1.1 Rappresentazione di numeri naturali ed interi

La ALU è in grado di eseguire operazioni **logiche** (AND, OR, NOT, etc.) su stringhe di bit ed operazioni **aritmetiche**, interpretando le stringhe di bit che maneggia come **numeri naturali in base 2, o come numeri interi rappresentati in complemento a 2**. La rappresentazione dei numeri naturali ed interi in complemento a 2 è già stata affrontata durante il corso di Fondamenti di Programmazione, e quindi viene richiamata brevemente.

1.1.1 Numeri naturali:

Su N bit sono rappresentabili i 2^N numeri naturali nell'intervallo $\left[0; 2^N - 1\right]$. Ciascuna configurazione di N bit $b_{N-1}, b_{N-2}, ..., b_1, b_0$ può essere vista come un numero naturale rappresentato in base 2, il numero $X = \sum_{i=0}^{N-1} b_i \cdot 2^i$. Per tale motivo, in una stringa di N bit il bit b_0 è detto **Bit Meno Significativo (LSB)**, mentre il bit b_{N-1} è detto **Bit Più Significativo (MSB)**.

Dualmente, il numero naturale X può essere codificato su una stringa di bit se si trovano le cifre della sua rappresentazione in base 2. Tali cifre si trovano **dividendo per due** successivamente il numero X, e considerando tutti i resti.

1.1.2 Numeri interi:

Su N bit sono rappresentabili tutti i 2^N numeri interi x compresi nell'intervallo

$$\left[-2^{N-1}; 2^{N-1} - 1\right]$$

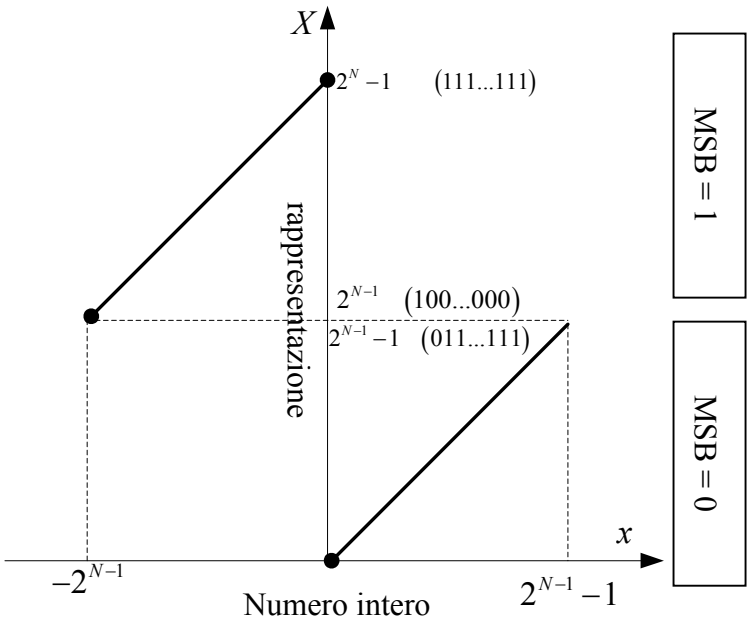
Quindi un numero intero x è rappresentabile su N bit o meno a seconda che entri o meno in questo intervallo. Su 8 e 16 bit l'intervallo è [-128, 127], [-32768, 32767], etc.

Il numero x viene rappresentato in C2 dalla stringa di bit che corrisponde al numero naturale X così calcolato:

$$X = \begin{cases} x & x \geq 0 \\ 2^N + x & x < 0 \end{cases}$$

Purché x sia un numero intero rappresentabile in C2 su N bit, cosa che deve essere verificata. In modo equivalente, si ha:

$$X = \left|x\right|_{2^N}$$



Come si vede dalla figura, tutte le stringhe di bit il cui bit più significativo è "0" rappresentano numeri interi *positivi*, mentre tutte quelle che cominciano per "1" rappresentano numeri interi *negativi*. Quindi, data una stringa di bit X che rappresenta un numero intero, il numero intero x che le corrisponde è il seguente:

$$x = \begin{cases} X & X_{N-1} = 0 \\ -\left(\overline{X} + 1\right) & X_{N-1} = 1 \end{cases}$$

Il *complemento* è l'operazione che cambia gli "0" in "1" e viceversa.

