

Rappresentazione dell'informazione

- Differenza tra *simbolo* e *significato*
 - la cifra (lettera) usata per scrivere è un **simbolo** che *rappresenta* l'informazione
 - il concetto di numero (suono) corrisponde al **significato** dell'informazione
- Per comunicare/rappresentare informazioni è quindi necessario usare dei *simboli*
 - necessaria una convenzione (*rappresentazione*, *codifica* o *codice*) per associare i simboli con il loro significato
- Per **codificare** l'informazione solitamente si usa un *alfabeto di simboli*
 - **Alfabeto** = insieme finito di *simboli* adottati per rappresentare informazione
 - Es: per rappresentare numeri nei calcolatori elettronici
 - Alfabeto binario: {0, 1}
 - Simboli associati con stati elettrici facilmente distinguibili
 - es.: conducibilità o meno di un transistor

Arch. Elab. - S. Orlando 2

Codifica o codice

- Dati:
 - un *Alfabeto A* (ad esempio, alfabeto binario: $A=\{0,1\}$)
 - s dati distinti $D=\{d_0, d_1, \dots, d_{s-1}\}$una *codifica* (o *codice*) fornisce una corrispondenza tra
 - *sequenze (stringhe, configurazioni)* di simboli in A , ed
 - i vari dati $d_i \in D$
- Solitamente, i *codici* fanno riferimento a *sequenze di simboli* di lunghezza finita
 - Alfabeto di N *simboli* e Sequenze di *lunghezza K*
 - N^K *configurazioni* possibili
 - Rispetto ad un *alfabeto binario*
 - numero totale di configurazioni: 2^K
 - $2^k \geq s$ (dove s è la cardinalità dell'insieme D)
 - Es.: se D comprende le 26 lettere dell'alfabeto inglese ($s=26$)
 - sono necessari almeno sequenze di K simboli binari, con $K \geq 5$, poiché $2^4 = 16 < 26 < 32 = 2^5$

Arch. Elab. - S. Orlando 3

Codifica dei numeri

- Codifica informazioni *non numeriche* può essere effettuata in maniera *semi arbitraria*.
 - Basta fissare una convenzione per permettere di *riconoscere* i dati
 - Es. Codice ASCII - American Standard Code for Information Exchange - è una codifica di *caratteri alfanumerici* su *sequenze di simboli binari* di lunghezza $k=8$
- Codifica dei *numeri*
 - accurata, perché è necessario effettuare operazioni (*sommare, moltiplicare* ecc.) usando le rappresentazioni dei numeri
 - di solito si adotta il sistema di numerazione *arabica*, o *posizionale*

Arch. Elab. - S. Orlando 4

Rappresentazione informazione Elementi di aritmetica dei computer Organizzazione della memoria e codici correttori

Salvatore Orlando

Arch. Elab. - S. Orlando 1

Sistema di codifica posizionale

- Sistema di numerazione arabica in base 10 ($B=10$)
 - cifre (simboli) appartenenti all'alfabeto di 10 simboli $A=\{0,1,\dots,9\}$
 - simboli con valore diverso in base alla posizione nella stringa di simboli in A (unità, decine, centinaia, migliaia, ecc.)
- Per codificare i numeri naturali in una generica base B
 - fissare un alfabeto A di B simboli
 - fissare una corrispondenza tra
 - i B simboli di A \leftrightarrow i primi B numeri naturali $\{0,1,2,\dots,B-1\}$
 - numeri maggiori di B rappresentabili come stringhe di simboli $d_i \in A$:
 - $d_{n-1} \dots d_1 d_0$
 - valore numerico della stringa, dove la *significatività* delle cifre è espressa in base alle varie potenze di B:
 - $B^{n-1} * d_{n-1} + \dots + B^1 * d_1 + B^0 * d_0$

