

# Soluzioni degli esercizi

---

## Esercizi capitolo 14 - Rappresentazione dei dati

### Valore dec di 0xD6 — senza segno

Usiamo la tabella di conversione da hex a nibble:  $0xD6 = 0b11010110$ .

Non c'è segno, quindi ogni bit conta come una potenza di 2, in ordine da  $2^0$  a destra, fino a  $2^7$  a sinistra.

I valori da sommare sono:  $2 + 4 + 16 + 64 + 128 = 214$  (dec).

### Valore di 0x6D — senza segno

Usiamo la tabella di conversione da hex a nibble:  $0x6D = 0b01101101$ .

Non c'è segno, quindi ogni bit conta come una potenza di 2, in ordine da  $2^0$  a destra, fino a  $2^7$  a sinistra.

I valori da sommare sono:  $1 + 4 + 8 + 32 + 64 = 109$  (dec).

### Valore di 0xC7 — senza segno

Come sopra: 199 (dec).

### Valore di 0x39 — senza segno

Come sopra: 57 (dec).

### Rappresentazione di 214 — 8 bit senza segno

Dividiamo il numero per 2, ripetutamente, segnandoci il resto.

214	0
107	1
53	1
26	0
13	1
6	0
3	1
1	1
0	

I resti corrispondono ai bit della rappresentazione, in alto il bit più a destra, in basso il bit più a sinistra: 0b11010110.

Usiamo la tabella di conversione da hex a nibble: 0b11010110 = 0xD6.

In **alternativa**, possiamo procedere per confronti e sottrazioni, bit a bit da sinistra a destra: 

1	1	0	1	0	1	1	0
---	---	---	---	---	---	---	---

.

1	$128 \leq 214 \implies n = 214 - 128 = 86$
1	$64 \leq 86 \implies n = 86 - 64 = 22$
0	$32 > 22$
1	$16 \leq 22 \implies n = 22 - 16 = 6$
0	$8 > 6$
1	$4 \leq 6 \implies n = 6 - 4 = 2$
1	$2 \leq 2 \implies n = 2 - 2 = 0$
0	$1 > 0$

## Rappresentazione di 109 — 8 bit senza segno

Dividiamo il numero per 2, ripetutamente, segnandoci il resto.

109	1
54	0
27	1
13	1
6	0
3	1
1	1
0	

I resti corrispondono ai bit della rappresentazione, in alto il bit più a destra, in basso il bit più a sinistra: 0b01101101. Abbiamo aggiunto a sinistra gli zeri necessari per completare gli 8 bit della rappresentazione.

Usiamo la tabella di conversione da hex a nibble: 0b01101101 = 0x6D.

In **alternativa**, possiamo procedere per confronti e sottrazioni, bit a bit da sinistra a destra: 

0	1	1	0	1	1	0	1
---	---	---	---	---	---	---	---

.

0	$128 > 109$
1	$64 \leq 109 \implies n = 109 - 64 = 45$
1	$32 \leq 45 \implies n = 45 - 32 = 13$
0	$16 > 13$
1	$8 \leq 13 \implies n = 13 - 8 = 5$
1	$4 \leq 5 \implies n = 5 - 4 = 1$
0	$2 > 1$
1	$1 \leq 1$

Rappresentazione di 199 — 8 bit senza segno

Come sopra: 0xC7.

Rappresentazione di 57 — 8 bit senza segno

Come sopra: 0x39.

Valore dec di 0xD6 — complemento a 2

Usiamo la tabella di conversione da hex a nibble: 0xD6 = 0b11010110.

Il bit più alto (a sx) è a 1, quindi il valore è negativo. Per scoprire il modulo, bisogna calcolare il complemento a 2: prima facciamo il complemento bit a bit e poi sommiamo 1.

```
11010110 ¬
-----
00101001 +
      1 =
-----
00101010
```

A questo punto, ogni bit conta come una potenza di 2, in ordine da 2<sup>0</sup> a destra, fino a 2<sup>7</sup> a sinistra.

I valori da sommare per ottenere il modulo sono: 2 + 8 + 32 = 42 (dec). Il valore rappresentato è dunque -42 (dec).

In **alternativa**, possiamo tener conto del valore di ciascun bit, sapendo che il bit più significativo ha segno negativo e su 8 bit vale -128.

