

Calcolatori elettronici

Nicolò Bianchi

Indice

1 Definizione e tipi di calcolatori	5
1.1 Evoluzione dei calcolatori	5
1.2 Hardware e software	5
1.3 Come nasce un programma	5
1.4 Modelli di esecuzione e linguaggi di programmazione	5
1.5 Architetture dei processori	6
1.5.1 Architettura di Von Neumann	6
1.5.2 Architettura Harvard	6
1.5.3 Architettura moderna	6
2 Rappresentazione dei dati e codici	7
2.1 Sistemi numerici posizionali	7
2.2 Conversione di base	7
2.2.1 Intervalli rappresentabili con la scrittura posizionale	7
2.3 Numeri negativi in base 2	8
2.3.1 Rappresentazione in modulo e segno	8
2.3.2 Complemento a uno	8
2.3.3 Complemento a due	9
2.3.4 Rappresentazione in eccesso	10
2.4 Numeri in virgola mobile	11
2.4.1 Rappresentazione di base	11
2.4.2 Rappresentazione (32 bit)	11
2.4.3 IEEE 754: tipi di numero	13
2.4.4 Massimo e minimo numero positivo rappresentabile	13
2.4.5 Errori di approssimazione	14
2.5 Codici ridondanti e irridondanti	15
2.5.1 Distanza di Hamming	15
2.5.2 Codici rivelatori di errori	15
2.5.3 Codice ASCII	16
2.5.4 Codice Binary Coded Decimal (BCD)	16
2.5.5 Codice di Gray	16
2.5.6 Codice di parità	17
2.5.7 Codici di Hamming	18
3 Algebra booleana	20
3.1 Variabili e funzioni di commutazione	20
3.2 Operatori e assiomi fondamentali dell'algebra booleana	20
3.3 Legge di dualità	20
3.4 Proprietà algebra booleana	20
3.4.1 Proprietà di idempotenza	20
3.4.2 Annichilatori funzionali	21
3.4.3 Legge dell'assorbimento	21
3.4.4 Teorema di De Morgan	21
3.5 Tabelle di verità	21
3.5.1 Tabelle di verità degli operatori fondamentali	22
3.6 Operatori derivati	22
3.6.1 OR esclusivo (XOR)	22

3.6.2	Not AND (NAND)	22
3.6.3	Not OR (NOR)	23
3.6.4	NOR esclusivo (XNOR)	23
3.7	Teorema di Shannon	23
3.8	Forma canonica in somma di prodotti	24
3.9	Forma canonica in prodotti di somme	24
3.10	Forme canoniche	24
3.10.1	Forma decimale	25
3.11	Forme semplificate	25
3.12	Mappe di Karnaugh	26
3.12.1	Semplificazioni mediante mappe di Karnaugh	26
3.12.2	Esempi di adiacenze	27
3.12.3	Don't care conditions	28
3.13	Operatori universali	29
3.13.1	Operatore universale NAND	29
3.13.2	Operatore universale NOR	29
4	Circuiti combinatori	30
4.1	Dalle funzioni booleane ai circuiti e viceversa	30
4.2	Bontà dei circuiti	32
4.2.1	Ritardo di commutazione	32
4.2.2	Costo di produzione	32
4.2.3	Minimizzazione del circuito	33
4.3	Forme canoniche come reti	33
4.4	Codificatore	34
4.5	Decodificatore	35
4.6	Multiplexer	36
4.6.1	Multiplexer come generatore di funzioni	36
4.7	Demultiplexer	37
4.8	Read Only Memory (ROM)	37
4.8.1	Schema logico di una ROM	37
4.8.2	Implementazione di una ROM	37
4.8.3	ROM paginata	38
4.8.4	Temporizzazione di una ROM	38
4.8.5	Programmable Logic Array	38
5	Reti iterative	40
5.1	Comparatori	40
5.1.1	Problemi reti iterative	41
5.1.2	Comparatori veloci	41
5.2	Reti iterative per la somma	42
5.2.1	Half adder	42
5.2.2	Full adder	42
5.2.3	Carry Lookahead Adder	43
5.3	Shifter	43
5.3.1	Barrel Shifter	44
5.4	Unità Logico Aritmetica (ALU)	45
6	Reti sequenziali	46
6.1	Circuito Latch	46
6.1.1	Problema del latch	46
6.2	Circuiti Flip Flop	46
6.2.1	Flip Flop SR	46
6.2.2	Flip Flop JK	47
6.2.3	Flip Flop D	47
6.2.4	Flip Flop T	47
6.3	Fronte di commutazione	48
6.4	Clock	48
6.5	Reti sequenziali	49

6.6	Macchine a stati finiti	49
6.6.1	Macchina di Moore	50
6.6.2	Macchina di Mealy	50
6.6.3	Rappresentazioni macchine di Moore e Mealy	50
6.6.4	Equivalenza tra modelli	51
6.6.5	Sintesi delle macchine	51
7	z64: Processing Unit (PU)	53
7.1	Instruction Set Architecture (ISA)	53
7.2	Architettura di von Neumann rivisitata	53
7.3	Registri	54
7.4	z64: architettura di alto livello	54
7.5	Processamento delle istruzioni	54
7.6	Modelli di esecuzione	55
7.7	Ciclo istruzione, ciclo macchina, stato macchina	55
7.8	Registri fisici e registri virtuali	56
7.9	Interconnessione tra registri	57
7.9.1	Interconnessione diretta	57
7.9.2	Interconnessione tramite multiplexer	57
7.9.3	Interconnessione tramite BUS interno	58
7.10	Instruction Pointer (RIP)	58
7.11	Instruction Register (IR)	58
7.12	Incremento del registro RIP	59
7.13	Modello di memoria	59
7.14	Trasferimento dati: il BUS	60
7.15	Interazione con la memoria (MAR, MDR)	60
7.16	Interconnessione tra registri e circuiti di calcolo	61
7.17	Banco dei registri	62
7.18	Registro FLAGS	62
7.18.1	Architettura registro FLAGS	62
7.18.2	Organizzazione bit registro FLAGS	63
7.18.3	Bit di stato e di controllo del registro FLAGS	63
7.19	Istruzioni e classi dello z64	63
7.20	Formato istruzioni macchina	64
7.20.1	Campo Opcode	64
7.20.2	Campo Mode	64
7.20.3	Campo SIB (scala, indice, base)	65
7.20.4	Campo R/M	66
7.21	Registri virtuali	66
7.21.1	Architettura rivisitata	67
7.22	Implementazione modalità di indirizzamento	68
7.23	Architettura finale della PU	69
8	Programmazione assembly	70
8.1	Alcune istruzioni assembly	70
8.2	Traduzioni istruzioni assembly in linguaggio macchina	70
8.3	Ordine dei byte: Memory Endianness	72
8.4	Istruzioni dello z64	73
8.5	Stack di programma	73
8.5.1	Gestione dello stack	73
8.6	Classi delle istruzioni dello z64	74
8.6.1	Classe 0: controllo hardware	74
8.6.2	Classe 1: istruzioni di movimento dati	74
8.6.3	Classe 2: istruzioni logico-aritmetiche	75
8.6.4	Classe 3: istruzioni di rotazione e shift	75
8.6.5	Classe 4: manipolazione dei bit di FLAGS	76
8.6.6	Classe 5: controllo del flusso di programma	76
8.6.7	Classe 6: controllo condizionale del flusso	76

9 z64: Control Unit (CU)	77
9.1 Microoperazioni	77
9.2 Realizzazione della CU	77
9.3 Organizzazione della CU	77
9.4 La CU come macchina di Mealy	78
10 La gerarchia di memoria	79
10.1 Memoria interna, principale e secondaria	79
10.1.1 Tecnologie delle memorie	79
10.2 Memoria principale	79
10.2.1 Organizzazione logica	79
10.2.2 Organizzazione in moduli	80
10.2.3 Allineamento e disallineamento	80
10.2.4 Organizzazione a matrice	81
10.2.5 Funzionamento organizzazione a matrice	81
11 Organizzazione a pipeline	82
11.1 Pipelining	82
11.2 Set istruzioni del processore z64 a pipeline	82
11.3 Schema di principio delle architetture pipeline	83
11.3.1 Scema temporale dell'esecuzione di più istruzioni	83
11.4 Progettazione del datapath: elementi di base	83
11.5 Istruzioni logico aritmetiche	84
11.5.1 Formato	84
11.5.2 Implementazione	84
11.5.3 Architettura finale a pipeline	85

1 Definizione e tipi di calcolatori

Dal dizionario Treccani: un calcolatore è una macchina elettromeccanica per il calcolo, capace di accettare e immagazzinare informazioni in una forma stabilita, di elaborarle, e di fornire quindi i risultati dell'elaborazione sotto forma o di dati alfanumerici o di segnali per il governo automatico di altre macchine o processi.

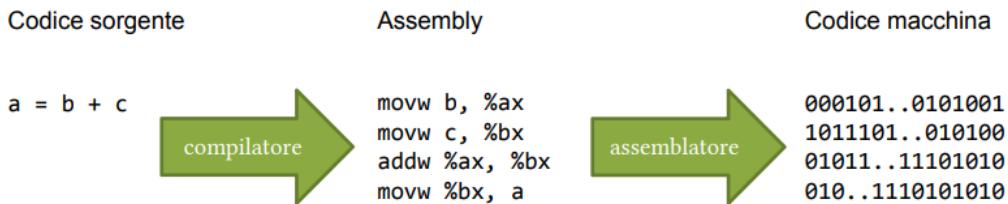
1.1 Evoluzione dei calcolatori

La prima macchina utilizzata per "calcolare" è stata l'*abaco*, la prima macchina programmabile, il *telaio di Jacquard*, fu brevettata da nel **1804**. Nel **1938** venne utilizzata la *bomba kryptologiczna* per la decifrazione a forza bruta dei messaggi segreti tedeschi, poi ripresa ed estesa da Alan Turing nel 1939. Nel **1946** l'*Eniac* fu il primo computer general purpose della storia.

1.2 Hardware e software

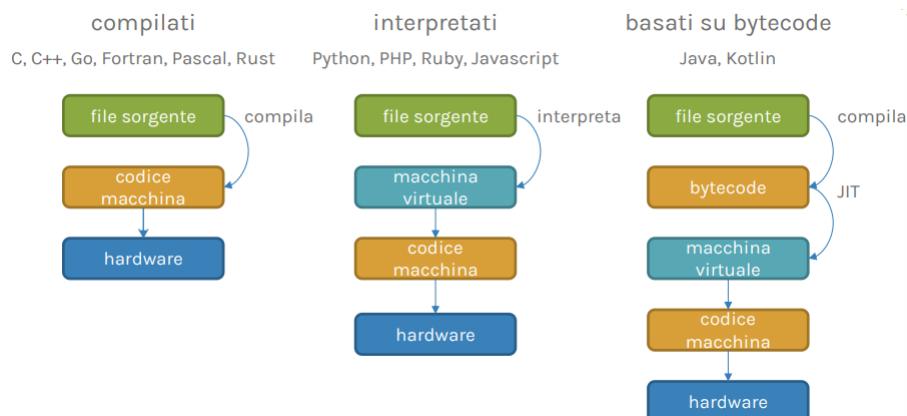
- L'**hardware** comprende le parti fisiche di un computer, la "rivoluzione informatica" è iniziata quando l'hardware è diventato programmabile.
- Il **software** è un insieme di dati e istruzioni che indicano al computer come lavorare. Le unità di elaborazione (CPU) parlano il codice macchina.

1.3 Come nasce un programma



Il codice Assembly non è univoco ma ne esistono molteplici e dipende dal processore, il codice macchina sono il set di istruzioni che vengono interpretate dal processore. Tra l'Assembly e il codice macchina c'è una corrispondenza 1 : 1 (una riga di Assembly corrisponde a una riga di codice macchina).

1.4 Modelli di esecuzione e linguaggi di programmazione

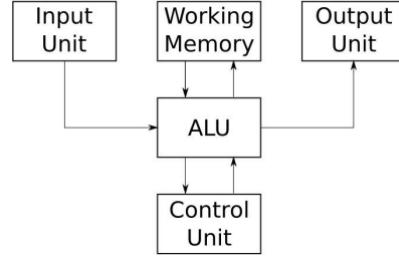


Alla base c'è sempre l'**hardware**.

1.5 Architetture dei processori

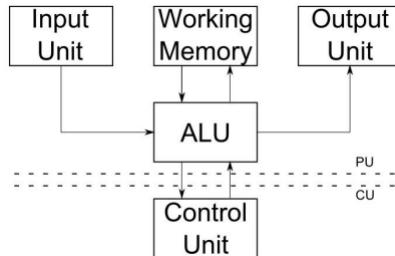
1.5.1 Architettura di Von Neumann

In questa architettura la componente centrale è l'unità logico aritmetica (ALU) che è pilotata dalla Control Unit secondo le nostre istruzioni. L' ALU comunica con dispositivi di ingresso e di uscita.



Organizzazione CU/PU La ALU e l'unità di controllo sono realizzate in maniera differente, dato che hanno requisiti e obiettivi diversi. Possiamo quindi operare una separazione logica tra:

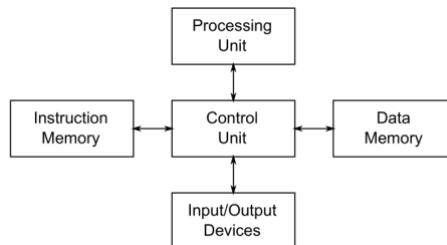
- Unità di **processamento** (PU): tutto ciò che "esegue il lavoro".
- Unità di **controllo** (CU): la parte "intelligente".



Il problema di questa architettura è che c'è una sola memoria che è contesa tra ALU e la Control Unit, questo significa latenza di accesso maggiore. Si crea così un **collo di bottiglia**.

1.5.2 Architettura Harvard

In questa architettura ci sono datapath alternativi per le istruzioni e i dati, quindi il sistema sarà più reattivo ma l'unità di controllo deve governare l'accesso alle risorse in maniera più complessa.



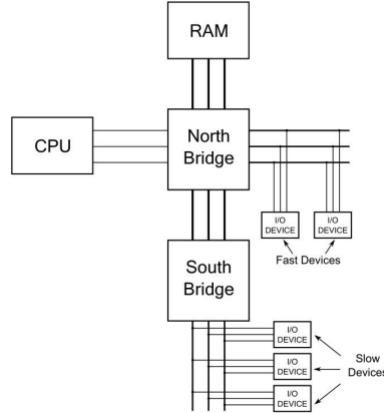
Questa architettura non è soggetta al problema della memoria contesa. Ma essendo le due memorie limitate se il programma è piccolo ma ci sono tanti dati (o viceversa) non si può eseguire. Infatti questa architettura è utilizzata solo in scenari molto specifici (cache).

1.5.3 Architettura moderna

Il processore è troppo veloce per la memoria e gli altri dispositivi e quindi perderebbe troppo tempo ad aspettarli. In questa architettura il processore si fa aiutare da alcuni **coprocessori** (North Bridge e South Bridge) che fanno da tramite.

Quindi la CPU che è il componente più veloce è mediato dai coprocessori per interagire con i dispositivi I/O (tastiera, schermo, ...).

Nei processori più moderni, i coprocessori sono all'interno della CPU.



2 Rappresentazione dei dati e codici

- Numero: entità astratta che esiste indipendentemente dalla nostra rappresentazione.
- Numerale: sequenza di caratteri che rappresenta un numero in un dato sistema di numerazione.

2.1 Sistemi numerici posizionali

Questo sistema per la rappresentazione funziona con qualsiasi base numerica

$$(x)_b = \langle a_n a_{n-1} \cdots a_1 a_0, c_1 c_2 c_3 \cdots \rangle = \sum_{i=0}^n (a_i \cdot b^i) + \sum_{k=1}^{\infty} c_k b^{-k}$$

2.2 Conversione di base

- Parte **intera**:

1. si effettuano divisioni intere successive per la base di destinazione;
2. ci si ferma quando si è arrivati al valore 0;
3. si ordinano i resti delle divisioni dall'ultimo al primo (\uparrow).

- Parte **frazionario**:

1. si effettuano moltiplicazioni successive per la base di destinazione;
2. si sottrae la parte intera;
3. ci si ferma quando si è arrivati al valore 0 o quando si individua un *periodo*;
4. si ordinano le parti intere sottratte dalla prima all'ultima (\downarrow).

I sistemi più comuni sono il sistema binario (base 2), il sistema ottale (base 8) e il sistema esadecimale (base 16).

2.2.1 Intervalli rappresentabili con la scrittura posizionale

Dato un numero k di cifre di un alfabeto associato ad un sistema numerico in base b , è possibile rappresentare tutti i valori nell'intervallo:

$$[0, b^k - 1].$$

Allo stesso modo, per rappresentare un numero n in base b sono necessarie un numero di cifre pari a:

$$k = \lceil \log_b n \rceil$$

dove $\lceil \cdot \rceil$ sta per la parte intera + 1.

Es.: per rappresentare $(54)_{10}$ in base 2 servono $k = \lceil \log_2 54 \rceil = \lceil 5.755 \rceil = 6$ cifre.

Gli intervalli rappresentabili sono importanti da conoscere perché le codifiche funzionano su insiemi finiti, e perché il processore è in grado di processare unicamente informazioni a **dimensione prefissata**.

2.3 Numeri negativi in base 2

È necessario prevedere una codifica anche per i numeri negativi, questa codifica deve essere:

- ragionevole per il processore;
- deve permettere operazioni veloci;
- deve permettere un'occupazione di memoria ridotta.

2.3.1 Rappresentazione in modulo e segno

In questa rappresentazione si utilizza uno dei bit per il segno.

1	0	0	0	1	1	0	0	= -12
0	0	0	0	1	1	0	0	= +12

È interessante notare che in questa rappresentazione esistono due zeri: $+0$ e -0 .

Questa rappresentazione è **inefficiente** dato che solo il segno occupa un bit, ciò richiede circuiti più complessi e prestazioni minori.

2.3.2 Complemento a uno

Qui i numeri negativi sono rappresentati come negativo aritmetico del valore del numero positivo, cioè con tutti i bit invertiti.

0	0	0	0	1	1	0	0	= +12
1	1	1	1	0	0	1	1	= -12

Il bit più significativo rappresenta ancora il segno ma fa parte anche della rappresentazione del numero. Il nome della rappresentazione viene dal fatto che, sommando un numero e il suo negato si ottiene una sequenza di tutti 1.

Con n cifre si possono rappresentare i numeri nell'intervallo $[-(2^{n-1} - 1), (2^{n-1} - 1)]$. Anche in questa rappresentazione esistono due zeri.

L'operazione di somma che coinvolge numeri negativi nella rappresentazione complemento a uno è più complessa, per esempio nella somma di (-1) e 2 in base binaria si otterrà:

$$\begin{array}{r} 1111\ 1110 \\ 0000\ 0010 \\ \hline 10000\ 0000 \end{array} \quad +$$

Il risultato è sbagliato, per ottenere il risultato corretto è necessario sommare al risultato ottenuto anche il bit di riporto ($0000\ 0000 + 1$).

2.3.3 Complemento a due

Questa è la rappresentazione che consente l'implementazione hardware più semplice ed efficiente.

Data una rappresentazione ad n cifre, il complemento a due di un numero x è definita come il complemento a $2^n - x$, ossia $2^n - x$.

Per calcolare il complemento a due di x possiamo ragionare sulla proprietà fondamentale del complemento a uno di x : $x + \bar{x} = 2^n - 1$ quindi:

$$\begin{aligned}x + \bar{x} &= 2^n - 1 \\x + \bar{x} + 1 &= 2^n \\ \bar{x} + 1 &= 2^n - x\end{aligned}$$

Es.: conversione del numero 6 in -6:

$$\begin{aligned}x &= (0110)_2 = (6)_{10} \\ \bar{x} &= (1001)_2 \\ \bar{x} + 1 &= (1010)_2 = (-6)_{10}\end{aligned}$$

Regola pratica: per calcolare il complemento a due di un numero, si parte dal bit meno significativo. Si lasciano inalterati tutti i bit fino a quando non si trova il primo uno. Quindi si invertono tutti i bit rimanenti (il primo uno resta uguale).

La **somma** nel complemento a due può essere svolta dimenticandosi del segno, per esempio nella somma di 15 e -5 si avrà:

$$\begin{array}{r} 0000\ 1111 \quad + \\ 1111\ 1011 \quad = \\ \hline \textcolor{red}{1}0000\ 1010 \end{array}$$

Il riporto in rosso può essere tralasciato, nella somma di numeri a 8 bit vogliamo un risultato a 8 bit.

Il complemento a due ci permette di effettuare le **sottrazioni** semplicemente negando il sottraendo ed effettuando la somma. Questo è molto utile perché è possibile utilizzare lo stesso circuito per due operazioni.

Con n cifre si rappresentano i numeri nell'intervallo $[-2^{n-1}, 2^{n-1} - 1]$.

Esempio se $n = 4$ cifre, possiamo rappresentare l'intervallo $[-2^{4-1}, 2^{4-1} - 1] = [-8, 7]$:

$(0000)_2 = 0$	$(1000)_2 = -8$
$(0001)_2 = 1$	$(1001)_2 = -7$
$(0010)_2 = 2$	$(1010)_2 = -6$
$(0011)_2 = 3$	$(1011)_2 = -5$
$(0100)_2 = 4$	$(1100)_2 = -4$
$(0101)_2 = 5$	$(1101)_2 = -3$
$(0110)_2 = 6$	$(1110)_2 = -2$
$(0111)_2 = 7$	$(1111)_2 = -1$

Notiamo che:

i numeri negativi cominciano sempre con un 1; l'ordinamento non è preservato; vi è un solo zero; si codifica il -8 e non l'8.

Condizione di overflow Ci sono due casi in cui il risultato di una somma o una sottrazione in complemento a due non è corretto, poiché soggetto a overflow.

- Primo caso: somma algebrica di due numeri **positivi** A e B . Si ha overflow se $A + B \geq 2^{n-1}$ (max range), dove n è il numero di bit usati per la rappresentazione.
- Secondo caso: somma algebrica di due numeri **negativi** A e B . Si ha overflow se $A + B \geq 2^{n-1}$, dove n è il numero di bit usati per la rappresentazione.

Quindi se il risultato di un'operazione in complemento a due è troppo grande o troppo piccolo per il range con cui stiamo lavorando ($[-2^{n-1}, 2^{n-1} - 1]$) si ha una condizione di overflow.

La condizione di overflow non si verifica mai se i due operandi hanno segno discorde, invece si verifica se gli ultimi due riporti sono discordi.

$$\begin{array}{r}
 \textcolor{red}{0} \textcolor{red}{1} \\
 01111 + \\
 00001 = \\
 \hline
 10000
 \end{array}
 \quad
 \begin{array}{r}
 \textcolor{green}{0} \textcolor{green}{0} \\
 01100 + \\
 00001 = \\
 \hline
 01101
 \end{array}$$

Nel primo caso stiamo sommando due numeri positivi quindi ci aspettiamo come risultato un numero positivo, invece il risultato è sbagliato poiché gli ultimi due riporti sono discordi.

2.3.4 Rappresentazione in eccesso

In questa rappresentazione si seleziona un numero k nell'intervallo rappresentabile, la codifica binaria di k viene utilizzata per rappresentare lo **zero**.

Quando si utilizzano n bit, tipicamente si pone $k = 2^{n-1}$ o $k = 2^{n-1} - 1$ e lo zero è rappresentato con un valore con la sola cifra più significativa pari a 1. Inoltre, a differenza del complemento a 2, viene conservato l'ordinamento dei numeri.

La **codifica** è molto semplice:

$$\begin{aligned}
 x' &= x + k \\
 x &= x' - k.
 \end{aligned}$$

Esiste solo una rappresentazione dello zero e l'intervallo rappresentabile è $[-2^{n-1}, 2^{n-1} - 1]$.

Esempio se abbiamo $n = 3$ bit, l'intervallo rappresentabile è $[0, 2^n - 1] = [0, 7]$. Utilizzando la rappresentazione in eccesso con $k = 2^{n-1} = 4$ l'intervallo sarà $[-2^{n-1}, 2^{n-1} - 1] = [-4, 3]$, quindi:

0	1	2	3	4	5	6	7
-4	-3	-2	-1	0	1	2	3

2.4 Numeri in virgola mobile

Esiste un numero massimo che può essere rappresentato data una parola di n bit, se dobbiamo rappresentare una quantità maggiore dobbiamo affidarci ad un dispositivo che è in grado di manipolarli, la Floating Point Unit (FPU).

2.4.1 Rappresentazione di base

La rappresentazione dei numeri in virgola mobile si basa su uguaglianze del tipo:

$$12,345 = 1,2345 \times 10^1$$

dove la parte prima del \times è la mantissa, 10 è la base e 1 l'esponente.

Il nome *virgola mobile* deriva dal fatto che la virgola può muoversi avanti e indietro, basta adattare l'esponente.

Esistono diversi standard, noi studieremo lo standard IEEE 754 a 32 bit.

2.4.2 Rappresentazione (32 bit)



Un float ha dimensione 32 bit così utilizzati:

- segno s : 1 bit (usiamo la rappresentazione modulo e segno);
- esponente e : 8 bit;
- mantissa m : 23 bit.

L'esponente decimale E è rappresentato in eccesso a 127 ($= 2^{n-1} - 1$):

$$e = E + 127.$$

0	1	2	...	127	...	253	254	255
-128	-127	-126	...	0	...	125	126	127

La mantissa rappresenta il valore binario $(1.m)_2$, la parte intera viene omessa nella rappresentazione.

Il valore decimale può essere calcolato come:

$$(-1) \cdot 2^{e-127} \cdot 1.m$$

La rappresentazione in eccesso a 127 (e non 128) viene chiamata *normalizzata*, ma lo standard permette di rappresentare altri tipi di numero.

Esempio 1 Conversione di $(-53.1)_{10}$ in base 2

53	1	0.1	0.2	U = 0
26	0	0.2	0.4	U = 0
13	1	0.4	0.8	U = 0
6	0	0.8	1.6	U = 1
3	1	0.6	1.2	U = 1
1	1	0.2		
0				

$$(53)_{10} = (110101)_2$$

$$(0.1)_{10} = (0.0\overline{0011})_2$$

Quindi $-53.1 = (-110101.0\overline{0011})_2 = (-1.101010\overline{0011} \cdot 2^5)$.

- $s = 1$;
- $e = 5 + 127 = 132$;
- $m = 101010 0011 0011 \dots$ (per 23 cifre).

$e = 132 = (10000100)_2$, quindi la rappresentazione sarà:

1	1000 0100	101010 0011 0011 0011 0011 0
---	-----------	------------------------------

In generale i passaggi da fare sono:

1. convertire il numero in base 2;
2. spostare la virgola in modo tale che il numero sia della forma $1.m \times 2^e$;
3. aggiungere il bias (+127 nel caso IEEE 754) all'esponente (e) → converti il numero in binario.

Esempio 2 Conversione di $(0.3)_{10}$ in binario con 4 bit per l'esponente e 5 bit per la mantissa.

0.3	0.6	U = 0	(0.0 1001 1001) = (1.001100 $\cdot 2^{-2}$)
0.6	1.2	U = 1	$m = (001100)$.
0.2	0.4	U = 0	
0.4	0.8	U = 1	Il bias nel caso in cui l'esponente ha 4 bit è
0.8	1.6	U = 1	$2^{n-1} - 1 = 7$ (arbitrario), quindi:
0.6			$e = 7 + (-2) = 5 = (0101)_2$.

$$(0.3)_{10} = (0.0\overline{1001})_2$$

La rappresentazione sarà:

0	0101	00110
---	------	-------

2.4.3 IEEE 754: tipi di numero

- Numeri **normalizzati**: sono la maggior parte dei numeri rappresentabili dallo standard.
- Numeri **denormalizzati**: valori molto prossimi allo zero. La parte intera omessa non è 1 ma 0.
- **Zeri**: è possibile rappresentare ± 0 , la maggior parte delle operazioni ignora il segno, ma dividere per ± 0 può dare come risultato $\pm\infty$.
- **Infiniti** ± 0 : sono il risultato di una divisione per 0, o di un'operazione che genera un overflow.
- **NaNs** (Not a number): sono il risultato di un'operazione che non ha significato (inf – inf, $0/0$, ...).

