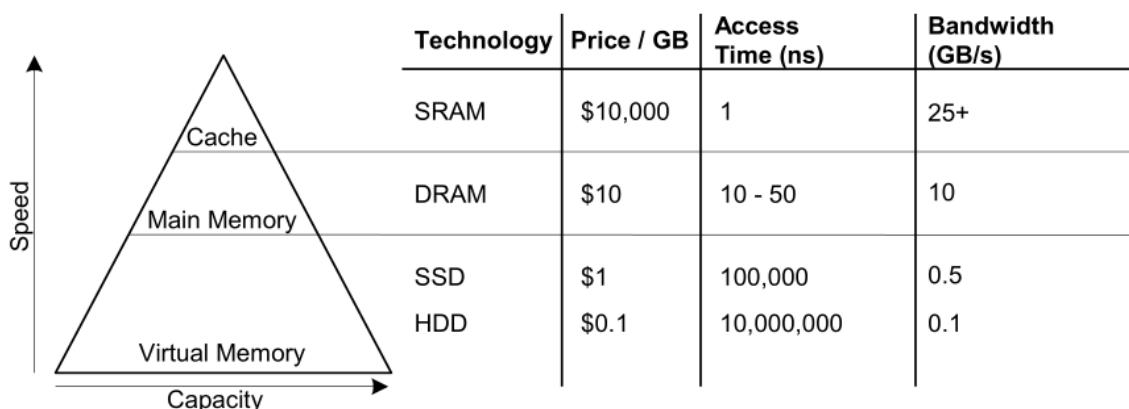
L'organizzazione è di tipo piramidale. Si sfrutta al meglio questa gerarchia di memorie per accedere rapidamente ai dati più usati offrendo comunque la capacità di immagazzinare grandi quantità di dati. Infatti idealmente una memoria dovrebbe essere veloce, grande ed economica: siccome non esiste una memoria del genere, combinando queste tipologie di memorie riusciamo ad ottimizzare tutti i parametri.

Si combina una memoria veloce, piccola ed economica con una memoria lenta, grande ed economica. La memoria veloce memorizza le istruzioni e i dati usati più di frequente, sfruttando la velocità di accesso e la capacità ridotta. La memoria grande il resto delle istruzioni e dei dati, avendo una grande capacità.

Nella gerarchia al crescere della capacità e della bandwidth, la velocità e quindi il tempo di accesso decresce.

Access time/Tempo di accesso: è la latenza, il tempo che intercorre da quando si fornisce un indirizzo da cui andare a leggere dei byte fino a quando viene letto in output il primo byte di memoria richiesto.

Bandwidth: Gigabyte di memoria letti e trasferiti al secondo dalla memoria appena inizia a leggere.



Località dei riferimenti

La cache deve essere sfruttata al meglio, essendo essa piccola ma estremamente veloce. Sappiamo che la maggior parte del tempo di esecuzione di una CPU è, di solito, impegnato da procedure in cui vengono eseguite ripetutamente le stesse istruzioni.

Questo concetto è noto come **località dei riferimenti**, cioè il fatto che alcune istruzioni in aree ben localizzate di un programma vengono eseguite ripetutamente in un determinato periodo di tempo, e si accede al resto del programma relativamente di rado.

Si manifesta in due modi: **località temporale** e **località spaziale**.

- La **località temporale** indica il fatto che un riferimento in memoria che è stato letto di recente ha un'alta probabilità di essere riletto nel prossimo immediato (futuro).
Significa che il processore ha un'elevata probabilità di accedere nuovamente nel prossimo futuro a un dato se lo ha utilizzato da poco. Oppure dopo aver inizializzato una variabile successivamente dopo poco la si utilizza in un'espressione. Oppure istruzioni in cicli.
- La **località spaziale** indica il fatto che se si richiede accesso ad un certo riferimento di memoria in lettura o scrittura, locato ad un certo indirizzo, molto probabilmente si richiederà di lì a poco un accesso in memoria ad un indirizzo vicino all'indirizzo richiesto nell'immediato.

Significa che quando il processore accede ad un certo dato ha un'elevata probabilità di accedere nel prossimo futuro ad altri dati in locazioni di memoria vicine al dato in questione. Esempio: array che è costituito da locazioni di memoria contigue (4 in 4), i riferimenti sono vicini. Anche per le istruzioni c'è una forte località spaziale, nonostante i branch che di solito fanno saltare poche istruzioni e sono abbastanza pochi.

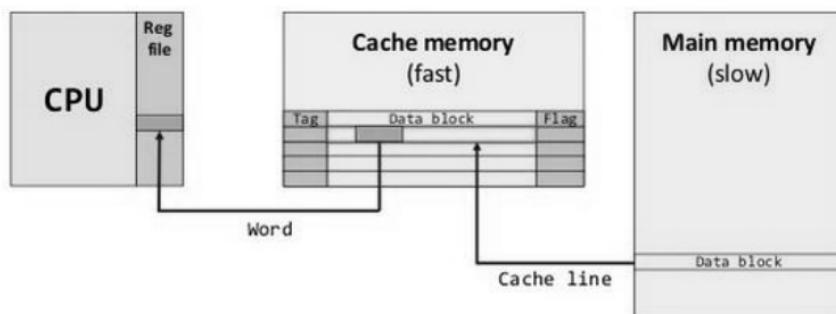
Questi aspetti in qualche modo giustificano l'utilizzo di una memoria piccola ma estremamente veloce. Si inseriscono nella memoria cache quei dati/istruzioni che hanno una maggiore probabilità di essere richiamate a breve termine.

Cache memory

Per la CPU l'organizzazione della memoria è come una black-box cioè la CPU ha una comunicazione con la memoria attraverso le operazioni di load e di store che prescinde da come è organizzata la memoria. Inoltre sappiamo che la velocità con cui la memoria risponde alle richieste di istruzioni e dati della CPU ha un peso determinante sulle prestazioni di un sistema.

La **memoria cache**, detta anche "memoria tampone" è una memoria che fa da filtro fra la CPU e la main memory che serve a velocizzare il tempo di accesso per prelevare o scrivere i dati o le istruzioni, in quanto contiene parti di programma e di dati che, volta per volta, interessano l'elaborazione. Essa è una memoria molto veloce e piccola, da qualche kb a qualche mb, ed è costituita generalmente da SRAM; è situata a bordo dello stesso chip del processore, eliminando così i ritardi dovuti alla propagazione dei segnali elettrici tra chip diversi. La cache può memorizzare sia dati che istruzioni e riduce il tempo totale di esecuzione in modo significativo ed impedisce alla CPU di "vedere" i tempi di risposta reali della memoria.

Principi di funzionamento cache



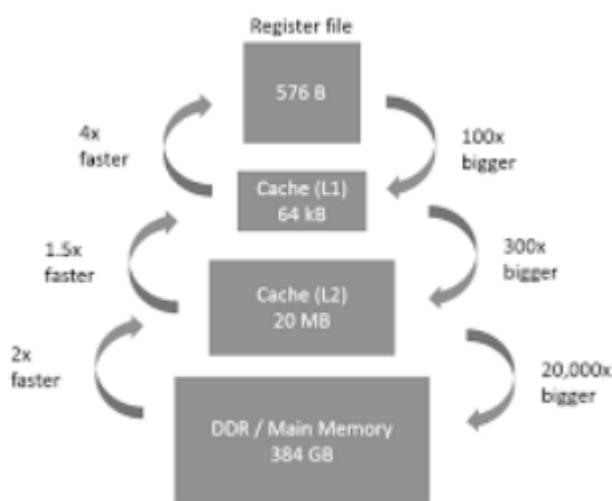
I circuiti di controllo della memoria cache sono progettati per avvantaggiarsi della proprietà della località dei riferimenti.