-128	64	32	16	8	4	2	1
------	----	----	----	---	---	---	---

La somma dei 7 bit meno significativi per 11010110 vale 2+4+16+64 = 86. Sottraendo infine 128, si ottiene -42.

Valore dec di 0x6D — complemento a 2

Usiamo la tabella di conversione da hex a nibble: 0x6D = 0b01101101.

Il bit più alto (a sx) è a 0, quindi il valore è positivo. Non bisogna calcolare il complemento a due. Ogni bit conta come una potenza di 2, in ordine da  $2^0$  a destra, fino a  $2^6$  a sinistra.

I valori da sommare sono:  $1 + 4 + 8 + 32 + 64 = 109$  (dec).

## Valore dec di 0xC7 — complemento a 2

Usiamo la tabella di conversione da hex a nibble:  $0xC7 = 0b11000111$ .

Il bit più alto (a sx) è a 1, quindi il valore è negativo. Per scoprire il modulo, bisogna calcolare il complemento a 2: prima facciamo il complemento bit a bit e poi sommiamo 1.

```
11000111 ~
-----
00111000 +
      1 =
-----
00111001
```

A questo punto, ogni bit conta come una potenza di 2, in ordine da  $2^0$  a destra, fino a  $2^7$  a sinistra.

I valori da sommare per ottenere il modulo sono:  $1 + 8 + 16 + 32 = 57$  (dec). Il valore rappresentato è dunque -57 (dec).

In **alternativa**, possiamo tener conto del valore di ciascun bit, sapendo che il bit più significativo ha segno negativo e su 8 bit vale -128.

La somma dei 7 bit meno significativi per 11000111 vale  $1 + 2 + 4 + 64 = 71$ . Sottraendo infine 128, si ottiene -57.

## Valore dec di 0x39 — complemento a 2

Usiamo la tabella di conversione da hex a nibble:  $0x39 = 0b00111001$ .

Il bit più alto (a sx) è a 0, quindi il valore è positivo. Non bisogna calcolare il complemento a due. Ogni bit conta come una potenza di 2, in ordine da  $2^0$  a destra, fino a  $2^6$  a sinistra.

I valori da sommare sono:  $1 + 8 + 16 + 32 = 57$  (dec).

## Rappresentazione di -42 — 8 bit in complemento a 2

Rappresentiamo in binario il modulo del numero: 42.

42		0
21		1
10		0
5		1
2		0
1		1
0		

I resti corrispondono ai bit della rappresentazione del modulo, in alto il bit più a destra, in basso il bit più a sinistra: 0b00101010. Abbiamo aggiunto a sinistra gli zeri necessari per completare gli 8 bit della rappresentazione.

Il valore da rappresentare è negativo, quindi dobbiamo calcolare il complemento a due del modulo.

```

00101010 ¬
-----
11010101 +
      1 =
-----
11010110

```

Usiamo infine la tabella di conversione da nibble a hex: 0b11010110 = 0xD6.

In **alternativa**, possiamo considerare il fatto che, per ottenere la rappresentazione di un numero negativo, sicuramente bisogna impostare a 1 il bit più significativo. Esso è l'unico ad avere valore negativo e su 8 bit vale  $-128$ . I restanti bit dovranno codificare un valore  $n_1$  tale che  $-128 + n_1 = -42$ , quindi  $n_1 = 128 - 42 = 86$ . A questo punto, possiamo procedere con i confronti e le sottrazioni.

1	$-42 < 0 \implies n_1 = 128 - 42 = 86$
1	$64 \leq 86 \implies n_1 = 86 - 64 = 22$
0	$32 > 22$
1	$16 \leq 22 \implies n_1 = 22 - 16 = 6$
0	$8 > 6$
1	$4 \leq 6 \implies n_1 = 6 - 4 = 2$
1	$2 \leq 2 \implies n_1 = 2 - 2 = 0$
0	$1 > 0$

## Rappresentazione di 109 — 8 bit in complemento a 2

Rappresentiamo in binario il modulo del numero: 109.

109	1
54	0
27	1
13	1
6	0
3	1
1	1
0	

I resti corrispondono ai bit della rappresentazione del modulo, in alto il bit più a destra, in basso il bit più a sinistra: 0b01101101. Abbiamo aggiunto a sinistra gli zeri necessari per completare gli 8 bit della rappresentazione.

