TWO'S COMPLEMENT IN C

LEONARDO TAMIANO

TABLE OF CONTENTS

INTRODUZIONE

Il seguente codice C può essere utilizzato per stampare la rappresentazione binaria di un dato intero.

```c
void print_binary(int value) {
  int bit_size = sizeof(int) * 8;
  int group_length = 4;

  int str_len = bit_size + 1 + ((bit_size / group_length) - 1);
  char bin_str[str_len];
  int bit_index;

  for (int i = 0, j = 0, c = 0; i < str_len; i++) {
    if (j > 0 && ((j % 4) == 0)) {
      bin_str[i] = ' ';
      j = 0;
      c++;
    } else {
      bit_index = bit_size - 1 - i + c;
      bin_str[i] = (value >> bit_index) & 1 ? '1' : '0';
      j += 1;
    }
  }

  bin_str[str_len] = '\0';

  printf("%s\n", bin_str);
}
```

Utilizzandolo vediamo la seguente cosa

```
Binary representation for: 1
        0000 0000 0000 0000 0000 0000 0000 0001

Binary representation for: -1
        1111 1111 1111 1111 1111 1111 1111 1111
```

La rappresentazione binaria di `1` è piuttosto intuitiva.

Quella di `-1` invece un po' meno.

Perché `-1` è rappresentato così?

1111 1111 1111 1111 1111 1111 1111 1111

SIGNED AND UNSIGNED TYPES

Il linguaggio di programmazione `C` offre due principali categorie di tipi per rappresentare i valori numerici interi: quelli `signed` e quelli `unsigned`.

I tipi `unsigned` sono utilizzati per rappresentare numeri interi senza segno.

```c
#include <stdio.h>

int main(int argc, char **argv) {
  unsigned short unsigned_short = 4321;
  unsigned int   unsigned_int   = 4294967295;
  unsigned long  unsigned_long  = 18446744073709551615ul;

  printf("[INFO] - unsigned short value = %u\n", unsigned_short);
  printf("[INFO] - unsigned int value   = %u\n", unsigned_int);
  printf("[INFO] - unsigned long value  = %lu\n", unsigned_long);

  return 0;
}
```

Comando:

```
[leo@archlinux code]$ gcc unsigned_values.c -o unsigned_values
[leo@archlinux code]$ ./unsigned_values
```

Output:

```
[INFO] — unsigned short value = 4321
[INFO] — unsigned int value   = 4294967295
[INFO] — unsigned long value  = 18446744073709551615
```

Quando utilizziamo i tipi `unsigned` tutti i bit a disposizione del tipo sono investiti per rappresentare il valore del numero.

Possiamo quindi rappresentare range di valori più grandi, ma non è possibile inserire informazioni sul segno.

```c
unsigned int   unsigned_int = -4294967295;
printf("%u\n", unsigned_int);  // prints 1
```

Tutti i numeri rappresentati sono positivi.

I tipi `signed` invece si utilizzano per rappresentare valori numerici interi con segno.

```c
#include <stdio.h>

int main(int argc, char **argv) {
  signed int  signed_short = -4321;
  signed int  signed_int   = -2147483644;
  signed long signed_long  = -9223372036854775807ul;

  printf("[INFO] - unsigned short value = %d\n", signed_short);
  printf("[INFO] - unsigned int value   = %d\n", signed_int);
  printf("[INFO] - unsigned long value  = %ld\n", signed_long);

  return 0;
}
```

Comando:

```
[leo@archlinux code]$ gcc signed_values.c -o signed_values
[leo@archlinux code]$ ./signed_values
```

Output:

```
[INFO] – unsigned short value = -4321
[INFO] – unsigned int value   = -2147483644
[INFO] – unsigned long value  = -9223372036854775807
```

I tipi `signed` permettono di rappresentare sia numeri positivi che numeri negativi.

I bit messi a disposizione devono quindi essere investiti per rappresentare due informazioni distinte:

- Il segno del numero.
- Il valore del numero.

Questo significa che il range di valori assoluti (senza segno) rappresentati è tipicamente minore (la metà) della rispettiva controparte `unsigned`.

```c
int value_int = 4294967295;
printf("%d\n", value_int); // print -1
```

Se non si esplicita nulla si assume che il tipo è `signed`, altrimenti si utilizza la keyword `unsigned` per specificare che si è solo interessati al valore positivo.

```
signed int   value_a = -1234;
int          value_b = -1234;
unsigned int value_c = 1234;
```

In generale poi, come abbiamo già visto, la keyword `unsigned` può essere utilizzata con tutti i tipi numerici standard.

```
unsigned short     short_value;
unsigned int       int_value;
unsigned long      long_value;
unsigned long long longlong_value;
```

TOWARDS A SIGNED REPRESENTATION FOR INTEGERS

Supponiamo di avere a disposizione `32 bit` per memorizzare un intero con segno.