Fare un paio di esempi.

Esercizio:

- rappresentare i seguenti numeri naturali su 8 bit: 32, 63, 130
- per ciascun numero x rappresentato, chiedersi quale sia la rappresentazione di $2x$, $4x$, … $2^k x$, e quale quella del quoziente della divisione intera $x/2$, $x/4$, …$x/2^k$. Ricavare una regola generale valida per i numeri naturali.

- rappresentare i numeri interi (su 8 bit in complemento a 2) -32, -1, -128, 127
- per ciascun numero x rappresentato, chiedersi quale sia la rappresentazione di $2x$, $4x$, … $2^k x$, e quale quella del quoziente della divisione intera $x/2$, $x/4$, …$x/2^k$. Ricavare una regola generale valida per i numeri interi.

- Rappresentare i numeri interi di cui alla precedente domanda su 16 bit in complemento a 2.
- Qual è la relazione tra la loro rappresentazione su 8 bit e la loro rappresentazione su 16 bit? Ricavare una regola generale per ottenere la seconda dalla prima.

1.1.3 Primo esempio di programma in linguaggio mnemonico

Il seguente esempio descrive un programma che fa quanto segue:

- preleva dalla memoria un operando a 32 bit, che si trova all'indirizzo `0x00000100`
- conta il numero di bit a 1 che ci sono in quell'operando
- scrive il risultato nella locazione di memoria di indirizzo `0x00000104`

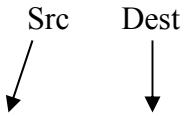
Vediamo come si fa nel dettaglio:

- si porta il contenuto della memoria in `EAX` (i conti si fanno meglio sui registri)
- si azzera un registro, ad esempio `CL`. Lo incrementeremo ogni volta che troviamo un 1 in `EAX`.
- Entriamo in un **ciclo**, nel quale facciamo scorrere verso (ad esempio) **destra** i bit di `EAX`, ed ogni volta che ne troviamo uno incrementiamo `CL`. Il ciclo termina quando `EAX` vale zero (potremmo farlo terminare dopo 32 passi, ma così si fa prima).
- Ricopiamo il contenuto di `CL` nella cella di memoria richiesta.

Scriviamo prima il codice che fa questa cosa, poi lo commentiamo. Per scriverlo, lo immaginiamo **contenuto in qualche locazione di memoria**, ad esempio a partire dall'indirizzo `0x00000200`.

Indirizzo **Contenuto**

```
0x00000100
0x00000101
0x00000102
0x00000103
0x00000104
…
0x00000200  MOVB $0x00, %CL
0x00000202  MOVL 0x00000100, %EAX
0x00000207  CMPL $0x00000000, %EAX
0x0000020A  JE %EIP+$0x07
0x0000020C  SHRL %EAX
0x0000020E  ADCB $0x00, %CL
0x00000211  JMP %EIP-$0x0C
0x00000213  MOVB %CL, 0x00000104
0x00000218  …
```



Note sintattiche (valide per l'Assembler, alle quali ci uniformiamo già da ora):

- Quando figurano come operandi in qualche istruzione (e solo in questo caso) le **costanti** vengono scritte premettendo il simbolo `$`. I numeri che non hanno il `$` davanti sono interpretati come **indirizzi.** La seconda istruzione riferisce **la cella di memoria di indirizzo** `0x00000100`, la terza il valore costante `$0x00000000`.