Il valore da rappresentare è positivo, quindi non dobbiamo calcolare il complemento a due.

Usiamo la tabella di conversione da nibble a hex:  $0b01101101 = 0x6D$ .

In **alternativa**, possiamo procedere con i confronti e le sottrazioni. Il bit più significativo vale  $-128$  e deve rimanere a 0 per ottenere la rappresentazione di un numero positivo:  $\boxed{0} \boxed{1} \boxed{1} \boxed{0} \boxed{1} \boxed{1} \boxed{0} \boxed{1}$ .

0	$109 \geq 0 \implies n_1 = 109$
1	$64 \leq 109 \implies n_1 = 109 - 64 = 45$
1	$32 \leq 45 \implies n_1 = 45 - 32 = 13$
0	$16 > 13$
1	$8 \leq 13 \implies n_1 = 13 - 8 = 5$
1	$4 \leq 5 \implies n_1 = 5 - 4 = 1$
0	$2 > 1$
1	$1 \leq 1$

## Rappresentazione di $-57$ — 8 bit in complemento a 2

Rappresentiamo in binario il modulo del numero: 57.

57	1
28	0
14	0
7	1
3	1
1	1
0	

I resti corrispondono ai bit della rappresentazione del modulo, in alto il bit più a destra, in basso il bit più a sinistra:  $0b00111001$ . Abbiamo aggiunto a sinistra gli zeri necessari per completare gli 8 bit della rappresentazione.

Il valore da rappresentare è negativo, quindi dobbiamo calcolare il complemento a due del modulo.

```
00111001 ~
-----
11000110 +
      1 =
-----
11000111
```

Usiamo infine la tabella di conversione da nibble a hex:  $0b11000111 = 0xC7$ .

In **alternativa**, possiamo considerare il fatto che, per ottenere la rappresentazione di un numero negativo, sicuramente bisogna impostare a 1 il bit più significativo. Esso è l'unico ad avere valore negativo e su 8 bit vale  $-128$ . I restanti bit dovranno codificare un valore  $n_1$  tale che  $-128 + n_1 = -57$ , quindi  $n_1 = 128 - 57 = 71$ . A questo punto, possiamo procedere con i confronti e le sottrazioni. Si ottiene  $\boxed{1} \boxed{1} \boxed{0} \boxed{0} \boxed{0} \boxed{1} \boxed{1} \boxed{1}$ .

1	$-57 < 0 \implies n_1 = 128 - 57 = 71$
1	$64 \leq 71 \implies n_1 = 71 - 64 = 7$
0	$32 > 7$
0	$16 > 7$
0	$8 > 7$
1	$4 \leq 7 \implies n_1 = 7 - 4 = 3$
1	$2 \leq 3 \implies n_1 = 3 - 2 = 1$
1	$1 \leq 1$

## Rappresentazione di 57 — 8 bit in complemento a 2

Rappresentiamo in binario il modulo del numero: 57.

57	1
28	0
14	0
7	1
3	1
1	1
0	

I resti corrispondono ai bit della rappresentazione del modulo, in alto il bit più a destra, in basso il bit più a sinistra: 0b00111001. Abbiamo aggiunto a sinistra gli zeri necessari per completare gli 8 bit della rappresentazione.

Il valore da rappresentare è positivo, quindi non dobbiamo calcolare il complemento a due.

Usiamo la tabella di conversione da nibble a hex: 0b00111001 = 0x39.

In **alternativa**, possiamo procedere con i confronti e le sottrazioni. Il bit più significativo vale  $-128$  e deve rimanere a 0 per ottenere la rappresentazione di un numero positivo: 0 0 1 1 1 0 0 1.

0	$57 \geq 0 \implies n_1 = 57$
0	$64 > 57$
1	$32 \leq 57 \implies n_1 = 57 - 32 = 25$
1	$16 \leq 25 \implies n_1 = 25 - 16 = 9$
1	$8 \leq 9 \implies n_1 = 9 - 8 = 1$
0	$4 > 1$
0	$2 > 1$
1	$1 \leq 1$

## Valore dec di 0x2500 — float16

Usiamo la tabella di conversione da nibble a bit.: 0b0010010100000000.

