

Modulo Calcolatori Elettronici

Proff. Gian Luca Marcialis, Giulia Orrù, Lorenzo Putzu, Fabio Roli

Corsi di Laurea in Ingegneria Biomedica Ingegneria Elettrica, Elettronica ed Informatica

Contatti:

marcialis@unica.it, giulia.orrù@unica.it, lorenzo.putzu@unica.it

Capitolo 5

Unità Centrale di Elaborazione, Aritmetica dei Calcolatori

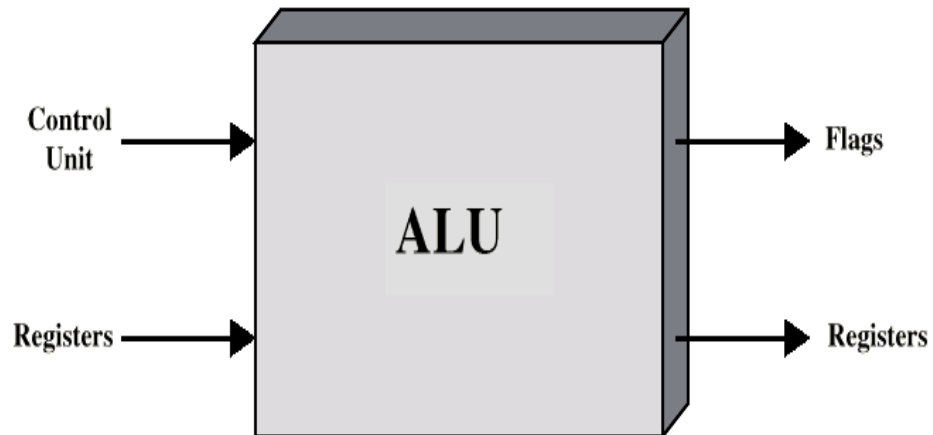
Fonti Principali: Patterson, A.D., Hennessy, J., "Struttura e progetto dei calcolatori elettronici", Zanichelli editore, 2019

Sommario

- L'unità aritmetico-logica (ALU)
- Rappresentazione degli interi
- Aritmetica degli interi
- Rappresentazione in virgola mobile
- Aritmetica dei numeri in virgola mobile
- Hardware della ALU

L'unità aritmetico-logica (ALU)

- Esegue le operazioni aritmetiche e logiche sui dati.
- Gli ingressi alla ALU sono:
 - I dati da elaborare, contenuti in alcuni registri interni.
 - I segnali provenienti dall' **unità di controllo**, che servono a controllare le operazioni svolte dall' ALU e lo spostamento dei dati dentro e fuori dall' ALU.
- Le uscite dell' ALU sono:
 - I risultati dell' operazione svolta, memorizzati in registri interni.
 - Eventuali “flag” (segnali di controllo).



Rappresentazione binaria dei numeri

- Rappresentazione binaria.
 - Si usano le sole cifre “0” e “1”.
- Il segno “ - ” per rappresentare i numeri negativi è rappresentato da un “1” nella posizione più significativa.
- Nel caso dei numeri frazionari la virgola “fissa” è “implicita”.

$$-13.3125_{10} = 11101.0101_2$$

- In generale, se una sequenza di n bit è interpretata come un intero senza segno, il suo valore è espresso attraverso la notazione posizionale:

$$A = \sum_{i=0}^{n-1} 2^i a_i$$

Rappresentazione degli interi in segno e valore

- La rappresentazione in segno e valore è una convenzione per rappresentare i numeri interi negativi.
- Il bit più a sinistra indica il segno del numero.
 - 0 per i numeri positivi
 - 1 per i numeri negativi
 - Es.: $+18 = 00010010$, $-18 = 10010010$.
- Questa rappresentazione è la più ovvia per noi, ma comporta dei problemi per i calcolatori:
 - E' necessario considerare sia il segno che il valore nell'esecuzione delle operazioni aritmetiche.
 - Vi sono due rappresentazioni per lo zero.
 - $+0 = 00000000$, $-0 = 10000000$.

Rappresentazione in complemento a due

- La regola “pratica” con cui si costruisce la rappresentazione in complemento a due di un intero negativo A di n bit è:
 - Si complementano tutti i bit, compreso quello di segno
 - Si considera il risultato come un intero senza segno
 - Si somma **uno** a tale risultato
- Da dove salta fuori questa regola “pratica” ?

Rappresentazione in complemento a due

- Con n bit sappiamo che si possono rappresentare 2^n configurazioni
- Le rappresentazioni dei numeri dipendono dal modo con cui si sceglie di usare queste configurazioni per rappresentare i numeri positivi e negativi

Decimal Representation	Sign-Magnitude Representation
+8	—
+7	0111
+6	0110
+5	0101
+4	0100
+3	0011
+2	0010
+1	0001
+0	0000
-0	1000
-1	1001
-2	1010
-3	1011
-4	1100
-5	1101
-6	1110
-7	1111
-8	—

• Ad esempio nella rappresentazione in segno e valore si è deciso di avere due configurazioni per lo “zero”. E di differenziare i positivi dai negativi semplicemente per il bit più significativo (di segno)

➤ Ma altre scelte sono possibili!

Rappresentazione in complemento a due

- Come vengono assegnate le 2^n configurazioni nel caso del complemento a due, e da dove scaturisce la regola “pratica” di costruzione della rappresentazione in complemento a due?
- Per capirlo bisogna prima di tutto rappresentare un numero A di n bit in questo modo:

$$A = -2^{n-1} a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i$$

➤ Si vede subito che il numero positivo più grande che si può rappresentare è $2^{n-1}-1$, mentre il numero negativo più piccolo è -2^{n-1} . Inoltre si vede subito che i numeri negativi avranno il bit più significativo ad 1. L'intervallo degli interi negativi sarà da -2^{n-1} a -1 .

