Codice BCD
Prima di passare alla rappresentazione dei numeri relativi in binario vediamo un tipo di codifica che ha una certa rilevanza in alcune applicazioni: il codice *BCD* (*Binary Coded Decimal*).
È un codice con le seguenti caratteristiche:

1. ciascuna cifra decimale è separatamente codificata in binario
2. ogni cifra corrisponde a 4 bit

Dato un numero decimale invece di tradurlo complessivamente viene decomposto cifra per cifra. Ognuna di tali cifre, variabili fra 0 e 9, viene fatta corrispondere a 4 bit. Tale codifica è utilizzata in molti calcolatori tascabili o in alcune bilance elettroniche e, in generale, in tutte le applicazioni in cui può essere conveniente una codifica interna che sia la più vicina possibile a quella che è la rappresentazione in cifre decimali.

Esempi
Si vuole rappresentare il numero $139_{(10)}$ in BCD:

Decimale	1	3	9
BCD	0001	0011	1001

utilizzando in totale 12 bit (a gruppi di 4).

Si vuole rappresentare il numero $3408_{(10)}$ in BCD:

Decimale	3	4	0	8
BCD	0011	0100	0000	1000

Il numero di bit necessari per rappresentare un certo numero nel sistema binario è minore o al più eguale del numero di bit necessari per rappresentare lo stesso numero in BCD.

Rappresentazione dei numeri relativi
Fino ad ora abbiamo trattato numeri interi positivi, ma per le esigenze di calcolo è importante saper rappresentare numeri relativi. Esistono almeno 3 modi per rappresentare i numeri relativi:

1. rappresentazione in *modulo e segno*
2. rappresentazione in *complemento alla base* (normalmente chiamata *complemento a 2*)
3. rappresentazione in *complemento alla base diminuita* (chiamata *complemento a 1*)

Di fatto tutti gli elaboratori (con pochissime eccezioni) utilizzano, per la rappresentazione dei numeri relativi, la rappresentazione in complemento a due.

Modulo e segno
È la più facile da intuire ed è quella che ha una corrispondenza diretta con la nostra usuale rappresentazione dei numeri. Noi anteponiamo al valore assoluto di un numero un simbolo che denota se esso è positivo o negativo. Utilizziamo 2 entità: il *modulo* ed il *segno*. Ad esempio +3 e -3 hanno stesso modulo (3) ma segno differente.
Tenendo presente che il segno deve essere rappresentato in binario possiamo facilmente intuire come ottenere un numero binario relativo. Stabilito il numero di bit, il bit più significativo ha valore di segno:

1. se 0 il segno è +
2. se 1 il segno è –

È evidente che se utilizziamo *n* bit, uno verrà sacrificato per rappresentare il segno. Si noti che anche in questo caso è fondamentale stabilire il numero di bit che si utilizzano.

Esempi:

Usando 4 bit: $1010_{(MS)} \rightarrow$ segno -, modulo $010 = 2_{(10)} \rightarrow -2_{(10)}$
Usando 4 bit: $0110_{(MS)} \rightarrow$ segno +, modulo $110 = 6_{(10)} \rightarrow 6_{(10)}$
Usando 5 bit: $11011_{(MS)} \rightarrow$ segno -, modulo $1011 = 11_{(10)} \rightarrow -11_{(10)}$

Complemento alla base
Stabilito il numero di bit, supponiamo *n*, il bit più significativo ha *peso*: $-(2^{n-1})$
Sulla base di questa considerazione possiamo riscrivere quella formula che, data la sequenza di bit, mi esprimeva il valore vero del numero: $N_V = - d_{n-1}\, 2^{n-1} + d_{n-2}\, 2^{n-2} + ... + d_0\, 2^0$

Data una sequenza di bit che rappresenta un numero relativo in complemento a due posso ricavare il valore vero del numero sulla base della relazione appena esposta.