<i>e</i>	<i>m</i>	Tipo di valore
[1,254]	qualsiasi	$(-1)^s \cdot 2^{e-127} \cdot 1.m$ (numeri normalizzati)
0	$\neq 0$	$(-1)^s \cdot 2^{-126} \cdot 0.m$ (numeri denormalizzati)
0	0	(-1) ^s · 0 (zero con segno)
255	0	$(-1)^s \cdot \infty$ (infinito con segno)
255	$\neq 0$	NaN

In alternativa ai NaN e infiniti è possibile richiedere alla FPU di sollevare delle eccezioni:

- **Invalid operation**: generata quando si calcola un'operazione non matematicamente corretta.
- **Overflow**: indica che il risultato di un numero è troppo grande per essere rappresentato da un numero in virgola mobile.
- **Division by zero**: quando si calcola $X/\pm 0$ e $x \neq 0$.
- **UnderFlow**: analoga dell'overflow per risultati troppo piccoli.
- **Inexact**: il risultato "reale" non può essere rappresentato (errore di arrotondamento).

2.4.4 Massimo e minimo numero positivo rappresentabile

Nel caso dei numeri **normalizzati**:

- La mantissa rappresenta i numeri nell'intervallo:
 $[(1.000000000000000000000000000000)_2, (1.1111111111111111111111)_2];$
- gli esponenti minimo e massimo sono -126 e 127;
- minimo: $2^{-127} \approx 1.1754943508 \times 10^{-38}$
- massimo: $2^{127} \times (2 - 2^{-23}) \approx 3.4028234664 \times 10^{38}$.

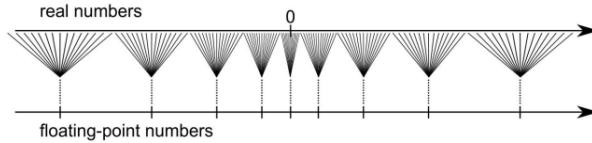
Nel caso dei numeri **denormalizzati**:

- La mantissa rappresenta i numeri nell'intervallo:
 $[(0.000000000000000000000000000001)_2, (0.1111111111111111111111)_2];$
- l'esponente è -126;
- minimo: $2^{-126} \times 2^{-23} = 2^{-149} \approx 1.4012984643 \times 10^{-45}$
- massimo: $2^{-126} \times (1 - 2^{-23}) \approx 1.1754942107 \times 10^{-38}$.

2.4.5 Errori di approssimazione

I numeri reali sono infiniti invece i bit utilizzati per la rappresentazione di un numero in virgola mobile sono finiti. Inoltre ogni volta che effettuiamo un'operazione su un numero in virgola mobile, commettiamo un **errore di approssimazione**.

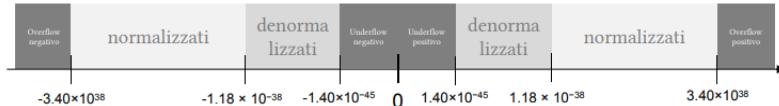
Il risultato è che, anche se l'insieme dei reali è totalmente connesso, l'insieme dei numeri in virgola mobile è *sparso*.



La presenza di numeri massimi e minimi crea dei buchi nella linea dei numeri rappresentabili. Può succedere anche tra la rappresentazione normalizzata e quella denormalizzata.

Se un numero non è rappresentabile in alcun modo può essere:

- *overflow*: numero troppo grande o troppo piccolo (negativo);
- *underflow*: arrotondamento allo zero.



La divisione del numero in mantissa e esponente fa nascere un'altra peculiarità del formato, quando siamo vicini allo zero, l'esponente è piccolo e quindi un incremento nella mantissa di 1 provoca un "salto" piccolo. Viceversa, per numeri grandi, questo salto sarà maggiore.

Questo significa che la **densità** dei numeri in virgola mobile non è costante:

- con 32 bit, circa metà dei numeri rappresentabili sono compresi tra -1 e 1,
- vi è la stessa quantità di numeri rappresentabili tra 65536 e 131072.

L'errore di approssimazione commesso nella rappresentazione di un numero in virgola mobile è quantificabile come:

- *errore assoluto*: è una variabile algebrica che ci dice quanto si è perso dell'informazione originale

$$\varepsilon_A = x - x',$$

- *errore relativo*: è una quantità adimensionale che indica se l'errore commesso è grande o piccolo

$$\varepsilon_R = \frac{x - x'}{x}.$$

Invece la quantità $-\log_{10} \varepsilon_R$ ci indica il numero di cifre non affette da errore.

Esempio consideriamo $x = 3.5648722189$ e supponiamo di volerlo rappresentare utilizzando soltanto quattro cifre decimali: $x \approx x' = 3.5648$.

Con questa approssimazione otteniamo un errore relativo pari a:

$$\varepsilon_R = \frac{x - x'}{x} = \frac{3.5648722189 - 3.5648}{3.5648722189} = 0.000020258.$$

Quindi il numero di cifre affidabili di \bar{x} è pari a:

$$-\log_{10}(0.000020258) = 4.69.$$

2.5 Codici ridondanti e irridondanti

Data una parola di n cifre binarie possiamo far corrispondere quello che vogliamo. Per esempio i giorni della settimana:

Giorno	Codifica
Lunedì	000
Martedì	001
Mercoledì	010
Giovedì	011
Venerdì	100
Sabato	101
Domenica	110

Dati N elementi da rappresentare, n cifre binarie disponibili per la codifica e $m = \lceil \log_2 N \rceil$ cifre necessarie per rappresentare N :

- Se $n = m$, allora il codice è **irridondante** (utilizziamo almeno una volta tutte le cifre).
- Se $n > m$, allora il codice è **ridondante**.

Nel caso di un codice ridondante le $k = n - m$ cifre aggiuntive sono chiamate **cifre di controllo**.

2.5.1 Distanza di Hamming

Si dice distanza di Hamming h il numero minimo di cifre diverse tra due parole del codice, quindi $h = \min(d(x, y))$ per ogni $x \neq y$ appartenenti al codice.

$$\begin{aligned} d(10010, 01001) &= 4 \\ d(11010, 11001) &= 2 \end{aligned}$$

Nel caso di codice irridondante la distanza è 1. Un codice ridondante è capace di rivelare errori di peso $\leq h - 1$.

I codici ridondanti sono di grande importanza pratica, perché la ridondanza permette di **riconoscere o correggere gli errori**. Gli errori legati alle inversioni di bit si possono verificare, per esempio errori di trasmissioni o per effetto di agenti esterni (particelle ionizzanti che colpiscono celle di memoria).

2.5.2 Codici rivelatori di errori

Elemento	Codice 1	Codice 2
A	000	000
B	100	011
C	011	101
D	111	110

Nel codice 1 $h = 1$, nel codice 2 $h = 2$, quindi se nel codice 2 si verifica un errore e si inverte un bit ce ne possiamo accorgere perché, dato che la distanza minima h è 2, la codifica non fa parte del dominio.

Esempio Se per un errore la rappresentazione di A diventa da (000) a (001) ce ne possiamo accorgere perché $(001) \notin$ Codice 2.

2.5.3 Codice ASCII

ASCII acronimo di American Standard Code for Information Interchange è la rappresentazione più comune per i caratteri, basata su un byte.

- ASCII tradizionale: 7 bit, permette di rappresentare 128 caratteri differenti.
 - I codici di controllo sono [0, 31] e 127.
- ASCII esteso: 8 bit, permette di rappresentare 256 caratteri (compresi i codici di controllo).

Binary	Dec	Ascii									
000 0000	0	NUL	010 0000	32	space	100 0000	64	@	110 0000	96	`
000 0001	1	SOH	010 0001	33	!	100 0001	65	A	110 0001	97	a
000 0010	2	STX	010 0010	34	"	100 0010	66	B	110 0010	98	b
000 0011	3	ETX	010 0011	35	#	100 0011	67	C	110 0011	99	c
000 0100	4	EOT	010 0100	36	\$	100 0100	68	D	110 0100	100	d
000 0101	5	ENQ	010 0101	37	%	100 0101	69	E	110 0101	101	e
000 0110	6	ACK	010 0110	38	&	100 0110	70	F	110 0110	102	f
000 0111	7	BEL	010 0111	39	'	100 0111	71	G	110 0111	103	g
000 1000	8	BS	010 1000	40	(100 1000	72	H	110 1000	104	h
000 1001	9	HT	010 1001	41)	100 1001	73	I	110 1001	105	i
000 1010	10	LF	010 1010	42	*	100 1010	74	J	110 1010	106	j
000 1011	11	VT	010 1011	43	+	100 1011	75	K	110 1011	107	k
000 1100	12	FF	010 1100	44	,	100 1100	76	L	110 1100	108	l
000 1101	13	CR	010 1101	45	-	100 1101	77	M	110 1101	109	m
000 1110	14	SO	010 1110	46	.	100 1110	78	N	110 1110	110	n
000 1111	15	SI	010 1111	47	/	100 1111	79	O	110 1111	111	o
001 0000	16	DLE	011 0000	48	0	101 0000	80	P	111 0000	112	p
001 0001	17	DC1	011 0001	49	1	101 0001	81	Q	111 0001	113	q
001 0010	18	DC2	011 0010	50	2	101 0010	82	R	111 0010	114	r
001 0011	19	DC3	011 0011	51	3	101 0011	83	S	111 0011	115	s
001 0100	20	DC4	011 0100	52	4	101 0100	84	T	111 0100	116	t
001 0101	21	NAK	011 0101	53	5	101 0101	85	U	111 0101	117	u
001 0110	22	SYN	011 0110	54	6	101 0110	86	V	111 0110	118	v
001 0111	23	ETB	011 0111	55	7	101 0111	87	W	111 0111	119	w
001 1000	24	CAN	011 1000	56	8	101 1000	88	X	111 1000	120	x
001 1001	25	EM	011 1001	57	9	101 1001	89	Y	111 1001	121	y
001 1010	26	SUB	011 1010	58	:	101 1010	90	Z	111 1010	122	z
001 1011	27	ESC	011 1011	59	;	101 1011	91	[111 1011	123	{
001 1100	28	FS	011 1100	60	<	101 1100	92	\	111 1100	124	
001 1101	29	GS	011 1101	61	=	101 1101	93]	111 1101	125	}
001 1110	30	RS	011 1110	62	>	101 1110	94	^	111 1110	126	~
001 1111	31	US	011 1111	63	?	101 1111	95	_	110 0000	127	DEL

2.5.4 Codice Binary Coded Decimal (BCD)

Il BCD è un codice irridondante per rappresentare le 10 cifre decimali usando quattro cifre binarie.

Base 10	BCD	Base 10	BCD
0	0000	5	0101
1	0001	6	0110
2	0010	7	0111
3	0011	8	1000
4	0100	9	1001

2.5.5 Codice di Gray

Il codice di Gray è un codice irridondante a lunghezza fissa, la rappresentazione è tale per cui tra due numeri adiacenti cambia una ed una sola cifra binaria (anche tra 7 e 0).

Base 10	Gray	Base 10	Gray
0	000	4	110
1	001	5	111
2	011	6	101
3	010	7	100

2.5.6 Codice di parità

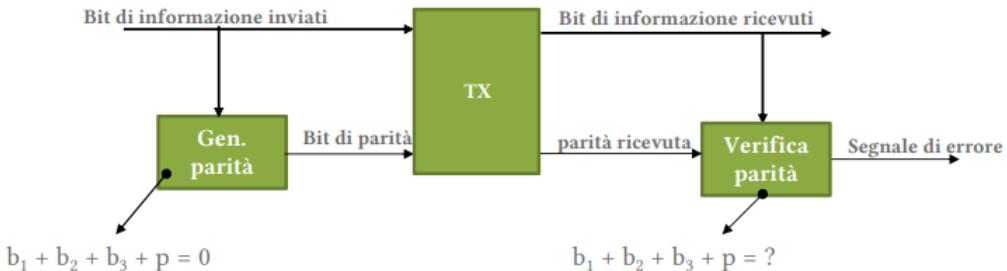
È un codice ridondante con $h = 2$, si ottiene aggiungendo una *cifra di parità* ad un codice irridondante.

Esistono due tipologie:

- parità (even parity): la cifra di parità vale 1 se il numero di 1 nella codifica irridondante è pari;
- disparità (odd parity): la cifra di parità vale 1 se il numero di 1 nella codifica irridondante è dispari.

Codice irridondante	Parità	Disparità
000	000 0	000 1
001	001 1	001 0
010	010 1	010 0
011	011 0	011 1
100	100 1	100 0
101	101 0	101 1
110	110 0	110 1
111	111 1	111 0

Con questa rappresentazione, supponendo di usare un codice di parità, si può determinare se c'è stato un (singolo) **errore di trasmissione** verificando la parità in ricezione.



Se in ricezione pari a zero non c'è stato errore di trasmissione (o c'è ne stato più di uno).

Esempio Vogliamo trasmettere (101), il generatore di parità calcola $p = 0$, viene trasmesso (1010).

Ricevuto	Parità	Segnale di errore
1010	uguale a zero	OK
1110	diversa da 0	ERRORE
1111	uguale a zero	OK

Se si verifica un errore e si "flippa" il bit 3 (secondo caso) la parità è $\neq 0$ quindi ci accorgiamo dell'errore. Nel terzo caso ci sono due errori (nel primo e terzo bit) qui la verifica non restituisce errore, perché il valore è legale (appartiene al dominio) anche se sbagliato.

2.5.7 Codici di Hamming

Il codice di Hamming è un metodo per la costruzione di codici a distanza $h \geq 3$.

Data una parola di codice di $m = n + k$ cifre, con $n \leq 2^k - k - 1$, dove n sono le cifre necessarie:

- i bit in posizione 2^i sono bit di **parità** (cifre aggiuntive);
- ciascun bit di parità controlla la **correttezza** dei bit di informazione la cui posizione, espressa in binario, ha un 1 nella potenza di 2 corrispondente al bit di parità.

Posizione cifra	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	
Dato codificato	p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8	d9	d10	d11	p16	d12	d13	d14	d15	
Copertura bit di parità	p1	✓		✓				✓				✓			✓						...
p2		✓	✓					✓	✓			✓	✓						✓	✓	
p4				✓	✓	✓	✓								✓	✓	✓	✓			
p8								✓	✓	✓	✓	✓	✓	✓	✓						
p16																	✓	✓	✓	✓	

8	4	2	1	
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	8
1	0	0	1	9
1	0	1	0	10
1	0	1	1	11
p8	p4	p2	p1	

Quindi i bit di parità sono

$$p_i \in \{1 (= 2^0), 2 (= 2^1), 4 (= 2^2), 8 (= 2^3), \dots\}.$$

p_1 ha \checkmark in $\{1, 3, 5, 7, 9, 11, \dots\}$, perché in corrispondenza ci sono gli 1.

p_2 ha \checkmark in $\{2, 3, 6, 7, 10, 11, \dots\}$, perché in corrispondenza ci sono gli 1.

E così via per p_4, p_8, p_{16}, \dots

Per determinare i bit di parità dobbiamo contare le \checkmark in corrispondenza degli uno, se sono dispari $p_i = 1$, se sono pari $p_i = 0$.

Esempio Trasformiamo la cifra ASCII 0 = (0110000) in un codice di Hamming $h = 3$.

Poiché $n = 7$ e $n \leq 2^k - k - 1 \Rightarrow k = 4$ e $m = 11$. I bit di parità sono in posizione 1, 2, 4, 8.

Posizione bit	1	2	3	4	5	6	7	8	9	10	11	
Dato codificato	p1	p2	0	p4	1	1	0	p8	0	0	0	
Copertura bit di parità	p1	✓		✓		✓		✓			✓	1
p2			✓	✓				✓	✓		✓	1
p4				✓	✓	✓	✓					0
p8								✓	✓	✓	✓	0

Dalla tabella precedente sostituiamo (d_1, d_2, d_3, \dots) con le cifre della rappresentazione, poi ricaviamo le cifre di parità.

- Nella prima riga c'è un 1 in corrispondenza delle \checkmark , quindi $p_1 = 1$;
- nella seconda riga c'è un 1 in corrispondenza delle \checkmark , quindi $p_2 = 1$;
- nella terza riga ci sono due 1 in corrispondenza delle \checkmark , quindi $p_4 = 0$;
- nella quarta riga ci sono zero 1 in corrispondenza delle \checkmark , quindi $p_8 = 0$.

Il valore codificato è quindi:

$$\begin{array}{cccccccccc} 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ p_1 & p_2 & & p_4 & & & & p_8 & & \end{array}$$

Ora supponiamo di ricevere (110110100) invece di (1101100000). Possiamo calcolare il **numero di controllo** $N_C = \langle p_8 p_4 p_2 p_1 \rangle$.

Essendo il bit 9 sbagliato, riceviamo:

Posizione bit	1	2	3	4	5	6	7	8	9	10	11	
Dato codificato	1	1	0	0	1	1	0	0	1	0	0	
Copertura bit di parità	p1	✓		✓		✓		✓		✓	✓	1
p2			✓	✓			✓	✓		✓	✓	0
p4					✓	✓	✓	✓				0
p8								✓	✓	✓	✓	1

quindi $(p_1 p_2 p_4 p_8)$ cambiano e diventano $(p_1 = 1, p_2 = 0, p_4 = 0, p_8 = 1)$. Il numero di controllo sarà $N_C = (1001)_2 = (9)_{10}$, sappiamo che si è verificato un errore di trasmissione (dato che i bit di parità sono diversi).

Inoltre sappiamo anche che il bit errato è in nona posizione, possiamo così ricostruire il valore corretto del dato trasmesso.

3 Algebra booleana

L'algebra Booleana è un tipo di algebra definita da George Bool. Nel 1936, si cominciò ad utilizzare questa algebra per studiare e progettare circuiti basati su relé, dato che entrambi si basano su due stati: vero/falso e aperto/chiuso.

3.1 Variabili e funzioni di commutazione

Una **variabile booleana** è una quantità algebrica x definita su un insieme $S = \{0, 1\}$, ossia che può assumere solo due valori.

Una **funzione di commutazione** di una variabile booleana è definita come la proiezione di $\{0, 1\}$ su $\{0, 1\}$:

$$f : \{0, 1\} \mapsto \{0, 1\}; \quad y = f(x)$$

Una funzione di commutazione di n variabili booleane è una funzione il cui dominio è dato da tutte le n -uple (x_1, x_2, \dots, x_n) in $\{0, 1\}^n$ ed il codominio è $\{0, 1\}$:

$$f : \{0, 1\}^n \mapsto \{0, 1\}; \quad y = f(x_1, x_2, \dots, x_n)$$

3.2 Operatori e assiomi fondamentali dell'algebra booleana

- *Somma logica*: si indica con il segno $+$;
- *prodotto logico*: si indica con il simbolo \cdot ;
- *negazione*: dato un valore x , il suo valore negato è \bar{x} .

Essi permettono di definire gli **assiomi fondamentali** sul dominio S :

- (i) chiusura: $a + b \in S, a \cdot b \in S; \forall a, b \in S$
- (ii) elemento identità: $\exists 0 \in S : a + 0 = a, \exists 1 \in S : a \cdot 1 = a; \forall a, b \in S$
- (iii) proprietà commutativa: $a + b = b + a$
- (iv) proprietà associativa: $(a + b) + c = a + (b + c), (a \cdot b) \cdot c = a \cdot (b \cdot c)$
- (v) proprietà distributiva: $a + b \cdot c = (a + b) \cdot (a + c), a \cdot (b + c) = a \cdot b + a \cdot c$
- (vi) elemento inverso: $\forall a \in S \exists \bar{a} \in S : a + \bar{a} = 1, a \cdot \bar{a} = 0$
- (vii) cardinalità: $|S| = 2^n$

3.3 Legge di dualità

Ogni identità booleana rimane invariata scambiando $+$ con \cdot e 0 con 1.

3.4 Proprietà algebra booleana

3.4.1 Proprietà di idempotenza

Nell'algebra booleana vale: $a + a = a$ e $a \cdot a = a$.

Infatti:

- $a = a + 0$
- $a = a + a \cdot \bar{a}$
- $a = a + a$
- $a = a \cdot a$ (per legge di dualità)

3.4.2 Annichilatori funzionali

Nell'algebra booleana vale: $a + 1 = 1$ e $a \cdot 0 = 0$.

Infatti:

- $a + 1 = a + a + \bar{a}$
- $a + 1 = a + \bar{a}$
- $a + 1 = 1$
- $a \cdot 0 = 0$ (per legge di dualità)

3.4.3 Legge dell'assorbimento

Nell'algebra booleana vale: $a + a \cdot b = a$ e $a \cdot (a + b) = a$

Infatti:

- $a + a \cdot b = a \cdot 1 + a \cdot b$
- $a + a \cdot b = a \cdot (1 + b)$
- $a + a \cdot b = a$
- $a \cdot (a + b) = a$ (per legge di dualità)

3.4.4 Teorema di De Morgan

Il teorema di De Morgan ci permette di esprimere gli operatori $+$ e \cdot in funzione degli altri due operatori fondamentali:

$$\overline{a + b} = \bar{a} \cdot \bar{b} \quad \overline{a \cdot b} = \bar{a} + \bar{b}$$

Questo è vero per un numero qualsiasi di variabili. Per dimostrare il teorema è sufficiente verificare che $\bar{a} \cdot \bar{b}$ è il complemento di $a + b$ (la somma di un elemento per il suo complemento è 1):

$$\begin{aligned} (a + b) + (\bar{a} \cdot \bar{b}) &= (a + b + \bar{b}) = \\ &= (1 + b) \cdot (1 + a) = (a + b) + (\bar{a} \cdot \bar{b}) = \\ &= 1 \cdot 1 = 1 \end{aligned}$$

3.5 Tabelle di verità

Una tabella di verità crea una relazione tra le variabili di input ed il valore di output della funzione.

Una tabella di verità di una funzione di n variabili è costituita da 2^n righe.

x	x_1	x_2	x_3	y
0	0	0	0	0
1	0	0	1	1
2	0	1	0	1
3	0	1	1	1
4	1	0	0	0
5	1	0	1	0
6	1	1	0	1
7	1	1	1	0

Nelle tabelle, spesso si utilizza in maniera intercambiabile il vettore $\mathbf{x} = \langle x_1, x_2, \dots, x_n \rangle$ e le variabili x_1, x_2, \dots, x_n .

3.5.1 Tabelle di verità degli operatori fondamentali

Anche gli operatori fondamentali possono essere visti come funzioni di commutazione su una o due variabili di commutazione.

Quindi è possibile esprimere gli operatori fondamentali come tabelle di verità.

Negazione		Somma logica			Prodotto logico		
x_1	y	x_1	x_2	y	x_1	x_2	y
0	1	0	0	0	0	0	0
1	0	0	1	1	0	1	0
		1	0	1	1	0	0
		1	1	1	1	1	1

3.6 Operatori derivati

Sulla base dei tre operatori fondamentali è possibile definire altri operatori, chiamati derivati.

3.6.1 OR esclusivo (XOR)

È utilizzato per verificare la disegualanza tra due variabili, ed è definito così:

$$a \oplus b = \bar{a}b + a\bar{b}$$

Proprietà principali:

- $a \oplus b = b \oplus a$
- $a \oplus (b \oplus c) = (a \oplus b) \oplus c$
- $a \oplus a = 0$
- $a \oplus \bar{a} = 1$
- $a \oplus 1 = \bar{a}$
- $\bar{a} \oplus b = a \oplus \bar{b} = \overline{a \oplus b}$

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

3.6.2 Not AND (NAND)

È definito così:

$$a|b = \overline{a \cdot b} = \bar{a} + \bar{b}$$

Proprietà principali:

- $a|b = b|a$
- $a|1 = \bar{a}$
- $a|0 = 1$
- $a|\bar{a} = 1$
- $a|(b|c) \neq (a|b)|c$

x_1	x_2	y
0	0	1
0	1	1
1	0	1
1	1	0

3.6.3 Not OR (NOR)

È definito così:

$$a \downarrow b = \overline{a + b} = \bar{a} \cdot \bar{b}$$

Proprietà principali:

- $a \downarrow b = b \downarrow a$
- $a \downarrow 1 = 0$
- $a \downarrow 0 = \bar{a}$
- $a \downarrow \bar{a} = 0$
- $a \downarrow (b \downarrow c) \neq (a \downarrow b) \downarrow c$

x_1	x_2	y
0	0	1
0	1	0
1	0	0
1	1	0

3.6.4 NOR esclusivo (XNOR)

Anche lui è utilizzato per la verifica di disuguaglianze, è definito così:

$$a \odot b = (\bar{a} + b) \cdot (a + \bar{b})$$

Proprietà principali:

- $a \odot b = b \odot a$
- $a \odot 1 = a$
- $a \odot 0 = \bar{a}$
- $a \odot \bar{a} = 0$
- $a \odot (b \odot c) = (a \odot b) \odot c$

x_1	x_2	y
0	0	1
0	1	0
1	0	0
1	1	1

3.7 Teorema di Shannon

Una qualsiasi funzione $y = f(x_1, x_2, \dots, x_n)$ può essere rappresentata in una delle due seguenti forme duali:

- somme di prodotti: $f(x_1, x_2, \dots, x_n) = x_1 \cdot f(1, x_2, \dots, x_n) + \bar{x}_1 \cdot f(0, x_2, \dots, x_n)$
- prodotti di somme: $f(x_1, x_2, \dots, x_n) = (x_1 + f(0, x_2, \dots, x_n)) \cdot (\bar{x}_1 + f(1, x_2, \dots, x_n))$

Applicando iterativamente il teorema ai residui si ottiene:

$$\begin{aligned} f(x_1, \dots, x_n) &= \bar{x}_1 \cdot \bar{x}_2 \cdot \dots \cdot \bar{x}_n \cdot f(0, 0, \dots, 0) + \\ &\quad + x_1 \cdot \bar{x}_2 \cdot \dots \cdot \bar{x}_n \cdot f(1, 0, \dots, 0) + \\ &\quad + \bar{x}_1 \cdot x_2 \cdot \dots \cdot \bar{x}_n \cdot f(0, 1, \dots, 0) + \\ &\quad + \dots + \\ &\quad + x_1 \cdot x_2 \cdot \dots \cdot \bar{x}_n \cdot f(1, 1, \dots, 0) + \\ &\quad + x_1 \cdot x_2 \cdot \dots \cdot x_n \cdot f(1, 1, \dots, 1) \end{aligned}$$

Generalizzando, quindi, possiamo esprimere ciascun termine come:

$$x_1^{\alpha_1} x_2^{\alpha_2} \dots x_n^{\alpha_n} f(\alpha_1, \alpha_2, \dots, \alpha_n)$$

con $\alpha_i \in \{0, 1\}$,

- $x_i^{\alpha_i} = x_i$ se $\alpha_i = 1$
- $x_i^{\alpha_i} = \bar{x}_i$ se $\alpha_i = 0$

3.8 Forma canonica in somma di prodotti

Prima forma canonica: somma di prodotti o forma canonica disgiuntiva:

$$f(x_1, x_2, \dots, x_n) = \sum_{k=0}^{2^n-1} \mathbf{m}_k f(\mathbf{k}).$$

Dove \mathbf{m}_k e \mathbf{k} sono vettori, \mathbf{m}_k viene chiamato **mintermine** ed è nella forma $x_1^{\alpha_1} x_2^{\alpha_2} \dots x_n^{\alpha_n}$. In ciascun mintermine le variabili compaiono una sola volta in forma diretta o negata.