➤ Ma allora le configurazioni da 111.....1 a 1000 sono assegnate alla rappresentazione dei negativi, le altre a quella dei positivi.

Rappresentazione in complemento a due

$$A = -2^{n-1} a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i$$

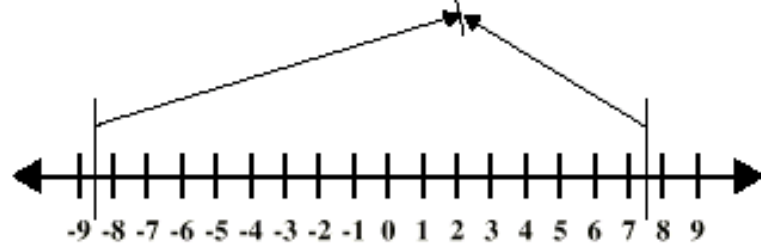
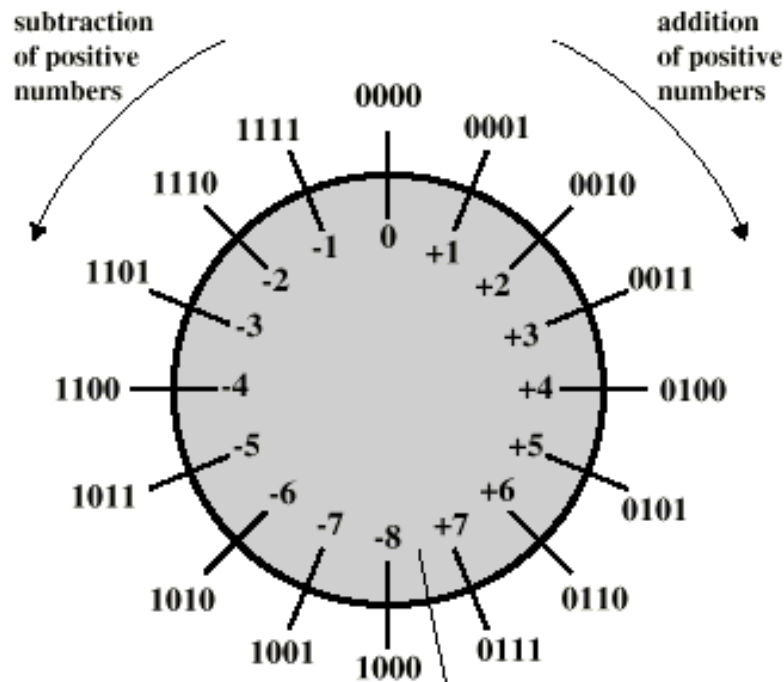
- a_{n-1} è il bit di “segno”
- I positivi hanno la stessa rappresentazione del caso in segno e valore
- Dalla formula si vede subito che quando ho un numero negativo parto da -2^{n-1} e poi vado a decrescere se i restanti $n-1$ bit sono diversi da 0
- Utilizzando questa rappresentazione si hanno i seguenti benefici:
 - Si ha una sola rappresentazione per lo zero.
 - Risultano più semplici l’addizione e la sottrazione (come vedremo in seguito).

Rappresentazione in complemento a due

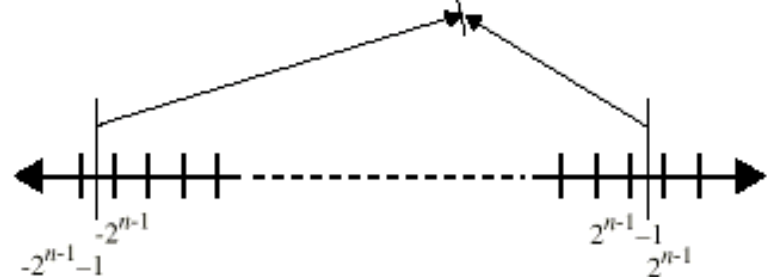
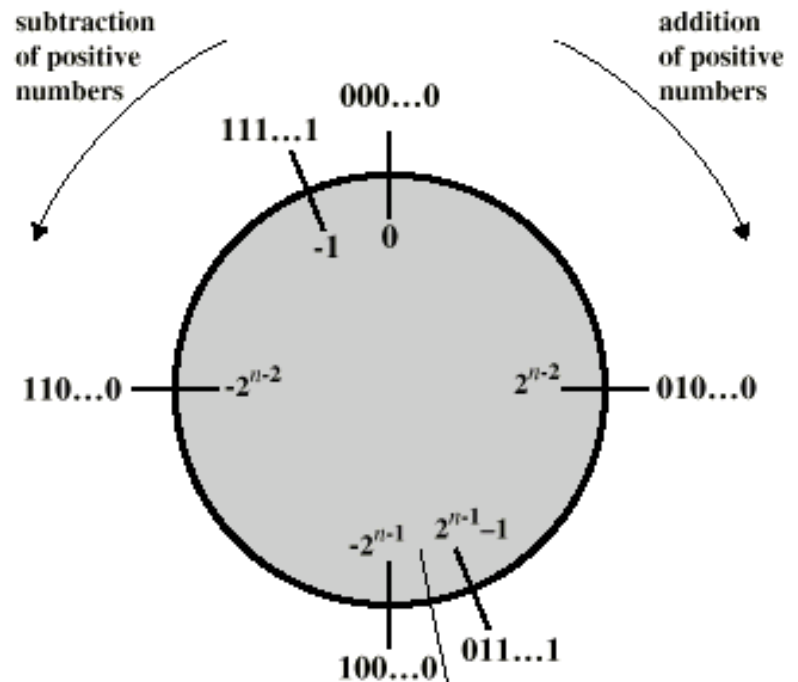
- Il seguente schema aiuta a capire come si calcola un numero negativo di 8 bit.
- La prima riga indica le potenze di 2 in ordine di posizione.
 - L' ultima casella a sinistra è negativa.
- Più “1” ci sono nella sequenza, più il numero si avvicina allo zero.
 - Se la sequenza è formata da soli “1” il valore è “-1”.