Il bit più a sx rappresenta il segno, in questo caso positivo.

I 5 bit successivi (01001) rappresentano l'esponente, ricordandosi però di sottrarre sempre 15 al loro valore, come richiede lo standard. Sommiamo le seguenti potenze di 2:  $1 + 8 = 9$ . Sottraiamo 15 e otteniamo come esponente -6.

Gli altri bit rappresentano la parte frazionaria, da porre per costruzione alla sinistra di un 1, dopo la virgola:  $1.01_2 \times 2^{-6}$ . Tutti i bit a zero in fondo non hanno valore.

La moltiplicazione per una potenza di 2 corrisponde a uno spostamento della virgola, in questo caso verso sinistra, visto che l'esponente è negativo: 0.00000101.

I bit alla destra della virgola rappresentano potenze negative di 2, in ordine da  $2^{-1}$  a sinistra, seguendo verso destra con  $2^{-2}$ ,  $2^{-3}$  ecc.

In questo caso dobbiamo sommare  $2^{-6} + 2^{-8} = 0.015625 + 0.00390625 = 0.01953125$ , usando la tabella delle potenze negative di 2.

In questo caso sappiamo che il segno è positivo, quindi il valore è proprio 0.01953125.

### Valore dec di 0xB500 — float16

Usiamo la tabella di conversione da nibble a bit.: 0b1011010100000000.

Il bit più a sx rappresenta il segno, in questo caso negativo.

I 5 bit successivi (01101) rappresentano l'esponente, ricordandosi però di sottrarre sempre 15 al loro valore, come richiede lo standard. Sommiamo le seguenti potenze di 2:  $1 + 4 + 8 = 13$ . Sottraiamo 15 e otteniamo come esponente -2.

Gli altri bit rappresentano la parte frazionaria, da porre per costruzione alla sinistra di un 1, dopo la virgola:  $1.01_2 \times 2^{-2}$ . Tutti i bit a zero in fondo non hanno valore.

La moltiplicazione per una potenza di 2 corrisponde a uno spostamento della virgola, in questo caso verso sinistra, visto che l'esponente è negativo: 0.0101.

I bit alla destra della virgola rappresentano potenze negative di 2, in ordine da  $2^{-1}$  a sinistra, seguendo verso destra con  $2^{-2}$ ,  $2^{-3}$  ecc.

In questo caso dobbiamo sommare  $2^{-2} + 2^{-4} = 0.25 + 0.0625 = 0.3125$ , usando la tabella delle potenze negative di 2.

In questo caso sappiamo che il segno è negativo, quindi il valore è -0.3125.

### Valore dec di 0x57D2 — float16

Usiamo la tabella di conversione da nibble a bit.: 0b0101011111010010.

Il bit più a sx rappresenta il segno, in questo caso positivo.

I 5 bit successivi (10101) rappresentano l'esponente, ricordandosi però di sottrarre sempre 15 al loro valore, come richiede lo standard. Sommiamo le seguenti potenze di 2:  $1 + 4 + 16 = 21$ . Sottraiamo 15 e otteniamo come esponente 6.

Gli altri bit rappresentano la parte frazionaria, da porre per costruzione alla sinistra di un 1, dopo la virgola:  $1.111101001_2 \times 2^6$ . Tutti i bit a zero in fondo non hanno valore.

La moltiplicazione per una potenza di 2 corrisponde a uno spostamento della virgola, in questo caso verso destra, visto che l'esponente è positivo: 1111101.001.

Per la parte intera, basta sommare le corrispondenti potenze positive di 2:  $1 + 4 + 8 + 16 + 32 + 64 = 125$ .

Per la parte frazionaria, basta sommare le corrispondenti potenze negative di 2:  $2^{-3} = 0.125$ .

In questo caso sappiamo che il segno è negativo, quindi il valore è 125.125.



## Valore dec di 0xAE00 — float16

Come sopra, -0.09375.

## Valore dec di 0x3C14 — float16

Usiamo la tabella di conversione da nibble a bit.: 0b0011110000010100.

Il bit più a sx rappresenta il segno, in questo caso positivo.