3.9 Forma canonica in prodotti di somme

Utilizzando il teorema di De Morgan possiamo trasformare la somma di prodotti in prodotti di somme

$$\overline{f(x_1, x_2, \dots, x_n)} = \sum_{k=0}^{2^n-1} \mathbf{m}_k \overline{f(\mathbf{k})} = \dots = \prod_{k=0}^{2^n-1} (\mathbf{M}_k f(\mathbf{k}))$$

Dove il termine \mathbf{M}_k viene chiamato **maxtermine** ed è nella forma $\mathbf{M}_k = \sum_{i=0}^{n-1} x_i^{\alpha_i}$, con $\alpha_i = \{0, 1\}$ e $x_i^{\alpha_i} = x_i$ se $\alpha_i = 0$, $x_i^{\alpha_i} = \bar{x}_i$ se $\alpha_i = 1$

Esempio Consideriamo la seguente funzione di commutazione definita mediante tabella di verità

\mathbf{k}	x_1	x_2	x_3	$f(\mathbf{k})$
0	0	0	0	1
1	0	0	1	0
2	0	1	0	0
3	0	1	1	0
4	1	0	0	1
5	1	0	1	1
6	1	1	0	0
7	1	1	1	1

- i **mintermini** sono le configurazioni $\mathbf{k} = 0, 4, 5, 7$ quindi $(0, 0, 0), (1, 0, 0), (1, 0, 1), (1, 1, 1)$
- i **maxtermini** sono le configurazioni $\mathbf{k} = 1, 2, 3, 6$ quindi $(0, 0, 1), (0, 1, 0), (0, 1, 1), (1, 1, 0)$

Dai mintermini, possiamo definire la seguente rappresentazione in **somma di prodotti**:

$$f(x_1, x_2, x_3) = \overline{x_1} \overline{x_2} \overline{x_3} + x_1 \overline{x_2} \overline{x_3} + x_1 \overline{x_2} x_3 + x_1 x_2 x_3$$

Dai maxtermini, possiamo definire la seguente rappresentazione in **prodotti di somme**:

$$f(x_1, x_2, x_3) = (x_1 + x_2 + \overline{x_3})(x_1 + \overline{x_2} + x_3)(x_1 + \overline{x_2} + \overline{x_3})(\overline{x_1} + \overline{x_2} + x_3)$$

3.10 Forme canoniche

Le forme canoniche sono rappresentazioni uniformi utilizzate per descrivere una funzione di commutazione.

Qualsiasi funzione di commutazione può essere trasformata in forma canonica, usando:

- tabella di verità* e utilizzando la tecnica dei mintermini o dei maxtermini;
- trasformazioni analitiche*: se una variabile x_i non è presente, si può moltiplicare per $(x_i + \overline{x_i})$.

Esempio la funzione $x_1 x_3 + \overline{x_1} (x_2 + \overline{x_3})$ può essere trasformata come segue:

$$\begin{aligned} x_1 x_3 + \overline{x_1} (x_2 + \overline{x_3}) &= x_1 x_3 (x_2 + \overline{x_2}) + \overline{x_1} x_2 + \overline{x_1} x_3 = \\ &= x_1 x_2 x_3 + x_1 \overline{x_2} x_3 + \overline{x_1} x_2 (x_3 + \overline{x_3}) + \overline{x_1} \overline{x_3} (x_2 + \overline{x_2}) = \\ &= x_1 x_2 x_3 + x_1 \overline{x_2} x_3 + \overline{x_1} x_2 x_3 + \overline{x_1} x_2 \overline{x_3} + \overline{x_1} x_2 \overline{x_3} + \overline{x_1} \overline{x_2} \overline{x_3} \end{aligned}$$

3.10.1 Forma decimale

La forma tabellare o le forme canoniche possono essere molto lunghe. A volte possono essere rappresentate nella forma decimale, in cui si indica l'interpretazione decimale delle variabili booleane associate a mintermini e maxtermini.

$$f(\mathbf{k}) = \sum(0, 4, 5, 7) = \prod(1, 2, 3, 6)$$

k	x_1	x_2	x_3	$f(\mathbf{k})$
0	0	0	0	1
1	0	0	1	0
2	0	1	0	0
3	0	1	1	0
4	1	0	0	1
5	1	0	1	1
6	1	1	0	0
7	1	1	1	1

3.11 Forme semplificate

Le forme canoniche non sono necessariamente le forme *minime* per rappresentare una funzione booleana.

Identificare le forme minime è importante poiché permette di realizzare circuiti più compatti. Per il processo di semplificazione possiamo usare:

- metodi analitici, applicando le proprietà e i teoremi dell'algebra booleana;
- metodi algoritmici.

Esempio Semplificazione analitica di una funzione

$$f(x, y, z, w) = xyzw + xyz\bar{w} + xy\bar{z}\bar{w} + xy\bar{z}w + \bar{x}yzw$$

Per la proprietà dell'idempotenza:

- primo e secondo mintermine: $xyzw + xyz\bar{w} = xyz$
- primo e quarto mintermine: $xyzw + xy\bar{z}w = xyw$
- primo e quinto mintermine: $xyzw + \bar{x}yzw = yzw$
- secondo e terzo mintermine: $xy\bar{z}\bar{w} + xy\bar{z}w = xy\bar{w}$

Quindi:

$$f(x, y, z, w) = xyz + xy\bar{w} + yzw + xy\bar{w}$$

sempre per l'idempotenza:

- secondo e ultimo mintermine: $xyz + xy\bar{w} = xy$

$$f(x, y, z, w) = xy + xyz + yzw$$

per la legge dell'assorbimento ($a + a \cdot b = a$):

$$f(x, y, z, w) = xy + yzw$$

In sintesi abbiamo trovato:

$$f(x, y, z, w) = xyzw + xyz\bar{w} + xy\bar{z}\bar{w} + xy\bar{z}w + \bar{x}yzw = xy + yzw$$

3.12 Mappe di Karnaugh

Le mappe di Karnaugh sono una rappresentazione differente delle tabelle di verità, le variabili vengono organizzate in tabelle quadrate o rettangolari, a seconda del loro numero.

I valori che possono assumere le variabili vengono ordinati secondo un codice di Gray (distanza di Hamming = 1), quindi spostandosi da una cella all'altra si causa il *cambiamento del valore di una sola variabile*.

Dato che $a + \bar{a} = 1$, è possibile eliminare una variabile se la funzione assume lo stesso valore in gruppi di celle adiacenti.

$\begin{array}{c cc} & \bar{x}_0 & x_0 \\ \hline x_1 & 0 & 1 \\ \hline 0 & f(00) & f(10) \\ 1 & f(01) & f(10) \end{array}$	2 variabili
--	-------------

$\begin{array}{c cccc} & \bar{x}_0 \bar{x}_1 & \bar{x}_0 x_1 & x_0 \bar{x}_1 & x_0 x_1 \\ \hline x_2 & 00 & 01 & 11 & 10 \\ \hline 0 & f(000) & f(010) & f(110) & f(100) \\ 1 & f(001) & f(011) & f(111) & f(101) \end{array}$	3 variabili
--	-------------

$\begin{array}{c cccc} & \bar{x}_0 \bar{x}_1 \bar{x}_2 & \bar{x}_0 \bar{x}_1 x_2 & \bar{x}_0 x_1 \bar{x}_2 & \bar{x}_0 x_1 x_2 & x_0 \bar{x}_1 \bar{x}_2 & x_0 \bar{x}_1 x_2 & x_0 x_1 \bar{x}_2 & x_0 x_1 x_2 \\ \hline x_3 & 00 & 01 & 11 & 10 \\ \hline 0 & f(0000) & f(0100) & f(1100) & f(1000) \\ 01 & f(0001) & f(0101) & f(1101) & f(1001) \\ 11 & f(0011) & f(0111) & f(1111) & f(1011) \\ 10 & f(0010) & f(0110) & f(1110) & f(1010) \end{array}$	4 variabili
---	-------------

Grazie all'utilizzo del codice di Gray anche le *celle agli estremi sono adiacenti* (effetto Pacman).

3.12.1 Semplificazioni mediante mappe di Karnaugh

Per sfruttare le adiacenze è possibile costruire **insiemi di copertura** di dimensione 2^i (1,2,4,8,16,...) celle. Gli insiemi devono coprire tutti i termini 1, in questo modo identifichiamo gli *implicanti primi*, ossia gli insiemi di termini che determinano la funzione equivalente minima.

In alternativa è possibile anche lavorare con i maxtermini, in tal caso si parla di implicanti minimi e le coperture avvengono sugli 0.

Esempio Semplifichiamo la funzione $f(x,y,z,w) = \sum(0, 1, 3, 7, 15)$

Quindi in corrispondenza di (0000),(0001),(0011),(0111),(1111) ci saranno gli uno. Rappresentiamo la tabella di verità su una mappa di Karnaugh:

$\begin{array}{c cccc} & \bar{x}_y & 00 & 01 & 11 & 10 \\ \hline \bar{x}_w & 00 & 1 & 0 & 0 & 0 \\ 01 & 1 & 0 & 0 & 0 & 0 \\ 11 & 1 & 1 & 1 & 0 & 0 \\ 10 & 0 & 0 & 0 & 0 & 0 \end{array}$
--

Identifichiamo gli implicanti primi selezionando degli insiemi, di dimensione 2^i , fino a coprire tutti gli 1 almeno una volta.

$\begin{array}{c cccc} & \bar{x}_y & 00 & 01 & 11 & 10 \\ \hline \bar{x}_w & 00 & 1 & 0 & 0 & 0 \\ 01 & 1 & 0 & 0 & 0 & 0 \\ 11 & 1 & 1 & 1 & 1 & 0 \\ 10 & 0 & 0 & 0 & 0 & 0 \end{array}$
--

- Insieme rosso: la variabile che cambia è w .

- Insieme verde: la variabile che cambia è y .
- Insieme giallo: la variabile che cambia è x .

Quindi, semplificando le variabili che cambiano valore nelle celle adiacenti in ciascun insieme, otteniamo:

$$f(x, y, z, w) = \overline{xyz} + \overline{x}zw + yzw$$

Dato che abbiamo creato insiemi che ricoprono gli 1 la funzione minima è una somma di prodotti.

Gli **uno** sono rappresentati dalla variabile in modalità diretta e gli **zero** dalle variabili in modalità inversa.

È possibile effettuare la semplificazione creando insiemi che ricoprono gli **zero**, in questo caso è necessario esprimere la funzione come **prodotto di somme**. Gli 1 saranno rappresentati dalle variabili in modalità inversa e gli 0 dalle variabili in modalità diretta.

3.12.2 Esempi di adiacenze

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

Si semplificano x_0 e x_2
 Si semplificano x_0 e x_2

Figura 1: semplificazioni funzioni di 4 variabili

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

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

$x_4 = 0$ $x_4 = 1$

Figura 2: semplificazioni funzioni di 5 variabili

Nella seconda foto gli insiemi giallo e rosso sono solo due (tabelle sovrapposte). L'insieme verde è il caso di un mintermine che è anche implicante primo.

3.12.3 Don't care conditions

A volte una funzione è *parzialmente specificata*, in questi casi, il valore di uscita non è definito per tutte le configurazioni delle variabili di ingresso: ci potrebbero essere delle **configurazioni non di interesse**.

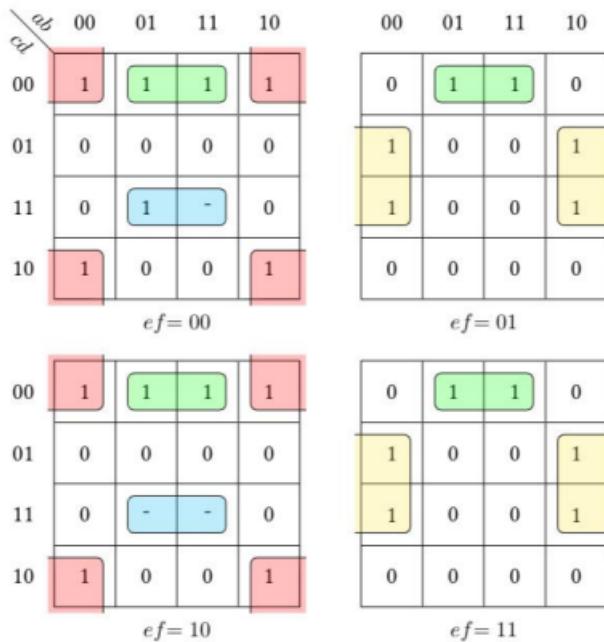
In questi casi i mintermini/maxtermini vengono associati (nella notazione decimale) a un insieme $\sum_{0/1}$ che rappresenta il fatto che non è di interesse (o non è noto) che il valore della funzione sia 0 o 1.

Nelle mappe di Karnaugh si indica tale condizione con un trattino (-) e si può far valere la funzione 1 o 0 a seconda di come è più comodo per la minimizzazione.

Esempio Semplifichiamo una funzione di 6 variabili con delle don't care conditions

$$\begin{aligned} f(a, b, c, d, e, f) = & \\ &= \sum(0, 1, 2, 3, 4, 5, 14, 16, 17, 18, 19, 20, 21, 29, 34, 35, 40, 41, 44, 50, 51, 56, 57, 60, 61) + \\ &+ \sum_{0/1}(15, 30, 31) \end{aligned}$$

Tale funzione ha 3 don't care conditions, che possono essere associate agli 0 o agli 1 come si preferisce.



L'insieme azzurro costruisce un insieme di copertura che racchiude 4 termini, permettendo una riduzione di 2 variabili. Se avessimo considerato il solo mintermine $\bar{a}\bar{b}\bar{c}\bar{d}\bar{e}\bar{f}$ l'espressione sarebbe stata più complessa.

3.13 Operatori universali

Abbiamo visto che esistono 3 operatori fondamentali ma, per definire l'algebra booleana, **ne bastano due**. In ogni caso, si può effettuare una doppia negazione e sfruttare il teorema di De Morgan.

Esempio

$$f(x, y, z) = x + yz + \overline{xz} = \overline{\overline{f}}(x, y, z) = \overline{\overline{x} \cdot \overline{yz} \cdot \overline{xz}}$$

Applicando nuovamente una doppia negazione e sfruttando il teorema di De Morgan, si sostituisce l'AND con l'OR.

Possiamo ridurre ancora l'algebra booleana ad **un solo operatore**

3.13.1 Operatore universale NAND

L'operatore NAND permette, da solo, di esprimere tutta l'algebra booleana, infatti:

- $a|a = \bar{a}$;
- $(a|a)|(b|b) = \overline{\bar{a} \cdot \bar{b}} = a + b$;
- $(a|b)|(a|b) = \overline{\overline{(a \cdot b)} \cdot \overline{(a \cdot b)}} = (a \cdot b) + (a \cdot b) = a \cdot b$;
- $a|(a|a) = \overline{a \cdot \bar{a}} = a + \bar{a} = 1$;
- $(a|(a|a))|(a|(a|a)) = 1|1 = 0$.

Quindi è possibile esprimere tutte le costanti e gli operatori fondamentali dell'algebra booleana sfruttando solo ed esclusivamente l'operatore NAND.

3.13.2 Operatore universale NOR

l'operatore NOR è stato definito come duale dell'operatore NAND, quindi anch'esso deve essere universale, infatti:

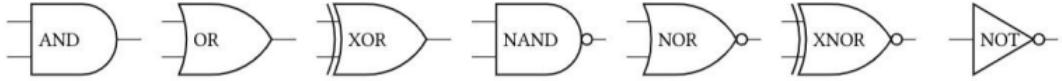
- $a \downarrow a = \bar{a}$;
- $(a \downarrow a)|(b \downarrow b) = \overline{\bar{a} + \bar{b}} = a \cdot b$;
- $(a \downarrow b) \downarrow (a \downarrow b) = \overline{\overline{(a + b)}} = a + b$;
- $a \downarrow (a \downarrow a) = 0$;
- $(a \downarrow (a \downarrow a)) \downarrow (a \downarrow (a \downarrow a)) = 1$.

Tali operatori universali possono portare benefici nell'implementazione dei circuiti perché la loro implementazione in hardware può richiedere un numero minore di componenti elettroniche.

4 Circuiti combinatori

I circuiti logici sono reti di componenti che accettano variabili booleane in input e restituiscono variabili booleane in output. Per la loro sintesi è utile affidarsi all'algebra booleana.

Esistono determinate *porte logiche* che implementano gli operatori booleani in hardware, tali operatori vengono astratte con dei simboli standard:



4.1 Dalle funzioni booleane ai circuiti e viceversa

Le porte logiche che abbiamo introdotto permettono di implementare in hardware qualsiasi funzione booleana.

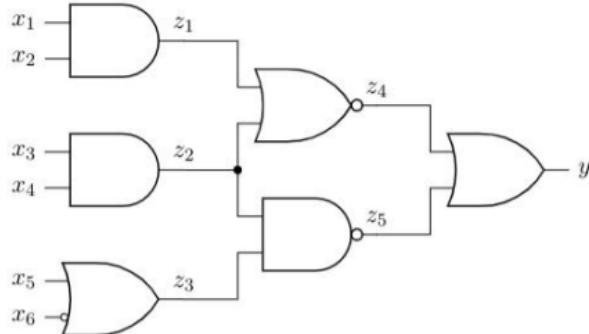
Per passare **dalla funzione al circuito**:

- si considerano i termini più interni (per precedenza) e si realizza quella parte della funzione come circuito;
- le uscite delle porte così definite vengono usate come input delle funzioni (porte) più esterne.

Per passare **dal circuito alla funzione**:

- si assegna un nome all'uscita di ogni porta;
- queste variabili ausiliarie vengono via via eliminate fino a quando non si ha un'espressione che usa solo le variabili di input.

Esempio Dal circuito alla funzione



$$\begin{aligned}
 y &= z_4 + z_5 = \\
 &= (z_1 \downarrow z_2) + (z_2 | z_3) = \\
 &= ((x_1 \cdot x_2) \downarrow (x_3 \cdot x_4)) + ((x_3 \cdot x_4) | (x_5 + \overline{x_6})) = \\
 &= \overline{x_1 x_2 + x_3 x_4} + \overline{x_3 x_4 (x_5 + \overline{x_6})}
 \end{aligned}$$

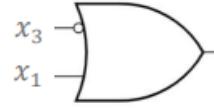
Esempio Dalla funzione al circuito

Consideriamo la funzione

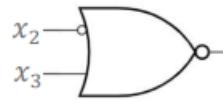
$$y = x_1 x_2 ((\overline{x}_2 \downarrow x_3) + (\overline{x}_1(x_1 + \overline{x}_3))).$$

Per costruire il circuito partiamo dai termini più interni e costruiamo le parti corrispondenti del circuito

- $(x_1 + \overline{x}_3)$

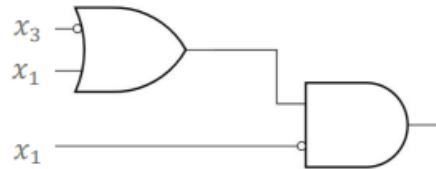


- $(\overline{x}_2 \downarrow x_3)$

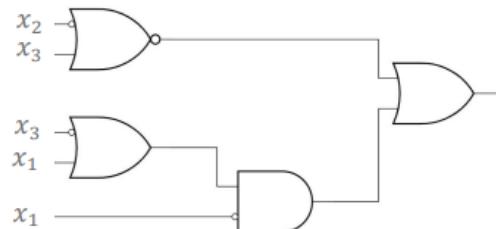


Poi ci spostiamo verso i termini più esterni

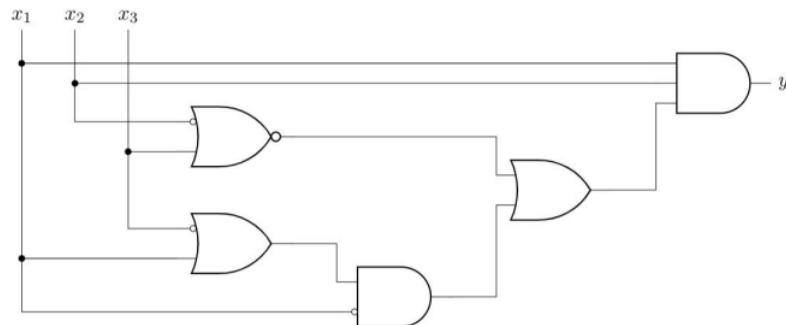
- $(\overline{x}_1(x_1 + \overline{x}_3))$



- $((\overline{x}_2 \downarrow x_3) + (\overline{x}_1(x_1 + \overline{x}_3)))$



Infine, otteniamo il circuito per l'intera funzione



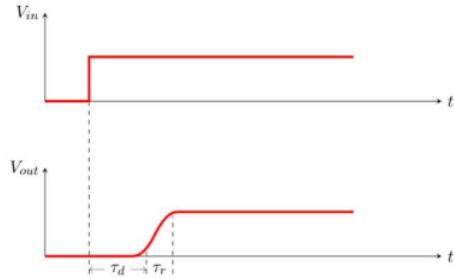
4.2 Bontà dei circuiti

Un qualsiasi circuito può essere valutato in due diversi modi:

- **costo**: il numero di componenti utilizzate per realizzare il circuito;
- **tempo**: la velocità del circuito a calcolare l'uscita dati gli ingressi.

4.2.1 Ritardo di commutazione

Una funzione booleana è impulsiva, un circuito di commutazione, invece, ha un ritardo di calcolo.

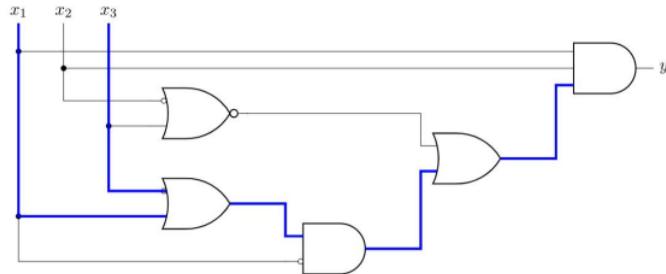


Essendo i circuiti componenti fisici, se applichiamo un gradino in ingresso, l'uscita si stabilizzerà dopo un po' di tempo:

- τ_d : ritardo di commutazione (tempo per arrivare al 10% del valore finale)
- τ_r : ritardo di salita (tempo per arrivare al 90% ¹ del valore finale)

Per semplicità, consideriamo un unico *tempo di propagazione*: $\tau_p = \tau_d + \tau_r$.

Il tempo di propagazione si **accumula** quando una funzione è implementata da più porte in cascata. Per stimare le prestazioni, si può ricorrere al *critical path* ovvero il percorso più lungo che un segnale di input attraversa in un circuito.



Nel caso in figura, possiamo stimare un ritardo pari a $4\tau_p$: il circuito è a quattro livelli.

4.2.2 Costo di produzione

Ogni porta logica richiede un certo numero di componenti per la sua realizzazione, che dipende anche dal numero di ingressi di ciascuna porta.

Se riduciamo sia il numero di porte che il numero di ingressi di ciascuna porta in un circuito, consumeremo meno transistor nell'implementazione finale.

¹il 90% è sufficiente dato che lavoriamo a solo due livelli

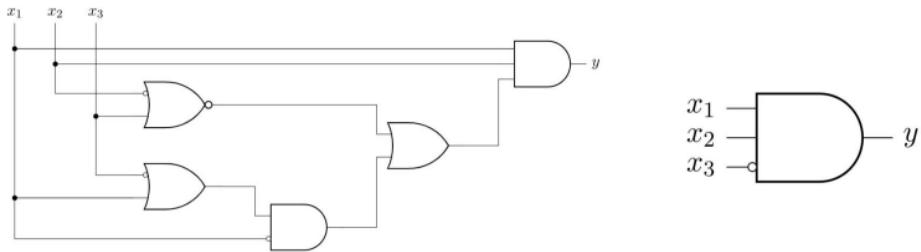
4.2.3 Minimizzazione del circuito

Possiamo sfruttare le tecniche di minimizzazione delle funzioni booleane per cercare di ridurre il costo e massimizzare le prestazioni del nostro circuito

Esempio

- $y = x_1 x_2 ((\bar{x}_2 \downarrow x_3) + (\bar{x}_1 (x_1 + \bar{x}_3)))$
- $y = x_1 x_2 ((x_2 \cdot \bar{x}_3) + (\bar{x}_1 (x_1 + \bar{x}_3)))$, definizione operatore NOR ($a \downarrow b = \overline{a + b} = \bar{a} \cdot \bar{b}$);
- $y = x_1 x_2 ((x_2 \cdot \bar{x}_3) + (\bar{x}_1 x_1 + \bar{x}_1 \bar{x}_3))$, proprietà distributiva;
- $y = x_1 x_2 (x_2 \cdot \bar{x}_3 + \bar{x}_1 \cdot \bar{x}_3)$, elemento inverso ($a \cdot \bar{a} = 0$) e elemento identità ($a + 0 = a$);
- $y = x_1 x_2 \bar{x}_3 + x_1 \bar{x}_1 x_2 \bar{x}_3$, proprietà distributiva e idempotenza ($a = a \cdot a$);
- $y = x_1 x_2 \bar{x}_3$, elemento inverso e elemento identità.

Grazie a queste minimizzazioni possiamo passare da un circuito più costoso a uno meno costoso:



È evidente la maggiore efficienza e il minor costo.

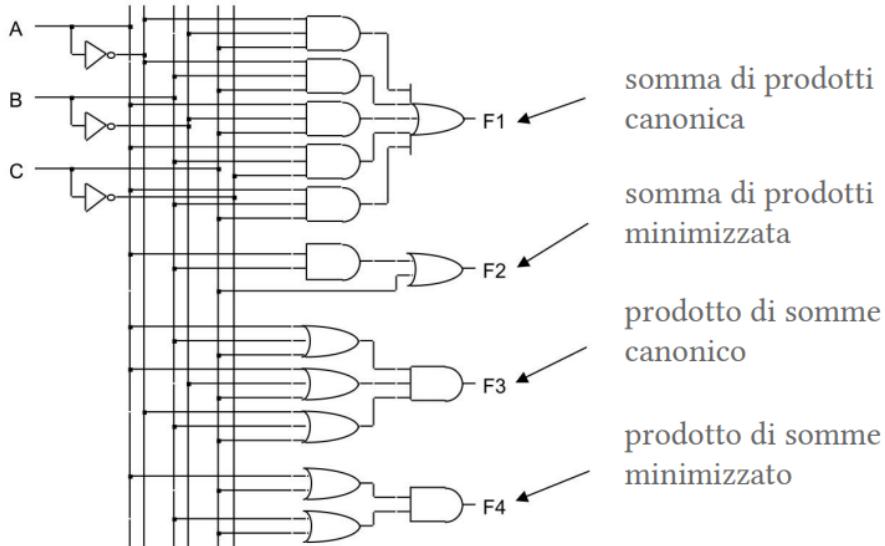
4.3 Forme canoniche come reti

Ogni funzione booleana y può essere espressa in forma canonica.

Le forme canoniche possono essere realizzate usando 2 soli livelli di porte logiche (AND/OR), questo ci garantisce una velocità elevata.

Non è detto che una rappresentazione in forma canonica sia in forma minima, minimizzare a partire da una forma canonica può permettere di mantenere la velocità legata ai due livelli.

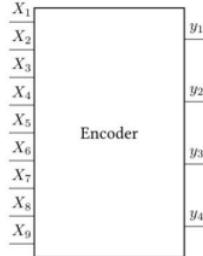
Esempio $f = ab + c$



4.4 Codificatore

Il codificatore (encoder) è un circuito che realizza la funzione di **codifica binaria**: associa ad ogni elemento di un certo insieme di codifica composto da m simboli una sequenza distinta di n bit.

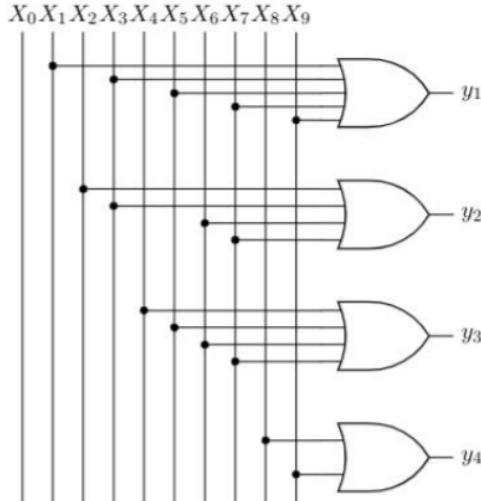
Per ogni simbolo, genera il codice corrispondente (con $2^n \geq m$). Il circuito ha quindi m linee di ingresso x_0, \dots, x_{m-1} ed n linee di uscita y_0, \dots, y_{n-1} .