-128	64	32	16	8	4	2	1	
1	0	0	0	1	0	1	0	
-128				8		2		= -118

Rappresentazione geometrica



(a) 4-bit numbers



(b) n -bit numbers

Operazioni sui numeri interi: “negazione”

- Con la rappresentazione in segno e valore, il “negativo” di un numero si ottiene invertendo il bit di segno.
- Con la notazione in complemento a due, il “negativo” di un numero si ottiene applicando le seguenti regole:
 - Complemento dei singoli bit
 - Somma di 1 al numero ottenuto.

$$\begin{array}{r} -18 = 11101110 \\ = 00010001 \\ + \quad \quad \quad 1 \\ \hline 00010010 \\ = +18 \end{array}$$

Perché vale la regola del complemento a due

- Il valore di un numero in complemento a due è:

$$A = -2^{n-1} a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i$$

- Secondo la regola, il corrispondente negativo è:

$$B = -2^{n-1} \overline{a_{n-1}} + 1 + \sum_{i=0}^{n-2} 2^i \overline{a_i}$$

- Deve essere $A = -B$, ovvero $A+B = 0$:

$$\begin{aligned} A + B &= -(a_{n-1} + \overline{a_{n-1}})2^{n-1} + 1 + \left(\sum_{i=0}^{n-2} 2^i (a_i + \overline{a_i}) \right) \\ &= -2^{n-1} + 1 + \left(\sum_{i=0}^{n-2} 2^i \right) \\ &= -2^{n-1} + 1 + (2^{n-1} - 1) = 0 \end{aligned}$$

Casi particolari

- Applicando la regola del complemento a due allo zero e troncando al numero di bit usati per la rappresentazione si riottiene la stessa rappresentazione dello zero:

$$\begin{aligned} 0 &= 00000000 = \\ &= 11111111 + 1 = \\ &= (1)00000000 = 0 \end{aligned}$$

- Applicando la regola al numero negativo più grande in valore assoluto, si ottiene il numero stesso:

$$\begin{aligned} -128 &= 10000000 \\ &= 01111111 \\ &\quad + \quad \quad \quad 1 \\ &= 10000000 = -128 \end{aligned}$$

- Ci sono un numero pari di configurazioni (2^n). Una sola per lo zero. E si è deciso di rappresentare più negativi che positivi. Ergo non esiste una configurazione per $+2^{n-1}$

- C'è una sola rappresentazione per lo 0. E non ne esiste una per $+2^{n-1}$.

L' addizione

- Viene eseguita con le regole consuete.
- Si ha **overflow** se sono necessari più bit di quelli a disposizione per rappresentare il risultato.

- Se si verifica, quest'ultimo è sbagliato.
- Si verifica se il segno del risultato discorde con quello dei due addendi.
- Si può avere overflow anche senza riporto dell'ultimo bit.

$\begin{array}{r} 1001 = -7 \\ +0101 = 5 \\ \hline 1110 = -2 \end{array}$ <p>(a) $(-7) + (+5)$</p>	$\begin{array}{r} 1100 = -4 \\ +0100 = 4 \\ \hline 10000 = 0 \end{array}$ <p>(b) $(-4) + (+4)$</p>
$\begin{array}{r} 0011 = 3 \\ +0100 = 4 \\ \hline 0111 = 7 \end{array}$ <p>(c) $(+3) + (+4)$</p>	$\begin{array}{r} 1100 = -4 \\ +1111 = -1 \\ \hline 11011 = -5 \end{array}$ <p>(d) $(-4) + (-1)$</p>
$\begin{array}{r} 0101 = 5 \\ +0100 = 4 \\ \hline 1001 = \text{Overflow} \end{array}$ <p>(e) $(+5) + (+4)$</p>	$\begin{array}{r} 1001 = -7 \\ +1010 = -6 \\ \hline 10011 = \text{Overflow} \end{array}$ <p>(f) $(-7) + (-6)$</p>

La sottrazione

- Si esegue sommando il minuendo al sottraendo negato in complemento a due.