Abbiamo la cpu col suo file register, essa mediante le operazioni di load e store richiede dei dati o delle istruzioni (program counter) dalla memoria. Nel mezzo abbiamo la memoria cache che fa da filtro fra la CPU e la main memory. Inizialmente se il processore richiede un dato che è direttamente presente nella cache memory, tale dato viene reso immediatamente disponibile al processore rapidamente, risparmiando il tempo di accesso alla main memory. In questo caso abbiamo un **cache hit**. Se il dato o l'istruzione richiesta dal processore non è direttamente presente nella cache memory, il processore lo recupera dalla main memory. In questo caso abbiamo un **cache miss**.

In questo caso, dopo aver prelevato il dato o l'istruzione dalla main memory per la prima volta, il **principio di località temporale** secondo cui questo dato potrà essere richiesto nel prossimo futuro, ci suggerisce di copiare il riferimento di memoria nella cache in modo tale che rimanga a disposizione nel caso di una nuova richiesta, minimizzando il tempo di accesso alla main memory e facendo verificare un hit al prossimo accesso. Inoltre per il principio di **località spaziale** se il processore ha richiesto un dato o istruzione in memoria locata ad un certo indirizzo, è molto probabile che richieda nel prossimo futuro altri riferimenti di memoria (dati o istruzioni) presenti in locazioni di memoria contigue (vicine) al dato/istruzione in questione.

Dunque la cache quando preleva una parola dalla main memory preleva anche altre parole adiacenti: questo gruppo di parole è denominato blocco. Quindi **un blocco o linea di cache** è un insieme di indirizzi contigui in memoria di una qualche dimensione.

Dimensioni cache



La cache memory è a sua volta suddivisa in due livelli L1 e L2. Il livello L1 è formato da una cache piccolissima e velocissima di 64kb, vicina alla velocità dei registri della CPU. Poi c'è una cache L2 che è più lenta ed è grande una decina di Mb. Poi c'è la main memory.

Dopo aver caricato un blocco di parole di memoria in cache, ogniqualvolta il processore 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 impone che sia piccola. Se è piccola non potrà contenere una frazione ridotta delle istruzioni ed i dati della memoria principale cioè solo i dati e le istruzioni di uso frequente.

Tecniche di gestione cache

La memoria cache ha un certo insieme di indirizzi; la main memory un altro insieme di indirizzi. La main memory è molto più grande della memoria cache quindi ha molti più indirizzi di essa. Dunque deve esserci un **“mapping” cioè una funzione / algoritmo di posizionamento**, che dato un blocco della main memory stabilisce dove esso va a posizionarsi nella memoria cache. Deve crearsi una corrispondenza tra gli indirizzi della memoria cache e gli indirizzi della memoria virtuale.

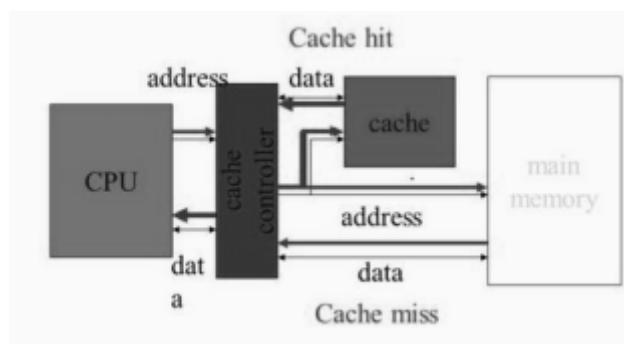
Inoltre poiché la memoria cache è più piccola della main memory, durante l'esecuzione del programma essa sicuramente diventerà **satura** (piena) quindi quando vogliamo aggiungere un nuovo blocco prelevato dalla main memory e non presente in cache si deve fare spazio

cancellando un altro blocco riscrivendolo in memoria. Questa gestione è affidata ad un **algoritmo di sostituzione** che individua quale blocco sostituire quando la cache è satura.

Tutte queste gestioni sono affidate ad un **controller della cache**.

La CPU, 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 (**controller**) che si fa carico di determinare se la parola richiesta è presente o meno nella cache.

Il controller ricerca la parola nella cache; se essa 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)** a seconda dell'istruzione. Se la parola non viene trovata in cache abbiamo un **read miss o un write miss**.



Gestione di un CACHE HIT

Se l'accesso alla cache ha avuto successo e quindi abbiamo una situazione di cache hit allora bisogna distinguere due tipi di situazioni:

Operazione di lettura: la memoria principale non viene coinvolta. Non c'è un coinvolgimento della main memory perchè la CPU richiede un indirizzo, il controller della cache si accorge che il dato a quell'indirizzo è già presente in cache e lo preleva da essa velocemente senza interpellare la main memory.

Operazione di scrittura: si può procedere con 2 strategie diverse: write-through o write-back

1) write-through: il dato da modificare in una certa locazione viene scritto simultaneamente sia nella memoria cache che nella main memory. Si mantiene una coerenza fra quello che c'è scritto nella memoria cache e quello che c'è scritto nella main memory. In realtà non viene riscritto solo un dato, poiché il passaggio fra cache e memoria avviene sempre in blocchi, quindi si aggiorna tutto il blocco in cache in cui è presente il dato da aggiornare e così si aggiorna anche il corrispondente blocco in memoria principale. Questa strategia è molto semplice e conservativa (safe) ma non è sempre efficace perchè se si va ad aggiornare lo stesso dato in qualche istruzione successiva, si deve di nuovo riaggiornare tutto il blocco in cache e in memoria: questa comunicazione fra le due memorie ha una certa latenza e costo e ciò inoltre richiede molti accessi alla memoria principale (molto lenta), aumentando i tempi. Quindi ci sono inutili operazioni di scrittura nella memoria principale, se una parola viene aggiornata più volte durante il periodo in cui risiede nella cache.

2) write-back: in questa strategia si aggiorna soltanto il blocco della memoria cache e ad ogni blocco è associato un **bit di modifica o dirty** che vale 1 se il blocco è stato modificato da almeno una scrittura, altrimenti vale 0 se non ha subito alcuna modifica in scrittura. Il blocco corrispondente in main memory viene aggiornato in seguito, quando il blocco modificato della memoria cache deve essere rimosso per far posto a un nuovo blocco. Quindi quando quel blocco della cache deve essere sostituito, se il suo bit di modifica è pari a 1 effettua il trasferimento di quel blocco nella memoria principale. 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 una parola richiesta in lettura dalla CPU non è presente nella cache si dice che l'accesso in lettura è fallito (**read miss**). Il blocco contenente la parola richiesta deve essere copiato dalla memoria principale nella memoria cache.

Abbiamo due modalità diverse per caricare il blocco in cache:

1- Si prende il blocco contenente la parola richiesta dalla memoria principale, lo si trasferisce in cache e poi lo si legge direttamente dalla memoria cache. Quindi dopo aver caricato l'intero blocco nella memoria cache, la parola richiesta viene inviata alla CPU.

2- **Load-through o early restart:** appena si trova la parola ricercata nel blocco della main memory si invia contemporaneamente la parola richiesta alla cpu e poi si sostituisce il blocco contenente la parola nella memoria cache. Quindi si invia contemporaneamente la parola alla CPU appena viene trovata in main memory. Ciò 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**). Non si trova un dato che devo scrivere in cache.