Esempi:

$N = 0101_{(C2)}$ $\quad N_V = -0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 4 + 1 = 5$
$N = 1011_{(C2)}$ $\quad N_V = -1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = -8 + 2 + 1 = -5$

Sulla base degli esempi possiamo osservare che anche nella rappresentazione in complemento a 2, così come valeva per la rappresentazione in modulo e segno, il primo bit mi dice se il numero è positivo o negativo, in particolare se il primo bit è 0 il numero è positivo, se il primo bit è 1 il numero è negativo.

Complemento alla base diminuita
Stabilito il numero di bit, supponiamo *n*, il bit più significativo ha *peso*: $-(2^{n-1}-1)$
Sulla base di questa considerazione possiamo riscrivere quella formula che, data la sequenza di bit, mi esprimeva il valore vero del numero: $N_V = - d_{n-1}\, (2^{n-1}-1) + d_{n-2}\, 2^{n-2} + ... + d_0\, 2^0$

Data una sequenza di bit che rappresenta un numero relativo in complemento a 1 posso ricavare il valore vero del numero sulla base della relazione appena esposta.

Esempi:

$N = 0101_{(C1)}$ $\quad N_V = -0 \times (2^3-1) + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 4 + 1 = 5$
$N = 1011_{(C1)}$ $\quad N_V = -1 \times (2^3-1) + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = -7 + 2 + 1 = -4$

Anche nella rappresentazione in complemento a 1 il primo bit mi dice se il numero è positivo o negativo, in particolare se il primo bit è 0 il numero è positivo, se il primo bit è 1 il numero è non positivo (ossia negativo o nullo).

Calcolo dell'opposto
Introduciamo un'*operazione unaria* detta *complementazione* o *negazione*. Essendo il sistema binario a due cifre (0 e 1) negare un bit vuol dire scambiare 0 con 1 o viceversa. La complementazione si può applicare a tutti i bit che compongono il numero.
Vediamo com'è possibile calcolare l'opposto di un numero nelle tre rappresentazioni appena viste.

Modulo e segno
Basta complementare il bit di segno.

Esempi:

$-2_{(10)} = 1010_{(MS)} \rightarrow 0010_{(MS)} = 2_{(10)}$
$+6_{(10)} = 0110_{(MS)} \rightarrow 1110_{(MS)} = -6_{(10)}$

Complemento alla base
La regola che mi consente di ottenere l'opposto è:
1. si complementano tutti i bit
2. al numero ottenuto si somma 1

Esempi:

$5_{(10)} = 0101_{(C2)} \rightarrow 1010 + 1 = 1011_{(C2)} = -8 + 2 + 1 = -5_{(10)}$
$-7_{(10)} = 1001_{(C2)} \rightarrow 0110 + 1 = 0111_{(C2)} = 3 + 2 + 1 = 7_{(10)}$

Attenzione ad un caso particolare:
Se il numero è negativo con tutti gli altri bit nulli non è possibile ottenere l'opposto:

$-8_{(10)} = 1000_{(C2)} \rightarrow 0111 + 1 = 1000_{(C2)} = -8_{(10)}$

Vedremo che questo problema è legato all'intervallo di rappresentazione del complemento alla base.

Complemento alla base diminuita
Per ottenere l'opposto basta complementare tutti i bit

Esempi:

$5_{(10)} = 0101_{(C1)} \rightarrow 1010_{(C1)} = -7 + 2 = -5_{(10)}$
$-4_{(10)} = 1011_{(C1)} \rightarrow 0100_{(C1)} = 4_{(10)}$

Conversione di un numero relativo dalla base 10 alla base 2
Partendo da un numero decimale relativo si vuole ottenere il binario in uno dei tre tipi di rappresentazione: modulo e segno, complemento alla base e complemento alla base diminuita.

Numero decimale positivo
Se il numero decimale è positivo per il modulo e segno, il complemento a due e il complemento a uno basta applicare semplicemente o il metodo delle potenze o l'algoritmo per divisioni, assicurandosi che il numero ottenuto rispetti la quantità di bit stabilita.