Esempio Codificatore per le cifre decimali in BCD (da decimale a binario)

Base 10	BCD	Base 10	BCD
0	0000	5	0101
1	0001	6	0110
2	0010	7	0111
3	0011	8	1000
4	0100	9	1001

In questo caso abbiamo $m = 10$ linee di ingresso e $n = 4$ linee di uscita.



y_1 controlla se la prima cifra è 1 (solo 1,3,5,7,9 hanno 1 in prima posizione), y_2 lo fa per la seconda cifra e così via. Quindi la cifra codificata sarà $y = \langle y_4 y_3 y_2 y_1 \rangle$.

Se per esempio vogliamo codificare 3 in BCD, la linea X_3 sarà 1 e le altre 0.

- y_1 sarà 1 perché almeno una delle linee dove è collegata è 1;
- y_2 sarà 1 perché almeno una delle linee dove è collegata è 1;
- y_3 sarà 0 perché nessuna delle linee dove è collegata è 1;
- y_4 sarà 0 perché nessuna delle linee dove è collegata è 1.

Quindi la codifica di 3 in BCD sarà $\langle y_4 y_3 y_2 y_1 \rangle = \langle 0011 \rangle$.

4.5 Decodificatore

Il decodificatore realizza la funzione inversa del codificatore, a partire da una parola di un codice binario, genera una uscita che identifica uno dei simboli dell'insieme di interesse.

Per ciascuna configurazione di ingresso, una sola uscita vale 1. Le altre uscite valgono 0.

Esempio Invertiamo il codificatore per BCD

Vogliamo generare uno dei segnali X_0, \dots, X_9 a partire dalla codifica $y = \langle y_4 y_3 y_2 y_1 \rangle$.

Il generico segnale X_i è generato dall' i -esimo mintermine delle variabili y_1, y_2, y_3, y_4 , quindi le equazioni sono del tipo:

$$X_0 = \overline{y_1} \overline{y_2} \overline{y_3} \overline{y_4}$$

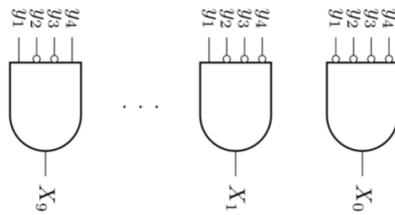
$$X_1 = y_1 \overline{y_2} \overline{y_3} \overline{y_4}$$

⋮

$$X_9 = y_1 \overline{y_2} \overline{y_3} y_4$$

Il circuito può quindi essere realizzato con una serie di porte AND.

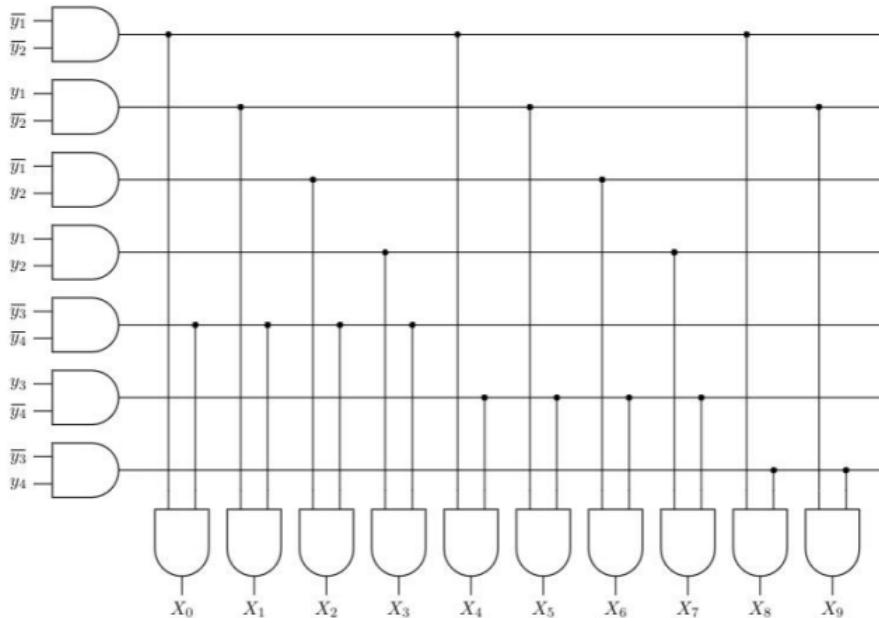
La prima realizzazione è immediata



Tuttavia, possiamo ridurre la complessità (aumentando il tempo di calcolo) usando un circuito a due livelli invece che a uno.

Decodifichiamo prima $y_1 y_2$ e poi $y_3 y_4$, infine calcoliamo i prodotti incrociati.

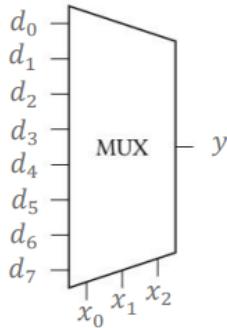
Il circuito sarà:



4.6 Multiplexer

In alcuni casi è necessario scegliere tra più segnali in input o in output. Il multiplexer è un circuito che permette di selezionare, tra un insieme di input, un solo output.

Il multiplexer funziona con n segnali di controllo (\underline{x}), 2^n segnali dati (\underline{d}) e una sola uscita (y).

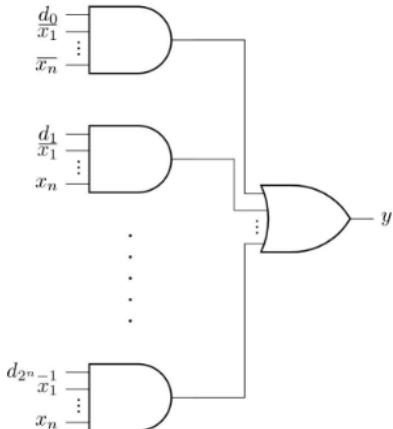


L'uscita assume il valore d_i quando $\underline{x} = i$ (\underline{x} è la conversione decimale di $x_1x_2\dots x_n$).

Possiamo quindi scrivere la funzione di uscita come somma logica tra il prodotto di tutti i mintermini di \underline{x} e i dati d :

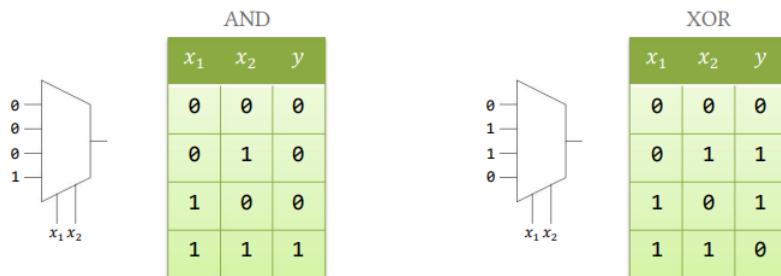
$$y = \sum_{i=0}^{2^n-1} d_i m_i$$

x	y
0	d_0
1	d_1
2	d_2
3	d_3
4	d_4
⋮	⋮
2^n-1	d_{2^n-1}



4.6.1 Multiplexer come generatore di funzioni

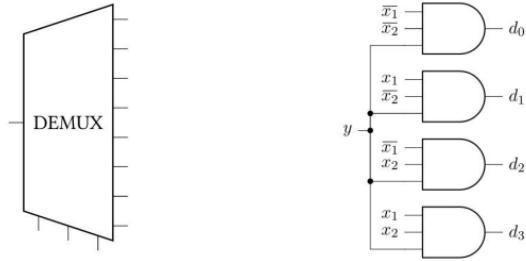
Un multiplexer può essere utilizzato per implementare qualsiasi funzione booleana di n variabili.



La rappresentazione in forma tabellare di una funzione determina per ogni configurazione dell'input il valore dell'output, un multiplexer può implementare tale tabella.

4.7 Demultiplexer

Il circuito duale del multiplexer è il demultiplexer, l'uscita i -esima assume il valore y quando la variabile di controllo $\underline{x} = i$.



Circuito simile al decodificatore.

4.8 Read Only Memory (ROM)

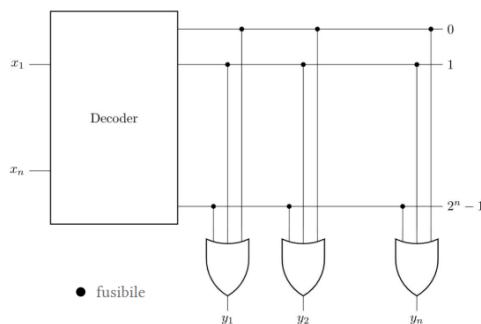
È una memoria di sola lettura, le locazioni di memoria possono essere lette specificandone l'indirizzo.



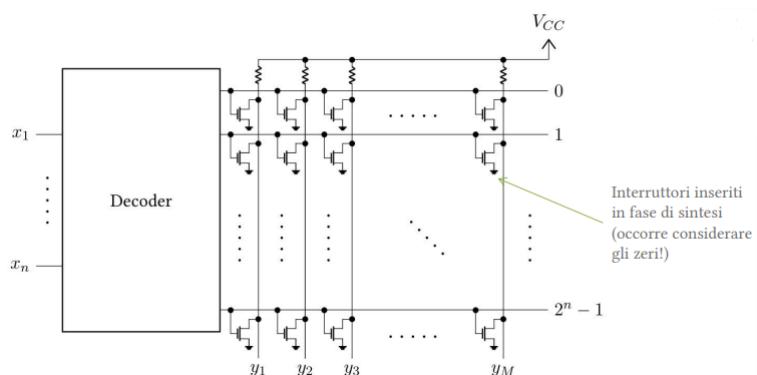
È un'implementazione alternativa di un circuito combinatorio: dato un ingresso, c'è una sola uscita.

4.8.1 Schema logico di una ROM

Le funzioni di commutazione sono realizzate come OR di mintermini.



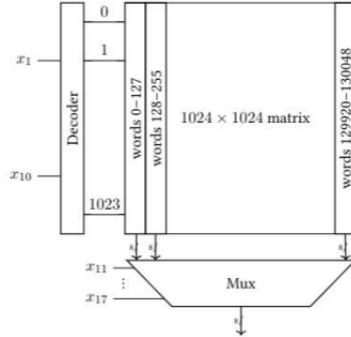
4.8.2 Implementazione di una ROM



4.8.3 ROM paginata

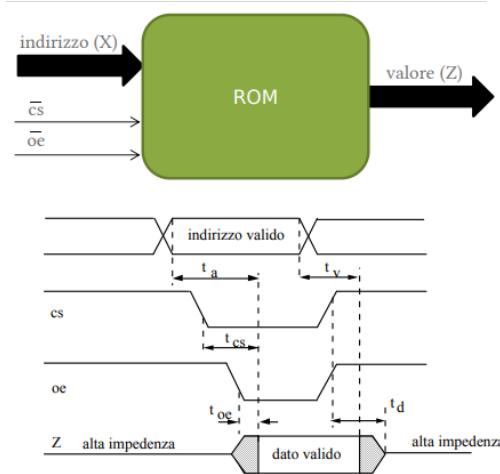
Per motivi di ottimizzazione di superficie, si cerca di realizzare le ROM in forma **quadrata**, quindi una ROM viene organizzata in "pagine".

Una parte dei bit dell'indirizzo viene usata per selezionare la pagina e la restante parte seleziona la parola all'interno della pagina.



4.8.4 Temporizzazione di una ROM

- t_a : tempo di propagazione dall'ingresso X all'uscita Z;
- t_{cs} : tempo di propagazione dall'ingresso cs all'uscita Z;
- t_{oe} : tempo di propagazione dall'ingresso oe all'uscita Z;
- t_v : tempo di mantenimento dell'uscita da quando commuta X o cs o oe ;
- t_d : tempo di disabilitazione dell'uscita da quando commuta cs o oe .

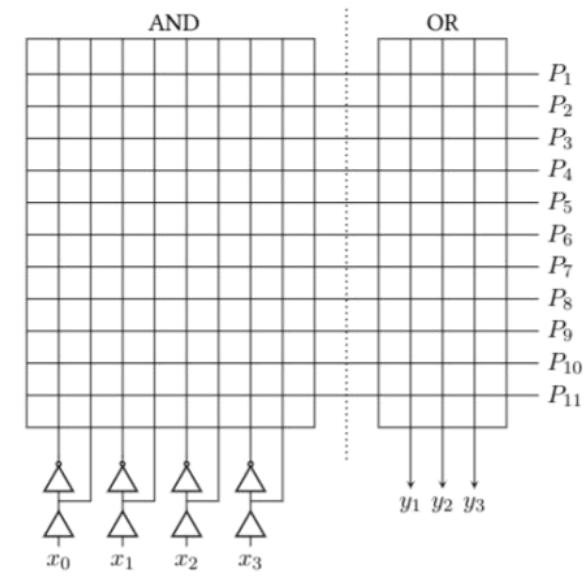


4.8.5 Programmable Logic Array

Logicamente una ROM è una matrice di AND (decoder) che implementa tutti i mintermini, accoppiata ad una matrice di OR che implementa le varie funzioni.

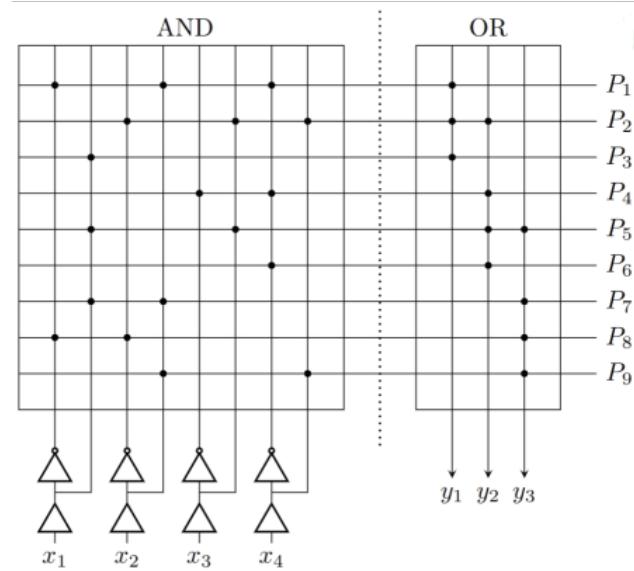
Si potrebbe rendere programmabile sia la rete AND che la rete OR.

La Programmable Logic Array (PLA) realizza questa struttura.



Esempio Implementazione funzioni booleane tramite PLA

$$\begin{aligned}y_1 &= \overline{x_1}x_2\overline{x_4} + \overline{x_2}x_3x_4 + x_1 \\y_2 &= \overline{x_2}x_3x_4 + \overline{x_3}\overline{x_4} + x_1x_3 + x_2\overline{x_4} \\y_3 &= x_1x_2 + \overline{x_1}\overline{x_2} + x_1x_3 + x_2x_4\end{aligned}$$



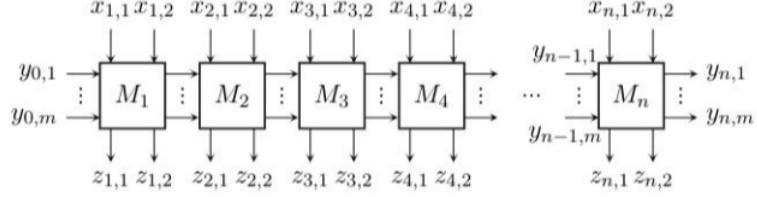
Sulle righe della parte sinistra, abbiamo tutte le configurazioni dei mintermini, sulle colonne della parte destra, ogni uscita somma uno o più mintermini.

- Nella prima riga a sinistra è rappresentato il mintermine: $\overline{x_1}x_2\overline{x_4}$.
- Nella prima colonna a sinistra vediamo che $y_1 = p_1 + p_2 + p_3$, ovvero è la somma dei primi tre mintermini (prime tre righe).

5 Reti iterative

Un circuito combinatorio realizzato da 16, 32, 64 bit può essere complesso da sintetizzare.

Possiamo organizzare i circuiti in maniera *iterativa*, significa che uno stesso circuito elementare tratta un sottoinsieme dei bit dei dati, riducendo il numero di variabili. Più circuiti elementari sono interconnessi tra loro, per calcolare la funzione finale.



- Vettore \mathbf{y} : rappresenta le informazioni di stato trasferite da un modulo al successivo, l'ultimo modulo può esporre parte di questa informazione all'esterno;
- vettore \mathbf{x} : rappresenta il dato in input, decomposto tra i vari moduli;
- vettore \mathbf{z} : rappresenta l'output \mathbf{z} , calcolato iterativamente dai moduli.

5.1 Comparatori

I comparatori sono dei circuiti che confrontano il valore di due numeri, A e B , rappresentati in formato binario. Il risultato di un comparatore determina se $A = B$.

Il confronto può essere effettuato su ciascuna coppia di bit in moduli separati, tuttavia, è necessario propagare il risultato della comparazione dei moduli precedenti.

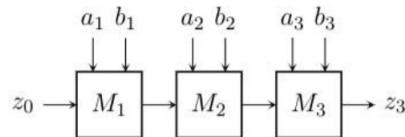
Esempio Implementazione comparatore uguaglianza

Date due parole A e B confrontiamo bit a bit a_i con b_i .

z_{i-1}	a_i	b_i	z_i
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

- La prima colonna vale 1 se la coppia di bit precedenti sono uguali;
- l'ultima colonna vale 1 se $z_{i-1} = 1$ e $a_i = b_i$.

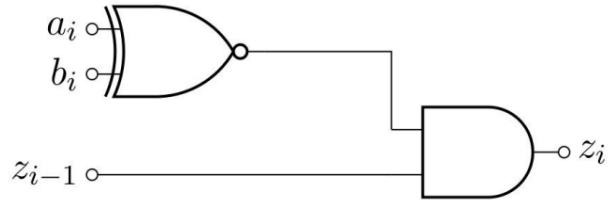
Quindi l'output è 1 se riceviamo un 1 dagli stadi precedenti (tutti i bit precedenti sono uguali) e se i due bit analizzati nel modulo corrente sono uguali.



I mintermini della funzione sono soltanto due:

$$z_i = z_{i-1}\bar{a}\bar{b} + z_{i-1}ab = z_{i-1}(a \odot b)$$

Quindi la realizzazione circuitale del modulo M_i è:

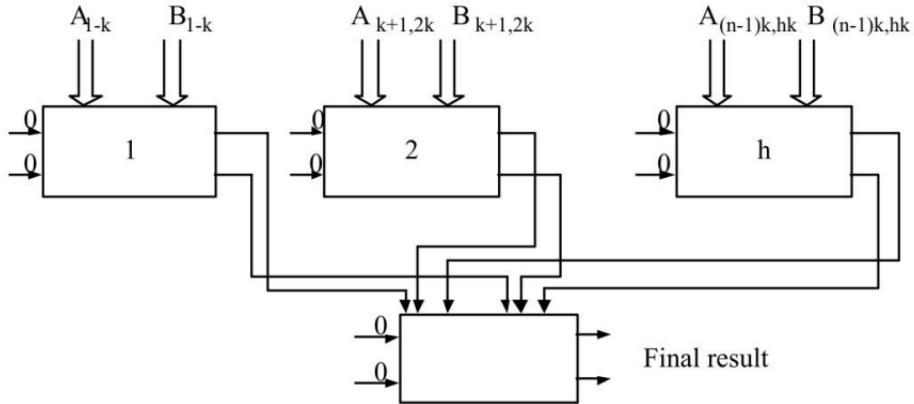


5.1.1 Problemi reti iterative

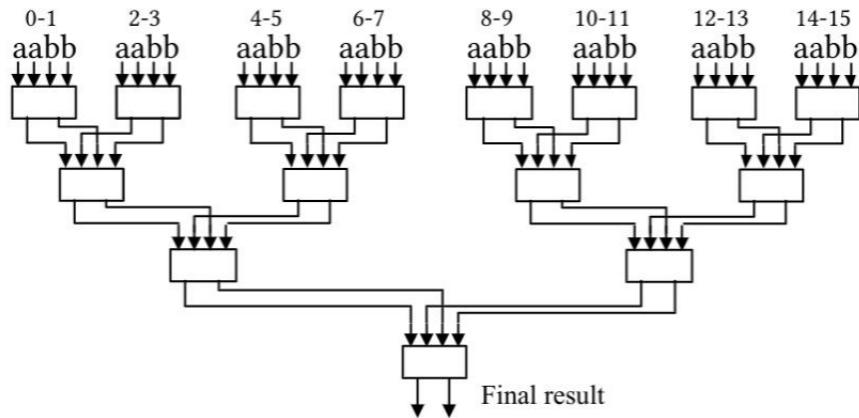
Analizzando la struttura del comparatore realizzato è evidente che il limite di queste reti è il **tempo di calcolo** della funzione, che può diventare inaccettabile se il numero di bit da processare è troppo elevato.

5.1.2 Comparatori veloci

I bit dei numeri da confrontare vengono divisi in h blocchi di k bit, ciascun blocco viene confrontato da un comparatore iterativo dedicato. Le uscite dei vari comparatori vengono processate da un comparatore aggiuntivo.



Poiché il numero di bit è tipicamente una potenza di 2, si possono organizzare i comparatori in una struttura ad albero a più livelli. Ad esempio, per interi a 16 bit:



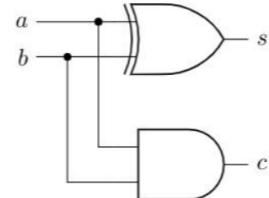
5.2 Reti iterative per la somma

5.2.1 Half adder

È il circuito più semplice per effettuare una somma di operandi ad un solo bit. Calcola il valore della somma e il valore del riporto:

$$s = a \oplus b \quad c = a \cdot b$$

a	b	s	c
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



5.2.2 Full adder

Per calcolare la somma di un intero a n bit possiamo realizzare una rete iterativa composta da n sommatori. Il circuito del modulo va modificato per considerare anche il riporto proveniente dai moduli precedenti.

c_{i-1}	a_i	b_i	s_i	c_i
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$s_i = f_1(a_i, b_i, c_{i-1})$$

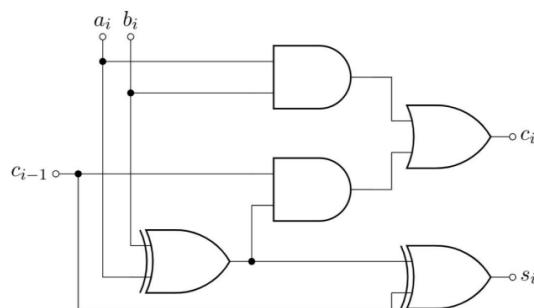
c_{i-1}	00	01	11	10
0	0	1	0	1
1	1	0	1	0

$$c = f_2(a_i, b_i, c_{i-1})$$

c_{i-1}	00	01	11	10
0	0	0	1	0
1	0	1	1	1

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

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



Circuitalmente, la porta XOR è complessa e ha un tempo di propagazione elevato. Quindi il tempo di propagazione del carry in una rete iterativa *non è accettabile*.

5.2.3 Carry Lookahead Adder

Una possibile soluzione è quella di calcolare i bit di carry in parallelo, effettuando questa decomposizione:

- $s_i = c_{i-1} \oplus a_i \oplus b_i$ e $c_i = a_i b_i + c_{i-1}(a_i \oplus b_i)$
- poniamo $G_i = a_i b_i$ e $P_i = a_i \oplus b_i$
- possiamo riscrivere l'equazione:
 - $s_i = P_i \oplus c_{i-1}$
 - $c_i = G_i + P_i c_{i-1}$

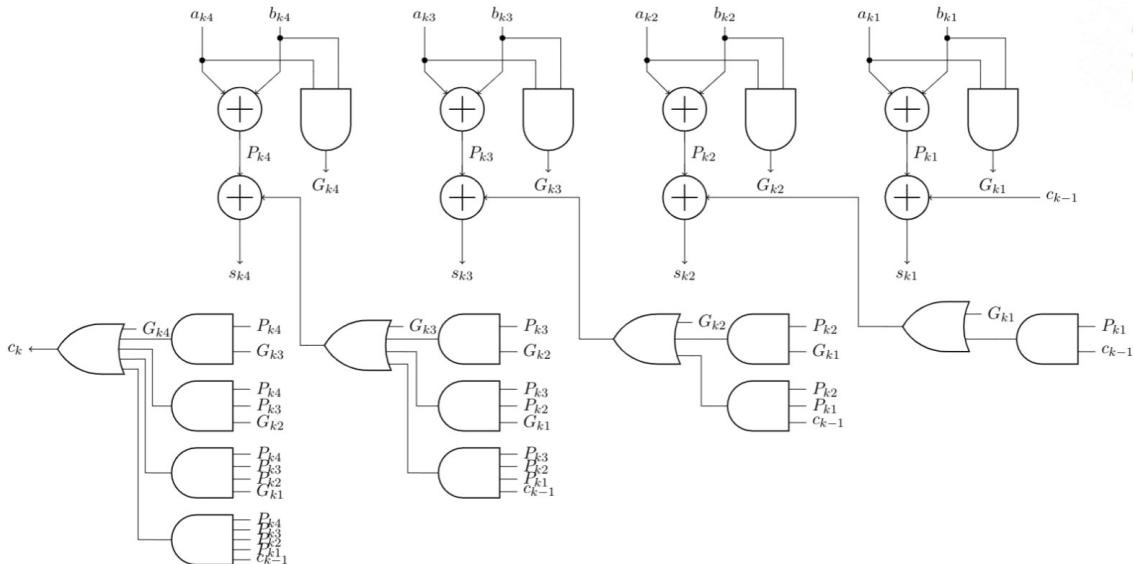
Il termine G_i viene chiamato **generatore di carry**.

Il termine P_i viene chiamato **propagatore di carry**.

Supponiamo di dividere i numeri A e B in gruppi di 4 bit, ciascun gruppo k riceverà un bit di carry dal modulo precedente $k - 1$ e ne invierà uno al modulo successivo $k + 1$.

Se riusciamo a calcolare velocemente il bit di carry c_k da inviare al modulo successivo, abbatteremo il ritardo dovuto all'organizzazione iterativa.

$$\begin{aligned} c_{k4} &= G_{k4} + P_{k4} c_{k3} = G_{k4} + P_{k4}(G_{k3} + P_{k3} c_{k2}) = G_{k4} + P_{k4}(G_{k3} + P_{k3}(G_{k2} + P_{k2} c_{k1})) = \\ &= G_{k4} + P_{k4}(G_{k3} + P_{k3}(G_{k2} + P_{k2}(G_{k1} + P_{k1} c_{k-1}))) = \\ &= G_{k4} + P_{k4}G_{k3} + P_{k4}P_{k3}G_{k2} + P_{k4}P_{k3}P_{k2}G_{k1} + P_{k4}P_{k3}P_{k2}P_{k1} c_{k-1} \end{aligned}$$



5.3 Shifter

L'operazione di *shift* consiste nel muovere a destra o a sinistra i bit di una parola binaria. Alcune cifre possono essere scartate, le posizioni lasciate libere possono essere riempite con degli zero.

Esempio, shift a destra di una posizione:

$$1101 \rightarrow 1010$$

Considerando la possibilità di spostare un bit di una sola posizione, dobbiamo prevedere i seguenti comandi per un circuito:

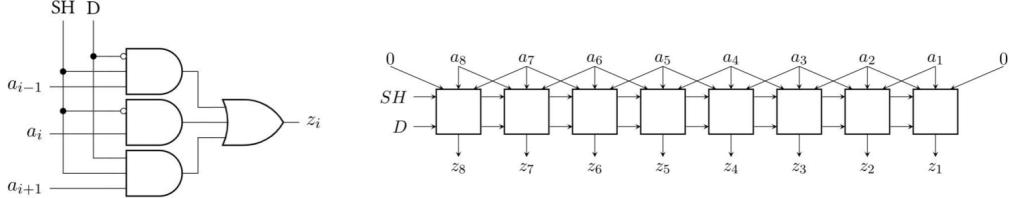
- $SH = 0$ non effettuare cambiamenti, $SH = 1$ effettua la traslazione;
- $D = 0$ trasla a sinistra, $D = 1$ trasla a destra.