Due alternative:

1- **se si utilizza un protocollo write-through**, le informazioni vengono scritte direttamente nella memoria principale, poi si trasferisce il blocco in cache.

2- **se si utilizza un protocollo write-back** il blocco contenente la parola indirizzata in main memory viene prima caricato nella cache, poi viene sovrascritto sempre nella cache con le nuove informazioni.

Prestazioni: hit rate, miss rate, amat

Una metrica per l'analisi delle prestazioni di una memoria sono i **tassi (o percentuali) di hit/miss (hit rate e miss rate)** e il **tempo medio di accesso in memoria (amat)**.

L'**hit rate** è il rapporto tra il numero hit (cioè il numero di volte in cui data una richiesta in memoria ho trovato l'istruzione o il dato nella cache) e il numero complessivo di accessi in memoria. E' uguale anche a **1-MISS RATE**

Il **miss rate** è il rapporto tra il numero di miss e il numero complessivo di accessi in memoria. E' uguale anche a **1-HIT RATE**

L'**AMAT** è l'**Average Memory Access Time** cioè il tempo medio di accesso in memoria, ossia il tempo medio di attesa da parte del processore per completare un'istruzione di lettura o scrittura dalla memoria. Influiscono su di esso hit rate e miss rate. Ipotizzando che vi sia anche la memoria virtuale e che il dato possa anche non trovarsi in main memory, l'amat si calcola come:

$$\text{AMAT} = t_{\text{cache}} + MR_{\text{cache}}[t_{MM} + MR_{MM}(t_{VM})]$$

Nel caso in cui non vi fosse memoria virtuale abbiamo:

$$\text{AMAT} = t_{\text{cache}} + MR_{\text{cache}} * t_{MM}$$

t_cache: tempo di accesso alla cache

MR_cache: miss rate della cache

t_MM: tempo di accesso alla main memory

MR_MM: miss rate della main memory

t_VM: tempo di accesso alla memoria virtuale

Terminologia cache

Capacity (C): è il numero di parole che la cache può contenere (numero di byte)

Blocco: è un gruppo di parole adiacenti della cache

Block size (b): è il numero di parole contenute in un singolo blocco di cache, ossia il numero di bytes che definisce la grandezza del blocco della cache.

Numero di blocchi della cache (B): $B=C/b$ è il rapporto fra la grandezza della cache e la dimensione del blocco di cache

Set della cache

Ogni cache è organizzata in S insiemi o **set**, ciascuno dei quali contiene uno o più blocchi N di parole. Quindi un **set** è un insieme della cache in cui sono contenuti uno o più blocchi N di parole della memoria.

Il **mapping** è una relazione tra l'indirizzo di un dato in memoria principale e la locazione di tale dato in cache. Ogni indirizzo di memoria è mappato in un set della cache. Questo serve al controller della cache per verificare dove andare a cercare il dato in cache.

Il **numero di blocchi in un set** è definito N (*degree of associativity*).

Il **numero di set in cache S=B/N** è dato dal rapporto tra il numero di blocchi totali della cache e il numero di blocchi in un set.

Tipologie di memorie cache

Abbiamo diverse categorie di memoria cache che sono organizzate in base al numero di blocchi che un set contiene:

- **Direct mapped**: **Ogni set è formato da 1 blocco di parole** e dunque un qualsiasi indirizzo di main memory è mappato in un solo blocco della cache. Il numero di set è uguale al numero di blocchi della cache. Il numero di righe è 1 poiché ogni set ha 1 solo blocco.

- **N-way set associative**: **Un set è formato da N blocchi di parole**. L'indirizzo di main memory è mappato in un set della cache, ma il dato corrispondente a tale indirizzo può finire in uno qualsiasi degli N blocchi di quel set. Il numero di vie è compreso fra $1 < N < B$. Il numero di set è da B/N ossia numero di blocchi totali della cache fratto il numero di blocchi in un set

- **Fully associative**: **C'è solo 1 set che contiene tutti gli B blocchi di parole della cache**, quindi il dato può andare in uno qualsiasi dei blocchi dell'unico set presente

Direct mapped cache

Una cache a mappatura diretta ha un solo blocco in ogni set, quindi è organizzata in tanti set quanti sono il numero di blocchi nella cache. Ogni indirizzo della memoria principale può essere mappato in un unico set. Immaginando che la main memory sia suddivisa anch'essa in blocchi di parole come la cache, possiamo dire che un indirizzo nel blocco 0 della main memory viene mappato nel set 0 della cache, un indirizzo del blocco 1 della main memory viene mappato nel set 1 della cache e così via fino all'ultimo set della cache.

Nella mappatura l'**indirizzo della main memory** viene logicamente suddiviso in 3 tre parti:

1- il **byte offset**, cioè i due bit meno significativi dell'indirizzo che sono sempre 00 perché si procede di parola in parola. Servono ad indicare un byte dentro la parola (1° byte, 2° byte ecc.).

2- i **bit di set** che sono usati per indicare in quale set della cache viene mappato l'indirizzo: ad esempio se la cache è formata da 8 set terremo conto dei $\log_2 8$ ($\log_2 S$) = 3 bit successivi e quindi si mappa l'indirizzo nel set in cache corrispondente ai bit considerati (000, 001, 010..111). Ogni indirizzo di memoria viene mappato in un solo set della cache. Gli indirizzi sono mappati in modo ciclico o circolare, poiché molti di essi sono mappati nel medesimo set.

3- i bit restanti più significativi sono il **tag (etichetta)** e indicano quale dei tanti possibili indirizzi è effettivamente presente in quel particolare set.

La cache è realizzata con una SRAM , nella quale abbiamo 8 linee che corrispondono in questo caso a 8 set. Ogni set costituisce una riga contenente **32 bit di dato, 27 bit di tag e 1 bit di validità**.

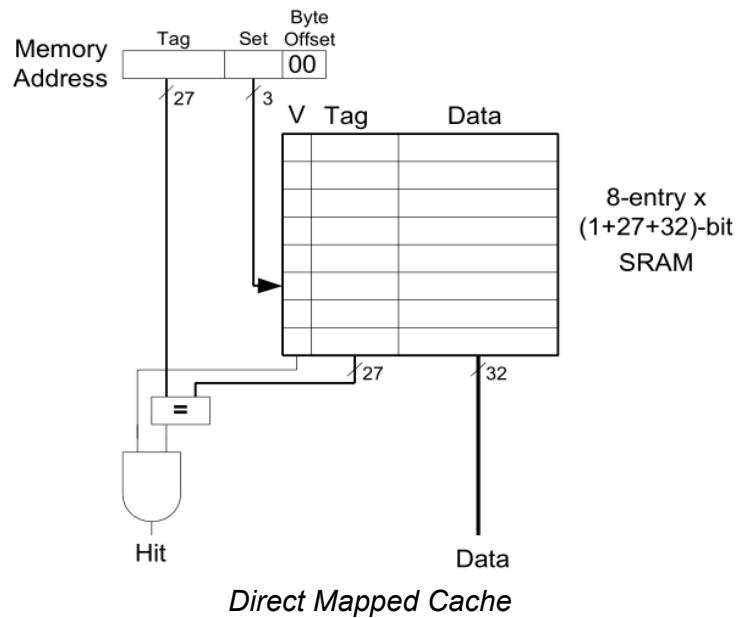
Il **bit di validità** indica se il set contiene dati significativi cioè se il dato è affidabile oppure no. Se il bit di validità è 0 il contenuto del set non è significativo, se 1 allora è valido.