I 5 bit successivi (01111) rappresentano l'esponente, ricordandosi però di sottrarre sempre 15 al loro valore, come richiede lo standard. Sommiamo le seguenti potenze di 2:  $1 + 2 + 4 + 8 = 15$ . Sottraiamo 15 e otteniamo come esponente 0.

Gli altri bit rappresentano la parte frazionaria, da porre per costruzione alla sinistra di un 1, dopo la virgola:  $1.00000101_2 \times 2^0$ . Tutti i bit a zero in fondo non hanno valore.

In questo caso, con esponente 0, non bisogna spostare la virgola.

Per la parte intera, basta sommare le corrispondenti potenze positive di 2: 1.

Per la parte frazionaria, basta sommare le corrispondenti potenze negative di 2:  $2^{-6} + 2^{-8} = 0.015625 + 0.00390625 = 0.01953125$ .

In questo caso sappiamo che il segno è positivo, quindi il valore è 1.01953125.

## Rappresentazione di 0.01953125 — float16

Prima di tutto, convertiamo in base binaria la parte intera. In questo caso vale 0.

Poi convertiamo in binario la parte frazionaria, moltiplicandola ripetutamente per 2 e segnando di volta in volta la parte intera del risultato.

0.01953125	
0.03906250	0
0.0781250	0
0.156250	0
0.31250	0
0.6250	0
1.250	1
0.50	0
1.0	1

La parte frazionaria si converte in  $0.00000101_2$ . Tutto il numero si converte nello stesso valore.

Ora bisogna spostare la virgola in modo da lasciare alla sua sinistra un solo bit a 1, contando gli spostamenti. In questo caso, si sposta la virgola di 6 posizioni verso destra, quindi si può scrivere:  $0.00000101_2 = 1.01_2 \times 2^{-6}$ .

A questo punto conosciamo il bit del segno del numero e i bit della sua parte frazionaria, cioè tutto ciò che è a destra della virgola nel valore che abbiamo trovato: 01. A destra aggiungeremo dei bit a 0 fino a occupare tutti e 10 i bit riservati alla parte frazionaria.

Non ci resta che trovare la rappresentazione dell'esponente, a cui però dobbiamo sempre aggiungere 15 come richiede lo standard:  $-6 + 15 = 9$ . Calcoliamo quindi la rappresentazione binaria di 9.

9		1
4		0
2		0
1		1
0		

Questi bit vanno allineati a destra nello spazio di 5 bit che lo standard destina all'esponente. Componendo le parti ottenute (1 bit di segno, 5 bit di esponente e 10 bit di parte frazionaria), si ha infine  $0b0010010100000000 = 0x2500$ .

## Rappresentazione di -0.3125 — float16

Prima di tutto, convertiamo in base binaria la parte intera. In questo caso vale 0.

Poi convertiamo in binario la parte frazionaria, moltiplicandola ripetutamente per 2 e segnando di volta in volta la parte intera del risultato.

0.3125		
0.6250		0
1.250		1
0.50		0
1.0		1

La parte frazionaria si converte in  $0.0101_2$ . Tutto il numero si converte nello stesso valore.

Ora bisogna spostare la virgola in modo da lasciare alla sua sinistra un solo bit a 1, contando gli spostamenti. In questo caso, si sposta la virgola di 2 posizioni verso destra, quindi si può scrivere:  $0.0101_2 = 1.01_2 \times 2^{-2}$ .

A questo punto conosciamo il bit del segno del numero e i bit della sua parte frazionaria, cioè tutto ciò che è a destra della virgola nel valore che abbiamo trovato: 01. A destra aggiungeremo dei bit a 0 fino a occupare tutti e 10 i bit riservati alla parte frazionaria.

Non ci resta che trovare la rappresentazione dell'esponente, a cui però dobbiamo sempre aggiungere 15 come richiede lo standard:  $-2 + 15 = 13$ . Calcoliamo quindi la rappresentazione binaria di 13.

13		1
6		0
3		1
1		1
0		

Questi bit vanno allineati a destra nello spazio di 5 bit che lo standard destina all'esponente. Componendo le parti ottenute (1 bit di segno, 5 bit di esponente e 10 bit di parte frazionaria), si ha infine  $0b1011010100000000 = 0xB500$ .

## Rappresentazione di 125.125 — float16

Prima di tutto, convertiamo in base binaria la parte intera, dividendola ripetutamente per 2.