Arch. Elab. - S. Orlando 5

Numeri naturali in base 2

- Alfabeto binario $A=\{0,1\}$, dove i simboli sono detti *bit*, con 0 corrispondente al numero zero ed 1 al numero uno
- Nei calcolatori i numeri sono rappresentati come sequenze di bit di lunghezza finita
 - numeri rappresentati in notazione arabica, con base $B=2$ (numeri binari)
 - $d_{n-1} \dots d_1 d_0$ dove $d_i \in \{0,1\}$
- Con stringhe di n bit, sono rappresentabili 2^n dati (numeri diversi)
 - dal numero 0 al numero 2^n-1
- Valore numerico corrispondente, dove la significatività delle cifre è espressa sulla base di una potenza di $B=2$:
 - $2^{n-1} * d_{n-1} + \dots + 2^1 * d_1 + 2^0 * d_0$
- Es: per trovare il valore della stringa di simboli 1010 in base 2
 - $1010_2 = 1*8 + 0*4 + 1*2 + 0*1 = 10_{10}$

Arch. Elab. - S. Orlando 6

Conversione inversa

- Da base 10 a base B
- Procedimento per divisione
- Sia dato un certo numero N rappresentabile in base B come stringa di n simboli $d_{n-1} \dots d_1 d_0$ il cui valore è:
$$N = B^{n-1} * d_{n-1} + \dots + B^1 * d_1 + B^0 * d_0$$
- Se dividiamo per B
 - otteniamo d_0 come resto
 - Quoziente: $B^{n-2} * d_{n-1} + \dots + B^0 * d_1$
 - Resto: d_0 , $0 \leq d_0 < B$
 - possiamo iterare il procedimento, ottenendo d_1, d_2, d_3 ecc. fino ad ottenere un Quoziente = 0

```
Q=N; i=0;
fintantoché è vero che (Q>0), ripeti:
{
    d_i = Q % B;           // Q mod B
    Q = Q / B;             // Q div B
    i = i+1;
}
```

Q	R
4	0
2	0
1	1
0	

$4_{10} = 100_2$

Arch. Elab. - S. Orlando 7

Rappresentazioni ottale ed esadecimale

- Ottale: $B = 8$
- Esadecimale: $B = 16$
- Usate per facilitare la comunicazione di numeri binari tra umani, o tra il computer e il programmatore
- Esiste infatti un metodo veloce per convertire tra base 8 (o base 16) e base 2, e viceversa

Arch. Elab. - S. Orlando 8

Rappresentazione ottale

- $B = 8$, $A = \{0,1,2,3,4,5,6,7\}$
- Come convertire:
 - Sia dato un numero binario di 10 cifre: $d_9 \dots d_1 d_0$, il cui valore è:
$$\sum_{i=0}^9 2^i \cdot d_i$$
 - Raggruppiamo le cifre: da destra, a 3 a 3
 - Poniamo in evidenza la più grande potenza di 2 possibile:
 $(2^0 d_0) 2^0 + (2^2 d_2 + 2^1 d_1 + 2^0 d_0) 2^3 + (2^5 d_5 + 2^4 d_4 + 2^3 d_3) 2^6 + (2^8 d_8 + 2^7 d_7 + 2^6 d_6) 2^9$
 - I termini tra parentesi sono numeri compresi tra 0 e 7
 - si possono far corrispondere ai simboli dell'alfabeto ottale
 - I fattori messi in evidenza corrispondono alle potenze di $B=8$:
 $2^0=8^0 \quad 2^3=8^1 \quad 2^6=8^2 \quad 2^9=8^3$
 - Da binario ad ottale: $1001010111_2 = 1 \ 001 \ 010 \ 111 = 1127_8$
 - Da ottale a binario: $267_8 = 010 \ 110 \ 111 = 10110111_2$

Arch. Elab. - S. Orlando 9

Numeri naturali (interi) binari

- Il processore che studieremo (MIPS) rappresenta i numeri interi su 32 bit (32 bit = 1 word)
- I numeri interi (unsigned) rappresentabili su 32 bit sono allora:

```
0000 0000 0000 0000 0000 0000 0000 0000 = 0ten
0000 0000 0000 0000 0000 0000 0001 0000 = 1ten
0000 0000 0000 0000 0000 0000 0010 0000 = 2ten
...
1111 1111 1111 1111 1111 1111 1101 0000 = 4,294,967,293ten
1111 1111 1111 1111 1111 1111 1110 0000 = 4,294,967,294ten
1111 1111 1111 1111 1111 1111 1111 0000 = 4,294,967,295ten
```