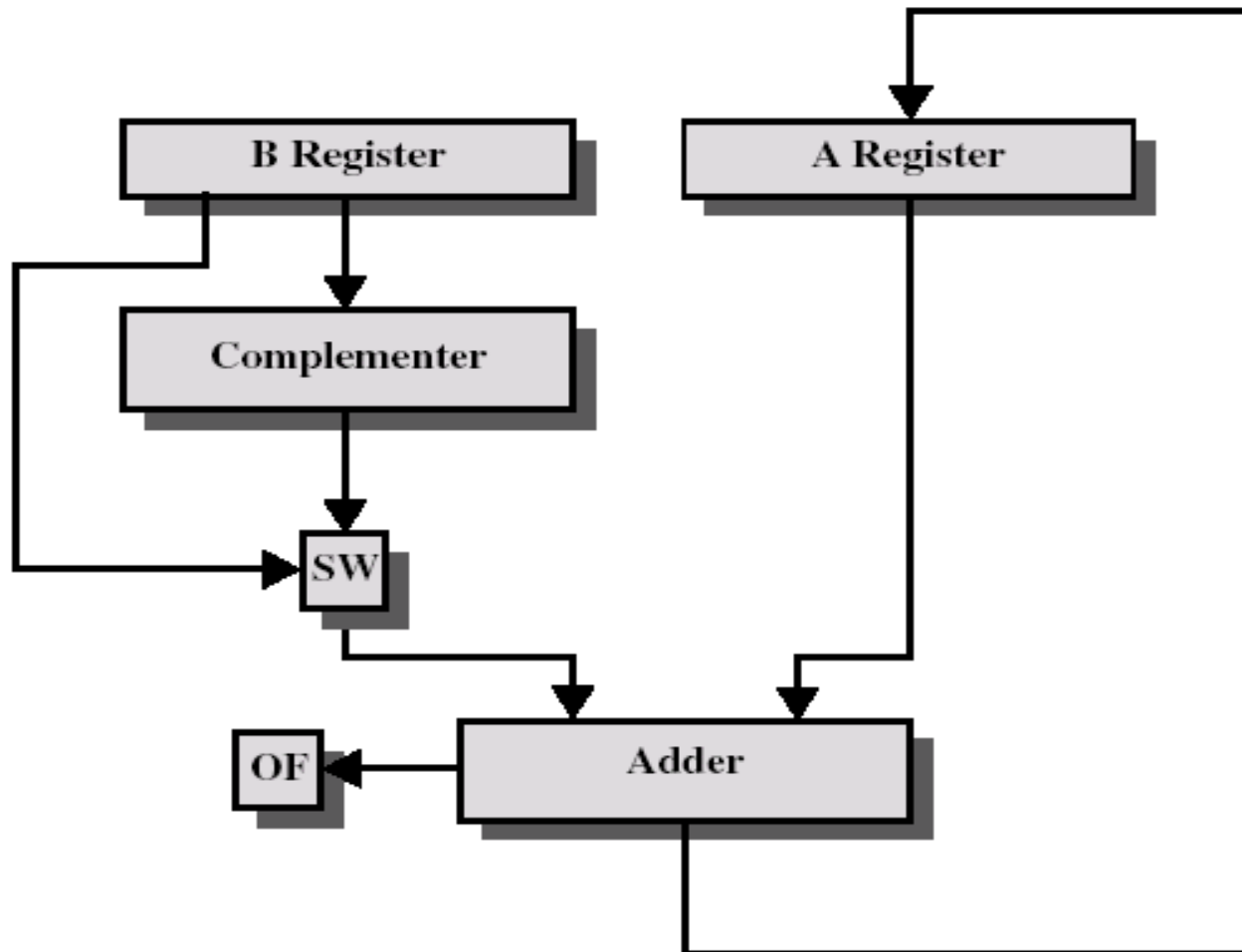
— Cioè: $a - b = a + (-b)$.

— Perciò servono solo i circuiti per l'addizione e per la negazione.

— Quindi i problemi con l'overflow sono gli stessi dell'addizione.

$\begin{array}{r} 0010 = 2 \\ + 1001 = -7 \\ \hline 1011 = -5 \end{array}$ <p>(a) M = 2 = 0010 S = 7 = 0111 -S = 1001</p>	$\begin{array}{r} 0101 = 5 \\ + 1110 = -2 \\ \hline 10011 = 3 \end{array}$ <p>(b) M = 5 = 0101 S = 2 = 0010 -S = 1110</p>
$\begin{array}{r} 1011 = -5 \\ + 1110 = -2 \\ \hline 11001 = -7 \end{array}$ <p>(c) M = -5 = 1011 S = 2 = 0010 -S = 1110</p>	$\begin{array}{r} 0101 = 5 \\ + 0010 = 2 \\ \hline 0111 = 7 \end{array}$ <p>(d) M = 5 = 0101 S = -2 = 1110 -S = 0010</p>
$\begin{array}{r} 0111 = 7 \\ + 0111 = 7 \\ \hline 1110 = \text{Overflow} \end{array}$ <p>(e) M = 7 = 0111 S = -7 = 1001 -S = 0111</p>	$\begin{array}{r} 1010 = -6 \\ + 1100 = -4 \\ \hline 10110 = \text{Overflow} \end{array}$ <p>(f) M = -6 = 1010 S = 4 = 0100 -S = 1100</p>

Schema a blocchi per l'addizione e la sottrazione



Nota: non è vincolante che il risultato sia inserito nel registro A.

Un terzo registro potrebbe essere destinato alla memorizzazione del risultato.

OF = overflow bit

SW = Switch (select addition or subtraction)

Cenni moltiplicazione fra numeri senza segno

- Algoritmo di base:
 - Si generano dei prodotti parziali, uno per ogni cifra del moltiplicatore, che si sommano per ottenere il prodotto finale.
- Per il calcolo di ogni prodotto parziale:
 - Se l' i -esimo bit del moltiplicatore è 0, il prodotto parziale è 0.
 - Se l' i -esimo bit del moltiplicatore è 1, il prodotto parziale è il moltiplicando.
- Per la somma dei prodotti parziali, ciascuno di questi è fatto scorrere di una posizione a sinistra rispetto al precedente prodotto parziale.

Esempio di moltiplicazione

- $4 \times 3 = 12$
- $100 \times 11 = 1100$

$$\begin{array}{r} 100 \times \\ 11 = \\ \hline 100 \\ 100- \\ \hline 1100 \end{array}$$