Come utilizziamo questi bit?

SIGN BIT

Una possibile idea iniziale potrebbe essere la seguente:

- Si utilizza un bit, detto **sign bit**, per memorizzare in modo esplicito il segno del numero.
- Si utilizzano i restanti bit per memorizzare il valore del numero.

$$
\begin{aligned}
3 &\longrightarrow 0 \mid 011 \\
-3 &\longrightarrow 1 \mid 011
\end{aligned}
$$

Questa rappresentazione richiederebbe di trattare i numeri senza segno (`unsigned`) e i numeri con il segno (`signed`) diversamente nelle tipiche operazioni aritmetiche di somma e sottrazione.

$$3 \longrightarrow \; 0011 \;\; (\text{unsigned})$$

$$\begin{aligned} 3 &\longrightarrow \; 0 \,|\, 011 \;\; (\text{signed}) \\ -3 &\longrightarrow \; 1 \,|\, 011 \;\; (\text{signed}) \end{aligned}$$

Questo necessiterebbe di aggiungere della logica non banale all'interno sistema, aumentandone la complessità totale.

Un sistema molto più intelligente per memorizzare i numeri con segno è detto **complemento a due**, in inglese **two's complement**.

La pecularità di questa rappresentazione è che **permette di trattare i numeri con segno e quelli senza segno nello stesso identico modo**.

Prima di vedere il **complemento a due** introduciamo il **complemento a uno**.

ONE'S COMPLEMENT

Il **complemento a uno** di un numero m memorizzato con N bit è calcolato andando ad invertire tutti i bit del numero.

Esempio $(N = 5,\ m = 4)$:

Dato che $m = 4$ è rappresentato in binario come 00100, invertendo i bit otteniamo

$$00100 \longrightarrow 11011$$

Quindi il complemento a uno di 4 è 11011.

Si potrebbe pensare di rappresentare i numeri negativi utilizzando il complemento a uno.

Esempio #1: $N = 3$

numero	binario
0	000
1	001
2	010
3	011

numero	binario
-0	111
-1	110
-2	101
-3	100

Notiamo che in questa rappresentazione il bit più a sinistra può essere considerato come un **sign bit**: se è 0 allora il numero è positivo, altrimenti il numero è negativo.

A differenza con lo schema precedente però, **il sign bit non è separato dai restanti numeri, ma è parte integrante del numero**.

Questo permette di effettuare le solite operazioni con i numeri senza dover aggiungere della logica extra per gestire il sign bit.

Detto questo il complemento a uno presenta dei leggeri problemi di calcolo.

Esempio #2: $N = 5,\ a = 10,\ b = 6$

Possiamo sottrarre b da a andando a sommare a con il complemento in base uno di b.

$$\begin{aligned} 10 - 6 &= 10 + (-6) \\ &= 01010_2 + 11001_2 \\ &= 00011 \\ &= 3 \neq 4 \end{aligned}$$

Per risolvere questi problemi si introduce **il complemento a due**.

TWO'S COMPLEMENT

Il **complemento a due** di un numero m memorizzato con N bit è calcolato nel seguente modo:

1. Si invertono tutti i bit del numero.
2. Si aggiunge 1 al numero risultante.

Esempio ($N = 5,\ m = 4$):

- $m = 4$ è rappresentato in binario come 00100.
- invertendo i bit otteniamo

$$00100 \longrightarrow 11011$$

- aggiungendo 1 si ottiene

$$11011 + 1 = 11100$$

Quindi il complemento a due di 4 è 11100.

L'idea è quella di rappresentare i numeri negativi tramite il complemento a due.

Esempi #1 $(N = 5,\ m = 4)$:

Per rappresentare -4 l'idea è quella di calcolare il complemento a due del numero 4.

$$\begin{aligned} 4 &\longrightarrow 00100 \\ -4 &\longrightarrow 11100 \end{aligned}$$

Esempio #2: $N = 3$

numero	binario
0	000
1	001
2	010
3	011

numero	binario
-1	111
-2	110
-3	101
-4	110

Cerchiamo adesso di capire le seguenti cose rispetto alla rappresentazione appena descritta supponendo di lavorare con $N = 32$, ovvero con `32 bit`:

1. Come è utilizzato e diviso lo spazio dei bit?
2. Perché viene utilizzato?
3. Perché funziona?

HOW IS THE SPACE USED?