Arch. Elab. - S. Orlando 11

Rappresentazione esadecimale

- $B = 16$, $A = \{0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f\}$
- Come convertire:
 - Si dato un numero binario di 10 cifre: $d_9 \dots d_1 d_0$, il cui valore è:
$$\sum_{i=0}^9 2^i \cdot d_i$$
 - Raggruppiamo le cifre: da destra, e a 4 a 4
 - Poniamo in evidenza la più grande potenza di 2 possibile:
 $(2^1 d_1 + 2^0 d_0) 2^8 + (2^3 d_3 + 2^2 d_2 + 2^1 d_1 + 2^0 d_0) 2^4 + (2^5 d_5 + 2^4 d_4 + 2^3 d_3 + 2^2 d_2 + 2^1 d_1 + 2^0 d_0) 2^0$
 - I termini tra parentesi sono numeri compresi tra 0 e 15
 - si possono far corrispondere ai simboli dell'alfabeto esadecimale
 - I fattori messi in evidenza corrispondono alle potenze di $B=16$:
 $2^0=16^0 \quad 2^4=16^1 \quad 2^8=16^2$
 - Da binario ad esadecimale: $10010101111_2 = 10 \ 0101 \ 1111 = 25f_{16}$
 - Da esadecimale a binario: $a67_{16} = 1010 \ 0110 \ 0111 = 101001100111_2$

Arch. Elab. - S. Orlando 10

Algoritmo di somma di numeri binari

- Per la somma di numeri rappresentati in binario possiamo adottare la stessa procedura usata per sommare numeri decimali
 - sommare via via i numeri dello stesso peso, più l'eventuale riporto:
- La tabella per sommare 3 cifre binarie è la seguente:

d0	d1	rip	RIS	RIP
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

Arch. Elab. - S. Orlando 12

Esempio di somma

- Sia $A = 13_{\text{dec}}$ e $B = 11_{\text{dec}}$

riporti:
A: 01101 +
B: 01011 =

11000 $\equiv 24_{\text{dec}}$

- L'algoritmo impiegato dal calcolatore per effettuare la somma è simile a quella carta e penna
 - le cifre sono prodotte una dopo l'altra, da quelle meno significative a quelle più significative

Overflow

- L'overflow si verifica quando il risultato è troppo grande per essere rappresentato nel numero finito di bit messo a disposizione dalle rappresentazioni dei numeri
 - ⇒ il riporto fluisce fuori
- Es.: la somma di due numeri di n-bit produce un numero non rappresentabile su n bit

1111
1111
+ 1001

11000

Sottrazione e numeri relativi

- L'algoritmo impiegato nei calcolatori per sottrarre numeri binari
 - è diverso da quello carta e penna, che usa la ben nota nozione di "prestito" delle cifre
- Non viene impiegata l'ovvia rappresentazione in modulo e segno per rappresentare i numeri relativi
 - si usa invece una particolare rappresentazione dei numeri negativi
- Questa particolare rappresentazione permette di usare lo stesso algoritmo efficiente già impiegato per la somma
- In pratica, nel calcolatore si usa lo stesso circuito
 - sia per la somma di numeri naturali (unsigned)
 - sia per la somma di numeri relativi (signed)

Possibili rappresentazioni

Modulo e Segno	One's Complement	Two's Complement
000 = + 0	000 = + 0	000 = + 0
001 = + 1	001 = + 1	001 = + 1
010 = + 2	010 = + 2	010 = + 2
011 = + 3	011 = + 3	011 = + 3
100 = - 0	100 = - 3	100 = - 4
101 = - 1	101 = - 2	101 = - 3
110 = - 2	110 = - 1	110 = - 2
111 = - 3	111 = - 0	111 = - 1

- Problemi:
 - bilanciamento**: nel Complemento a Due, nessun numero positivo corrisponde al più piccolo valore negativo
 - numero di zeri**: le rappresentazioni in Modulo e Segno, e quella in Complemento a Uno, hanno 2 rappresentazioni per lo zero
 - semplicità delle operazioni**: per il Modulo e Segno bisogna prima guardare i segni e confrontare i moduli, per decidere sia il segno del risultato, e sia per decidere se bisogna sommare o sottrarre. Il Complemento a uno non permette di sommare numeri negativi.
- Qual è quindi la migliore rappresentazione e perché?