I 27 bit di tag vengono confrontati con i tag dell'indirizzo di memoria poiché il mapping non è 1:1 e dato che la cache ha una capacità minore della main memory e quindi in un particolare set possono essere memorizzati più indirizzi diversi.

Dati i bit di set dell'indirizzo, il controller verifica la linea corrispondente al set in cache. Poi confronta il tag dell'indirizzo che la cpu ha richiesto col tag presente nella linea di cache. Il confronto avviene con un comparatore : se i due tag sono uguali vuol dire che l'indirizzo presente nella linea di cache è proprio quello richiesto dalla CPU.

Il tag poi va in AND col bit di validità; se il bit di validità è 1 e il tag è 1 (cioè il tag di cache corrisponde al tag di indirizzo) allora si ha un **HIT. Cioè un 1 che indica che il dato è stato trovato in cache**. A questo punto si può leggere o scrivere il dato attraverso il bus data. In caso di miss il controller deve prelevare il dato dalla memoria principale.

Svantaggio: miss di conflitto.



Miss obbligatori e di conflitto

Quando il processore all'inizio richiede dei dati avremo una situazione in cui abbiamo dei **miss obbligatori** dovuti al fatto che al primo accesso in cache questi dati non saranno presenti e bisogna prima caricarli dalla main memory ; il bit di validità corrispondente ai vari set sarà 0. Quindi poi i dati vengono prelevati dalla main memory e copiati in cache, settando il bit di validità a 1. Agli accessi successivi avremo degli hit in quanto i dati sono presenti nei vari set della cache con i vari bit di validità a 1. Dipendono dal fatto che una certa locazione di memoria sia essa contenente un istruzione o un dato viene richiesta per la prima volta (in lettura o scrittura), come se fosse un inizializzazione.

Quando due indirizzi richiesti di recente dal processore si mappano nel medesimo set della cache si verifica un **miss di conflitto**. Dunque l'indirizzo che deve essere mappato più recente espelle quello precedentemente mappato nel set. Il dato viene sovrascritto una volta che è stato prelevato dalla main memory. Anche il tag del set è sovrascritto.

Dipendono dal fatto che essendo la memoria cache come capacità inferiore alla memoria principale, il mapping fra gli indirizzi della memoria principale e i set della memoria cache non è 1:1 (funzione iniettiva) quindi più blocchi della main memory possono memorizzarsi nello stesso set della cache.

Siccome le cache a mappatura diretta hanno un solo blocco per ogni set, due indirizzi che si mappano nel medesimo set causano sempre un miss di conflitto. miss rate pari al 100% e tempo di accesso in memoria alto.

N-Way Associative cache

Una **tecnica efficace per ridurre i miss di conflitto** è quella di passare da una direct mapped cache ad una **n-way associative cache**, cioè una cache parzialmente associativa a N vie. Infatti lo scopo di una N-Way associative cache è proprio quello di ridurre i miss di conflitto causati dall'uso di una direct mapped cache.

In questa tipologia di cache per ogni set possiamo mappare più blocchi.

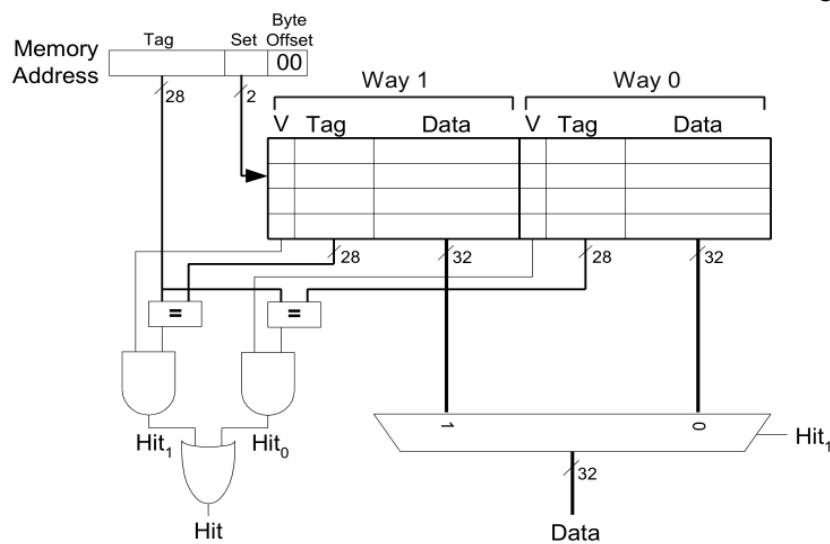
Quindi ci sono N blocchi in ciascun set: l'indirizzo di memoria viene mappato in uno specifico set, ma può essere copiato in uno qualsiasi degli N blocchi del set. Gli N blocchi presenti in ciascun set rappresentano il grado di associatività della cache. Nell'esempio considerato abbiamo 2 blocchi, quindi 2 vie.

In ciascun set sono presenti quindi 2 parole, ognuna con il suo tag e i dati corrispondenti ed il bit di validità. Siccome ogni set contiene 2 parole e la cache è formata da 8 parole, vuol dire che avremo 4 set in totale, quindi ci servono solo **2 bit identificativi di set**.

I primi due bit dell'indirizzo sono sempre il **byte offset** per discriminare il byte all'interno della parola. Il **tag dell'indirizzo di memoria** viene confrontato contemporaneamente con entrambi i tag (delle 2 way presenti) del set corrispondente. **Quindi per ogni indirizzo di memoria si estrae il tag e lo si confronta con gli N tag delle way associate al set corrispondente, con N che dipende dal grado di associatività della cache.**

Il confronto dei tag avviene sempre attraverso dei comparatori, dopodiché si mette il risultato in AND col bit di validità. Abbiamo 2 possibili situazioni di hit *Hit₁* (way 1) e *Hit₀* (way 0) che corrispondono al fatto che la prima via può combaciare col tag dell'indirizzo di memoria o che l'altra via combacia col tag. Abbiamo un hit se il tag dell'indirizzo richiesto è presente in almeno una delle vie corrispondenti al set, quindi usiamo una porta OR per stabilire quale dei due Hit sia andato a buon fine. Se si verifica un hit in una delle due vie, dobbiamo selezionare i dati della via in cui si è verificato l'hit: per farlo usiamo un multiplexer 2:1 che seleziona di volta in volta i dati o della way1 o della way0. Esso è pilotato dal valore di *Hit₁*: se esso è pari a 1 allora il mux seleziona il dato corrispondente alla way1, se esso è pari a 0 allora il mux seleziona il dato corrispondente alla way0, nel caso in cui il valore finale *Hit* sia pari a 1.