125	1
62	0
31	1
15	1
7	1
3	1
1	1
0	

Poi, convertiamo in base binaria la parte frazionaria, moltiplicandola ripetutamente per 2.

0.125	
0.250	0
0.50	0
1.0	1

Componiamo tutto il numero, in base binaria:  $1111101.001_2$ .

Ora bisogna spostare la virgola in modo da lasciare alla sua sinistra un solo bit a 1, contando gli spostamenti. In questo caso, si sposta la virgola di 6 posizioni verso sinistra, quindi si può scrivere:  $1111101.001_2 = 1.111101001_2 \times 2^6$ .

A questo punto conosciamo il bit del segno del numero e i bit della sua parte frazionaria, cioè tutto ciò che è a destra della virgola nel valore che abbiamo trovato: 111101001. A destra aggiungeremo dei bit a 0 fino a occupare tutti e 10 i bit riservati alla parte frazionaria.

Non ci resta che trovare la rappresentazione dell'esponente, a cui però dobbiamo sempre aggiungere 15 come richiede lo standard:  $6 + 15 = 21$ . Calcoliamo quindi la rappresentazione binaria di 21.

21	1
10	0
5	1
2	0
1	1
0	

Questi bit vanno allineati a destra nello spazio di 5 bit che lo standard destina all'esponente. Componendo le parti ottenute (1 bit di segno, 5 bit di esponente e 10 bit di parte frazionaria), si ha infine  $0b0101011111010010 = 0x57D2$ .

## Rappresentazione di -0.09375 — float16

Prima di tutto, convertiamo in base binaria la parte intera, dividendola ripetutamente per 2. In questo caso, vale 0.

Poi, convertiamo in base binaria la parte frazionaria, moltiplicandola ripetutamente per 2.

0.09375	
0.18750	0
0.3750	0
0.750	0
1.50	1
1.0	1

Componiamo tutto il numero, in base binaria:  $0.00011_2$ .

Ora bisogna spostare la virgola in modo da lasciare alla sua sinistra un solo bit a 1, contando gli spostamenti. In questo caso, si sposta la virgola di 4 posizioni verso destra, quindi si può scrivere:  $0.00011_2 = 1.1_2 \times 2^{-4}$ .

A questo punto conosciamo il bit del segno del numero e i bit della sua parte frazionaria, cioè tutto ciò che è a destra della virgola nel valore che abbiamo trovato: 1. A destra aggiungeremo dei bit a 0 fino a occupare tutti e 10 i bit riservati alla parte frazionaria.

Non ci resta che trovare la rappresentazione dell'esponente, a cui però dobbiamo sempre aggiungere 15 come richiede lo standard:  $-4 + 15 = 11$ . Calcoliamo quindi la rappresentazione binaria di 11 (dec).

11		1
5		1
2		0
1		1
0		

Questi bit vanno allineati a destra nello spazio di 5 bit che lo standard destina all'esponente. Componendo le parti ottenute (1 bit di segno, 5 bit di esponente e 10 bit di parte frazionaria), si ha infine  $0b1010111000000000 = 0xAE00$ .

## Rappresentazione di 1.02 — float16

Prima di tutto, convertiamo in base binaria la parte intera, dividendola ripetutamente per 2. In questo caso vale 1.

Poi, convertiamo in base binaria la parte frazionaria, moltiplicandola ripetutamente per 2. Attenzione, in questo caso non otterremo una conversione precisa, ma dovremo accontentarci di una approssimazione.

0.02		0.04		0
0.04		0.08		0
0.08		0.16		0
0.16		0.32		0
0.32		0.64		0
0.64		1.28		1
0.28		0.56		0
0.56		1.12		1
0.12		0.24		0
0.24		0.48		0
0.48		0.96		0
0.96		1.92		1
0.92		1.84		1 ...

Componiamo tutto il numero, in base binaria:  $1.0000010100011_2$ .


Ora bisogna spostare la virgola in modo da lasciare alla sua sinistra un solo bit a 1, contando gli spostamenti. In questo caso, non bisogna spostare la virgola quindi l'esponente vale 0.

A questo punto conosciamo il bit del segno del numero e i bit della sua parte frazionaria, cioè tutto ciò che è a destra della virgola nel valore che abbiamo trovato: 0000010100. Ci limitiamo a occupare tutti e 10 i bit riservati alla parte frazionaria, accontentandoci di una approssimazione.