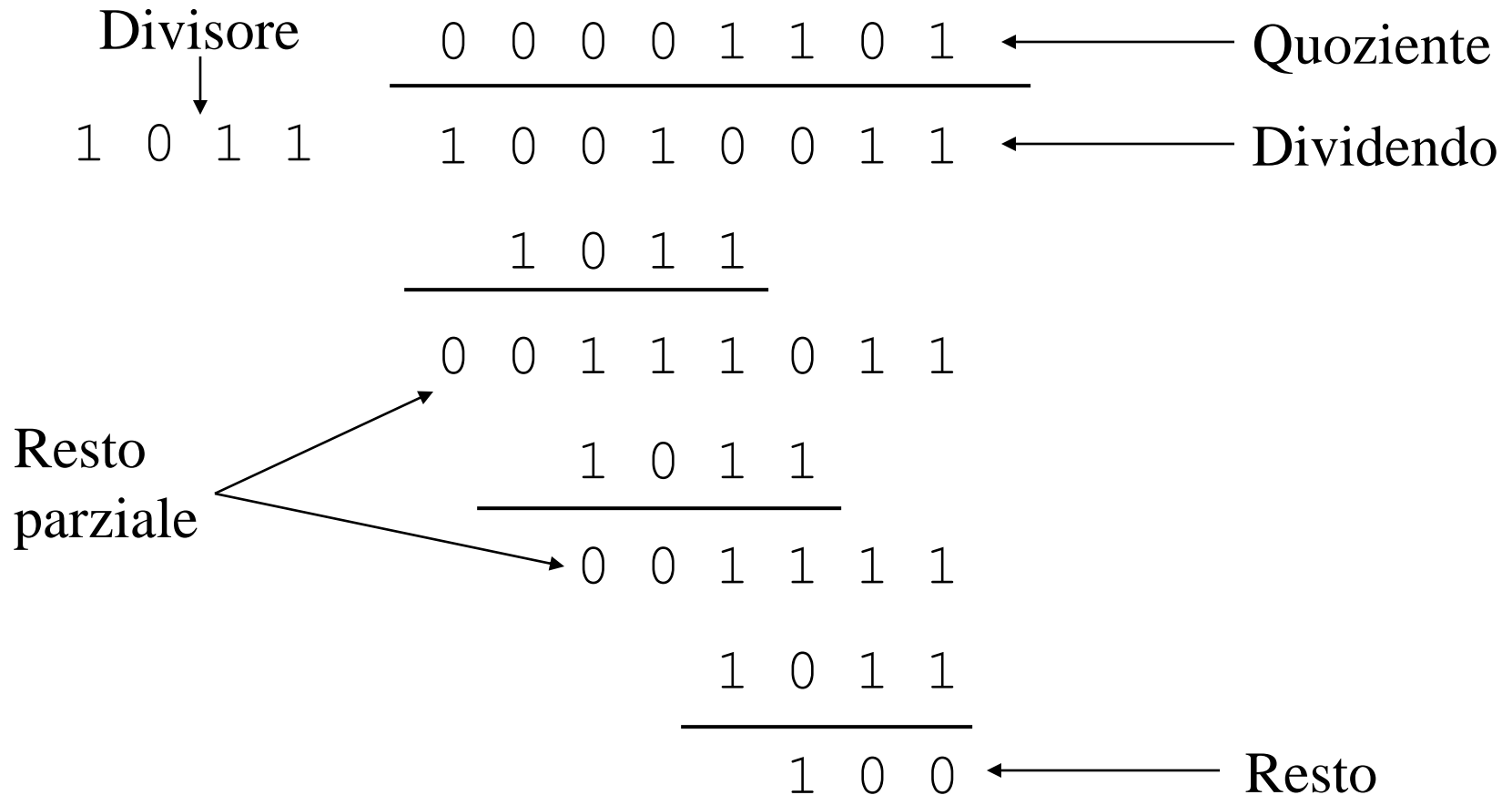
Moltiplicazione di numeri negativi

- L' algoritmo visto non funziona.
 - Infatti, si dovrebbero considerare i prodotti parziali come numeri da $2n$ bit, completati con degli “1” nelle posizioni più a sinistra se uno dei fattori è negativo.
- Soluzione 1
 - Convertire i fattori in numeri positivi.
 - Applicare l' algoritmo per i numeri senza segno.
 - Se i due fattori sono discordi, negare il risultato.
- Soluzione 2
 - Applicare l' algoritmo di Booth (vedi libro di testo)
 - moltiplicazione più rapida
 - non si richiede la negazione del risultato

La divisione: cenni

- Ancora più complessa della moltiplicazione
- L' algoritmo base è quello delle sottrazioni ripetute
- In caso di numeri interi senza segno, l'algoritmo di base è il seguente:
 - I bit del dividendo sono esaminati da sinistra verso destra
 - Si mettono tanti zeri nel quoziente finché non si trova un numero maggiore o uguale al divisore
 - Si mette un "1" nel quoziente
 - Si sottrae il divisore al dividendo parziale
 - Si aggiungono al resto parziale le cifre del dividendo non usate
 - Si ripete finché tutte le cifre del dividendo sono state usate

Esempio di divisione tra interi senza segno



I numeri reali

- Un modo di rappresentare i numeri reali è il “fixed point” (virgola fissa).
 - Il punto di separazione tra parte intera e decimale è in una posizione fissa.
- In questo modo però non si possono rappresentare numeri molto grandi o molto piccoli, a meno di non usare un numero molto grande di bit.

Rappresentazione in “floating point”

- Risulta utile usare la notazione scientifica:
$$\pm S \times B^E$$
- Un numero reale può essere memorizzato con una parola con tre campi:
 - Segno (\pm)
 - Mantissa (S), anche detta “significand”
 - Esponente (E).
- Questa notazione è conosciuta come “floating point” (“virgola mobile”).

Rappresentazione in “floating point”



- Il numero viene normalizzato nella forma $1.bbb...b \times 2^E$.
 - Il valore ‘1’ in evidenza è il primo bit diverso da zero del numero espresso in virgola fissa
 - Poichè questo “1.” è sempre presente lo si considera “implicito”, e si memorizzano solo i bit successivi.
- L’ esponente è scalato rispetto ad un valore costante detto “polarizzazione” o “eccesso”.
 - In genere è pari a $2^{k-1}-1$, dove k è il numero di bit del campo esponente.
 - L’ intervallo di base dell’ esponente è $0 - (2^k-1)$.
 - Si somma la polarizzazione all’ esponente per ottenere il valore reale del campo esponente.
 - Considerando questa polarizzazione, l’intervallo dell’esponente diventa $-(2^{k-1} - 1) - (+ 2^{k-1})$.