Questa tipologia di cache ha un miss rate inferiore alla cache direct mapped, tuttavia è generalmente più lenta e più costosa da realizzare a causa del mux di uscita e dei comparatori aggiuntivi. Sorge inoltre il problema di quale via "sostituire" quando entrambe sono piene. L'associatività riduce i miss di conflitto. Abbiamo solo i miss obbligatori iniziali

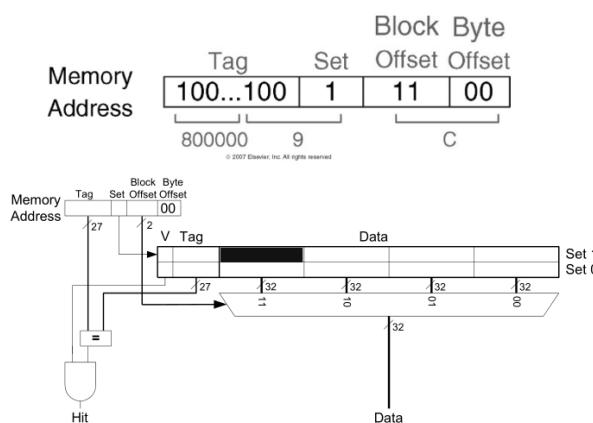


Sfruttare la località spaziale: dimensione del block size

Nell'ipotesi che **aumentiamo il block size**, cioè il numero di parole presenti in ogni blocco, sfruttiamo al meglio il principio di **località spaziale e riduciamo i miss obbligatori** poiché carichiamo indirizzi vicini nel blocco senza doverli andare a riprendere uno per uno in main memory. Sfruttiamo la località spaziale in quanto è probabile che nel futuro ci servirà uno degli indirizzi vicini a quello richiesto. Quindi una cache utilizza blocchi di dimensioni maggiori per memorizzare più parole di memoria consecutive col vantaggio che in caso di miss vengono copiate in cache la parola non trovata e anche le parole adiacenti nel blocco di memoria principale. Gli accessi successivi hanno dunque maggiore probabilità di dare luogo a hit grazie alla località spaziale.

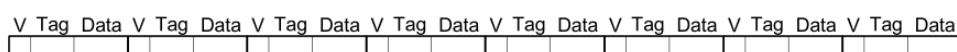
Nell'esempio considerato abbiamo una cache di capacità 8 parole di tipologia direct mapped (ogni blocco corrisponde ad un set). Se un blocco ha un block size di 4 parole, significa che avremo 2 set, con 1 bit di set. Ogni set conterrà quindi 4 parole, che corrispondono ad 1 blocco. Avremo dunque due bit prima del bit di set, detti **block offset** che ci dicono quale parola all'interno del set scegliere (1°, 2°, 3°...). Si confronta il tag e poi si sceglie la parola corrispondente attraverso un mux che seleziona il contenuto associato alla parola giusta, dato il block offset.

Una dimensione del block size maggiore significa a parità di capacità della cache che questa ha meno blocchi, aumentano quindi i miss di conflitto. Inoltre serve più tempo in caso di miss per prelevare dalla main memory tutte le parole del blocco e trasferirle in cache.



Fully associative cache

Questa tipologia di cache ha una grande versatilità in quanto è costituita da 1 unico set, nel quale sono presenti tante vie corrispondenti al numero B di blocchi. Un indirizzo di memoria può essere mappato in una qualsiasi delle vie associate all'unico set. Essa consente la massima flessibilità nella scelta della locazione nella cache in cui posizionare un blocco di memoria quindi lo spazio può essere utilizzato in modo efficiente comportando una riduzione dei miss di conflitto. Lo svantaggio principale è che il costo della ricerca di un blocco in cache è notevole: alla richiesta di un dato dobbiamo confrontare tanti tag quanti ne sono i blocchi presenti nell'unico set, dal momento che il dato potrebbe trovarsi in un blocco qualsiasi. Un mux con tanti ingressi e un'uscita seleziona il dato corretto in caso di hit. Quindi questa cache richiede hardware aggiuntivo per i confronti dei tag ed un elevato numero di comparatori.



Bit di validità: dettaglio

Siccome deve esserci una certa coerenza fra il contenuto di un blocco in cache e il contenuto di un blocco della main memory, viene introdotto un bit di validità per mantenere questa coerenza. Esso è un bit di controllo, ed è necessario per ogni blocco. Indica se il blocco contiene o meno dati validi.

Non deve però essere confuso con il **bit di modifica**, che indica se il blocco è stato modificato o meno durante il periodo in cui è stato nella cache, esso 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 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 (swap) dal disco (virtual memory), 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 della main memory che deve essere aggiornato mediante swap dalla virtual memory, il suo bit di validità viene posto a “0”; in tal modo, si garantisce che nella cache non ci siano dati obsoleti e si mantiene una certa coerenza tra cache e main memory. Il bit di validità a “0” candida immediatamente il blocco della cache all’interno di un set ad essere sovrascritto e quindi sostituito.

Il Direct Memory Access - DMA

I trasferimenti dal disco(memoria di massa) alla memoria principale vengono effettuati mediante una logica di controllo non gestita dalla CPU, bensì da un meccanismo detto di DMA (Direct Memory Access). La CPU non si accorge dello scambio di pagine fra la memoria virtuale e la main memory. Durante il trasferimento la CPU non interviene, perché è la logica di controllo che gestisce le linee indirizzo e le linee dati, fornendo i segnali di controllo necessari. 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.

Le **pagine** quindi sono segmenti di riferimenti di memoria contigui, come i blocchi della cache , ma molto più grandi (ordine di Kb), in quanto la virtual memory e la main memory hanno capacità maggiori. Normalmente, le pagine di programma e quelle di dati vanno nella memoria centrale senza coinvolgere la cache.

Svuotamento (flush) della cache. Problema della coerenza fra cache, mm, vm

In una gerarchia di memoria a 3 livelli (cache, main memory, virtual memory) si deve sempre garantire che vi sia una certa coerenza fra i livelli. Questa gerarchia di livelli fa sì che la cache comunichi con la main memory (*end to end*) e la memoria principale comunichi con la virtual memory. La cache non comunica direttamente con la virtual memory. La memoria principale si trova in mezzo a richieste che vengono contemporaneamente dalla cache e dalla virtual memory e c’è bisogno di un meccanismo che mantenga la coerenza in modo tale che la memoria principale soddisfi le richieste sia della cache che della virtual memory. In particolare quando abbiamo un write hit nella cache, se la politica adottata è quella di write-back significa che si sovrascrive il blocco in cache senza sovrascrivere anche la

memoria principale. Quindi è come se si perdesse la “coerenza” fra le due memorie: si deve far sì che questa perdita momentanea di coerenza non infici sull’intero sistema. Quindi si pone il bit di modifica del blocco in cache pari a 1 e quando deve essere sostituito per far spazio ad un nuovo blocco lo dobbiamo sovrascrivere in memoria principale nel suo indirizzo corrispondente , che otteniamo guardando il tag del blocco. Un altro caso lo abbiamo quando sovrascriviamo una parola all’interno del blocco in cache ma non sovrascriviamo il blocco in main memory e questo blocco fa parte di una pagina più grande che dobbiamo swappare con la virtual memory. Quando dobbiamo swappare la pagina, i dati nella pagina devono essere aggiornati : ciò significa che dobbiamo prendere tutti i blocchi che fanno parte della pagina che stanno in cache che hanno il bit di modifica pari a 1 e dobbiamo fare uno svuotamento, cioè un **flush** del blocco della cache che deve essere spostato in main memory . Poichè la pagina viene poi swappata con la virtual memory per far contenere qualcosa di nuovo, tutti i bit di validità del blocco in cache poi vengono settati a 0 poichè i dati non sono più coerenti con i nuovi presenti nella pagina aggiornata in main memory.

Il sistema operativo svolge tutte queste operazioni necessarie in modo molto efficiente. 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 (di un blocco di parole in cache)

In una **cache ad indirizzamento diretto**, la posizione di ogni blocco è predefinita ed abbiamo che ogni set corrisponde ad un blocco, quindi **non esiste alcuna strategia di sostituzione** visto che la sostituzione avviene per sovrascrizione dell’unico blocco presente nel set col nuovo blocco aggiornato.