Esempi:

Usando 4 bit:
$5_{(10)}$ → algoritmo per divisioni → 101 → $0101_{(MS)} = 0101_{(C2)} = 0101_{(C1)}$
$8_{(10)}$ → algoritmo per divisioni → 1000 → non va bene perché coinvolge il bit più significativo

Usando 5 bit:
$5_{(10)}$ → algoritmo per divisioni → 101 → $00101_{(MS)} = 00101_{(C2)} = 00101_{(C1)}$
$8_{(10)}$ → algoritmo per divisioni → 1000 → $01000_{(MS)} = 01000_{(C2)} = 01000_{(C1)}$

Numero decimale negativo
Se il numero decimale è negativo si considerare il suo valore assoluto e applicare o il metodo delle potenze o l'algoritmo per divisioni, assicurandosi sempre che il numero ottenuto rispetti la quantità di bit stabilita. Ricordando che il binario ottenuto è positivo, si deve calcolare l'opposto. Vediamo degli esempi.

Modulo e segno

Usando 4 bit:
$-5_{(10)}$ → $5_{(10)}$ → algoritmo per divisioni → 101 → $0101_{(MS)}$ → opposto → $1101_{(MS)}$
$-8_{(10)}$ → $8_{(10)}$ → algoritmo per divisioni → 1000 → non va bene
Usando 5 bit:
$-5_{(10)}$ → $5_{(10)}$ → algoritmo per divisioni → 101 → $00101_{(MS)}$ → opposto → $10101_{(MS)}$
$-8_{(10)}$ → $8_{(10)}$ → algoritmo per divisioni → 1000 → $01000_{(MS)}$ → opposto → $11000_{(MS)}$

Complemento alla base

Usando 4 bit:
$-5_{(10)}$ → $5_{(10)}$ → algoritmo per divisioni → 101 → $0101_{(C2)}$ → opposto → $1011_{(C2)}$
$-9_{(10)}$ → $9_{(10)}$ → algoritmo per divisioni → 1001 → non va bene
Caso particolare: $-8_{(10)}$ → $1000_{(C2)}$

Usando 5 bit:
$-5_{(10)}$ → $5_{(10)}$ → algoritmo per divisioni → 101 → $00101_{(C2)}$ → opposto → $11011_{(C2)}$
$-9_{(10)}$ → $9_{(10)}$ → algoritmo per divisioni → 1001 → $01001_{(C2)}$ → opposto → $10111_{(C2)}$

Complemento alla base diminuita

Usando 4 bit:
$-5_{(10)}$ → $5_{(10)}$ → algoritmo per divisioni → 101 → $0101_{(C1)}$ → opposto → $1010_{(C1)}$
$-9_{(10)}$ → $9_{(10)}$ → algoritmo per divisioni → 1001 → non va bene

Usando 5 bit:
$-5_{(10)}$ → $5_{(10)}$ → algoritmo per divisioni → 101 → $00101_{(C1)}$ → opposto → $11010_{(C1)}$
$-9_{(10)}$ → $9_{(10)}$ → algoritmo per divisioni → 1001 → $01001_{(C1)}$ → opposto → $10110_{(C1)}$

Intervallo di rappresentazione dei numeri relativi binari
Riferiamoci ai numeri relativi interi: possono essere rappresentati o in modulo e segno o in complemento a 2 o in complemento a 1. Vediamo nella tabella seguente i valori rappresentabili utilizzando 4 bit. Ci attendiamo che ad ogni combinazione 0/1 dei bit corrisponda un diverso valore.