Esempio di numero in “floating point”

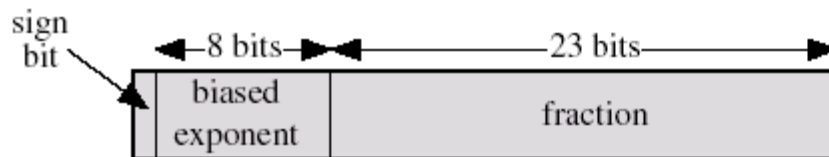
- Esprimere in floating point il numero $53_{(10)}$ sapendo che:
 - La mantissa è di 8 bit
 - L'esponente è di 4 bit, rappresentato in eccesso 7
- Conversione decimale-binario:
 - $53_{(10)} \rightarrow 110101_{(2)}$
- Normalizzazione della mantissa:
 - $110101 \rightarrow 1.\mathbf{10101} * 2^5$
- Rappresentazione dell'esponente in eccesso 7:
 - $5 \rightarrow 101 + 111 = 1100$

S	E	M
0	1100	110101000

- Il bit implicito (in corsivo) si omette in quanto la normalizzazione adottata lo sottintende

IEEE 754

- E' lo standard per la rappresentazione dei numeri in floating point.
- Sono previsti due formati: 32 e 64 bit.
- L' esponente ha 8 e 11 bit rispettivamente
- La mantissa ha 23 e 52 bit rispettivamente.
- Esistono anche dei formati estesi (sia per mantissa che per esponente) che sono usati per i calcoli intermedi.



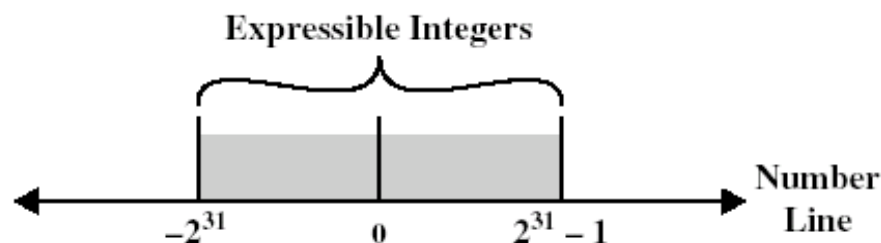
(a) Single format



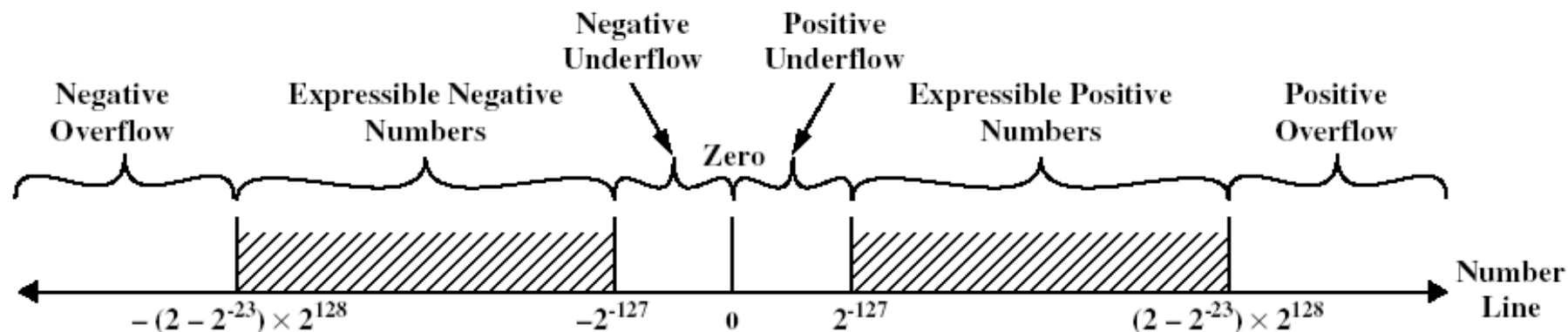
(b) Double format

Valori esprimibili nel formato a 32 bit

- Non vi è una rappresentazione dello zero.
 - Si usa una particolare configurazione per rappresentare lo zero: bit esponente e mantissa tutti a zero



(a) Twos Complement Integers



(b) Floating-Point Numbers

Caratteristiche del IEEE 754 a 32 bit

- Esponente rappresentato in eccesso 127
- Mantissa frazionaria e normalizzata con bit implicito (1.M)

11111111 128

...

10000000 1

01111111 0

...

00000000 -127

- Es.

— $(51)_{10} = 1.10011 \cdot 2^5$

— Mantissa: 10011

— Esponente:

— $5 = 101 + 01111111$
 $= 10000100$

Densità dei numeri in floating point

- I numeri rappresentabili non sono equispaziati.
 - La “densità” maggiore di valori rappresentabili si ha vicino all’origine.
- Un valore reale compreso tra due valori rappresentabili viene approssimato al più vicino
 - Laddove v’è “scarsa densità” abbiamo minore precisione nell’approssimazione
- C’è un legame tra numero di bit, intervallo di valori e precisione.
 - A parità di numero di bit complessivi, aumentando l’intervallo di valori rappresentabili (riducendo i numeri di bit della mantissa a favore dell’esponente), diminuisce la precisione.
 - A parità di intervallo (numero di bit per l’esponente), aumentando il numero di bit complessivi aumenta la precisione.
 - Singola e doppia precisione