In una **cache associativa a N vie** quando dobbiamo trascrivere un nuovo blocco in un set dobbiamo scegliere quale blocco sostituire quando il set è pieno. Nel caso ci fossero dei blocchi in un set con bit di validità pari a 0 si incomincia a sovrascrivere quelli in quanto contengono dati non più significativi. Se il set è pieno e tutti i bit di validità valgono 1 e si verifica la situazione in cui dobbiamo sostituire in memoria cache un blocco con un nuovo blocco richiesto dalla cpu e prelevato dalla main memory , dobbiamo scegliere uno dei blocchi in cache con bit di validità pari a 1 da sostituire. In questo caso si sfrutta il principio della **località temporale** per cui dati a cui abbiamo fatto accesso di recente saranno quelli con più probabilità di essere riusati nel prossimo futuro.

Riassumendo, 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, sovrascrivere. In generale, l’obiettivo è quello di mantenere nella cache quei blocchi che hanno una maggior possibilità di essere nuovamente utilizzati nel prossimo futuro. 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 elevata probabilità di essere utilizzati nuovamente entro breve tempo.

Algoritmo LRU

Una strategia di sostituzione è quella definita dall **algoritmo di sostituzione LRU (Least Recently Used)** in cui viene sovrascritto il blocco a cui non si accede da più tempo nel set. Tale blocco prende il nome di blocco utilizzato meno di recente LRU. Per utilizzare l’algoritmo LRU, il controllore della cache deve mantenere traccia di tutti gli accessi ai blocchi mentre l’elaborazione prosegue attraverso un **contatore degli accessi al blocco**. Si stabilisce una “graduatoria” dei blocchi nel set per stabilire quello meno usato di recente,

stabilendo una classifica di posizionamento dei blocchi. Ad esempio se abbiamo 4 blocchi, con due bit indicheremo con 00 il blocco usato più recente in prima posizione, 01 il secondo, 10 il terzo e con 11 il blocco usato meno di recente.

Gestione del contatore degli accessi nell'algoritmo LRU

Quando si ha un successo nell'accesso al blocco di un set (**hit**):

- il contatore di quel blocco viene posto a 0 poiché diventa il blocco usato più di recente LRU. I contatori con valori inferiori a quelli del blocco a cui si accede vengono incrementati di uno, mentre tutti gli altri con valore superiore rimangono invariati.

Quando, invece, l'accesso fallisce (**miss**) si aprono due possibilità:

- il set non è pieno, quindi abbiamo un bit di validità pari a 0 : viene caricato il nuovo blocco dalla memoria principale viene e il contatore degli accessi viene posto a 0 , il bit di validità passa a 1 e il valore di tutti gli altri contatori incrementa di 1.
- tutto il set è pieno, si rimuove il blocco meno usato di recente, il cui contatore ha valore 3, e si pone il nuovo blocco al suo posto, mettendo il contatore a 0. Gli altri tre contatori del set vengono incrementati di 1.

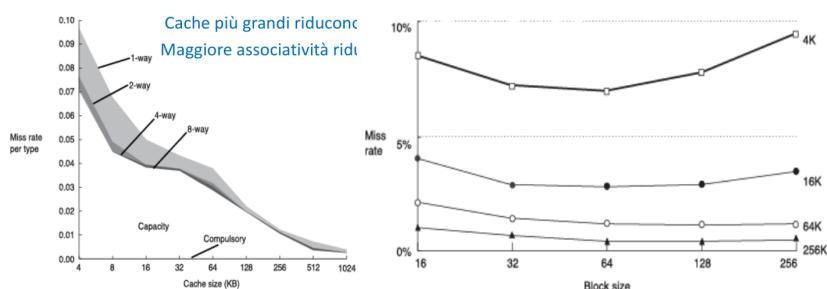
Questa strategia può generare molti miss ad esempio nel caso di accessi sequenziali a un vettore di elementi che è leggermente troppo grande per poter stare tutto nella cache e dunque le prestazioni sono molto scadenti con molti miss. **Quindi ci sono strategie diverse che utilizzano sempre l'algoritmo LRU ma ogni 5-6 accessi sostituiscono un blocco in modo casuale (random).**

Tipi di miss. Analisi del miss rate.

- Compulsory:** primo accesso ad un dato ossia i **miss obbligatori**
- Capacity:** la cache è troppo piccola per contenere tutti i dati di interesse **miss di capacità**
- Conflict:** due dati sono indirizzati nella stessa locazione nella cache ossia i **miss di conflitto**

Dai grafici si mostra che avendo fissata una dimensione del blocco in cache , all'aumentare della capacità della cache , fissato il numero di vie , il numero di set aumenta dato che aumentiamo la capacità. Quindi si riducono i miss di capacità e avremo meno miss in generale. Inoltre fissata una certa dimensione, se aumentiamo il numero di vie si diminuiscono i miss di conflitto, però aumentare a dismisura il numero di vie ad un certo punto significa complicare l'hardware e non ottenere un reale beneficio .

Mentre fissato il cache size e un numero n di vie , notiamo che all'aumentare del block size il miss rate prima migliora col numero di miss obbligatori dato che sfruttiamo la località spaziale con blocchi più grandi, ma poi ad un certo punto peggiora all'aumentare del block size. Questo perchè se aumentiamo il block size in modo molto grande arriveremo in un punto in cui diminuisce ovviamente il numero di set e quindi avremo più miss di conflitto , perchè abbiamo pochi set.



Frazionamento della memoria in moduli

E utile che i trasferimenti da/verso le unità di memoria possano essere effettuati alla stessa frequenza dell'unità più veloce, cioè la cache. 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 (spaziale) nella gestione della memoria principale consiste nell'impiego di un'organizzazione in più moduli della memoria. 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 di trasmissione delle parole da o verso il sistema di memoria principale ed inoltre possiamo trasferire più parole in parallelo, impiegando meno tempo.

Abbiamo un metodo interlacciato e un metodo non interlacciato della memoria che corrispondono a due metodi di distribuzione degli indirizzi.

1- Non interlacciato.

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**.

I k bit attraverso un decoder vanno ad identificare un certo modulo, gli m bit meno significativi vanno a finire in un **address register** ed identificano l'offset all'interno del modulo, cioè quale parola all'interno del modulo (che contiene tutti indirizzi di memoria contigui) ci stiamo andando a riferire. La parola viene presa in considerazione e il dato corrispondente viene messo in un **data register** e poi verrà inviato dal bus fino alla CPU o alla memoria virtuale. **I blocchi di cache saranno contenuti tutti all'interno di un modulo.** Quindi quando avremo un trasferimento da cache alla main memory di un blocco questo andrà ad essere copiato nel modulo corrispondente della main memory, in base ai k bit dell'indirizzo che specificano il numero del modulo. Intanto attraverso il DMA negli altri moduli che sono liberi può avvenire il trasferimento di una pagina con la virtual memory.