Numero decimale relativo	Modulo e segno	Complemento a 1	Complemento a 2
+7	0111	0111	0111
+6	0110	0110	0110
+5	0101	0101	0101
+4	0100	0100	0100
+3	0011	0011	0011
+2	0010	0010	0010
+1	0001	0001	0001
0	0000 1000	0000 1111	0000
-1	1001	1110	1111
-2	1010	1101	1110
-3	1011	1100	1101
-4	1100	1011	1100
-5	1101	1010	1011
-6	1110	1001	1010
-7	1111	1000	1001
-8	-	-	1000

I numeri positivi hanno la stessa codifica in tutte e tre le rappresentazioni. Vediamo che in modulo e segno vi sono due zeri $0000_{(MS)} = 1000_{(MS)}$, così come succede anche per il complemento a 1: $0000_{(C1)} = 1111_{(C1)}$. Ne consegue che complemento a uno e modulo e segno associano alle $2^4 = 16$ possibili combinazioni 0/1 solo 15 numeri. Ciò non giova all'elaborazione automatica dove è opportuno avere una corrispondenza biunivoca tra entità e sua rappresentazione. I numeri negativi sono rappresentati in forme e regole diverse secondo il tipo di rappresentazione. Infine il numero -8 non può essere rappresentato in modulo e segno e in complemento a 1, utilizzando 4 bit.
Nelle rappresentazioni modulo e segno e complemento a 1, l'intervallo di rappresentazione è simmetrico (da +7 a -7); nel caso del complemento a 2 si va da +7 a -8. Ciò fa intuire il vantaggio di utilizzare la rappresentazione in complemento alla base.

In conclusione, con *n* bit (n-1, n-2, ..., 1, 0) l'intervallo di rappresentazione è:

complemento a 2 → $[-(2^{n-1}), +2^{n-1}-1]$
modulo e segno e complemento a 1 → $[-(2^{n-1}-1), +2^{n-1}-1]$

Esempi:

Usando 3 bit:
Per modulo e segno e complemento a 1: $[-(2^{3-1}-1) , +2^{3-1}-1] = [-3, 3]$
Per complemento a 2: $[-(2^{3-1}) , +2^{3-1}-1] = [-4, 3]$

Usando 8 bit:
Per modulo e segno e complemento a 1: $[-(2^{8-1}-1) , +2^{8-1}-1] = [-127, 127]$
Per complemento a 2: $[-(2^{8-1}) , +2^{8-1}-1] = [-128, 127]$

Usando 6 bit:
Per modulo e segno e complemento a 1: $[-(2^{6-1}-1) , +2^{6-1}-1] = [-31, 31]$
Per complemento a 2: $[-(2^{8-1}) , +2^{8-1}-1] = [-32, 31]$

Vantaggi del complemento a due
Chiediamoci adesso perché i calcolatori utilizzano al loro interno la notazione in complemento a 2.

Corrispondenza univoca fra rappresentazione binaria e numero decimale
Come abbiamo visto prima il complemento a due è l'unico sistema di rappresentazione che con *n* bit esprime esattamente 2^n numeri interi. È evidente il vantaggio di associare ad ogni combinazione 0/1 un valore univoco.

Semplificazione dei circuiti
Un altro vantaggio è che la rappresentazione in complemento alla base semplifica la realizzazione dei circuiti. Si supponga di essere in modulo e segno e di rappresentare in questa notazione i numeri +5 e –5, utilizzando 4 bit:

$$\begin{array}{ll} +5 & 0101_{(MS)} \\ -5 & 1101_{(MS)} \end{array}$$

supponiamo di volerli sommare: applichiamo le regole viste circa la somma dei numeri binari:

$$\begin{array}{l} 0101+ \\ \underline{1101=} \\ 0010_{(MS)} \end{array}$$

Il risultato è +2. Dove è l'errore? Abbiamo applicato le regole algebriche di somma bit a bit tra numeri binari senza tenere conto che il bit più significativo ha la funzione di segno. L'errore è stato quello di considerare i numeri 0101 e 1101 come numeri binari assoluti. In effetti se dovessimo effettuare l'operazione correttamente dovremmo scorporare il primo bit dagli altri e non potremmo applicare le regole di somma tra numeri binari opposti in segno. Dovremmo ricordarci che la somma tra numeri di segno opposto di fatto corrisponde alla differenza. Tutti questi ragionamenti dobbiamo farli effettuare ad un elaboratore. Considerazioni analoghe valgono anche per il complemento alla base diminuita. Consideriamo, invece, la rappresentazione in complemento a due, scriviamo il numero +5 ed il numero –5, utilizzando 4 bit, e sommiamoli:

$$\begin{array}{ll} +5 & 0101+ \\ -5 & \underline{1011=} \\ & 0000_{(C2)} \end{array}$$