Complemento a 2

- La rappresentazione in *complemento a 2* è quella adottata dai calcolatori per i numeri con segno (signed)
- Il bit più significativo corrisponde al segno (0 *positivo*, 1 *negativo*)
- MIPS: Numeri relativi (signed) su 32 bit:

[illegible]

Complemento a 2

- Rappresentazione di numeri in complemento a 2 su n bit dei numeri *signed*:
 - 0: 0.....0
 - $2^{n-1}-1$ numeri positivi:
 - 1 (001)
 - $2^{n-1}-1$ (*massimo*) (0111)
 - 2^{n-1} numeri negativi
 - $-|N|$ rappresentato dal numero *unsigned* ottenuto tramite la seguente operazione: $2^n - |N|$
 - -1: $2^n - 1$ (1.....1)
 - -2^{n-1} (*minimo*): $2^n - 2^{n-1} = 2^{n-1}$ (10.....0)

Complemento a 2

- Il valore corrispondente alla rappresentazione dei numeri *positivi* è quella solita
- Per quanto riguarda i numeri negativi, per ottenere direttamente il valore di un numero negativo su *n* posizioni, basta considerare
 - il **bit di segno** (=1) in posizione *n-1* con **peso: -2^{n-1}**
 - tutti gli altri bit in posizione *i* con peso 2^i

- **Dimostrazione:**

- $-|N|$ viene rappresentato in complemento a 2 dal numero unsigned $2^n - |N|$
- supponiamo che $2^{n-|N|}$ corrisponda alla n-upla

$$\begin{aligned}
 1 \, d_{n-2} \dots d_1 d_0 &\Rightarrow \\
 2^{n-1} |n| &= 2^{n-1} + 2^{n-2} * d_{n-2} + \dots + 2^1 * d_1 + 2^0 * d_0 \\
 -|n| &= -2^{n-1} + 2^{n-2} * d_{n-2} + \dots + 2^1 * d_1 + 2^0 * d_0 \\
 |n| &= -2^{n-1} + 2^{n-2} * d_{n-2} + \dots + 2^1 * d_1 + 2^0 * d_0
 \end{aligned}$$

Complemento a 2

- Dato un numero positivo N , con bit di segno uguale a 0
- Per ottenere la rappresentazione in complemento a 2 di $-N$ è possibile impiegare equivalentemente
 - Alg. 1: inverti tutti i bit (ovvero **Complementa a uno**) e somma 1
 - Alg. 2: inverti tutti i bit a sinistra della cifra “1” meno significativa

Regole per complementare a 2

- Esempio Alg. 1

00010101000

↓ Complementa a uno

11101010111 +
1
11101011000

- Esempio Alg. 2

00010101000

↓ Complementa a uno fino alla cifra 1 meno significativa

11101011000

Regole per complementare a 2

Alg. 1: inverti tutti i bit e somma 1 ([dimostrazione](#))

- La rappresentazione in complemento a 2 del numero negativo $-|N|$ è:

$1\ d_{n-2} \dots d_1\ d_0$,

- Il valore è:

$$-|N| = -2^{n-1} + 2^{n-2} * d_{n-2} + \dots + 2^1 * d_1 + 2^0 * d_0$$

- Allora:

$$\begin{aligned} |N| &= 2^{n-1} - 2^{n-2} * d_{n-2} - \dots - 2^1 * d_1 - 2^0 * d_0 = \leftarrow \text{Sommando e sottraendo 1} \\ &= (2^{n-1}-1) + 1 - (2^{n-2} * d_{n-2} + \dots + 2^1 * d_1 + 2^0 * d_0) = \\ &\quad \leftarrow (2^{n-2} * 1 + \dots + 2^1 * 1 + 2^0 * 1) - \\ &= (2^{n-2} * d_{n-2} + \dots + 2^1 * d_1 + 2^0 * d_0) + 1 = \\ &= (2^{n-2} * (1 - d_{n-2}) + \dots + 2^0 * (1 - d_0)) + 1 \end{aligned}$$