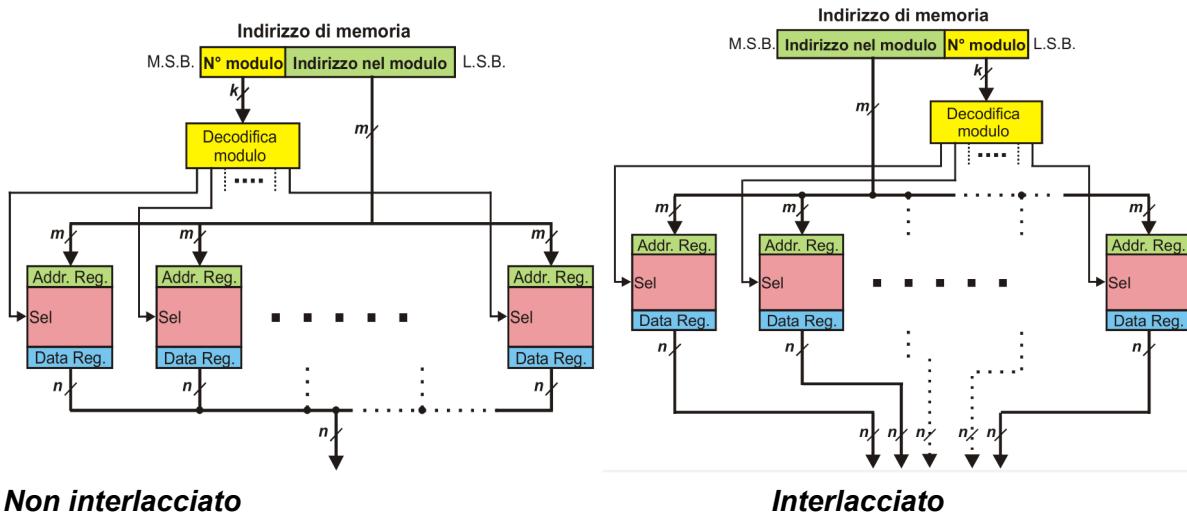
Quindi può avvenire in contemporanea un trasferimento di una pagina con la memoria virtuale in un modulo ed un trasferimento di un blocco dalla cache ad un altro modulo. Questa frazionamento in moduli allora consente il trasferimento in contemporanea di dati dalla cache e dalla virtual memory verso la main memory e viceversa purché si riferiscano a moduli differenti.

Questo tipo di organizzazione **non è quello più efficiente** perchè i trasferimenti di blocchi della memoria cache che ha una ridotta capacità e quindi si satura facilmente, con la main memory sono più frequenti rispetto ai trasferimenti di pagina tra la memoria virtuale e la main memory che sono più lenti e meno frequenti. **Per questo si passa ad un tipo di indirizzamento interlacciato**

2- Interlacciato

Funziona in maniera duale: i **bit meno significativi (k)** **selezionano un modulo**, mentre i **bit più significativi (m)** **identificano una parola nel modulo**.

Questo vuol dire che indirizzi contigui che stanno nello stesso blocco corrispondono a moduli differenti, in cui hanno le parole dello stesso blocco sono posizionate allo stesso offset, ma ovviamente in blocchi differenti. Il trasferimento di un blocco avviene andando a leggere i differenti moduli e può avvenire in parallelo. La lettura dei blocchi dalla cache alla main memory avviene più velocemente, però siccome si impegnano tutti i moduli se c'è un trasferimento alla memoria virtuale finché non si libera un modulo questo trasferimento non può avvenire.



Cache multilivello

Oggiorno la memoria cache è costituita da una gerarchia di 2 o 3 livelli: **L1, L2, L3**. Questa tipologia di cache è detta multilivello. I livelli successivi sono cache un po' più grandi ma più lente ovviamente della cache di primo livello. In questi casi le politiche di gestione tra cache e main memory si ripetono anche fra i livelli di cache.

Vi possono essere due tipologie di cache multilivello: **inclusiva** o **esclusiva**

1- Inclusiva: la cache L2 contiene tutti i dati presenti in L1 , in quanto è più grande in termini di capacità di L1 e può contenerla. Quindi L1 è un sottoinsieme più veloce di L2. (similmente L2 è contenuta in L3). Il tipo di politica di solito è **write-through**, quindi si mantiene subito una coerenza fra i vari livelli di cache. Quando abbiamo un **miss** in L1, il controller vede se il blocco ricercato è contenuto in L2: se lo trova il blocco viene caricato in L1 e poi nel caso avviene una sostituzione di un blocco in L1 con un algoritmo di sostituzione che fa sì che il blocco che viene sovrascritto venga caricato in L2 (**copy back**). Se abbiamo un miss in L1 e in L2 il blocco viene prelevato dalla main memory e viene copiato sia in L1 che in L2 per mantenere la coerenza fra i livelli.

2-Esclusiva: la cache L2 contiene solo i dati di copy back da L1 (victim cache)

quindi non contiene gli stessi dati ma solo i dati che vengono passati da L1 quando c'è una sostituzione di un blocco di L1 (blocco vittima).

La memoria complessiva della cache è data dalla somma delle capacità dei vari livelli. Se abbiamo un miss in L1 e hit in L2 il blocco trovato in L2 deve passare in L1 : quindi avviene uno scambio fra i blocchi cioè quello meno recente in L1 viene sostituito con quello più recente appena trovato (hit) in L2, ovviamente rispettando i bit di set.

Se abbiamo un miss in L1 e L2: il blocco viene letto dalla main memory ed è memorizzato direttamente in L1 e verrà trasferito in L2 solo quando esso diventa un blocco vittima cioè quando L1 diventa satura e dunque il blocco deve essere sostituito.

Cache condivise

Nelle architetture con più core i livelli esterni della cache L2 o L3 possono essere condivisi fra più core di uno stesso processore. Cio' consente un uso più efficiente del livello condiviso quando per esempio un core è inattivo e allora l'altro può utilizzare per sé il livello condiviso oppure entrambi sfruttano la stessa cache L2.

La memoria virtuale

Al gradino più basso della gerarchia delle memorie abbiamo il **disco rigido** in tecnologia magnetica o a stato solido. Esso è **grande ed economico ma terribilmente lento**.

I programmi e i software risiedono in memoria di massa; è il **sistema operativo che si fa carico del mapping degli indirizzi di memoria virtuale del programma che risiede in memoria di massa in indirizzi di memoria principale**. Inoltre il sistema operativo carica il programma dalla memoria di massa in main memory in una locazione **casuale** ed in modo dinamico. La tecnica della memoria virtuale prevede quindi che il sistema operativo assegni una parte degli spazi liberi nella memoria centrale ai **segmenti** di un programma installato su disco. Infatti per un problema di efficienza il sistema operativo non è detto che carichi il programma in un insieme contiguo di indirizzi, ma in realtà lo può caricare in diversi segmenti dato che la memoria principale è suddivisa in **pagine** ossia dei blocchi di indirizzi continui (decine di kb) . È probabile che il programma segmentato venga caricato in più pagine nella main memory non necessariamente contigue fra loro. La suddivisione in pagine consente una gestione più efficace della memoria principale.

Swapping

Anche la **main memory si satura** : un programma molto pesante che non ha sufficiente spazio in main memory per continuare ad essere eseguito deve salvare parte di esso di nuovo nella memoria virtuale: questa viene detta un'**operazione di swapping**. La parte del programma caricata nella virtual memory viene ripresa successivamente.

Lo swapping dipende dal fatto che la main memory ha una capacità ovviamente molto minore della virtual memory. Le operazioni di swapping essendo la memoria di massa più lenta della main memory sono delle operazioni "pesanti" dal punto di vista del tempo perso. Bisogna cercare di rendere questa operazione di swapping meno onerosa possibile.