Esempio

- Sono dati i seguenti formati per la rappresentazione dei numeri in un calcolatore (campo complessivo 48 bit):
 - Interi senza segno
 - Reali in virgola fissa con bit di segno, parte intera e 16 bit per la parte frazionaria
 - Reali in virgola mobile con bit di segno, esponente in eccesso 63 (7 bit) e mantissa frazionaria e normalizzata in segno e valore 1.M
- Si richiede il minimo e massimo valore positivi, escluso lo zero

Esempio

- Interi senza segno
 - Minimo= 1
 - Massimo= $2^{48} - 1$
- Reali in virgola fissa con bit di segno, parte intera e 16 bit per la parte frazionaria
 - Minimo= 2^{-16}
 - Massimo= $2^{+31} - 2^{-16}$
- Reali in virgola mobile con bit di segno, esponente in eccesso 63 (7 bit) e mantissa frazionaria e normalizzata in segno e valore 1.M
 - Minimo= 2^{-63}
 - Massimo= $2^{+64} (2 - 2^{-40})$

Aritmetica in floating point

- Per addizione/sottrazione si eseguono i seguenti 4 passi:
 - Si controlla se uno dei due operandi è nullo.
 - Si allineano le mantisse (aggiustando gli esponenti).
 - Si esegue l'addizione o la sottrazione delle mantisse.
 - Si normalizza il risultato.
- Moltiplicazione/Divisione sono più semplici

Floating Point Numbers	Arithmetic Operations
$X = X_S \times B^{X_E}$ $Y = Y_S \times B^{Y_E}$	$\left. \begin{aligned} X + Y &= (X_S \times B^{X_E - Y_E} + Y_S) \times B^{Y_E} \\ X - Y &= (X_S \times B^{X_E - Y_E} - Y_S) \times B^{Y_E} \end{aligned} \right\} X_E \leq Y_E$ $X \times Y = (X_S \times Y_S) \times B^{X_E + Y_E}$ $\frac{X}{Y} = \left(\frac{X_S}{Y_S} \right) \times B^{X_E - Y_E}$

Esempio di somma tra due numeri FP

	S	E	M
A	0	1100	1 10101000
B	0	1011	1 11001100

- L'esponente di A è maggiore di quello di B, è necessario allineare B rispetto a A, e poi sommare le mantisse, senza scordarsi del *bit implicito delle rispettive mantisse* (in corsivo):

A	0	1100	1 10101000	+
B	0	1100	0 11100110	=
<hr/>				
Somma	0	1100	10 10001110	

- Poiché il riporto finale (in neretto) è diverso da zero, è necessario incrementare l'esponente per normalizzare la mantissa:

Somma	0	1101	1 01000111
--------------	---	------	------------

Requisiti elementari dell'hardware di una ALU

- Registri per conservare gli operandi in ingresso e parziali
 - registri a scorrimento
- Bit di controllo per controllare quali operazioni svolgere
- MUX o opportune reti combinatorie per l'interpretazione dei bit di controllo
- Una rete combinatoria che effettui le operazioni (di solito un Parallel Adder)

Floating Point Unit di una ALU

- Serve per effettuare le operazioni fra numeri in virgola mobile
- Realizzata con due unità aritmetiche in virgola fissa, una per l'esponente e una per la mantissa, che vengono accoppiate
- L'unità per la mantissa si occupa di eseguire le operazioni aritmetiche di base sulla mantissa (somma algebrica, moltiplicazione e divisione).
- L'unità per l'esponente deve eseguire solo operazioni di somma algebrica e di confronto fra numeri interi
 - Il confronto può essere effettuato attraverso una sottrazione degli esponenti

Floating Point Unit di una ALU

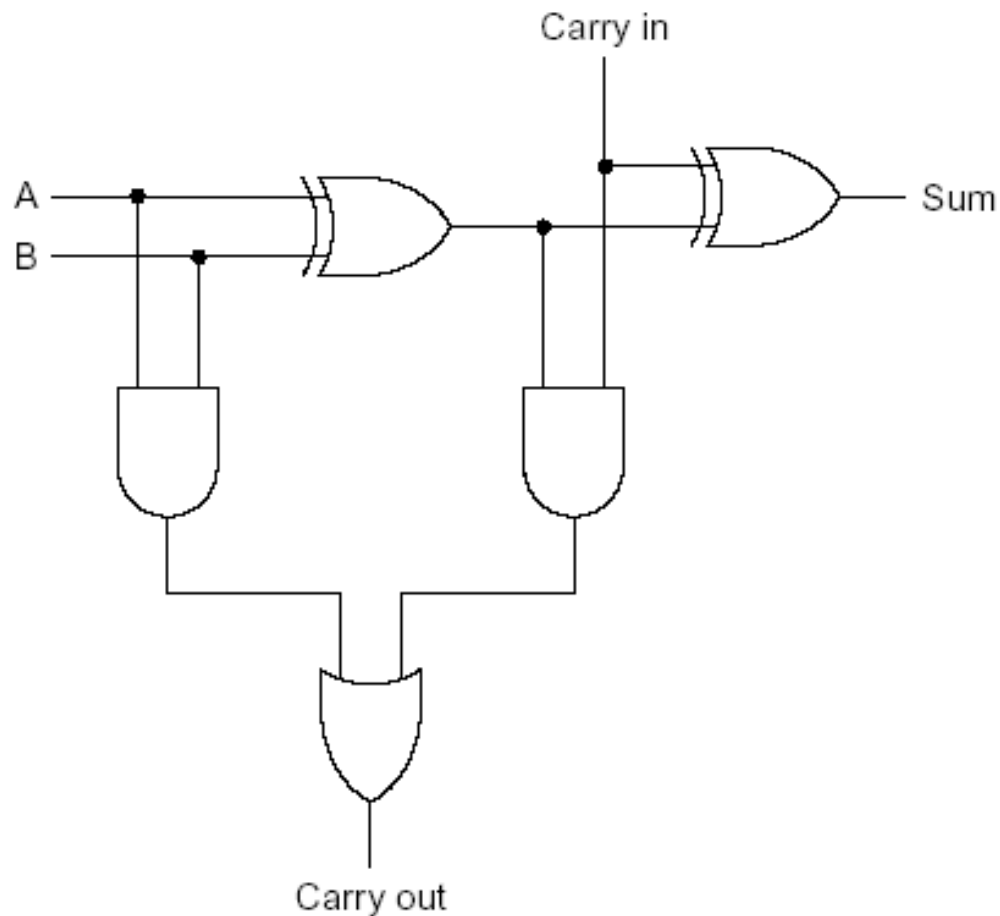
- Per eseguire la somma di due numeri in FP si fa la differenza degli esponenti.
 - Il segno del risultato indica quale dei due esponenti è il più piccolo
 - Il modulo indica il numero di scorrimenti verso destra che devono essere eseguiti sulla mantissa del numero più piccolo
- Il registro che contiene la mantissa deve dunque essere del tipo a scorrimento
- Gli scorrimenti possono essere pilotati da un contatore caricato con la differenza fra gli esponenti
 - ad ogni scorrimento il contatore viene decrementato finché non viene raggiunto il valore zero