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

poiché (serie geometrica):

$$\sum_{i=0}^n q^i = \frac{1 - q^{n+1}}{1 - q}$$

⇒ Invertendo tutti i bit della rappresentazione di $-|N|$ otteniamo $0(1 - d_{n-2}) \dots (1 - d_0)$

dove $0 = 1 - d_{n-1}$

Il valore del numero **complementato** (positivo) è:

$$2^{n-2} * (1 - d_{n-2}) + \dots + 2^0 * (1 - d_0)$$

Sommando 1, otteniamo proprio il valore di $|N|$ sopra derivato

Regole per complementare a 2

Alg. 1: inverti tutti i bit e somma 1 ([dimostrazione - continuazione](#))

- Se N è un numero positivo, la rappresentazione di N sarà

$0\ d_{n-2} \dots d_1\ d_0$

il cui valore è:

$$2^{n-2} * d_{n-2} + \dots + 2^1 * d_1 + 2^0 * d_0$$

- Quindi il valore del numero negativo -N sarà uguale a

$$\begin{aligned} -N &= -2^{n-2} * d_{n-2} - \dots - 2^0 * d_0 = \\ &= (2^{n-1}-1) - (2^{n-1}-1) - 2^{n-2} * d_{n-2} - \dots - 2^0 * d_0 = \\ &= (2^{n-2} * 1 + \dots + 2^0 * 1) - (2^{n-1}-1) - \\ &\quad (2^{n-2} * d_{n-2} + \dots + 2^0 * d_0) = \\ &= -2^{n-1} + (2^{n-2} * (1 - d_{n-2}) + \dots + 2^0 * (1 - d_0)) + 1 \end{aligned}$$

Sommando e sottraendo $(2^{n-2}-1)$

⇒ Invertendo tutti i bit della rappresentazione di N otteniamo

$$1(1 - d_{n-2}) \dots (1 - d_0)$$

dove $1 = 1 - d_{n-1}$

Il valore del numero **complementato** (negativo) è:

$$-2^{n-2} * (1 - d_{n-2}) + \dots + 2^0 * (1 - d_0)$$

Sommando 1, otteniamo proprio il valore di -N sopra derivato

Estensione del numero bit della rappresentazione

- Regola: copiare il bit più significativo (bit di segno) negli altri bit

$$0010 \rightarrow 0000\ 0010$$

$$1010 \rightarrow 1111\ 1010$$

- L'estensione del bit di segno funziona anche per i numeri negativi

- il complemento a 2 del numero negativo 1010 è 110, indipendentemente dal numero di 1 iniziali (es. 1...1010)

- Esempio di applicabilità dell'estensione del segno:

- un operando di una istruzione macchina può essere più corto di una word (32 bit)

- l'operando deve essere esteso nella corrispondente rappresentazione a 32 bit prima che i circuiti della CPU possano effettuare l'operazione aritmetica richiesta dall'istruzione

Addizioni & Sottrazioni

- Operazioni di numeri binari in complemento a 2 sono *facili*
 - sottraiamo usando semplicemente l'algoritmo dell'addizione
 - il sottraendo (negativo) deve essere espresso in complemento a 2

- Esempio:

Sottrazione dei valori assoluti vs Somma dei numeri relativi in compl. 2

$$\begin{array}{r} 7 - \\ 6 = \\ 1 \end{array} \quad \begin{array}{r} 0111- \\ 0110= \\ 0001 \end{array} \quad \begin{array}{r} 7 + \\ (-6) = \\ 1 \end{array} \quad \begin{array}{r} 0111+ \\ 1010= \\ 0001 \end{array}$$