Possiamo descrivere il circuito che calcola il valore del bit z_i in uscita con il seguente sistema:

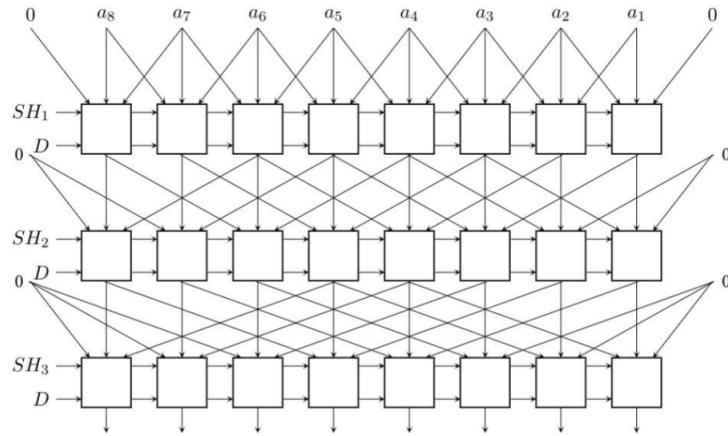
$$z_i = \begin{cases} a_i & \text{se } SH = 0 \\ a_{i-1} \cdot \overline{D} + a_{i+1} \cdot D & \text{se } SH = 1 \end{cases}$$

Pertanto l'equazione che calcola il valore del bit diventa:

$$z_i = \overline{SH} \cdot a_i + SH \cdot (a_{i-1} \cdot \overline{D} + a_{i+1} \cdot D)$$



Per supportare una traslazione di più posizioni, possiamo costruire una rete iterativa di shifter. Utilizzando un vettore di controllo SH possiamo controllare i vari livelli.

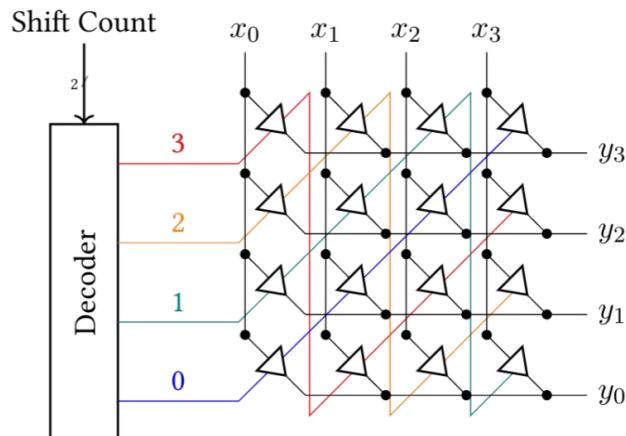


Il costo di questa rete può essere molto elevato.

5.3.1 Barrel Shifter

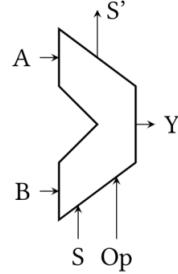
È una rete basata su interruttori posti in posizioni predefinite.

Attivando gli interruttori corretti, è possibile implementare in tempo costante tutte le traslazioni a destra e a sinistra (considerando che in una parola di 4 bit, uno shift a sinistra di 3 è uguale a uno shift a destra di 1).



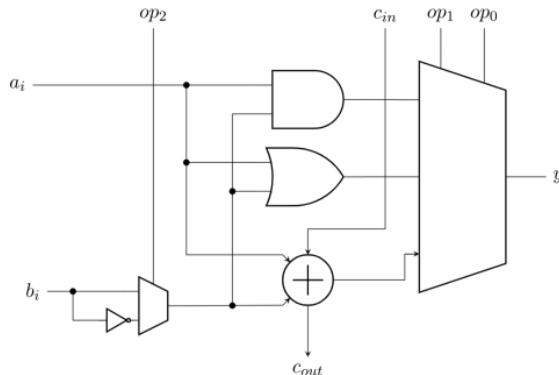
5.4 Unità Logico Aritmetica (ALU)

Molte operazioni operano su parole binarie di dimensione fissa. Per **risparmiare spazio**, è possibile accoppare le varie operazioni in un singolo circuito iterativo, chiamato ALU. La struttura delle reti iterative è la stessa, cambia soltanto la logica per implementare l'operazione.



Mediante alcuni segnali di controllo (Op), si specifica qual'è l'operazione che si vuole eseguire sulle parole in input.

Esempio Un modulo come quello in figura permette di implementare varie operazioni, a seconda dei bit di controllo forniti in input.



op_2	op_1	op_0	c_{in}	y
0	0	0	-	$a \cdot b$
0	0	1	-	$a + b$
0	1	0	-	$a \oplus b \oplus c_{in}$
1	1	0	0	$a + \bar{b}$
1	1	0	1	$a - b$ (complemento a 2)

Tramite la configurazione di $\langle op_1 \, op_0 \rangle$ possiamo scegliere l'ingresso che il multiplexer seleziona, ogni ingresso del multiplexer corrisponde ad un'operazione.

Tramite op_2 possiamo decidere se negare o meno b_i . Complessivamente i bit di configurazione sono $\langle op_2 \, op_1 \, op_0 \rangle$.

Per effettuare la sottrazione $(a_i - b_i)$ in complemento a due effettuiamo la somma con b_i negativo in complemento a due $(a_i + (-b_i))$. Per rappresentare b_i in complemento a due lo neghiamo ($op_2 = 1$) e sommiamo 1 grazie al carry c_{in} .

6 Reti sequenziali

I circuiti visti fino ad ora possono essere definiti combinatori. Dato un input \underline{x} , essi calcolano un output \underline{y} secondo la relazione:

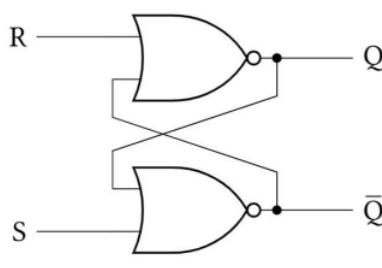
$$\underline{y} = f(\underline{x})$$

Quindi questo tipo di circuiti hanno un'uscita che dipende esclusivamente dall'input.

Con a disposizione solo questo tipo di circuito, non potremmo implementare elementi di memoria (dato che cambiano stato in base all'input).

6.1 Circuito Latch

Un circuito latch (*lucchetto*) "cattura" il valore di input utilizzando degli anelli a *retroazione*. È un circuito che consente di immagazzinare un'informazione digitale, ha due ingressi S (set) e R (reset).



S	R	Q	Q'
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	indeterminato
1	1	1	indeterminato

Quindi nei circuiti latch:

- $R=0, S=0$: mantiene lo stato precedente;
- $R=1, S=0$: azzerà l'uscita Q;
- $R=0, S=1$: imposta lo stato logico 1 sull'uscita Q.

6.1.1 Problema del latch

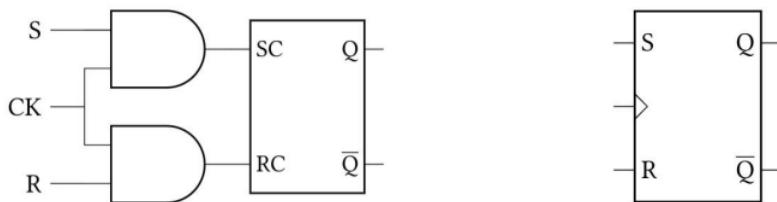
Una configurazione in ingresso in cui $S = 1$ e $R = 1$ manda il circuito in uno stato oscillante. Dato che i valori di S e R possono essere calcolati da altre reti combinatorie, non è possibile garantire che, a causa dei ritardi di propagazione, non si osservi mai una configurazione transiente pari a $S = 1$ e $R = 1$.

Per evitare questo problema, possiamo campionare il valore di S e R quando siamo certi che gli input si sono stabilizzati. In questo caso il circuito assume il nome di *flip flop*.

6.2 Circuiti Flip Flop

6.2.1 Flip Flop SR

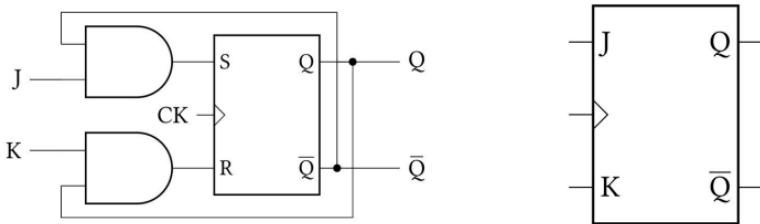
Si basa sull'aggiunta di un segnale di *clock* al latch.



È ancora possibile che gli input vengano impostati a $S = 1$ e $R = 1$.

6.2.2 Flip Flop JK

Il flip flop JK aggiunge una rete di controllo ai segnali in ingresso, tale rete rende impossibile che si verifichi la condizione $S = 1$ e $R = 1$ in input al flip flop SR interno.



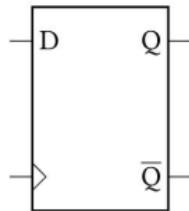
J	K	Q	\bar{Q}
0	0	nc	nc
1	0	0	1
0	1	1	0
1	1	\bar{Q}	Q

Dove "nc" sta per "nessun cambiamento". La condizione di input n cui $S = 1$ e $R = 1$ inverte lo stato logico delle uscite.

6.2.3 Flip Flop D

I flip flop visti fino ad ora hanno la necessità di due segnali di controllo *opposti*.

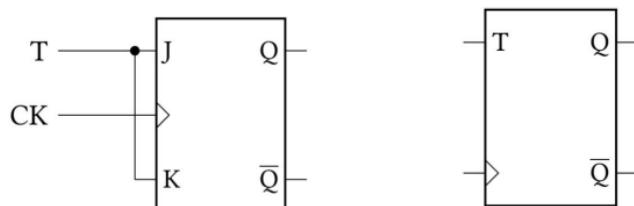
Il flip flop D si comporta come ritardo, l'input viene propagato in output dopo un periodo di clock.



6.2.4 Flip Flop T

È un flip flop che si comporta come switch:

- al primo segnale, commuta da 0 a 1;
- al secondo segnale, commuta da 1 a 0.



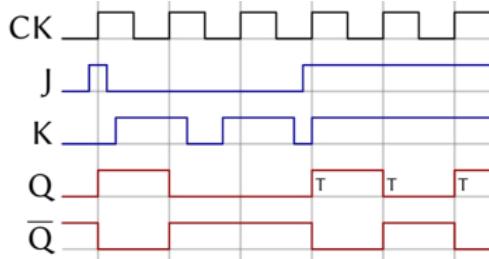
T	Q	\bar{Q}
0	Q	Q
1	\bar{Q}	Q

6.3 Fronte di commutazione

Per funzionare correttamente, questi flip flop devono avere i segnali di controllo stabili per tutta la durata del periodo di clock.

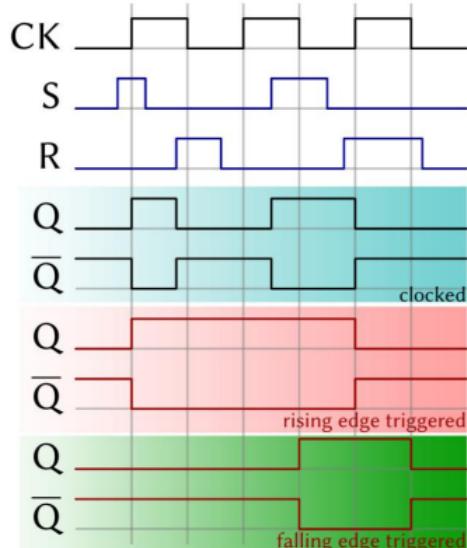
È utile prevedere circuiti che effettuino la commutazione in finestre temporali precisi e di durata più breve.

Edge-triggered flip flop: commutano sul fronte di salita o discesa.



Rising Edge-Triggered JK Flip-Flop.

Il comportamento dello stesso flip flop può essere estremamente differente, in alcuni casi, alcuni segnali potrebbero essere ignorati.

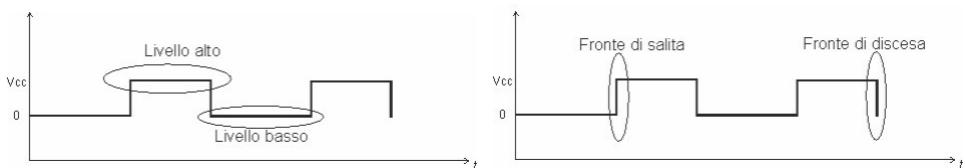


In figura sono riportati i differenti comportamenti per un flip flop SR.

6.4 Clock

Il segnale di clock è generato da un circuito (realizzato con un opportuno cristallo) che emette un segnale impulsivo periodico con una precisa durata (pulse width) e con un preciso intervallo tra due impulsi consecutivi.

Il clock è un segnale free-running ossia che continua indefinitamente (almeno finché il sistema e' alimentato), di tipo periodico, con un periodo detto *tempo di clock* Tck (clock cycle time); il suo reciproco è la *frequenza di clock* fck o f.



6.5 Reti sequenziali

I circuiti che implementano i flip flop sono semplici reti sequenziali. Una rete sequenziale ha un funzionamento che evolve nel tempo, quindi è necessario definire uno stato interno per descrivere l'evoluzione nel tempo.

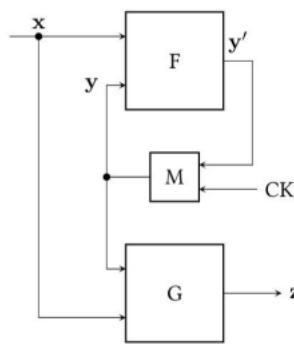
L'equazione non sarà più una ($y = f(x)$) ma due:

$$\begin{cases} \underline{y}' = f(\underline{x}, \underline{y}) \\ \underline{z} = g(\underline{x}, \underline{y}) \end{cases}$$

Esistono due classi principali di reti sequenziali:

- *sincrone*: se la transizione di stato avviene in istanti temporali controllabili dall'esterno;
- *asincrone*: se le transizioni di stato non sono controllate esternamente

Studieremo esclusivamente le prime, che sono schematizzate così:



Questo è il diagramma a blocchi delle due funzioni viste prima, dove:

- F e G sono due circuiti combinatori;
- M è un modulo di memoria formato da flip flop;
- scriviamo e leggiamo dati sul fronte del clock.

6.6 Macchine a stati finiti

Una macchina a stati finiti è un modello matematico per la descrizione della computazione. È una macchina astratta che rispetta queste regole:

- in un dato istante si trova in un solo stato;
- eventi esterni provocano una transizione da uno stato ad un altro;
- la macchina può generare output verso l'esterno.

Esistono due varianti principali di macchina a stati finiti:

- macchina di Moore: l'output dipende unicamente dallo stato;
- macchina di Mealy: l'output dipende dallo stato e dalla transizione innescata.

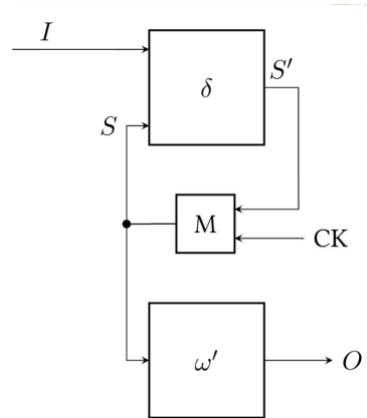
Le due macchine differiscono solo in come viene generato l'output. A seconda del progetto e della descrizione a parole si può decidere di realizzare un automa di Moore o di Mealy. Di solito l'automa di Mealy ha meno stati (quindi meno elementi di memoria) ma ha le reti combinatorie ed in particolare la rete combinatoria delle uscite più complessa e quindi potenzialmente più lenta. Spesso si realizza l'automa di Moore perché è concettualmente più semplice.

6.6.1 Macchina di Moore

Una rete sequenziale è definita dalla sestupla

$$M = \langle I, S, s_0, O, \delta, \omega \rangle$$

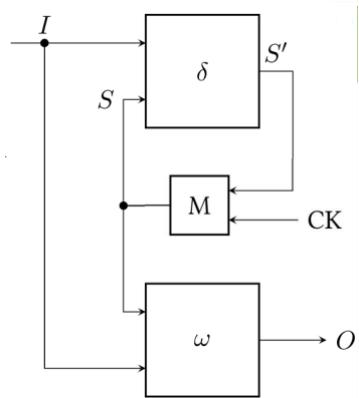
- $I = \{i_1, i_2, \dots, i_p\}$: alfabeto di ingresso;
- $S = \{s_1, s_2, \dots, s_n\}$: insieme degli stati interni
- $s_0 \in S$: stato iniziale;
- $O = \{o_1, o_2, \dots, o_r\}$: alfabeto di uscita;
- $\delta : I \times S \mapsto S$: funzione di transizione;
- $\omega : S \mapsto O$: funzione che calcola l'output.



6.6.2 Macchina di Mealy

$$M = \langle I, S, s_0, O, \delta, \omega \rangle$$

- $I = \{i_1, i_2, \dots, i_p\}$: alfabeto di ingresso;
- $S = \{s_1, s_2, \dots, s_n\}$: insieme degli stati interni
- $s_0 \in S$: stato iniziale;
- $O = \{o_1, o_2, \dots, o_r\}$: alfabeto di uscita;
- $\delta : I \times S \mapsto S$: funzione di transizione;
- $\omega : I \times S \mapsto O$: funzione che calcola l'output.



La macchina di Mealy, a differenza di quella di Moore, deve sapere sia lo stato corrente che l'input per passare allo stato successivo.

6.6.3 Rappresentazioni macchine di Moore e Mealy

Per rappresentare una macchina a stati è possibile utilizzare due formalismi:

- *diagramma degli stati*: è un grafo che mostra graficamente le relazioni tra gli stati e le transizioni, identificando anche i caratteri di output.
- *tavella degli stati e delle transizioni*: è una rappresentazione equivalente in forma tabellare.

Diagramma degli stati:

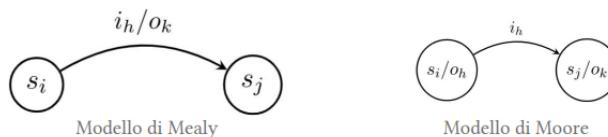


Tabella degli stati e delle transizioni:

	i_1	i_2	\dots	i_j	\dots	i_p
s_1						
s_2						
\vdots						
s_i				$\delta(i_i, s_i)/\omega(i_j, s_i)$		
\vdots						
s_n						

Modello di Mealy

	i_1	i_2	\dots	i_j	\dots	i_p	ω'
s_1							
s_2							
\vdots							
s_i					$\delta(i_i, s_i)$		
\vdots							
s_n							

Modello di Moore

6.6.4 Equivalenza tra modelli

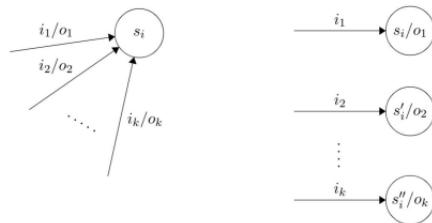
Esiste sempre una trasformazione tra una macchina di Mealy e una macchina di Moore, quindi i due modelli sono equivalenti.

- Trasformazione da Moore a Mealy:

- gli alfabeti di input e output sono gli stessi;
- gli stati sono gli stessi;
- se in uno stato s_i raggiunto da una transizione causata da un carattere i_j viene generato un output o_k , quello stesso output viene generato durante la transizione i_j verso lo stato s_i .

- Trasformazione da Mealy a Moore:

- questa trasformazione è meno immediata;
- possiamo avere più transizioni verso lo stesso stato che generano output differenti;
- in questo caso, lo stato di destinazione deve essere decomposto in più stati differenti;
- stati Mealy \leq stati Moore.



6.6.5 Sintesi delle macchine

Per realizzare circuitualmente una macchina è necessario:

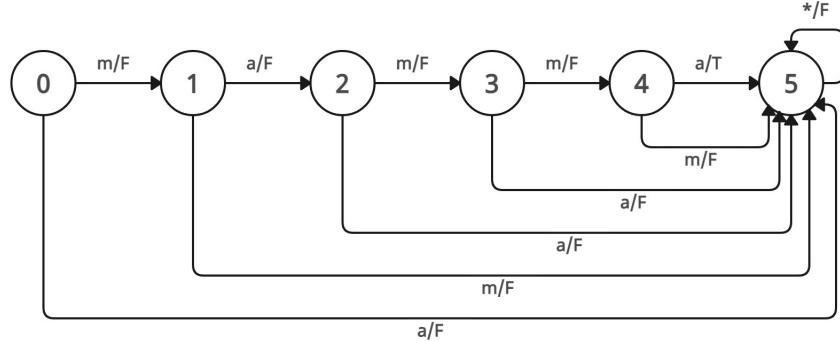
- realizzare il blocco M: lo facciamo utilizzando un numero sufficiente di flip flop D;
- realizzare la rete δ : è necessario realizzare un circuito di commutazione per aggiornare lo stato di ciascun flip flop D (*equazione di eccitazione di un flip flop*);
- realizzare la rete ω : è necessario realizzare un circuito di commutazione per generare ciascun bit dei caratteri di output.

Trattandosi di reti combinatorie, è possibile utilizzare le tecniche di sintesi e minimizzazione (Karnaugh, analiticamente) che abbiamo studiato per le funzioni booleane.

La sintesi può essere svolta a partire dalla tabella degli stati e transizioni.

Esempio Sintesi della macchina che accetta la stringa "mamma" in input.

- Diagramma degli stati:



- Alfabeto di ingresso: $I = \{m, a\}$ (due caratteri che compongono la stringa "mamma").
- Alfabeto di uscita: $O = \{F, T\}$ (False, True).
- Insieme degli stati: $S = \{0, 1, 2, 3, 4, 5\}$.

S	y_0	y_1	y_2
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1

I	x_0
m	0
a	1

O	z_0
F	0
T	1

- Tabella degli stati e delle transizioni:

y_0	y_1	y_2	x_0	y'_0	y'_1	y'_2	z_0
0	0	0	0	0	0	1	0
0	0	0	1	1	0	1	0
0	0	1	0	1	0	1	0
0	0	1	1	0	1	0	0
0	1	0	0	0	1	1	0
0	1	0	1	1	0	1	0
0	1	1	0	1	0	0	0
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	0
1	0	0	1	1	0	1	1
1	0	1	0	1	0	1	0
1	0	1	1	1	0	1	0

- Le prime tre colonne rappresentano lo stato corrente della macchina;
- la quarta colonna rappresenta l'ingresso;
- la quinta, sesta, settima colonna rappresentano lo stato successivo della macchina in base allo stato precedente e l'input;
- l'ultima colonna rappresenta l'uscita.

7 z64: Processing Unit (PU)

7.1 Instruction Set Architecture (ISA)

L'ISA di un processore definisce l'insieme di istruzioni, tipi di dato e registri che è possibile utilizzare per scrivere programmi in assembly.

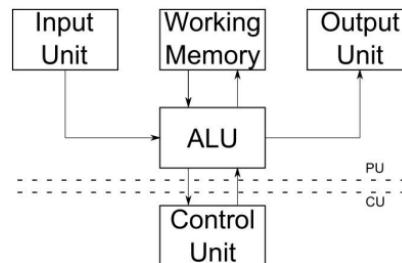
Le ISA sono, in generale, classificabili in due categorie:

- Reduced Instruction Set Computers (**RISC**):
 - comprende un piccolo insieme di istruzioni;
 - richiede programmi più lunghi;
 - più efficiente, sia nelle performance che nel consumo energetico;
 - è più semplice da progettare.
- Complex Instruction Set Computers (**CISC**):
 - comprende una grande quantità di istruzioni;
 - programmi più compatti;
 - più lento e più energivoro;
 - è di più complessa progettazione.

7.2 Architettura di von Neumann rivisitata

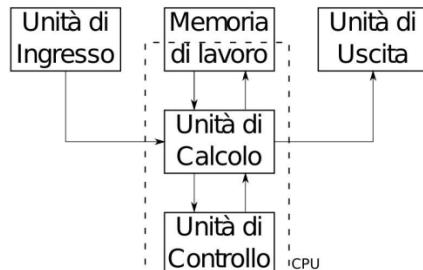
Inizialmente, abbiamo suddiviso logicamente l'architettura in due blocchi:

- Unita di calcolo (PU): tutto ciò che esegue il lavoro;
- Unità di controllo (CU): la parte intelligente.



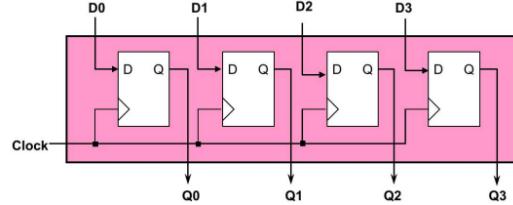
In realtà, all'interno della nostra CPU troviamo:

- l'unità di controllo;
- l'unità di calcolo (circuito combinatorio);
- *parte* della memoria di lavoro (registri).



7.3 Registri

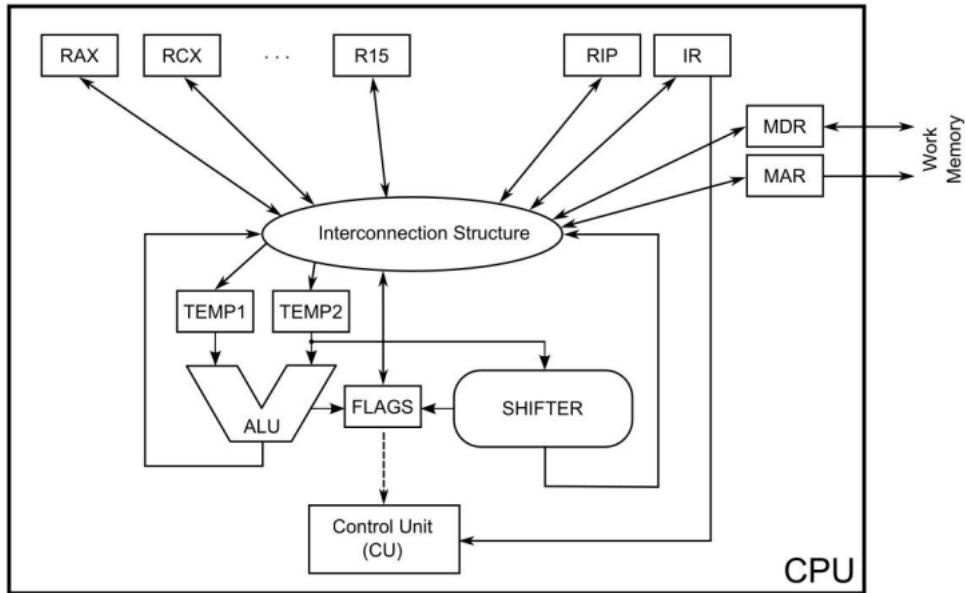
I registri sono delle unità di memoria interne al processore, compongono la parte di memoria di lavoro interna alla CPU. Permettono di memorizzare parole binarie. I registri sono realizzati a partire da flip flop, la quantità di memoria disponibile è estremamente limitata.



I registri di una CPU possono essere suddivisi in classi principali:

- *Registri principali*: sono i registri senza i quali non è possibile realizzare un'architettura di von Neumann.
- *Registri visibili al programmatore*: sono registri che il programmatore può utilizzare esplicitamente nel suo programma.
- *Registri invisibili al programmatore*: sono registri che il programmatore può modificare solo indirettamente e non programmaticamente.

7.4 z64: architettura di alto livello



7.5 Processamento delle istruzioni

Le istruzioni sono contenute in memoria di lavoro in forma binaria (codice macchina), il loro processamento (*ciclo istruzione*) prevede tre fasi:

- **fetch**: prelievo dell'istruzione, in rappresentazione binaria, dalla memoria di lavoro;
- **decodifica**: l'istruzione prelevata viene interpretata per capire come questa dovrà essere eseguita;
- **esecuzione**: la semantica dell'istruzione viene effettivamente implementata.

7.6 Modelli di esecuzione

Esistono due principali modelli che possiamo utilizzare per realizzare l'unità di calcolo:

- Modello **uniciclo**:

- un'istruzione viene eseguita in un solo colpo di clock;
- il periodo di clock deve essere tarato per garantire che tutta la rete si stabilizzi (clock dipende dal critical path);
- non è possibile la condivisione delle componenti, è necessario prevedere componenti dedicate. Per esempio, se devo eseguire due somme differenti per eseguire un'istruzione, avrò bisogno di due ALU.