- i numeri sono interpretati in base 10, a meno che non siano preceduti dal prefisso `0x`, nel qual caso sono esadecimali.
- i nomi dei **registri** devono essere preceduti dal simbolo di `%`.
- In alcune istruzioni, i letterali `L` e `B` sono i suffissi di lunghezza. La stessa istruzione (`MOV`) può lavorare con operandi a 8 (`B`), 16 (`W`), 32 (`L`) bit

Descriviamo **informalmente** il funzionamento del programma.

```
0x00000200  MOVB $0x00, %CL
```
`MOV`: "sposta". Istruzione di **assegnamento**. Assegna il byte `0x00` al registro `CL`. In quest'istruzione entrambi gli operandi sono ad 8 bit, donde il suffisso "`B`". Quest'istruzione **occupa 2 byte** (infatti, la successiva comincia a 0x00000202). Ne prendiamo atto, preannunciando che **non sarà necessario ricordarlo.**

```
0x00000202  MOVL 0x00000100, %EAX
```
Lo stesso che prima. Adesso, però, l'istruzione lavora su operandi a 32 bit (suffisso "`L`"). In questo caso, mette nel registro `EAX` il contenuto della (quadrupla) locazione di memoria di indirizzo `0x00000100`. Quest'istruzione **occupa 5 byte**.

```
0x00000207  CMPL $0x0000000, %EAX
```
`CMP`: "confronta". Confronta la costante 0 ed `EAX`. Nel farlo, scriverà qualcosa nel registro dei flag.

```
0x0000020A  JE EIP+$0x0B
```
`JE`: "Jump if equal" Se da un'analisi del registro dei flag (che è stato settato dal precedente confronto) è emerso che i due operandi erano uguali (cioè che il contenuto di `EAX` era zero), si prosegue ad eseguire il programma dalla cella di memoria che sta `0x07` indirizzi più avanti. Se uno si fa il conto (che è noiosissimo), viene fuori **l'ultima** istruzione. Infatti, mentre sto eseguendo quest'istruzione, EIP ha **già** il valore della prossima locazione di memoria (in quanto viene incrementato subito dopo che ho prelevato un'istruzione), cioè `0x0000020C`. Se a questo valore sommo `0x07`, ottengo `0x00000213`. Quest'istruzione e la precedente sono ciò che mi consente di **uscire dal ciclo**.

```
0x0000020C  SHRL %EAX
```
`SHRL`: "Shift right". Quest'istruzione trasla a destra il contenuto di `EAX`. Da sinistra (bit più significativo) viene inserito uno 0, ed il bit meno significativo finisce nel flag `CF`. A questo punto, se il bit che è finito in CF è pari ad uno, devo incrementare un contatore.

`0x0000020E  ADCB $0x00, %CL`
`ADC`: “Add with Carry”. Quest’istruzione somma all’operando **destinatario** (`CL`) l’operando sorgente **ed il contenuto del flag** `CF`. (in C++ scriverei: `CL += 0x00 + CF`). In questo caso, l’operando sorgente è zero, quindi somma il contenuto del flag solamente. Quindi, se il precedente shift ha messo in `CF` un bit pari ad uno, il registro `CL` viene incrementato, altrimenti no.

`0x00000211  JMP EIP-$0x0C`
`JMP`: “Jump”. Devo continuare il ciclo. Devo quindi saltare nuovamente alla prima istruzione del ciclo, quella cioè nella quale confronto `EAX` con zero. Fatti i debiti conti, e tenuto conto del fatto che `EIP` vale l’indirizzo della **successiva** istruzione, ottengo che devo tornare indietro di 12 locazioni.

`0x00000213  MOVB %CL, 0x00000104`
Posso arrivare qui soltanto se sono uscito dal ciclo. Quindi, a questo punto, CL contiene il numero di bit a 1 che erano originariamente nel registro `EAX`. Come da specifica, copio questo numero nella cella di indirizzo `0x00000104`.

Per scrivere un programma in questo modo è necessario conoscere **la lunghezza delle istruzioni**, e fare **i conti a mano** per i salti (noioso e pericoloso). Noi **non programmeremo in questo modo**, ma lo faremo utilizzando un linguaggio (Assembler) che è sostanzialmente **identico** al linguaggio mnemonico, ma permette di specificare in modo **simbolico** gli indirizzi delle istruzioni e delle locazioni di memoria. Il programma che scriveremo dovrà quindi essere **tradotto** in linguaggio macchina da un programma assemblatore. La traduzione di un assemblatore è solamente **sintattica,** ed avviene **riga per riga**: ad un’istruzione Assembler corrisponde un’istruzione in linguaggio macchina.