Non ci resta che trovare la rappresentazione dell'esponente, a cui però dobbiamo sempre aggiungere 15 come richiede lo standard:  $0 + 15 = 15$ . Calcoliamo quindi la rappresentazione binaria di 15.

15		1
7		1
3		1
1		1
0		

Questi bit vanno allineati a destra nello spazio di 5 bit che lo standard destina all'esponente. Componendo le parti ottenute (1 bit di segno, 5 bit di esponente e 10 bit di parte frazionaria), si ha infine  $0b0011110000010100 = 0x3C14$ .

 Questa è la migliore rappresentazione approssimata di 1.02 che riusciamo a ottenere in float16. Abbiamo calcolato in uno degli esercizi precedenti il valore rappresentato da 0x3C14, che non è proprio 1.02 ma 0.01953125.

## Codifica del code-point 0x7B — UTF-8

Scriviamo il code-point in binario e contiamo i bit necessari:  $0b01111011$ , servono 7 bit.

UTF-8 prevede quattro schemi di codifica:

1 byte	0xxxxxxx	code-point fino a 7 bit
2 byte	110xxxxx 10xxxxxx	code-point fino a 11 bit
3 byte	1110xxxx 10xxxxxx 10xxxxxx	code-point fino a 16 bit
4 byte	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx	code-point fino a 21 bit

Per 7 bit basta un byte.

Inseriamo semplicemente i bit del code-point al posto delle x, allineandoli a destra:  $0b01111011 = 0x7B$ .

Si tratta infatti di un carattere ASCII, che mantiene invariata la sua codifica in UTF-8.

## Codifica del code-point 0xAC — UTF-8

Scriviamo il code-point in binario e contiamo i bit necessari:  $0b10101100$ , servono 8 bit.

UTF-8 richiede 2 byte, per codificare un code-point di 8 bit:  $110xxxxx 10xxxxxx$ .

Inseriamo i bit del code-point al posto delle x, allineandoli a destra:  $11000010 10101100 = 0xC2AC$ .

## Codifica del code-point 0x3BB — UTF-8

Scriviamo il code-point in binario e contiamo i bit necessari:  $0b001110111011$ , servono 10 bit senza contare gli zeri a sinistra.

UTF-8 richiede 2 byte, per codificare un code-point di 10 bit:  $110xxxxx 10xxxxxx$ .

Inseriamo i bit del code-point al posto delle x, allineandoli a destra:  $11001110 10111011 = 0xCEBB$ .

### Codifica del code-point 0x221A — UTF-8

Scriviamo il code-point in binario e contiamo i bit necessari: 0b0010001000011010, servono 14 bit senza contare gli zeri a sinistra.

UTF-8 richiede 2 byte, per codificare un code-point di 14 bit: 1110xxxx 10xxxxxx 10xxxxxx.

Inseriamo i bit del code-point al posto delle x, allineandoli a destra: 11100010 10001000 10011010 = 0xE2889A.

### Codifica del code-point 0x1F385 — UTF-8

Scriviamo il code-point in binario e contiamo i bit necessari: 0b00011111001110000101, servono 17 bit senza contare gli zeri a sinistra.

UTF-8 richiede 4 byte, per codificare un code-point di 17 bit: 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx.

Inseriamo i bit del code-point al posto delle x, allineandoli a destra: 11110000 10011111 10001110 10000101 = 0xF09F8E85.

### Decodifica dei byte 0x7B — UTF-8

Scriviamo la sequenza in binario: 01111011.

Controlliamo che i bit di intestazione siano corretti: 0xxxxxxx.

Eliminiamoli: 0111 1011 = 0x7B.

Si tratta infatti di un carattere ASCII, che mantiene invariata la sua codifica in UTF-8.

### Decodifica dei byte 0xC2AC — UTF-8

Scriviamo la sequenza in binario: 11000010 10101100.

Controlliamo che i bit di intestazione siano corretti: 110xxxxx 10xxxxxx.

Eliminiamoli: 1010 1100 = 0xAC.

### Decodifica dei byte 0xCEBB — UTF-8

Scriviamo la sequenza in binario: 11001110 10111011.