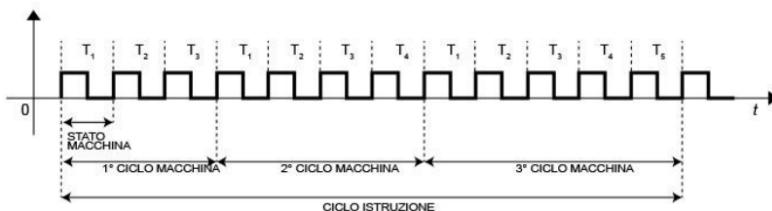
- Modello **multiciclo**:

- un'istruzione è eseguita in più colpi di clock;
- le componenti possono essere riutilizzate tra un ciclo ed il successivo;
- la fase di esecuzione è spezzata in più fasi.

uniciclo		multiciclo	
PRO	CONTRO	PRO	CONTRO
più semplice	lento più HW necessario	più complesso	veloce HW riutilizzabile

7.7 Ciclo istruzione, ciclo macchina, stato macchina

- **Ciclo istruzione**: intervallo temporale necessario ad eseguire una istruzione nella sua interezza.
- **Ciclo macchina**: intervallo temporale necessario ad eseguire una fase (fetch, decode, execute). Possono essere necessario un numero diverso di cicli macchina a seconda dell'istruzione.
- **Stato macchina**: periodo di tempo necessario per stabilizzare la rete dell'unità di calcolo (corrispondente al clock).



Esempio Esecuzione di un'istruzione.

- Consideriamo l'istruzione di alto livello $a = a + b$.
- La semantica dell'istruzione è: memorizza nella variabile di nome a la somma dei valori contenuti nelle variabili di nome a e b .
- L'esecuzione dovrà portare a questo cambiamento di stato:

$a:$ <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>15</td></tr><tr><td>9</td></tr></table> $b:$ <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>9</td></tr></table>	15	9	9	$\xrightarrow{a=a+b}$	$a:$ <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>24</td></tr><tr><td>9</td></tr></table> $b:$ <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>9</td></tr></table>	24	9	9
15								
9								
9								
24								
9								
9								
<i>Prima</i>		<i>Dopo</i>						

Per eseguire questa istruzione è necessario:

1. stabilire dove sono memorizzati i valori da sommare;
2. decidere quale operazione svolgere;
3. stabilire dove va scritto il risultato dell'operazione.

Stabiliamo questi tre punti per eseguire un istruzione.

1. Nello z64, gli operandi sono memorizzati nei registri interni alla CPU (registri di uso generale) o in memoria.
2. Il formato dell'istruzione (nella sintassi AT&T) è:

MOV_s <sorgente>, <destinazione>

dove s può assumere 4 valori:

- B: operandi a 8 bit;
- W: operandi a 16 bit;
- L: operandi a 32 bit;
- Q: operandi a 64 bit.

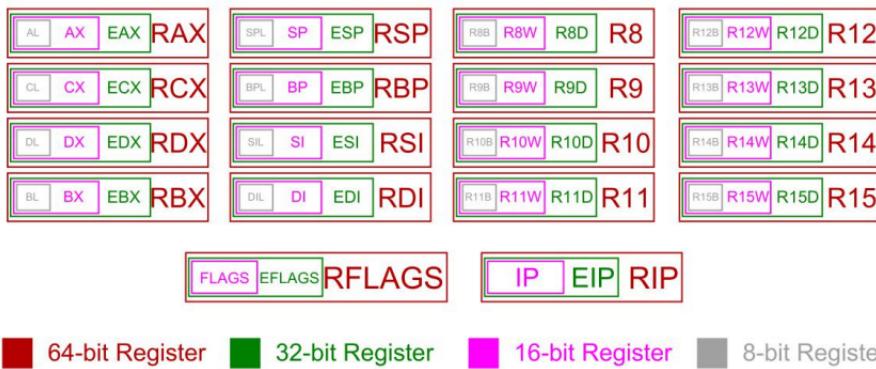
Il nome dei registri varia di conseguenza.

3. Il campo destinazione è un registro che contiene il valore iniziale di un operando e sarà modificato:

MOVW %ax, %bx

7.8 Registri fisici e registri virtuali

I registri di uso generale (*general purpose*) visibili al programmatore e alcuni dei registri fondamentali sono i seguenti (ogni registro è grande 64 bit):

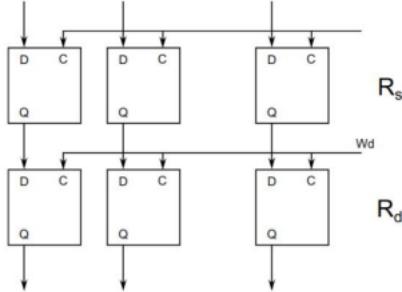


- RFLAGS: contiene i bit di stato;
- RIP: tiene conto dell'ultima operazione eseguita.

7.9 Interconnessione tra registri

Per effettuare operazioni e memorizzare dati, i registri devono essere connessi tra loro.

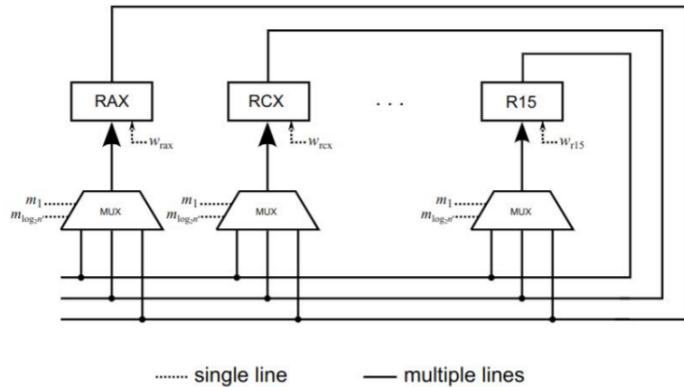
7.9.1 Interconnessione diretta



Tutti i registri sono collegati tra loro. Quindi da ogni registro escono 16 canali da 64 fili.

PRO	CONTRO
semplice da implementare	difficoltà se è necessario interconnettere più registri tra loro (troppi fili)

7.9.2 Interconnessione tramite multiplexer



Tutti i registri sono collegati tra loro tramite un solo canale (composto da 64 fili).

Per sovrascrivere un registro:

- il multiplexer seleziona l'ingresso corrispondente al registro che contiene i dati che vogliamo inserire nel nuovo registro;
- l'unità di controllo deve impostare $w_{reg} = 1$, dove per w intendiamo *write enable*. Possiamo scrivere in un registro solo se $w_{reg} = 1$.

Abbiamo bisogno di w messo in and con il clock per selezionare il registro in cui vogliamo scrivere, dato che i fili portano le stesse informazioni a tutti i registri.

PRO	CONTRO
semplice da implementare	costo eccessivo dei multiplexer e delle linee di interconnessione
possibilità di trasferire più dati contemporaneamente	

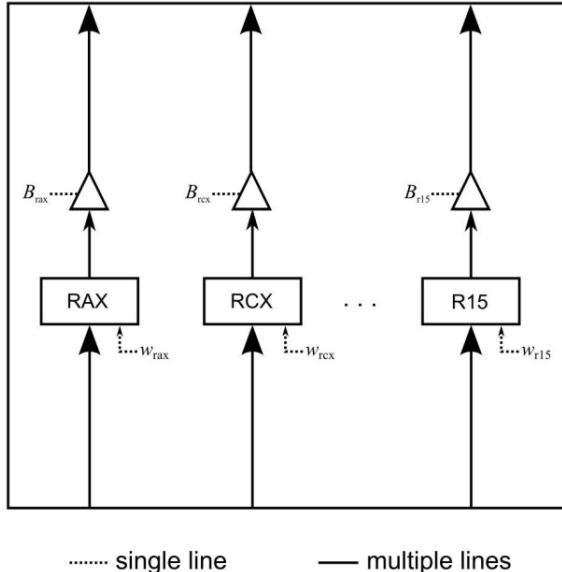
7.9.3 Interconnessione tramite BUS interno

Nel caso precedente, da ogni registro usciva un filo, in questo caso abbiamo un unico filo, chiamato BUS, che connette in lettura e scrittura tutti i registri.

Il BUS è caratterizzato da due operazioni:

- **ricezione dati:** i bit presenti sul BUS sono memorizzati in un registro;
- **trasmissione dati:** il contenuto di un registro è posto sul BUS.

Sul BUS può scrivere al massimo un registro alla volta. Utilizziamo dei *segnali di controllo* opportunamente generati: *write enable* (W_{reg}) e *buffer three-state* (B_{reg}).



Si può avere una sola scrittura per volta sul BUS, pertanto, se si hanno n registri, si dovranno prevedere $2n$ segnali di controllo.

- $W_{reg} = 1$: si può leggere dal BUS;
- $B_{reg} = 1$: si può scrivere sul BUS.

7.10 Instruction Pointer (RIP)

Nell'architettura di von Neumann sia i dati che i programmi si trovano nella memoria di lavoro, per conoscere l'**indirizzo della prossima istruzione** da eseguire, il processore utilizza un registro fondamentale: l'*Instruction Pointer* (in alcune architetture è chiamato Program Counter (PC)).

Ogni volta che un'istruzione viene eseguita, RIP viene aggiornato automaticamente per puntare a quella successiva. Quindi le istruzioni devono essere *contigue* in memoria, anche se il flusso di esecuzione può non seguire il loro ordine in memoria, a causa di salti.

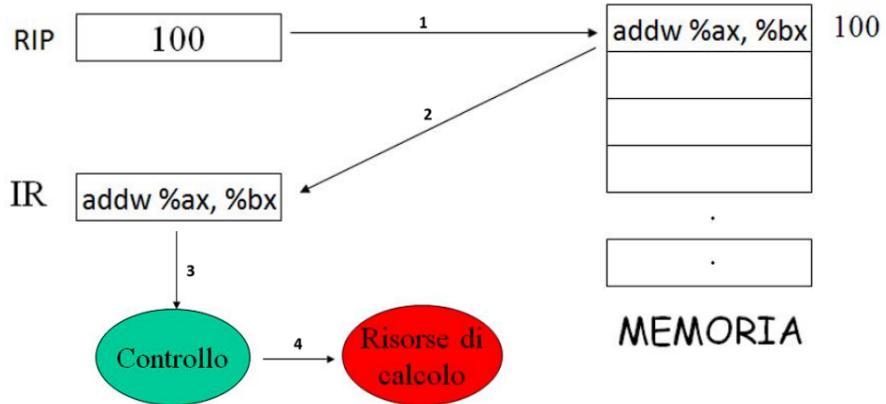
7.11 Instruction Register (IR)

L'IR è un registro fondamentale in cui un **istruzione viene copiata prima di essere eseguita**.

Questo perché l'esecuzione di un programma è divisa in due fasi: decodifica e esecuzione. La fase di esecuzione può richiedere un numero variabile di cicli macchina, in cui può essere necessario accedere nuovamente in memoria. Dato che accedere alla memoria è un'operazione costosa, copiamo l'istruzione nel Instruction Register. Quindi le fasi diventano tre: fetch (copia), decode, execute.

- I **registri** dei processori sono realizzati con tecnologia molto più **veloce**, ma più costosa.
- La **memoria**, avendo dimensione più grande, è realizzata con tecnologie più economiche ma più **lenta**.

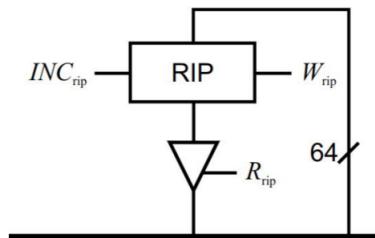
Esempio Esecuzione di un'istruzione



1. RIP punta alla prossima istruzione da eseguire;
2. durante la fase di fetch, l'istruzione viene copiata dalla memoria di lavoro nell'IR;
3. durante la fase di decodifica, l'istruzione viene interpretata per identificare il micropogramma che la possa eseguire;
4. la CU pilota la PU per implementare la semantica dell'istruzione (in questo caso $W_{RAX} = 1$ e $B_{RBX} = 1$).

7.12 Incremento del registro RIP

Il registro RIP viene incrementato dopo l'esecuzione di ogni istruzione, quindi, per ottimizzare le prestazioni, è possibile realizzarlo come *registro a incremento*. Il registro è accoppiato ad un sommatore veloce dedicato. Quindi il procedimento di incremento sarà più veloce ma servirà un circuito più complesso.



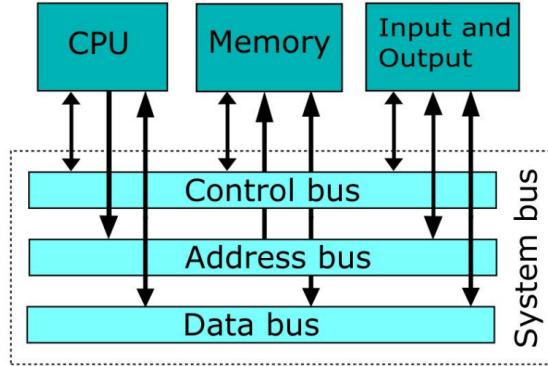
Il sommatore comincia ad incrementare dal terzo bit meno significativo, facendo così sommiamo 8 al contenuto di RIP (passiamo alla cella di memoria successiva).

Il RIP punta all'istruzione da eseguire solo nella fase di fetch, subito dopo che l'istruzione è stata copiata nel IR il RIP punta all'istruzione successiva.

7.13 Modello di memoria

- La memoria è un nastro molto lungo diviso in celle (modello di memoria piatta).
- Ciascuna cella è identificata da un numero intero (*indirizzo*).
- Ogni cella ha una dimensione prefissata (1 byte).
- Una "testina virtuale" si muove sul nastro per leggere o scrivere dati da/sulle celle (può essere effettuata solo una delle due operazioni alla volta).
- Una cella viene necessariamente scritta nella sua interezza.

7.14 Trasferimento dati: il BUS



L'Address bus nel x86 ha 48 bit e non 64.

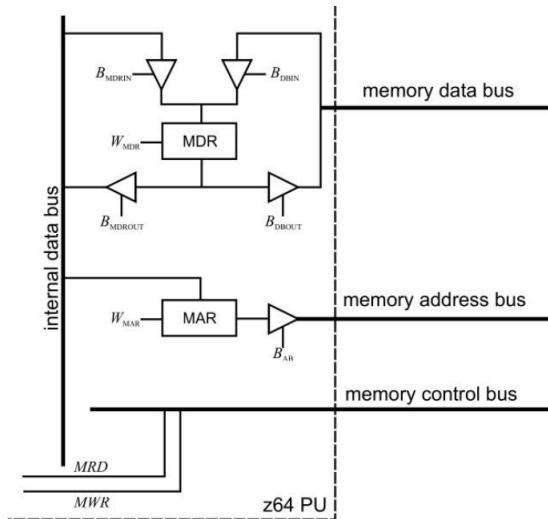
7.15 Interazione con la memoria (MAR, MDR)

La memoria contiene sia dati che le istruzioni, può essere sia scritta sia letta.

È necessario quindi poter:

- prelevare istruzioni;
- leggere i dati;
- scrivere dati.

Le parole ricevute dalla memoria devono essere instradate correttamente verso i registri di interesse (sia nel caso di dati sia nel caso di istruzioni). I dati da scrivere in memoria devono essere letti dal registro corretto.



- Internal data bus è il bus interno (7.9.3), collega i registri tra loro.
- Memory data bus, memory address bus e memory control bus formano il System BUS (7.14).

Il memory address register (**MAR**) è un registro (a 48 bit) della CPU contenente l'indirizzo della locazione di memoria RAM in cui si andrà a leggere o scrivere un dato. In altre parole, il MAR contiene l'indirizzo di memoria del dato a cui la CPU dovrà accedere.

Il Memory Data Register (**MDR**) è un registro (a 64 bit) a cui l'unità logica aritmetica ha accesso diretto e che contiene momentaneamente i dati da/per la CPU.

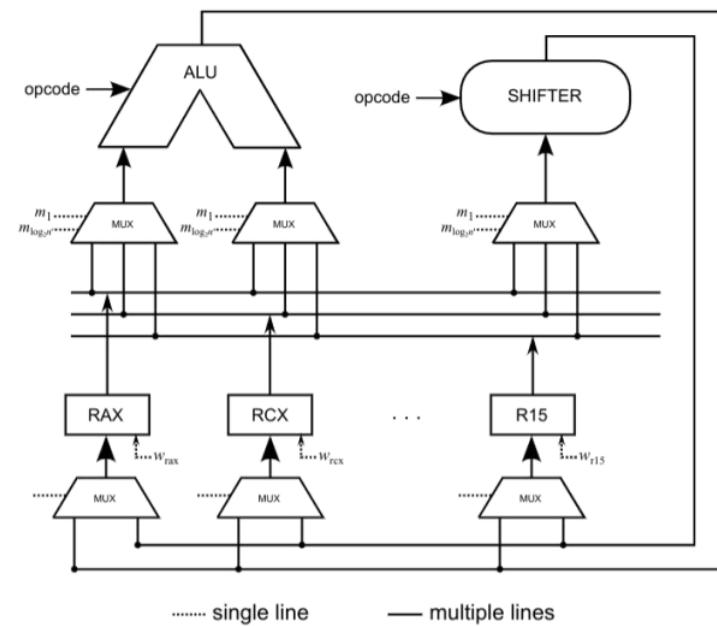
MDR e MAR, interfacciano quindi la CPU con la memoria centrale utilizzando i microprogrammi.

7.16 Interconnessione tra registri e circuiti di calcolo

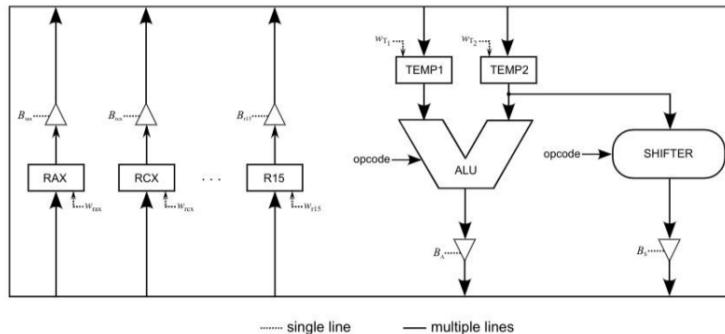
Le operazioni sono realizzate dalla ALU e dallo Shifter, che ricevono gli input dai registri.

Per connettere i registri ai componenti di calcolo possiamo utilizzare un insieme di multiplexer.

Questa implementazione ha due problemi: è molto costosa, possiamo leggere o scrivere un solo registro alla volta.



Una soluzione più economica ma più lenta prevede la possibilità di usare dei registri tampone:



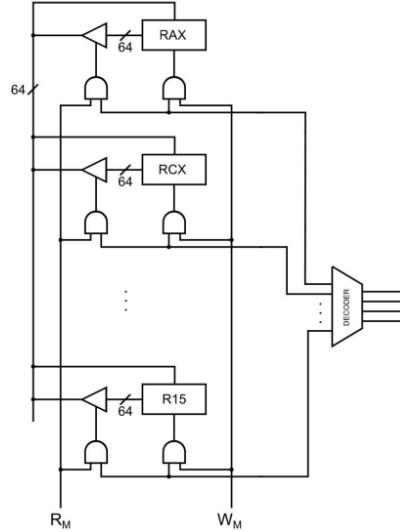
- TEMP1 e TEMP2 sono i registri tampone, collegati al data BUS interno;
- dato che possiamo scrivere o leggere solo un dato alla volta, salviamo il primo operando in TEMP1 e dopo salviamo il secondo operando in TEMP2. Usiamo i segnali di controllo B_{reg} per scrivere sul BUS e i segnali W_{T_i} per scrivere sui registri tampone;
- nel tempo in cui TEMP1 è stato già sovrascritto ma TEMP2 ancora no, la ALU calcola dei risultati sbagliati (non conosce il secondo operando), per ignorare i risultati errati basta impostare $B_A = 0$;
- $B_A = 1$ solo quando nei registri tampone ci sono i due operandi e la ALU ha effettuato la somma corretta;
- quando la ALU ha eseguito l'operazione richiesta salviamo il risultato in uno dei 16 registri, usando $W_{reg} = 1$.

7.17 Banco dei registri

Sono l'insieme dei 16 registri di uso generale. Sono tutti controllati da solo due **segnali di controllo** globali:

- W_M : scrittura sul registro;
- R_M : scrittura del contenuto del registro sul BUS interno.

L'input del Decoder è un **codice di registro** che ci permette di identificare il registro di interesse (tramite la sua conversione in binario [0,15]).



Per scrivere sul registro RAX devo:

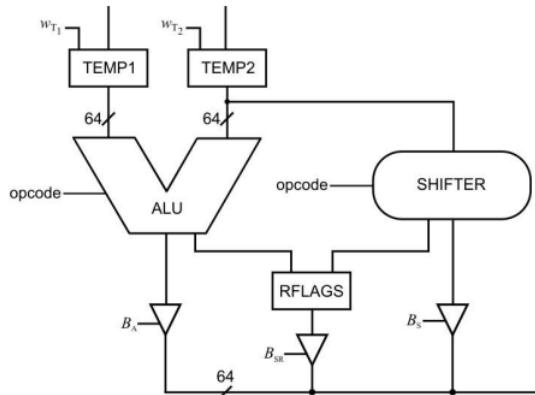
- impostare $W_M = 1$;
- selezionarlo tramite il codice di registro, in questo caso (0000).

Questa è effettivamente l'architettura utilizzata in tutti i processori.

7.18 Registro FLAGS

La ALU e lo Shifter sono reti iterative che emettono dei bit di stato (carry). I bit di carry devono essere memorizzati da qualche parte, per farlo usiamo il registro fondamentale FLAGS.

7.18.1 Architettura registro FLAGS



Quindi il registro FLAGS riceve informazioni di controllo dalla ALU o dallo Shifter.

Per poter leggere le informazioni memorizzate basta impostare $B_{SR} = 1$, così da far viaggiare i dati sul BUS.

7.18.2 Organizzazione bit registro FLAGS

I bit nel registro FLAGS si dividono in **status** e **control** bit:



- I bit di **stato** sono aggiornati dalla ALU e dallo Shifter per memorizzare informazioni sull'ultima operazione eseguita (carry).
- I bit di **controllo** sono modificabili dal programmatore per modificare alcune funzionalità del processore.
- I bit **riservati** sono inutilizzati (e non vanno modificati), il contenuto di questi bit sono delle costanti, 0 nei bit 3,4,5,8,12,13,14,15 e 1 nel bit 1, che possiamo usare in caso di necessità.

7.18.3 Bit di stato e di controllo del registro FLAGS

Bit di stato:

- **carry** (CF): vale 1 se l'ultima operazione ha prodotto un riporto;
- **parity** (PF): vale 1 se nel risultato dell'ultima operazione c'è un numero di 1 pari;
- **zero** (ZF): vale 1 se l'ultima operazione ha come risultato 0;
- **sign** (SF): vale 1 se l'ultima operazione ha prodotto un risultato negativo, nel caso di operazioni in complemento a due;
- **overflow** (OF): vale 1 se il risultato dell'ultima operazione supera la capacità di rappresentazione (xor ultimi due riporti).

Bit di controllo:

- **interrupt enable** (IF): indica se c'è la possibilità di interrompere l'esecuzione del programma in corso;
- **direction** (DF): modifica il comportamento delle operazioni su stringhe.

7.19 Istruzioni e classi dello z64

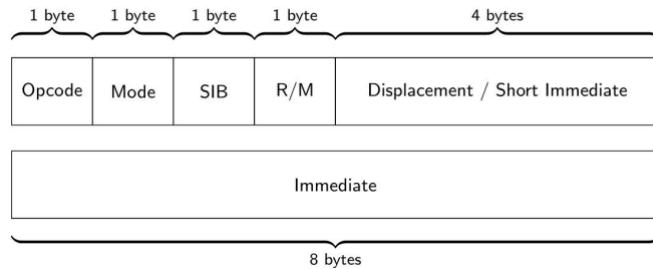
Le istruzioni sono organizzate in 8 classi:

- Classe 0: controllo dell'hardware.
- Classe 1: spostamento dati.
- Classe 2: aritmetiche su interi e logiche.
- Classe 3: rotazione e shift.
- Classe 4: operazioni sui bit di FLAGS.
- Classe 5: controllo del flusso d'esecuzione del programma.
- Classe 6: controllo condizionale del flusso d'esecuzione del programma.
- Classe 7: ingresso/uscita di dati.

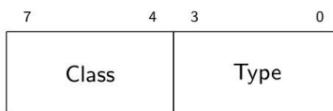
Le istruzioni con comportamenti simili sono nella stessa classe. Le classi sono utili per codificare le varie istruzioni.

7.20 Formato istruzioni macchina

Nel formato dello z64, le istruzioni hanno un formato variabile



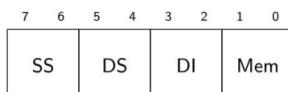
7.20.1 Campo Opcode



Il campo Opcode è diviso in due parti da 4 bit ciascuna:

- **Class** è un codice di 4 bit che identifica la classe di istruzioni cui appartiene quella corrente (ci sono 8 classi, sono codificabili con 3 bit, quindi un bit sarà inutilizzato).
- **Type** è un codice di 4 bit che identifica la precisa istruzione nella famiglia (ogni classe ha 16 operazioni).

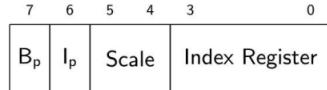
7.20.2 Campo Mode



- **SS** e **DS** contengono rispettivamente la codifica della dimensione dell'operando sorgente e destinazione.
- **DI** è un campo di 2 bit che indica o meno la presenza di un displacement e di un dato immediato.
- **Mem** indica quali degli operandi (sorgente o destinazione) sono da considerarsi operandi in memoria.

Campo	Valore	Significato
SS	00	La sorgente è un byte
	01	La sorgente è una word
	10	La sorgente è una longword
	11	La sorgente è una quadword
DS	00	La destinazione è un byte
	01	La destinazione è una word
	10	La destinazione è una longword
	11	La destinazione è una quadword
DI	00	Spiazzamento non utilizzato, immediato non presente
	01	Immediato presente
	10	Spiazzamento utilizzato
	11	Spiazzamento utilizzato, immediato presente
Mem	00	Sia la sorgente che la destinazione sono registri
	01	La sorgente è un registro, la destinazione è in memoria
	10	La sorgente è in memoria, la destinazione è un registro
	11	Condizione impossibile (genera un'eccezione a runtime)

7.20.3 Campo SIB (scala, indice, base)



- B_p e I_p indicano se la modalità di indirizzamento in memoria dell'istruzione corrente utilizza una base e/o un indice.
 - Se $I_p = 1$, il campo **Index** contiene la codifica binaria del registro indice.
 - Se $I_p = 0$, il campo Index contiene delle *don't care conditions*.
- **Scale** contiene il valore della scala, i cui valori sono:
 - 1 codificato come 00;
 - 2 codificato come 01;
 - 4 codificato come 10;
 - 8 codificato come 11.

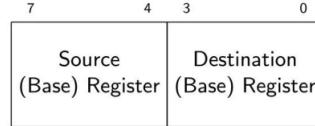
$$\left[\begin{array}{c} AX \\ BX \\ CX \\ DX \\ SP \\ BP \\ SI \\ DI \\ R8 \\ R9 \\ R10 \\ R11 \\ R12 \\ R13 \\ R14 \\ R15 \end{array} \right] + \left[\begin{array}{c} AX \\ BX \\ CX \\ DX \\ SP \\ BP \\ SI \\ DI \\ R8 \\ R9 \\ R10 \\ R11 \\ R12 \\ R13 \\ R14 \\ R15 \end{array} \right] * \left\{ \begin{array}{c} 1 \\ 2 \\ 4 \\ 8 \end{array} \right\} + [\text{spiazzamento}]$$

Codifica dei registri fisici

Nome	Codifica	Uso Comune
RAX	0000	Registro Accumulatore
RCX	0001	Registro Contatore
RDX	0010	Registro Dati
RBX	0011	Registro Base
RSP	0100	Stack Pointer
RBP	0101	Base Pointer
RSI	0110	Registro Sorgente
RDI	0111	Registro Destinazione
R8	1000	Registro di uso generale
R9	1001	Registro di uso generale
R10	1010	Registro di uso generale
R11	1011	Registro di uso generale
R12	1100	Registro di uso generale
R13	1101	Registro di uso generale
R14	1110	Registro di uso generale
R15	1111	Registro di uso generale

- Alcune istruzioni utilizzano degli operandi impliciti.
- Alcuni registri hanno un ruolo particolare, sfruttato da queste istruzioni.
- Se si utilizzano istruzioni senza operandi impliciti, i registri possono essere usati come general purpose indicandoli come operandi espliciti.