Tale somma viene 0, esattamente come ci si aspetta. In effetti in notazione complemento a 2 si possono effettuare le operazioni aritmetiche applicando le regole usuali dell'algebra binaria senza dover scorporare il bit di segno. Da un punto di vista dell'architettura del calcolatore i circuiti saranno sempre gli stessi trattando tutti i bit allo stesso modo e semplificando enormemente la loro realizzazione e il costo finale dell'elaboratore. Nel caso del modulo e segno avremmo dovuto avere un circuito che trattava il modulo e uno che trattava il segno, e quindi il circuito sarebbe stato complessivamente più critico, più complesso.

Riduzione del numero di circuiti
Poiché in complemento a due i circuiti trattano tutti i bit allo stesso modo si può ad esempio ricondurre la sottrazione sempre ad una somma. Si supponga di voler eseguire l'operazione: +5 - (+3). Dovremmo avere un circuito che realizza la sottrazione. In realtà la sottrazione può essere effettuata come una somma: +5 + (-3), basta individuare l'opposto del secondo addendo.

Ne consegue che possiamo eliminare il circuito che fa la sottrazione dotando gli elaboratori di un circuito che fa la somma e di uno che fa il complemento. Quest'ultimo circuito è sicuramente molto più semplice del circuito che fa la sottrazione.

5- 3= 2	→	0101- 0011=	→	0101+ 1101= 0010

Da quanto visto si evince l'estrema convenienza nell'utilizzare all'interno dell'elaboratore la rappresentazione in complemento a due.

Controllo dell'overflow error
Avendo un numero di bit limitato per rappresentare i valori è necessario controllare cosa succede se l'operazione causa la necessità di utilizzare un numero di bit superiore di quelli a disposizione. Potrebbe generarsi quello che viene chiamato *overflow error* o *overflow di tipo aritmetico*. Supponiamo in complemento a due di voler sommare 0111 (ossia 7) ad 1. Il risultato è 1000 apparentemente giusto, ma in realtà per la regola del valore vero associato al complemento alla base il numero 1000 vale –8. Questo è un errore connesso all'overflow. Vediamo altri esempi.

Sia $n = 4$ e consideriamo i numeri:
$-5_{(10)} = 1011_{(C2)}$
$+2_{(10)} = 0010_{(C2)}$
la somma è $1101_{(C2)}$ ossia $-3_{(10)}$. I numeri esprimibili in complemento a 2 utilizzando 4 bit sono quelli compresi fra -8 e +7 e poiché -3 è compreso in tale intervallo non si ha alcuna problema.

Sempre sotto l'ipotesi che sia $n = 4$ consideriamo i numeri:
$+5_{(10)} = 0101_{(C2)}$
$+4_{(10)} = 0100_{(C2)}$
la somma è $1001_{(C2)}$ ossia $-7_{(10)}$. La somma di due numeri positivi dà un numero negativo. Questa è una situazione di overflow, infatti, il numero +9 non rientra nell'intervallo [-8, 7].
Controllare l'overflow error in complemento a due è relativamente semplice: si ha overflow quando gli ultimi due riporti (quello sul bit più significativo e quello che va sul bit successivo) sono diversi fra loro (01 oppure 10).

Vediamo qualche esempio:

0010 –5+ 1011+ +2= 0010= –3 1101 **No overflow**	**01**00 +5+ 0101+ +4= 0100= +9 1001 **Overflow**	**11**00 +4+ 0100+ –3= 1101= +1 0001 **No overflow**	**10**00 –2+ 1110+ –7= 1001= –9 0111 **Overflow**