Il numero senza segno (`unsigned`) più grande che possiamo rappresentare con `32 bit` è

$$2^{32} - 1 = 4294967295$$

Se però vogliamo rappresentare numeri con segno dobbiamo dimezzare lo spazio tra numeri positivi e numeri negativi.

Il numero 0

```
0000 0000 0000 0000 0000 0000 0000 0000
```

Il range dei **numeri positivi**

$$[1, \quad 2147483647 = 2^{31} - 1]$$

```
0000 0000 0000 0000 0000 0000 0000 0001 --> 1
0000 0000 0000 0000 0000 0000 0000 0010 --> 2
0000 0000 0000 0000 0000 0000 0000 0011 --> 3
0000 0000 0000 0000 0000 0000 0000 0100 --> 4
                ...
0100 0000 0000 0000 0000 0000 0000 0000 --> 1073741824
0100 0000 0000 0000 0000 0000 0000 0001 --> 1073741825
0100 0000 0000 0000 0000 0000 0000 0010 --> 1073741826
0100 0000 0000 0000 0000 0000 0000 0011 --> 1073741827
0100 0000 0000 0000 0000 0000 0000 0100 --> 1073741828
                ...
0111 1111 1111 1111 1111 1111 1111 1100 --> 2147483644
0111 1111 1111 1111 1111 1111 1111 1101 --> 2147483645
0111 1111 1111 1111 1111 1111 1111 1110 --> 2147483646
0111 1111 1111 1111 1111 1111 1111 1111 --> 2147483647
```

Il range dei **numeri negativi**

$$[-1, \quad -2147483648 = -2^{31}]$$

```
1000 0000 0000 0000 0000 0000 0000 0000 --> -2147483648
1000 0000 0000 0000 0000 0000 0000 0001 --> -2147483647
1000 0000 0000 0000 0000 0000 0000 0010 --> -2147483646
1000 0000 0000 0000 0000 0000 0000 0011 --> -2147483645
1000 0000 0000 0000 0000 0000 0000 0100 --> -2147483644
               ...
1111 1111 1111 1111 1111 1111 1111 1011 --> -5
1111 1111 1111 1111 1111 1111 1111 1100 --> -4
1111 1111 1111 1111 1111 1111 1111 1101 --> -3
1111 1111 1111 1111 1111 1111 1111 1110 --> -2
1111 1111 1111 1111 1111 1111 1111 1111 --> -1
```

WHY IS IT USED

Il fatto che rende il complemento a due molto comodo da utilizzare è che

somma e sottrazione possono essere implementate a livello dell'hardware utilizzando gli stessi circuiti, sia per i numeri signed che per quelli unsigned.

Esempio #1: $N = 8,\ a = 3,\ b = 15$

Possiamo sommare a e b direttamente per ottenere

$$\begin{aligned} 3 + 15 &= 00000011_2 + 00001111_2 \\ &= 00010010_2 \\ &= 18 \end{aligned}$$

Esempio #2: $N = 8,\ a = 3,\ b = 15$

Possiamo sottrarre b da a, andando a sommare a con il complemento in base due di b.

$$\begin{aligned} 3 - 15 &= 3 + (-15) \\ &= 00000011_2 + 11110001_2 \\ &= 11110100_2 \\ &= -12 \end{aligned}$$

WHY DOES IT WORK?

Siano m_1 e m_2 due numeri interi positivi potenzialmente diversi.

numero	valore rappresentazione binaria
m_1	m_1
$-m_2$	$(2^N - 1 - m_2) + 1$

Se li sommiamo otteniamo

$$m_1 + (2^N - 1 - m_2 + 1) = 2^N + (m_1 - m_2)$$

Dato poi che lavoriamo con numeri da N bit, abbiamo che 2^N va in **overflow**, ritornando ad essere 0. Otteniamo quindi

$$\begin{aligned} m_1 + (2^N - 1 - m_2 + 1) &= 2^N + (m_1 - m_2) \\ &= 0 + (m_1 - m_2) \\ &= m_1 - m_2 \end{aligned}$$

In altre parole,

È il meccanismo di overflow, ovvero la finitezza dei bit utilizzati, che ci permette di utilizzare il complemento a due per rappresentare i numeri negativi.

WHAT ABOUT DECIMALS?

In questa lezione abbiamo analizzato come poter memorizzare i numeri interi, sia quelli positivi che quelli negativi.

E per i numeri decimali?

$$\phi = 1.61803398875\ldots$$
$$\pi = 3.14159265359\ldots$$
$$e = 2.71828182845\ldots$$

Come possiamo memorizzare i numeri decimali, ovvero i numeri con la virgola?

Prossimamente cercherò di rispondere anche a questa domanda, andando ad analizzare lo standard IEEE per la Floating-Point Arithmetic (**IEEE 754**).