7.20.4 Campo R/M



Questo campo ha spazio per contenere esattamente la codifica di due registri. L'interpretazione di questi campi dipende dai valori di Mem e del bit B_p di SIB.

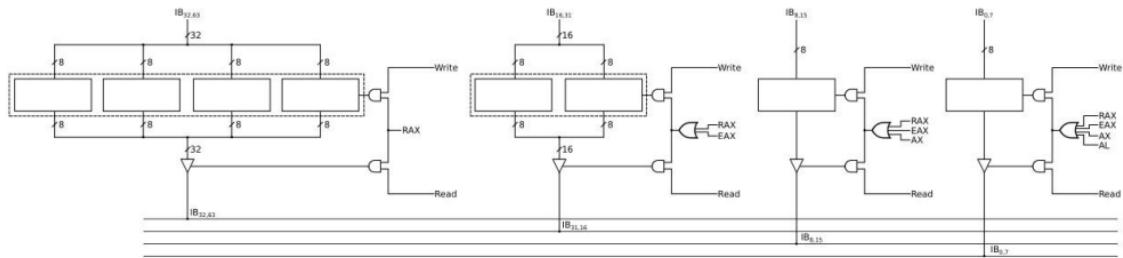
I registri possono quindi essere interpretati come registri semplici oppure come registri di base.

7.21 Registri virtuali

Le istruzioni assembly accettano un suffisso per indicare la dimensione del dato trattato:

- b (byte): 8 bit;
- w (word): 16 bit;
- l (longword): 32 bit;
- q (quadword): 64 bit.

I registri fisici sono tutti a 64 bit, ma visto che è necessario accedere a sottoporzioni dei dati nei registri, utilizziamo questa architettura:



Un registro fisico è composto da 4 registri virtuali.

- Se vogliamo accedere a un dato a **64 bit** dobbiamo accedere al valore di tutti i flip flop (suffisso istruzione: Q, registro: RAX).
- Se vogliamo accedere a un dato a **32 bit** dobbiamo accedere solo al valore dei primi 4 flip flop (suffisso istruzione: L, registro: EAX).
- Se vogliamo accedere a un dato a **16 bit** dobbiamo accedere solo al valore dei primi 2 flip flop (suffisso istruzione: W, registro: AX).
- Se vogliamo accedere a un dato a **8 bit** dobbiamo accedere solo al valore del primo flip flop (suffisso istruzione: B, registro: AL).

Per fare questo aggiungiamo dei segnali di controllo che specificano la taglia del registro a cui vogliamo accedere.

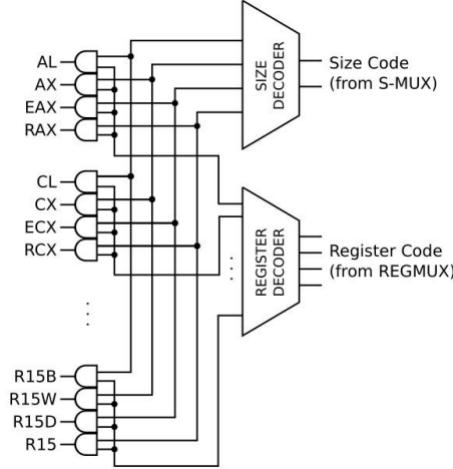
Per **scrivere** su un registro virtuale dobbiamo:

- selezionare la taglia del registro, lo facciamo grazie ai segnali di controllo messi in OR tra loro:
 - nel primo registro virtuale ci sono tutti e quattro i segnali di controllo messi in OR perché qualunque sia la taglia a cui voglio accedere il primo flip flop sarà compreso.
 - nel secondo registro virtuale ci sono i segnali di controllo per accedere ai registri di 16, 32, 64 bit. Ecc...

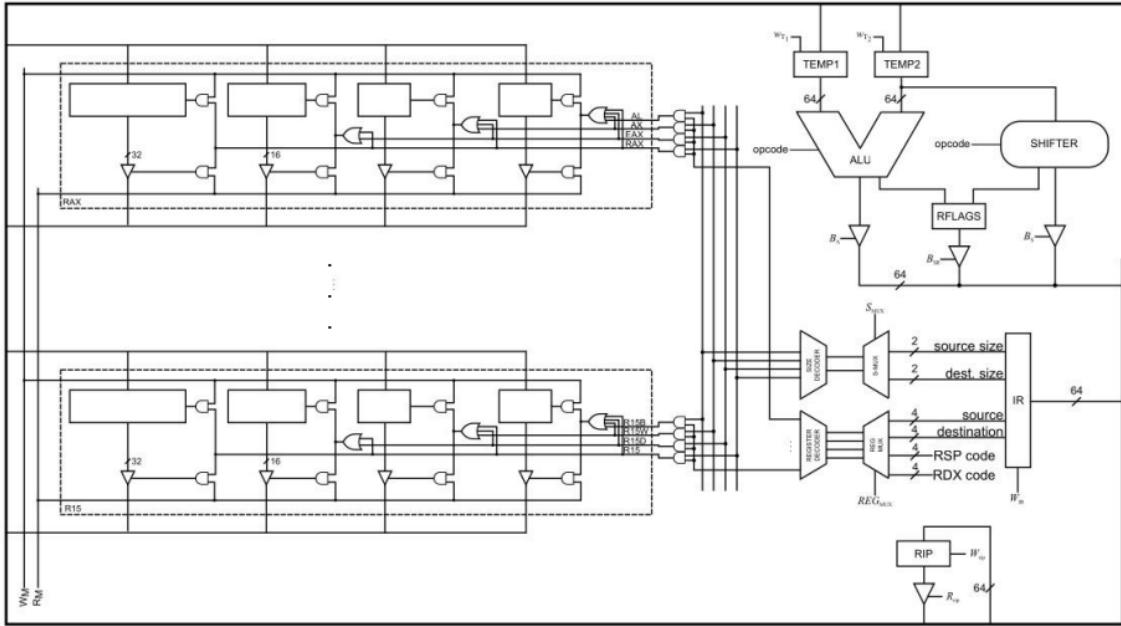
- impostare il segnale di controllo di Write a 1. Write è messo in AND con il segnale che ci fornisce la taglia del registro virtuale, quindi scriveremo solo sui registri virtuali selezionati.

Per **leggere** i dati dei registri virtuali, quindi scrivere il loro contenuto sull'internal bus dobbiamo abilitare il buffer three-state. L'abilitazione del buffer è in AND con il segnale di Read e il segnale che seleziona la taglia.

Il segnale di controllo che ci fornisce la dimensione del registro a cui vogliamo accedere ci viene fornito da SS e DS del campo Mode nell'istruzione register.



7.21.1 Architettura rivisitata



Quindi l'accesso ai banchi dei registri è organizzato in questo modo:

- Il *register decoder* seleziona il registro fisico a cui vogliamo accedere.
- Il *size decoder* seleziona il registro virtuale, dentro al registro fisico selezionato, a cui vogliamo accedere.
- Il *S-Mux* fornisce l'input al size decoder, seleziona da IR il campo SS o DS.
- Il *Reg-Mux* seleziona da IR source o destination.

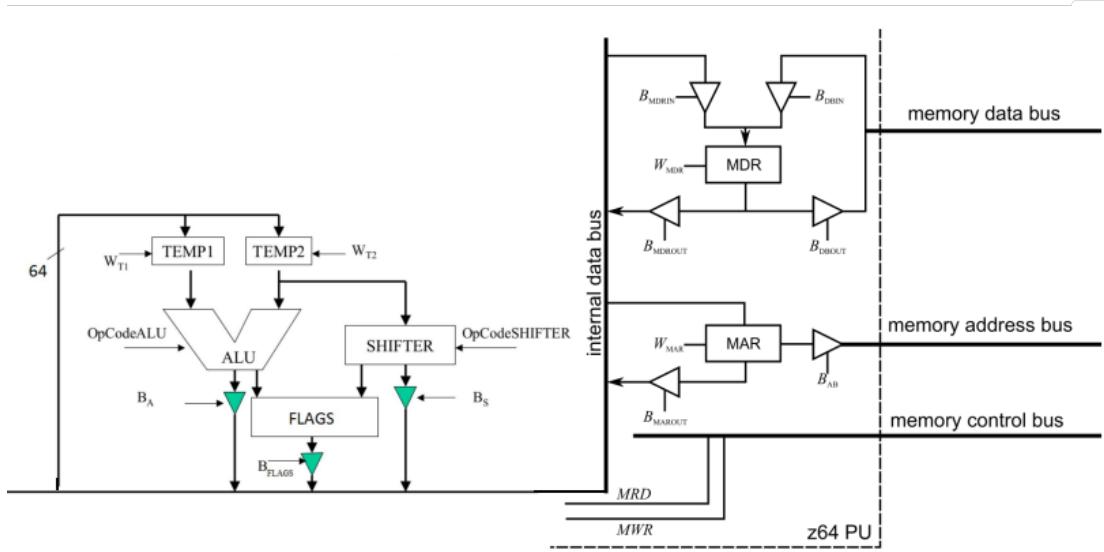
7.22 Implementazione modalità di indirizzamento

La determinazione di un indirizzo di memoria è un'operazione complessa, dobbiamo effettuare due somme e una moltiplicazione.

Queste operazioni possiamo effettuarle con la ALU e lo shifter (hardware che abbiamo già a disposizione, possiamo utilizzare lo shifter perché dobbiamo moltiplicare solo per multipli di 2, scala $\in \{1,2,4,8\}$).

Il risultato finale (registro effettivo) deve andare a finire dentro MAR (registro invisibile al programmatore). Dato che dobbiamo effettuare tre operazioni, dobbiamo rispettare la gerarchia ed effettuare prima la moltiplicazione e per salvare questo risultato intermedio in un registro invisibile al programmatore (così non rischiamo di sovrascrivere dati) possiamo usare MAR.

Nell'architettura di MAR che abbiamo implementato precedente (7.15) questo non è possibile, dato che l'internal data bus è unidirezionale e non possiamo leggere il valore di MAR. Usiamo un'altra architettura:



Quindi per effettuare l'operazione: [base + (indice · scala) + spiazzamento] dobbiamo:

- calcolare [indice · scala]:

1. Il contenuto del registro indice viaggia sull'internal data bus per arrivare a TEMP2.
2. Da TEMP2 lo propago allo shifter che, a seconda dell'OpCode, lo moltiplica per 1,2,4 o 8 (con degli shift verso sinistra, rispettivamente: 0,1,2,3).
3. Abilito l'uscita dello shifter (B_S) e attraverso l'internal data bus (che al momento è libero) porto il risultato della moltiplicazione sul registro MAR, abilitando la scrittura (W_{MAR}).

- calcolare [base + (indice · scala)]:

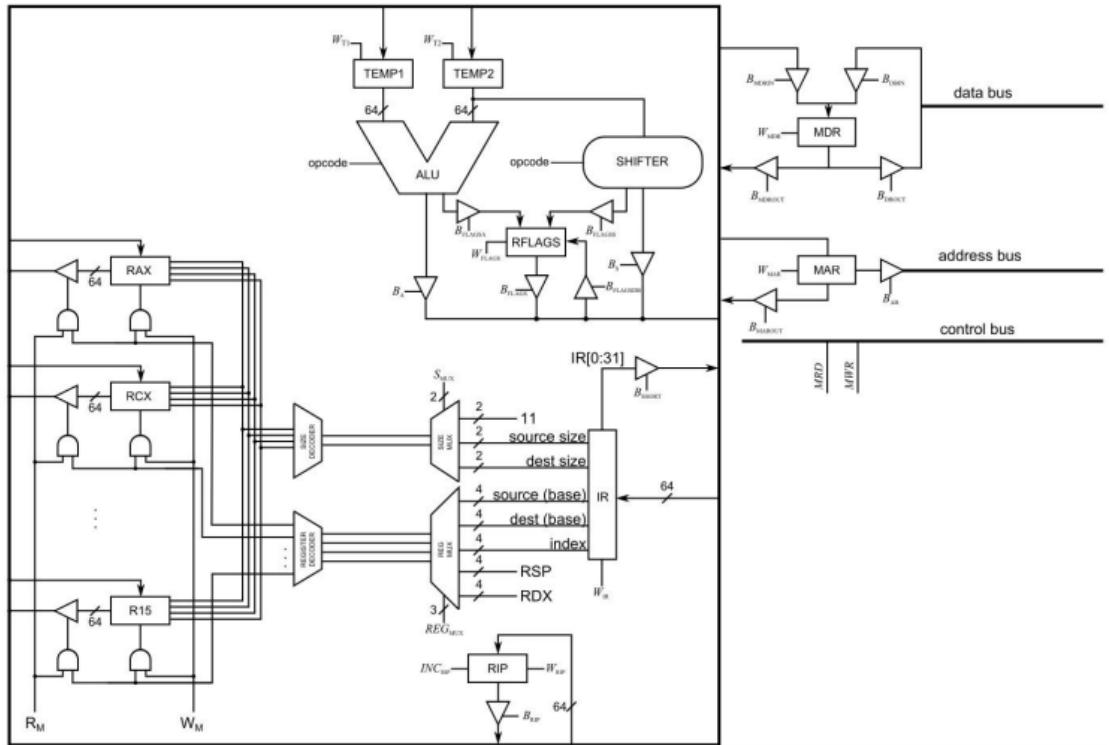
1. Scriviamo (indice · scala) su TEMP1 (copiandolo da MAR) abilitando B_{MAROUT} e W_{T1} .
2. Prendiamo il contenuto del registro base e lo scriviamo su TEMP2.
3. Configuriamo l'OpCode della ALU come somma, scriviamo il risultato generato dalla ALU su MAR (abilitando B_A e W_{MAR}).

- calcolare [base + (indice · scala) + spiazzamento]:

1. Prendiamo lo spiazzamento (dai 32 bit meno significativi dell'IR) e lo copiamo su TEMP1.
2. Copiamo [base + (indice · scala)] da MAR su TEMP2.
3. Impostiamo l'OpCode della ALU su somma e copiamo il risultato finale su MAR.

Tutte le microoperazioni che effettuiamo avvengono in colpi di clock differenti.

7.23 Architettura finale della PU



- BUS interno (7.9.3);
- Instruction Pointer (RIP) (7.10), Instruction Register (IR) (7.11);
- ALU (5.4), Shifter (5.3);
- MDR, MAR (7.15);
- registro FLAGS (7.18);
- banco dei registri (7.21), registri fisici e virtuali (7.8);
- system BUS (7.14).

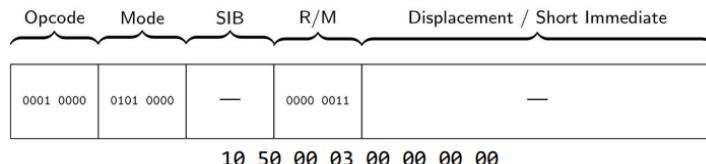
8 Programmazione assembly

8.1 Alcune istruzioni assembly

- **movb %al, %bl**: copia il contenuto del byte meno significativo di RAX in RBX (il suffisso è b quindi usiamo il registro virtuale AL di 8 bit).
- **movw %ax, (%rdi)**: copia il contenuto dei 2 byte meno significativi di RAX nei due byte di memoria il cui indirizzo iniziale è memorizzato in RDI (dentro RDI c'è l'indirizzo del primo byte di memoria, il secondo byte è il successivo).
- **movl (%rsi), %eax**: copia nei 4 byte meno significativi di RAX il contenuto dei 4 byte di memoria il cui indirizzo è specificato in RSI (RSI contiene l'indirizzo del primo byte, gli altri 3 sono i successivi).
- **movq (%rsi, %rcx, 8), %rax**: copia nel registro RAX il contenuto degli 8 byte di memoria il cui indirizzo iniziale è calcolato come $RSI + RCX \cdot 8$ ("contenuto di RCX per 8 + contenuto di RSI").
- **subl %eax, %edx**: sottrai il contenuto dei 4 byte meno significativi di RAX dai 4 byte meno significativi di RDX e aggiorna il contenuto dei 4 byte meno significativi di RDX ("destinazione - sorgente" usando la rappresentazione in complemento a due).
- **addb \$d, %al**: somma al byte meno significativo di RAX la quantità costante d (il dollaro indica una costante).
- **addb d, %al**: somma al byte meno significativo di RAX il byte contenuto all'indirizzo di memoria d (in questo caso d è lo spiazzamento della modalità di indirizzamento in memoria).

8.2 Traduzioni istruzioni assembly in linguaggio macchina

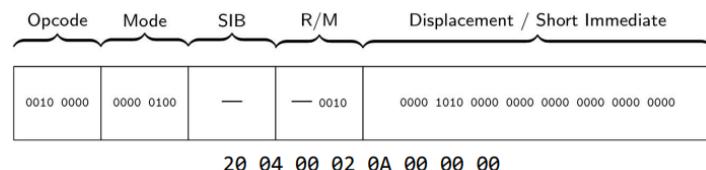
- **movw %ax, %bx**



1. Class: 0001, Type: 0000.
2. SS: 01 (16 bit), DS: 01 (16 bit), DI: 00, Mem: 00.
3. B_P : 0, I_P : 0, Scale: 00, Index Register: — (dato che $I_P = 0$).
4. Source: 0000 (RAX), Destination: 0011 (RBX).
5. Displacement: —...—.

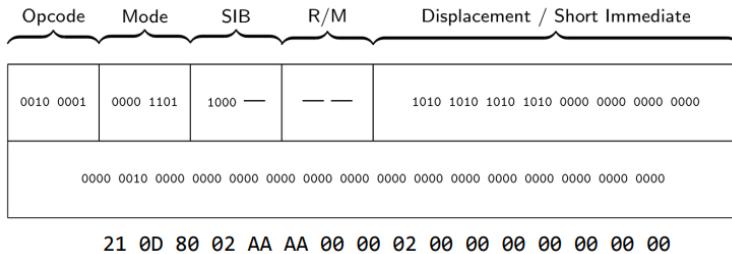
10 50 00 03 00 00 00 00 è la codifica esadecimale dell'istruzione (le don't care conditions sono considerate come zeri).

- **addb \$0xa, %dl**



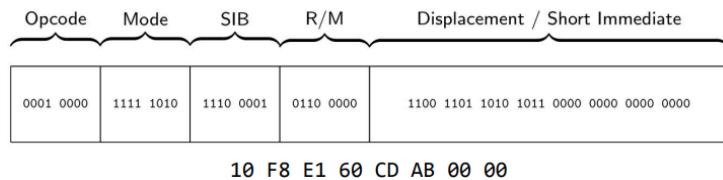
Utilizziamo una costante numerica come operando (in R/M primi 4 bit don't care conditions dato che una costante non può essere una destinazione). Lo 0x prima di una costante indica che la costante è espressa in esadecimale.

- subb \$0x2, 0xAAAA



In questo caso è presente uno spiazzamento.

- `movq 0xABCD(%rsi, %rcx, 4), %rax`



- OpCode:
 - Class: 0001, classe 1 (spostamenti dati);
 - Type: 0000, prima istruzione della classe.
 - Mode:
 - SS: 11, la sorgente è una quadword (64 bit);
 - DS: 11, la destinazione è una quadword (64 bit);
 - DI: 10, spiazzamento utilizzato;
 - Mem: 10, la sorgente è in memoria, la destinazione è un registro.
 - SIB:
 - B_P : 1, utilizziamo la base;
 - I_P : 1, utilizziamo un indice;
 - Scale: 10, la scala è 4;
 - Index register: 0001, il registro indice è RCX.
 - R/M:
 - Source: 0110, il registro sorgente è RSI;
 - Destination: 0000, il regitro destinazione è RAX.
 - Displacement:
 - la codifica di 0xABCD in binario a 32 bit è 0000 0000 0000 0000 1010 1011 1100 1101;
 - dato che lo z64 utilizza il sistema little-endian per la memorizzazione dei byte raggruppiamo la codifica in gruppi da 8 bit: 00000000 00000000 10101011 11001101 e li scriviamo partendo dal meno significativo;
 - 11001101 10101011 00000000 00000000 = 1100 1101 1010 1011 0000 0000 0000 0000.

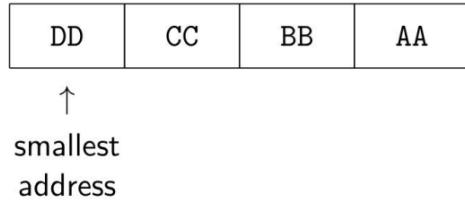
8.3 Ordine dei byte: Memory Endianness

L'ordine dei byte descrive la modalità utilizzata dai calcolatori per immagazzinare in memoria dati di dimensione superiore al byte.

La differenza dei sistemi è data dall'ordine con il quale i byte costituenti il dato da immagazzinare vengono memorizzati:

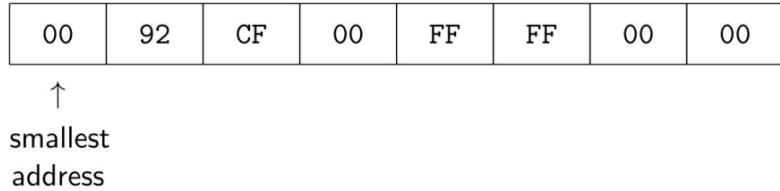
- *little-endian*: memorizzazione che inizia dal byte meno significativo per finire col più significativo (usato dai processori Intel e dallo z64).
- *big-endian*: memorizzazione che inizia dal byte più significativo per finire col meno significativo (Network-byte order).
- *middle-endian*: ordine dei byte né crescente né decrescente.

Esempio Nell'organizzazione a little-endian il valore 0xAABBCCDD è posto nel layout di memoria come segue:



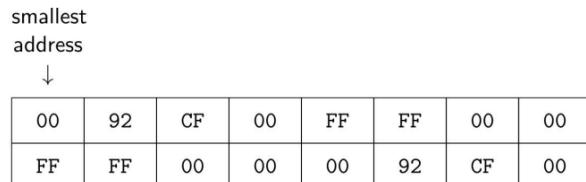
Un numero in esadecimale occupa 4 bit ($16 = 2^4$) quindi due numeri in esadecimale occupano un byte in binario.

Esempio Memorizzazione di due interi di 4 byte (0x00cf9200 e 0x0000ffff) in modo consecutivo in memoria.



I byte di ogni intero sono scambiati, ma non l'ordine degli interi.

Esempio Memorizzazione di due interi a 4 byte, seguiti da un intero a 8 byte (0x00cf9200, 0x0000ffff, 0x00cf92000000ffff)



L'indirizzo di memoria non cambia se voglio accedere ad una sottoporzione del dato invece che a tutto il dato. I processori Big-Endian devono invece calcolare uno spiazzamento corretto per accedere a sottoporzioni del dato.

8.4 Istruzioni dello z64

Queste convenzioni vengono utilizzate per rappresentare gli operandi delle istruzioni:

- **B**: l'operando è un registro di uso generale, un indirizzo di memoria o un valore immediato. In caso di un indirizzo di memoria, qualsiasi combinazione delle modalità di indirizzamento è lecita. In caso di un immediato, la sua posizione dipende dalla possibile presenza dello spiazzamento e dalla sua dimensione.
- **E**: L'operando è un registro di uso generale, o un indirizzo di memoria. In caso di un indirizzo di memoria, qualsiasi combinazione delle modalità di indirizzamento è lecita.
- **G**: L'operando è un registro di uso generale.
- **K**: L'operando è una costante numerica non segnata di valore fino a $2^{32} - 1$.
- **M**: L'operando è una locazione di memoria, codificata come uno spiazzamento a partire dal contenuto del registro RIP dopo l'esecuzione della fase di fetch.
- **ImmK**: L'operando è un dato immediato di K cifre binarie.

8.5 Stack di programma

Il processore ha un numero estremamente limitato di registri e dato che un programma potrebbe avere bisogno di gestire tante variabili o variabili molto grandi, è possibile utilizzare un'area di memoria come "area di appoggio": lo stack di programma.

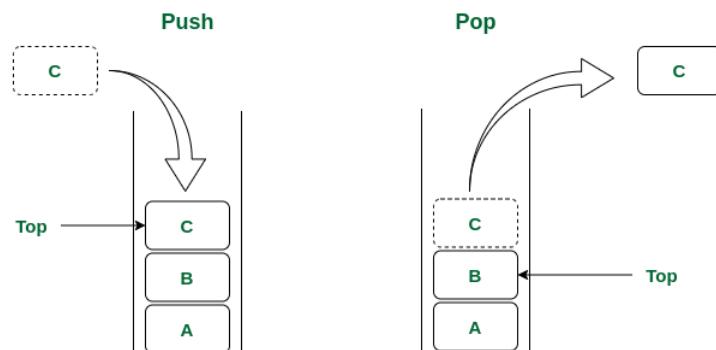
Lo stack o pila è una struttura dati di tipo LIFO (Last In First Out): il primo elemento che può essere prelevato è l'ultimo ad essere stato memorizzato.

Si possono effettuare due operazioni su questa struttura dati:

- **push**: viene inserito un elemento sulla sommità (top) della pila;
- **pop**: viene prelevato l'elemento superiore dalla pila.

8.5.1 Gestione dello stack

- Lo stack è composto da quadword (non si può eseguire una push di un singolo byte, viene esteso a quadword);
- la cima dello stack è individuata dall'indirizzo memorizzato in un registro specifico chiamato SP (Stack Pointer, registro general purpose);
- modificare il valore di SP significa perdere il riferimento alla cima dello stack, e quindi a tutto il suo contenuto (non farlo);
- lo stack è posto in fondo alla memoria e cresce "all'indietro";
- quindi lo stack "cresce" se il valore contenuto in SP diminuisce, "decresce" se il valore contenuto in SP cresce.



8.6 Classi delle istruzioni dello z64

8.6.1 Classe 0: controllo hardware

Tipo	Mnemonico	Operandi	O	S	Z	P	C	Descrizione
1	hlt	-	-	-	-	-	-	Mette la CPU in modalità di basso consumo energetico, finché non viene ricevuta l'interruzione successiva
2	nop	-	-	-	-	-	-	Nessuna operazione
3	int	Imm8	-	-	-	-	-	Chiama esplicitamente un gestore di interruzioni

8.6.2 Classe 1: istruzioni di movimento dati

Tipo	Mnemonico	Operandi	O	S	Z	P	C	Descrizione
0	mov	B, E	-	-	-	-	-	Fa una copia di B in E
1	movsx	E, G	-	-	-	-	-	Fa una copia di E in G con estensione del segno
2	movzx	E, G	-	-	-	-	-	Fa una copia di E in G con estensione dello zero
3	lea	E, G	-	-	-	-	-	Valuta la modalità di indirizzamento, salva il risultato in G
4	push	E	-	-	-	-	-	Copia il contenuto di E sulla cima dello stack
5	pop	E	-	-	-	-	-	Copia il contenuto della cima dello stack in E
6	pushf	-	-	-	-	-	-	Copia sulla cima dello stack il registro FLAGS
7	popf	-	-	-	-	-	-	Copia nel registro FLAGS il contenuto della cima dello stack
8	movs	-	-	-	-	-	-	Esegue una copia memoria-memoria
9	stos	-	-	-	-	-	-	Imposta una regione di memoria ad un dato valore

Per supportare l'istruzione di popf è necessario permettere la scrittura esplicita del registro FLAGS. Occorre quindi aggiungere un collegamento verso il data bus interno (come nella architettura finale della PU (7.23)).