I Sommatore di un modulo ALU

- Il sommatore elementare è noto come “half adder”: dati due bit in ingresso, presenta in uscita la somma e il riporto.
- Il passo successivo è il “full adder”, per cui, dati in ingresso i due bit e il riporto, calcola in uscita somma e riporto
- Connettendo in serie n full adder attraverso le linee del riporto, è possibile effettuare la somma di due operandi a n bit: si ottiene il cosiddetto “parallel adder”

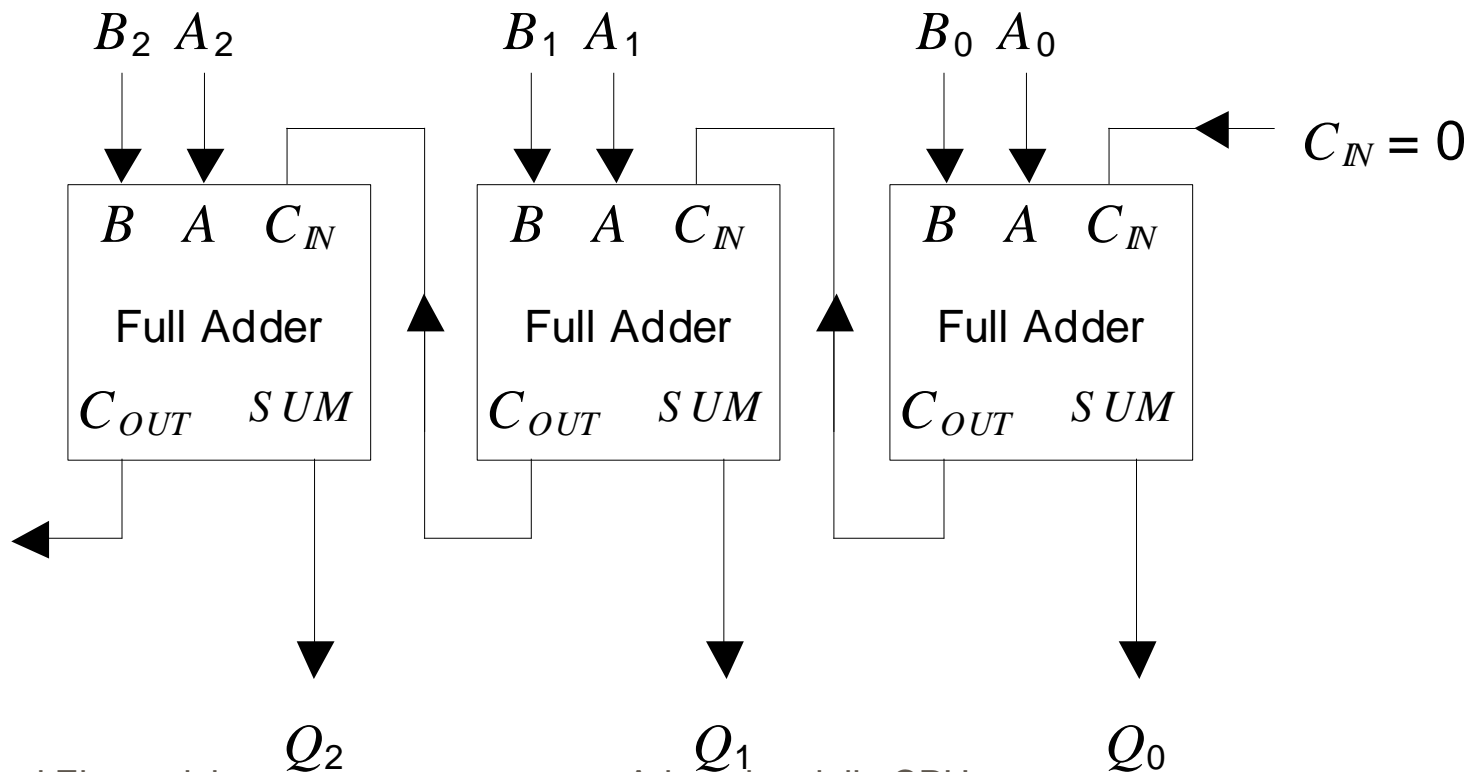
Schema logico di un full adder

A	B	Carry in	Sum	Carry out
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



Il Parallel Adder (o Ripple Carry Adder)

- La somma complessiva dipende dal tempo che ogni singolo full adder impiega a generare il riporto per lo stadio successivo (ritardo)
- Se δ è il ritardo del singolo modulo, il ritardo impiegato da un Parallel Adder a n bit è dato da $n \delta$



Schema di base di una ALU a n bit