Controlliamo che i bit di intestazione siano corretti: 110xxxxx 10xxxxxx.

Eliminiamoli: 0011 1011 1011 = 0x3BB.

### Decodifica dei byte 0xE2889A — UTF-8

Scriviamo la sequenza in binario: 11100010 10001000 10011010.

Controlliamo che i bit di intestazione siano corretti: 1110xxxx 10xxxxxx 10xxxxxx.

Eliminiamoli: 0010 0010 0001 1010 = 0x221A.

## Decodifica dei byte 0xF09F8E85 — UTF-8

Scriviamo la sequenza in binario: 11110000 10011111 10001110 10000101.

Controlliamo che i bit di intestazione siano corretti: 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx.

Eliminiamoli: 0001 1111 0011 1000 0101 = 0x1F385.

## Dimensioni di BMP 1

Larghezza: 25 pixel; altezza: 25 pixel; profondità: 16 bpp; palette: assente; intestazione: 54 byte.

Dobbiamo calcolare l'occupazione di ogni riga, in byte. Cominciamo a moltiplicare la larghezza per la profondità:  $25 \times 2 = 50$ . Non è un multiplo di 4, allora cerchiamo il multiplo di 4 successivo: ogni riga occupa 52 byte.

Adesso moltiplichiamo per l'altezza, aggiungendo intestazione e palette:  $52 + 54 = 1354$ .

## Dimensioni di BMP 2

Larghezza: 45 pixel; altezza: 45 pixel; profondità: 4 bpp; palette: 12 colori; intestazione: 54 byte.

Ogni colore nella palette occupa sempre 4 byte. Una palette di 12 colori occupa  $12 \times 4 = 48$  byte.

Dobbiamo calcolare l'occupazione di ogni riga, in byte. Cominciamo a moltiplicare la larghezza per la profondità:  $45 \times 0.5 = 22.5$ . Non è un multiplo di 4, allora cerchiamo il multiplo di 4 successivo: ogni riga occupa 24 byte.

Adesso moltiplichiamo per l'altezza, aggiungendo intestazione e palette:  $24 + 54 + 48 = 1182$ .

## Dimensioni di BMP 3

Larghezza: 50 pixel; altezza: 40 pixel; profondità: 8 bpp; palette: 32 colori; intestazione: 54 byte.

Ogni colore nella palette occupa sempre 4 byte. Una palette di 32 colori occupa  $32 \times 4 = 128$  byte.

Dobbiamo calcolare l'occupazione di ogni riga, in byte. Cominciamo a moltiplicare la larghezza per la profondità:  $50 \times 1 = 50$ . Non è un multiplo di 4, allora cerchiamo il multiplo di 4 successivo: ogni riga occupa 52 byte.

Adesso moltiplichiamo per l'altezza, aggiungendo intestazione e palette:  $52 + 54 + 128 = 2262$ .

## Dimensioni di WAV 1

Durata: 10 sec; campionamento: 8000 Hz; bits-per-sample: 8 bit; stereo; intestaz.: 44 byte.

Per un audio stereo, a ogni campionamento vengono campionati 2 canali. Per ogni canale viene occupato 1 byte. In ogni secondo, questo spazio viene richiesto 8000 volte.

La dimensione del WAV è  $2 \times 1 \times 8000 \times 10 + 44 = 160044$ .

## **Dimensioni di WAV 2**

Durata: 6 sec; campionamento: 16000 Hz; bits-per-sample: 8 bit; 6 canali (5.1 surround); intestaz.: 44 byte.

A ogni campionamento vengono campionati 6 canali. Per ogni canale viene occupato 1 byte. In ogni secondo, questo spazio viene richiesto 16000 volte.

La dimensione del WAV è  $6 \times 1 \times 16000 \times 6 + 44 = 576044$ .

## **Dimensioni di WAV 3**

Durata: 4 sec; campionamento: 44100 Hz; bits-per-sample: 16 bit; mono; intestaz.: 44 byte.

Per un audio mono, a ogni campionamento viene campionato 1 solo canale. Per ogni canale vengono occupati 2 byte. In ogni secondo, questo spazio viene richiesto 44100 volte.

La dimensione del WAV è  $1 \times 2 \times 44100 \times 4 + 44 = 352844$ .