La località spaziale e temporale alleggeriscono le operazioni di swapping dato che è molto probabile che vengano richiesti indirizzi contenuti in una stessa pagina di memoria, senza necessariamente dover effettuare operazioni di swap continuamente.

Inoltre in un'operazione di swap è possibile che vengano scambiate molte pagine.

La lentezza delle operazioni di swapping dipende anche dal fatto che le memorie virtuali sono costruite con una tecnologia più lenta di quella delle cache e delle ram. Infatti un'unità a dischi rigidi magnetici contiene uno o più dischi rigidi ciascuno dotato di una testina di lettura/scrittura. La testina si sposta sulla corretta locazione e sfrutta l'elettromagnetismo per leggere o scrivere dati sul disco in rotazione sotto di lei. La testina impiega vari millisecondi per raggiungere la corretta locazione sul disco: un tempo milioni di volte più lento di quello del processore. I dischi rigidi magnetici sono progressivamente stati sostituiti da dischi a stato solido perché in questi ultimi la velocità di lettura ha ordini di grandezza maggiori.

Quindi la bandwidth è minore di una cache o una ram poichè dipende dalla velocità in cui gira il supporto magnetico ed essendo meccanico non raggiunge mai le velocità di sistemi elettronici come le sram o le dram. Inoltre c'è un overhead nel tempo di lettura e scrittura dovuto alla latenza per trovare il punto iniziale da cui andare a scrivere. Questo richiede il movimento meccanico della testina, operazione lenta. La latenza maggiore ad oggi si ha quando bisogna caricare inizialmente un programma in main memory. Per questo oggi usiamo una memoria a stato solido che ha un tempo di accesso più veloce di un disco rigido.

L'obiettivo di aggiungere un disco rigido alla gerarchia di memoria è quello di dare l'illusione di uno spazio di memoria molto grande pur mantenendo la velocità delle memorie più rapide per la maggior parte degli accessi.

Gestione della memoria virtuale

I programmi possono accedere a un qualsiasi dato della memoria virtuale, quindi devono utilizzare **indirizzi virtuali** che specificano le locazioni dei dati nella memoria virtuale. Quindi gli indirizzi virtuali appartengono alla memoria di massa. Poi come sappiamo abbiamo la main memory che però è una memoria con una capacità minore della memoria di massa: essa contiene un **sottoinsieme della parte di memoria virtuale più recentemente utilizzata**. In questo modo la main memory (memoria fisica) agisce da cache della memoria virtuale. Anche in questo caso avviene quindi un'**operazione di mapping** ossia gli **indirizzi virtuali vengono tradotti in indirizzi fisici della main memory**. In questo caso il mapping avviene seguendo una tabella degli indirizzi chiamata **lookup table** e non in modo automatico perchè lo swapping tra main memory e virtual memory è più raro ed è quindi gestito dal sistema operativo che gestisce la tabella degli indirizzi. Quindi c'è una politica di mapping fully associative dove si vede una tabella per effettuare il mapping da indirizzi virtuali a fisici. Infatti gli indirizzi virtuali corrispondono a varie pagine nell'hard disk ed una parte di essi, che deve essere caricata in main memory, viene tradotta in indirizzi fisici. La memoria virtuale è suddivisa in pagine virtuali, così come la memoria fisica è suddivisa in pagine fisiche della stessa dimensione. Per evitare mancanze di pagina dovute ai conflitti ogni pagina virtuale può essere quindi mappata in una qualsiasi pagina fisica, in questo modo la memoria fisica si comporta da cache fully associative per la memoria virtuale.

Tabella delle pagine - lookup table e il TLB (Translation Lookaside Buffer)

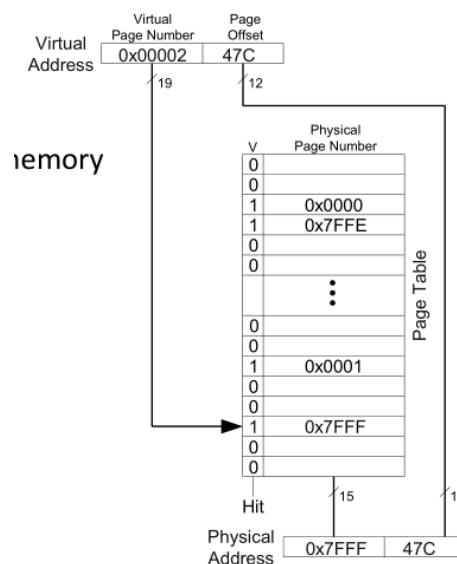
Si usa la tabella delle pagine per effettuare la traduzione dell'indirizzo virtuale in fisico. Essa ha un entry per ogni pagina virtuale che indica in quale pagina fisica tale pagina virtuale si trova oppure che è presente solo su disco. Ogni istruzione di lettura o scrittura in memoria richiede quindi un accesso alla tabella delle pagine seguito da un accesso alla memoria fisica: l'accesso alla tabella delle pagine traduce l'indirizzo virtuale usato dal programma in indirizzo fisico, che viene poi utilizzato per l'effettivo accesso di lettura o scrittura del dato. Ogni entry della pagina virtuale contiene quindi un numero di pagina fisica e un bit di validità: se esso vale 1, la pagina virtuale viene mappata nella pagina fisica specificata nell'elemento, altrimenti la pagina virtuale va caricata dal disco e poi si aggiorna la tabella.

La tabella delle pagine è normalmente così grande da essere memorizzata in memoria fisica.

Le operazioni di load/store richiedono 2 accessi alla main memory:

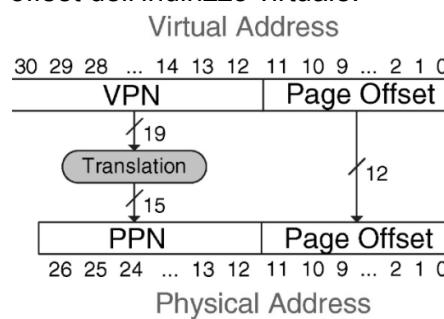
- Il primo per la traduzione (lettura della page table)
- Il secondo è l'accesso vero e proprio al dato dopo la traduzione

Questo dimezzerebbe le performance della main memory e per evitare ciò si usa il **Translation Lookaside Buffer TLB** che tiene in cache poche entry delle migliaia della tabella delle pagine che vengono utilizzate più di frequente. Infatti il processore tiene traccia di un certo numero di entry della tabella delle pagine nel TLB e guarda in esso per trovare la traduzione dell'indirizzo prima di dover accedere alla memoria principale per leggerla dalla tabella delle pagine. Nei programmi reali la stragrande maggioranza degli accessi nel TLB da luogo a hit (99%) evitando i lenti accessi in memoria fisica per leggerla la tabella delle pagine. Il TLB è organizzato come una cache fully associative. 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.



Come avviene la traduzione dell'indirizzo virtuale in fisico

I 12 bit meno significativi dell'indirizzo virtuale da 31 bit indicano il page offset cioè all'interno della pagina a quale locazione ci stiamo riferendo e non richiedono traduzione.
I successivi 19 bit più significativi indicano il numero di pagina virtuale e vengono tradotti nei 15 del numero di pagina fisica. Per prendere la locazione corrispondente nella pagina fisica mappata viene preso il page offset dell'indirizzo virtuale.



La MMU - Memory Management Unit

Il compito di tradurre l'indirizzo virtuale 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. Essa controlla la lookup table e fa il mapping dall'indirizzo virtuale a fisico.