Estensione del segno

Istruzione	Tipo di conversione
movsbw %al, %ax	Estendi il segno da byte a word
movsbl %al, %eax	Estendi il segno da byte a longword
movsbq %al, %rax	Estendi il segno da byte a quadword
movswl %ax, %eax	Estendi il segno da word a longword
movswq %ax, %rax	Estendi il segno da word a quadword
movslq %eax, %rax	Estendi il segno da longword a quadword

8.6.3 Classe 2: istruzioni logico-aritmetiche

Tipo	Mnemonico	Operandi	O S Z P C	Descrizione
0	add	B, E	↑↑↑↑↑↑↑↑	Memorizza in E il risultato di E + B
1	sub	B, E	↑↑↑↑↑↑↑↑	Memorizza in E il risultato di E - B
2	adc	B, E	↑↑↑↑↑↑↑↑	Memorizza in D il risultato di E + B + CF
3	sbb	B, E	↑↑↑↑↑↑↑↑	Memorizza in D il risultato di E - (B + neg(CF))
4	cmp	B, E	↑↑↑↑↑↑↑↑	Confronta i valori di B ed E calcolando E - B, il risultato viene poi scartato
5	test	B, E	↑↑↑↑↑↑↑↑	Calcola l'and logico bit a bit di B ed E, il risultato viene poi scartato
6	neg	E	↑↑↑↑↑↑↑↑	Rimpiazza il valore di E con il suo complemento a 2

Tipo	Mnemonico	Operandi	O S Z P C	Descrizione
7	and	B, E	0↑↑↑↑↑0	Memorizza in E il risultato dell'and bit a bit tra B ed E
8	or	B, E	0↑↑↑↑↑0	Memorizza in E il risultato dell'or bit a bit tra B ed E
9	xor	B, E	0↑↑↑↑↑0	Memorizza in E il risultato dello xor bit a bit tra B ed E
10	not	E	0↑↑↑↑↑0	Rimpiazza il valore di E con il suo complemento a uno
11	bt	K, E	- - - - ↑	Imposta CF al valore del K-simo bit di E (bit testing)

Con lo z64 è possibile effettuare operazioni aritmetiche usando dati a 64 bit, se vogliamo effettuare operazioni con dati più grandi possiamo usare le istruzioni **adc** e **sbb** che ci permettono di realizzare programmi a precisione arbitraria.

```
movq $operand_1_high, %rax
      movq $operand_1_low, %rbx
      movq $operand_2_high, %rcx
      movq $operand_2_low, %rdx
      addq %rbx, %rdx
      adcq %rax, %rcx
```

8.6.4 Classe 3: istruzioni di rotazione e shift

Tipo	Mnemonico	Operandi	O S Z P C	Descrizione
0	sal	K, G	↑↑↑↑↑↑↑↑	Moltiplica per 2, K volte
1	sal	G	↑↑↑↑↑↑↑↑	Moltiplica per 2, RCX volte
0	shl	K, G	↑↑↑↑↑↑↑↑	Moltiplica per 2, K volte
1	shl	G	↑↑↑↑↑↑↑↑	Moltiplica per 2, RCX volte
2	sar	K, G	↑↑↑↑↑↑↑↑	Dividi (con segno) per 2, K volte
3	sar	G	↑↑↑↑↑↑↑↑	Dividi (con segno) per 2, RCX volte
4	shr	K, G	↑↑↑↑↑↑↑↑	Dividi (senza segno) per 2, K volte
5	shr	G	↑↑↑↑↑↑↑↑	Dividi (senza segno) per 2, RCX volte
6	rcl	K, G	↓ - - - - ↑	Ruota a sinistra, K volte
7	rcl	G	↓ - - - - ↑	Ruota a sinistra, RCX volte
8	rcr	K, G	↑ - - - - ↑	Ruota a destra, K volte
9	rcr	G	↑ - - - - ↑	Ruota a destra, RCX volte
10	rol	K, G	↓ - - - - ↑	Ruota a sinistra, K volte
11	rol	G	↓ - - - - ↑	Ruota a sinistra, RCX volte
12	ror	K, G	↓ - - - - ↑	Ruota a destra, K volte
13	ror	G	↓ - - - - ↑	Ruota a destra, RCX volte

8.6.5 Classe 4: manipolazione dei bit di FLAGS

Tipo	Mnemonico	Operandi	O S Z P C	Descrizione
0	clc	-	- - - - 0	Resetta CF
1	clp [†]	-	- - - 0 -	Resetta PF
2	clz [†]	-	- - 0 - -	Resetta ZF
3	cls [†]	-	- 0 - - -	Resetta SF
4	cli	-	- - - - -	Resetta IF
5	cld	-	- - - - -	Resetta DF
6	clo [†]	-	0 - - - -	Resetta OF
7	stc	-	- - - - 1	Imposta CF
8	stp [†]	-	- - - 1 -	Imposta PF
9	stz [†]	-	- - 1 - -	Imposta ZF
10	sts [†]	-	- 1 - - -	Imposta SF
11	sti	-	- - - - -	Imposta IF
12	std	-	- - - - -	Imposta DF
13	sto [†]	-	1 - - - -	Imposta OF

[†]: non esiste un'istruzione corrispondente nell'assembly x86

8.6.6 Classe 5: controllo del flusso di programma

Tipo	Mnemonico	Operandi	O S Z P C	Descrizione
0	jmp	M	- - - - -	Esegue un salto relativo
1	jmp	*G	- - - - -	Esegui un salto assoluto
2	call	M	- - - - -	Esegue una chiamata a subroutine relativa
3	call	*G	- - - - -	Esegue una chiamata a subroutine assoluta
4	ret	-	- - - - -	Ritorna da una subroutine
5	iret	-	↔ ↔ ↔ ↔ ↔ ↔	Ritorna dal gestore di una interruzione

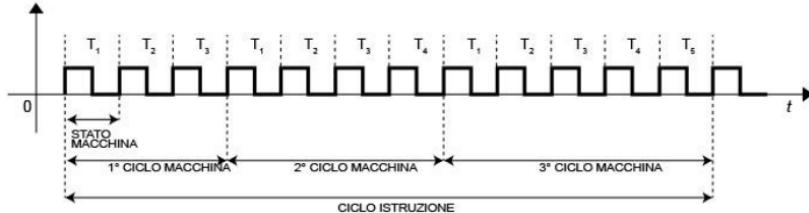
8.6.7 Classe 6: controllo condizionale del flusso

Tipo	Mnemonico	Operandi	O S Z P C	Descrizione
0	jc	M	- - - - -	Salta a M se CF è impostato
1	jp	M	- - - - -	Salta a M se PF è impostato
2	jz	M	- - - - -	Salta a M se ZF è impostato
3	js	M	- - - - -	Salta a M se SF è impostato
4	jo	M	- - - - -	Salta a M se OF è impostato
5	jnc	M	- - - - -	Salta a M se CF non è impostato
6	jnp	M	- - - - -	Salta a M se PF non è impostato
7	jnz	M	- - - - -	Salta a M se ZF non è impostato
8	jns	M	- - - - -	Salta a M se SF non è impostato
9	jno	M	- - - - -	Salta a M se OF non è impostato

9 z64: Control Unit (CU)

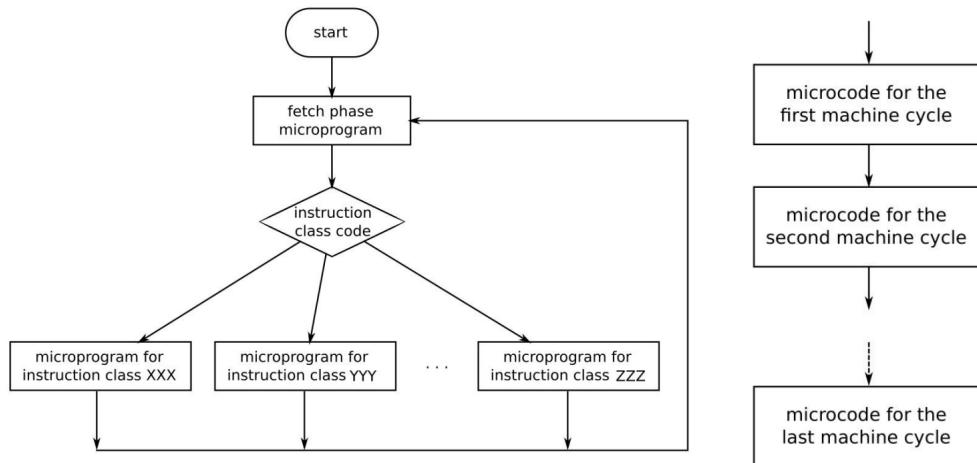
9.1 Microoperazioni

Dato che lo z64 è un processore multiciclo, l'esecuzione di un'istruzione è suddivisa in più cicli macchina e ogni ciclo macchina è suddiviso in più stati macchina.



La CU implementa un'istruzione con un insieme di *microoperazioni*, ogni microoperazione è definita dai segnali di controllo abilitati in uno specifico stato macchina.

9.2 Realizzazione della CU



Il primo blocco dopo lo start identifica il microprogramma della fase di *fetch*. Il blocco successivo è la fase di *decode*, a seconda della classe dell'istruzione finiamo in stati diversi in cui ci sono i microprogrammi per implementare le varie istruzioni

9.3 Organizzazione della CU

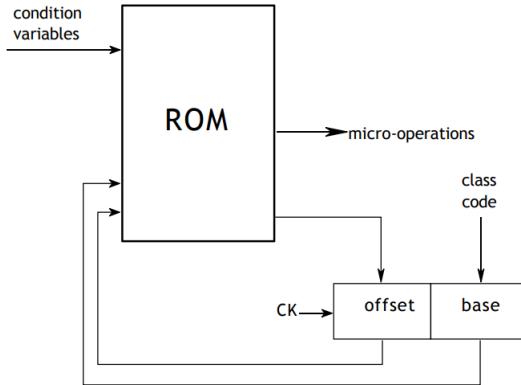
Per eseguire il microprogramma di un'istruzione, la CU userà diversi input (conditions variables):

- classe e tipo dell'istruzione (contenuta in IR);
- variabili di condizione che arrivano al di fuori della CPU;
- variabili di condizione che arrivano da dentro la CPU (bit del registro FLAGS);
- modalità di indirizzamento degli operandi dell'istruzione (dedotta da IR).

Il numero di segnali di output della CU dipende dall'implementazione della PU e dei moduli esterni.

L'organizzazione della CU dipende dal costo implementativo, dalla performance desiderata e dal tipo di macchina a stati finiti scelta (Mealy o Moore).

9.4 La CU come macchina di Mealy



La CU è una rete sequenziale complessa, con molte variabili in input e output.

Le funzioni δ (aggiornamento stato macchina) e ω (generazione microoperazioni) possono essere rappresentate tramite una ROM, paginata in funzione dei microprogrammi. Quindi la ROM è formata da n pagine di dimensione uguale tra loro e all'interno di ciascuna pagina sono rappresentate le microoperazioni. Ogni pagina rappresenta un microprogramma.

Anche se i microprogrammi possono avere dimensioni differenti la dimensione delle pagine è la stessa, questo causa uno spreco di memoria quando abbiamo microprogrammi corti. Teniamo le pagine con la stessa dimensione perché con i soli 4 bit che identificano la classe nell'IR e co-piandoli all'interno del registro base sappiamo immediatamente quale è la posizione alla quale il microprogramma deve saltare, all'interno della ROM, per poter eseguire l'istruzione dopo la fase di fetch.

10 La gerarchia di memoria

La memoria è un dispositivo esterno utilizzato per memorizzare informazioni.

10.1 Memoria interna, principale e secondaria

- Registri interni alla CPU:
 - visibili o meno al programmatore;
 - memorizzano temporaneamente dati e istruzioni;
 - dimensione: KByte;
 - velocità: stessa della CPU.
- Memoria principale (DRAM):
 - memorizza dati e istruzioni dei programmi;
 - velocità: intermedia.
- Memoria secondaria (dischi, DVD, ...):
 - dimensioni: GByte, TByte;
 - velocità: millisecondi.

10.1.1 Tecnologie delle memorie

Le memorie sono realizzate con tecnologie diverse e si differenziano per:

- costo per bit immagazzinato;
- tempo di accesso (*latenza*): ritardo tra l'istante in cui avviene la richiesta e l'istante in cui il dato è disponibile;
- modo di accesso (*seriale* o *casuale*).

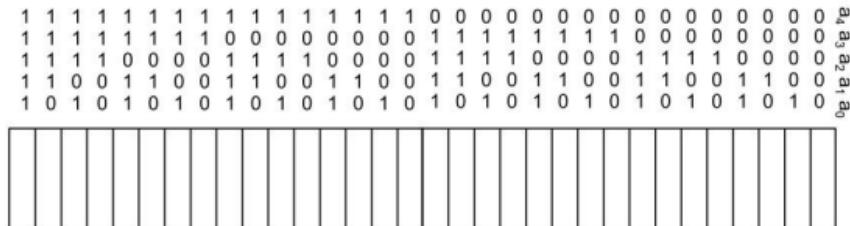
La velocità e il costo della memoria sono direttamente proporzionali. Quindi sono necessari diversi tipi di memorie, *più economiche* dove poter immagazzinare grandi quantità di dati e *memorie più piccole*, ma estremamente più veloci, con le quali possiamo interagire con la CPU.

10.2 Memoria principale

10.2.1 Organizzazione logica

La memoria principale si basa sul modello di memoria **flat**: un lungo vettore di celle di memoria (byte). Ogni cella è identificata da un **indirizzo**.

Esempio Organizzazione logica a vettore di 32 celle, l'indirizzo è composto da 5 bit ($2^5 = 32$).



Il problema di questa organizzazione è che potremmo dover leggere dati di 1, 2, 4, 8 byte (B, W, L, Q) ma la memoria è indirizzata al singolo byte.

10.2.2 Organizzazione in moduli

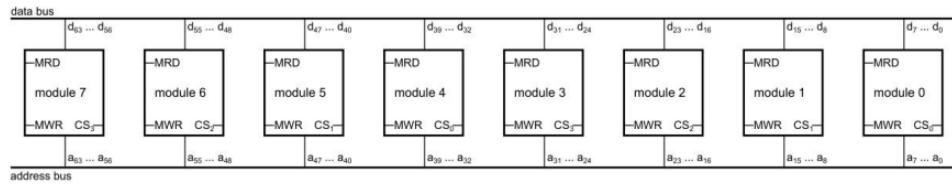
Possiamo organizzare la memoria in moduli paralleli in modo da poter leggere più byte alla volta. In questa organizzazione i tre bit meno significativi identificano il modulo, quelli più significativi la riga.

Esempio Lettura della riga 01 (due bit più significativi):



In questo caso selezionando una riga leggiamo sempre 8 byte, dobbiamo implementare un modo per leggere/scrivere meno di 8 byte.

Per fare questo introduciamo il segnale di Chip Select (CS):



in questo modo dopo aver selezionato una riga possiamo abilitare il segnale di CS solo su 1, 2, 4 o 8 moduli (MDR e MWR per sapere se vogliamo scrivere o leggere dati).

Per il segnale di Chip Select sono necessari 2 bit (abbiamo 4 possibili taglie: 1, 2, 4, 8) che possiamo prendere dai due bit di SS o DS nel campo Mode dell'Instruction Register (operando in memoria).

Anche in questa organizzazione c'è un problema: potrei dover leggere 8 byte partendo non dall'inizio delle righe, in questo caso gli ultimi byte si troverebbero sulla riga successiva.

Quindi il problema è che con questa organizzazione possiamo leggere fino ad 8 byte ma solo se sono tutti allineati (dobbiamo selezionare prima la riga).

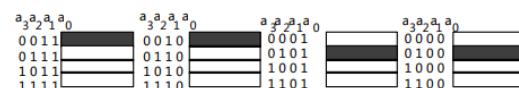
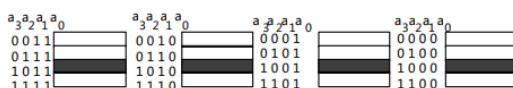
10.2.3 Allineamento e disallineamento

Una lettura disallineata può coinvolgere al più due righe.

Abbiamo due possibili soluzioni:

- Soluzione hardware: effettuiamo più letture e ricostruiamo il dato.
- Soluzione software: utilizziamo il *padding*, cioè spostiamo le celle che vogliamo leggere sulla stessa riga.

In questa soluzione il problema è che devo spostare degli elementi e quindi il loro indirizzo in memoria, se per esempio uno di questi elementi fa parte di una struct cambiando indirizzo cambiano anche gli offset che devo andare ad applicare per accedere ai membri della struct. I compilatori usano l'operazione di padding, cioè sprecano dei byte all'interno delle struct per aumentare la probabilità che i singoli membri siano sulla stessa linea di memoria. (Motivo per il quale `sizeof(struct)` ≥ somma dimensione singoli membri della struct).



Quattro byte allineati sullo stesso indirizzo di riga.

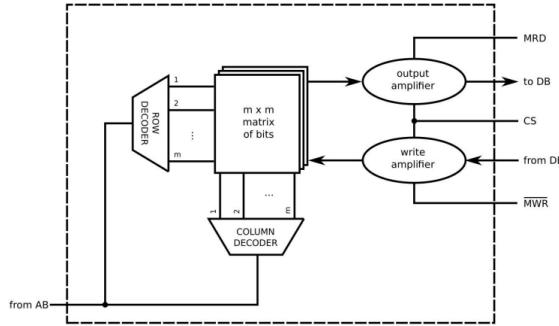
Quattro byte disallineati.



Due byte disallineati

Quindi possiamo generalizzare questa organizzazione ad una matrice.

10.2.4 Organizzazione a matrice



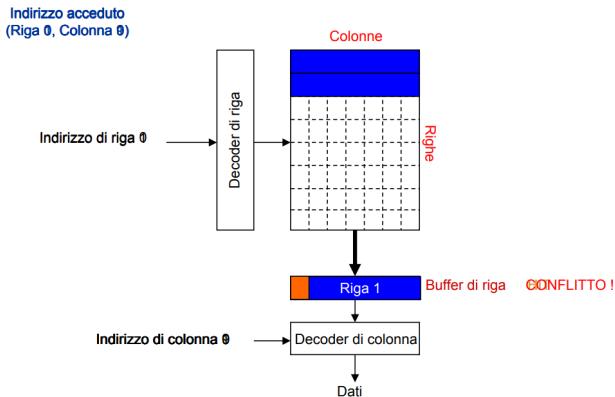
Quando forniamo un indirizzo di memoria questo viene utilizzato per selezionare una riga e poi una colonna (modulo) dalla quale cominciare a scrivere o leggere.

Questi componenti funzionano con poca elettricità quindi a seconda del segnale che è attivo tra MRD e MWR c'è un amplificatore che potenziano la quantità di corrente in arrivo dalla RAM (lettura) o in arrivo dal processore (scrittura) in modo tale da avere corrente sufficiente.

I decoder di riga e colonna sono circuiti combinatori, hanno un ritardo tra arrivo dell'input e uscita dell'output, quindi questo sommato al fatto che i percorsi dei fili che connettono i vari componenti sono molto lunghi implica che ci possono essere ritardi non minimi.

10.2.5 Funzionamento organizzazione a matrice

Per evitare ritardi troppo lunghi, la memoria lavora in questo modo:



Il buffer di riga all'interno della RAM è un registro tampone.

1. Il buffer di riga è vuoto.
2. Arriva un indirizzo dal processore, il decoder di riga seleziona una riga.
3. La riga viene letta e amplificata tutta e copiata all'interno del buffer di riga.
4. L'indirizzo di colonna viene inviato al decoder di colonna che seleziona solo il byte che ci interessa (p. es. byte 0).
5. Il dato viene inviato al data bus.

Dopo aver fatto l'accesso ad un byte ci sono due possibilità:

- **Hit:** la richiesta del byte successivo è sulla stessa riga del precedente. In questo caso non c'è bisogno di ricaricare la riga nel buffer e questo ci permette di ridurre notevolmente il tempo di ritardo.
- **Miss:** la richiesta del byte non successivo è sulla stessa riga del byte precedente. In questo caso la riga va ricaricata nel buffer.

11 Organizzazione a pipeline

L'organizzazione a multiciclo utilizzata finora è efficace perché ci permette il riuso dell'hardware per svolgere compiti differenti, ma essendo un'organizzazione *completamente sequenziale*, in un istante di tempo, c'è una sola istruzione (microoperazione) in esecuzione nella CPU.

Con un organizzazione multiciclo, il tempo di esecuzione:

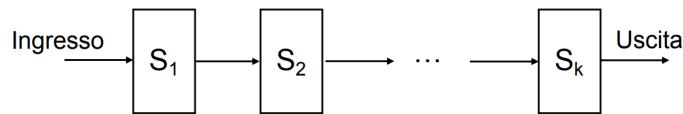
$$T_{esec} = n_{cicli} \cdot T_{clock} \Leftrightarrow T_{esec} = \frac{n_{cicli}}{f_{clock}}$$

con: $n_{cicli} = n_{insn} \cdot CPI$ (numero istruzioni · clock per instruction).

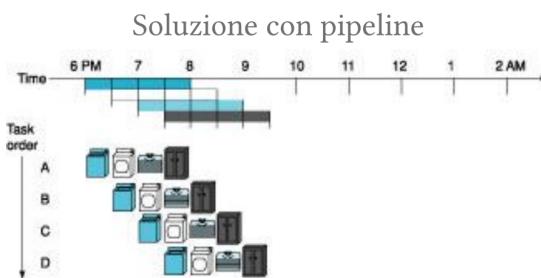
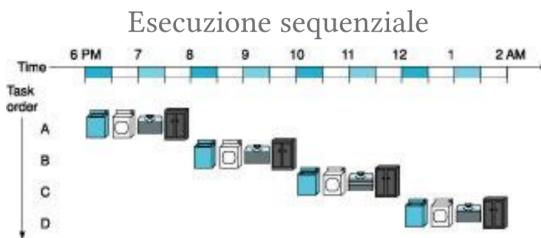
11.1 Pipelining

È una tecnica per progettare un'architettura in grado di eseguire contemporaneamente parti differenti delle istruzioni.

Il lavoro svolto dal processore è suddiviso in passi (stadi della pipeline), connessi sequenzialmente tra loro, che richiedono una frazione di tempo d'esecuzione di un'intera istruzione.



Esempio Fare il bucato con le diverse organizzazioni.



11.2 Set istruzioni del processore z64 a pipeline

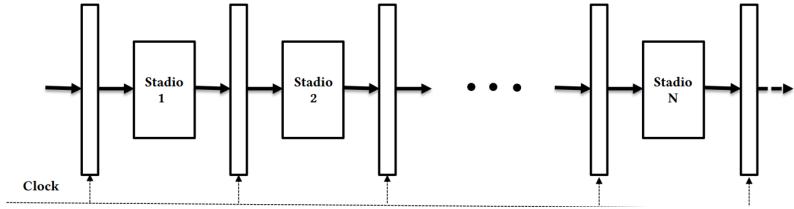
Implementeremo le seguenti istruzioni z64

Class	Instruction	Syntax	Semantics
HW	Non-operational	nop	No operation (beyond the fetch phase)
L/A	Sum	addq %regsorg, %regdest	regdest = regsorg + regdest
	Subtraction	subq %regsorg, %regdest	regdest = regdest - regsorg
	Logic product	andq %regsorg, %regdest	regdest = regsorg and regdest
	Logic sum	orq %regsorg, %regdest	regdest = regsorg or regdest
L/S	Logic negation	notq %register	register = not register
	Loading of word	movq offset(%regbase), %regdest	regdest = memory[offset+regbase]
	Storage of word	movq %regsorg, offset(%regbase)	memory[offset+regbase] = regsorg
J	jump if flag X == 1	jX displacement	if flag X == 1 then RIP = RIP + displacement

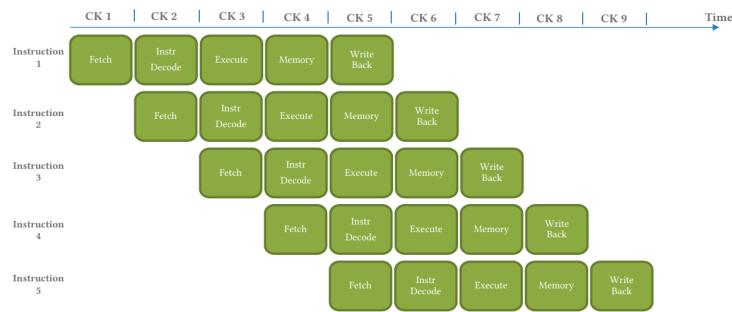
Il formato delle istruzioni è identico a quello dell'organizzazione a multiciclo.

11.3 Schema di principio delle architetture pipeline

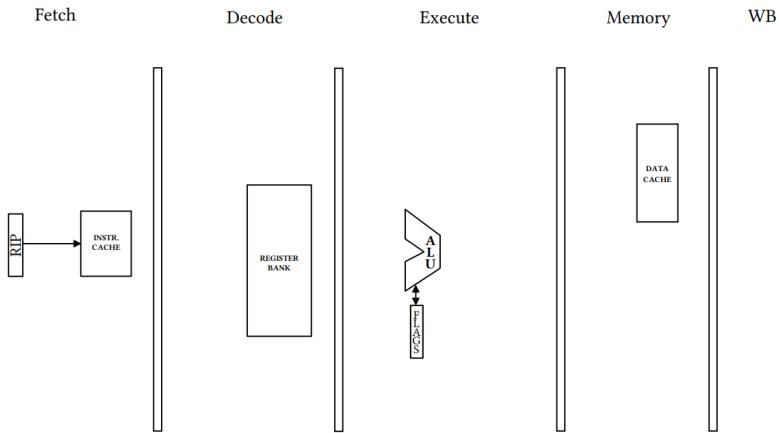
La pipeline è organizzata in N stadi, dove N è il numero di passi elementari necessario ad eseguire tutte le istruzioni (nello z64 $N=5$). Ogni stadio sarà un circuito combinatorio indipendente dagli altri, per rendere gli stadi indipendenti sono necessari dei registri tamponi (registri di pipeline).



11.3.1 Scema temporale dell'esecuzione di più istruzioni



11.4 Progettazione del datapath: elementi di base



- **Fetch:**

- RIP: dobbiamo sapere la prossima istruzione.
- Accesso in memoria: lo facciamo grazie alla cache istruzioni (L1).

- **Decode:**

- Banco registri: mentre abbiamo eseguito la decodifica dell'istruzione possiamo già leggere il registro base, sorgente e destinazione.

- **Execute:**

- ALU: ci serve per effettuare le operazioni logico aritmetiche, aggiornando il registro FLAGS.

- **Memory:**

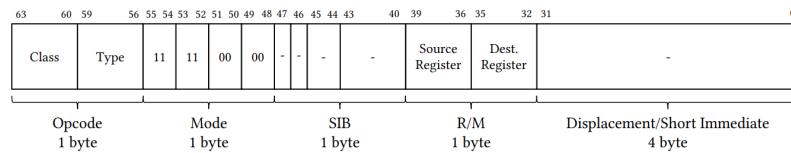
- Cache dati: ci serve per leggere o scrivere dati in memoria dati.

- **Writeback:** scriviamo all'interno del banco dei registri i risultati dell'istruzione.

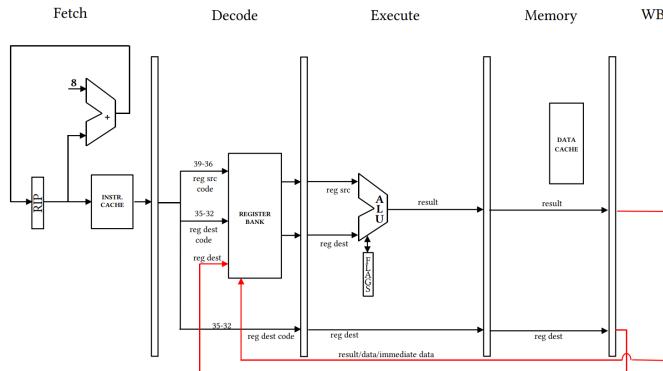
11.5 Istruzioni logico aritmetiche

11.5.1 Formato

- SIB: - (non si possono effettuare operazioni in memoria);
- Displacement: - (non si possono usare costanti);
- SS/DS: 11 (solo operandi a 64 bit);
- DI: 00 (né spiazzamento né dati immediati);
- Mem: 00 (entrambi gli operandi sono registri).



11.5.2 Implementazione



11.5.3 Architettura finale a pipeline