Scoprire gli Overflow

- No overflow se somma di numeri con segno discorde
- No overflow se sottrazione di numeri con segno concorde
- Overflow se si ottiene un numero con segno diverso da quello aspettato, ovvero se si sommano algebricamente due numeri con segno concorde, e il segno del risultato è diverso. Quindi otteniamo **overflow**
 - se sommando due positivi si ottiene un negativo
 - se sommando due negativi si ottiene un positivo
 - se sottraendo un negativo da un positivo si ottiene un negativo
 - se sottraendo un positivo da un negativo si ottiene un positivo
- Considera le operazioni $A + B$, e $A - B$
 - Può verificarsi overflow se $B \leq 0$?
 - Può verificarsi overflow se $A \leq 0$?

Addizioni & Sottrazioni

- Per sottrarre $N1 - N2$ (numeri di n-bit), $N1 > 0$ e $N2 > 0$
 - sommiamo $(N1 + (2^n - N2)) \bmod 2^n$
- Perché questo tipo di *somma algebrica* funziona ?
Perché in questo caso non possiamo avere un **overflow** ?
- se $N1 > N2$, il risultato dovrà essere **positivo**. Otterremo un bit di peso n che non verrà considerato (a causa del modulo 2^n)
 $\Rightarrow (N1 + 2^n - N2) \bmod 2^n = N1 - N2$ poiché $(N1 + 2^n - N2) > 2^n$
$$\begin{array}{r} 7 - \\ 6 = \\ 1 \end{array} \quad \begin{array}{r} 0111+ \\ 1010= \\ 10001 \end{array}$$
- se $N1 < N2$, il risultato dovrà essere **negativo**. Il modulo non avrà effetto, poiché $(N1 + 2^n - N2) < 2^n$
 $\Rightarrow (N1 + 2^n - N2) \bmod 2^n = 2^n - (N2 - N1)$, che corrisponde proprio alla rappresentazione in complemento a 2 di $(N2 - N1) > 0$
$$\begin{array}{r} 5 - \\ 6 = \\ -1 \end{array} \quad \begin{array}{r} 0101+ \\ 1010= \\ 1111 \end{array}$$

Scoprire gli Overflow

- Somma algebrica di due numeri positivi A e B la cui somma non può essere rappresentata su n bit in complemento a 2
 - Overflow se $A+B \geq 2^{n-1}$
 $A=01111$ $B=00001$ (**OVERFLOW** \Rightarrow due ultimi riporti **discordi**)
 $A=01100$ $B=00001$ (**NON OVERFLOW** \Rightarrow due ultimi riporti concordi)
$$\begin{array}{r} 01 \\ 01111+ \\ 00001= \\ 10000 \end{array} \quad \begin{array}{r} 00 \\ 01100+ \\ 00001= \\ 01101 \end{array}$$
- Somma algebrica di due numeri negativi A e B la cui somma non può essere rappresentata su n-bit in complemento a 2
 - Overflow se $|A|+|B| > 2^{n-1}$
 $A=10100$ $B=10101$ (**OVERFLOW** \Rightarrow due ultimi riporti **discordi**)
 $A=10111$ $B=11101$ (**NON OVERFLOW** \Rightarrow due ultimi riporti concordi)
$$\begin{array}{r} 10 \\ 10100+ \\ 10101= \\ 01001 \end{array} \quad \begin{array}{r} 11 \\ 10111+ \\ 11101= \\ 10100 \end{array}$$

Numeri razionali (a virgola fissa)

- Numeri con la **virgola** (o con il **punto**, secondo la convenzione anglosassone)
- Nel sistema di numerazione posizionale in base B, con *n* cifre intere e *m* cifre frazionarie:

$$d_{n-1} \dots d_1 d_0, d_{-1} d_{-2} \dots d_{-m}$$

Significatività:

$$B^{n-1} * d_{n-1} + \dots + B^1 * d_1 + B^0 * d_0 + B^{-1} * d_{-1} + B^{-2} * d_{-2} + \dots + B^{-m} * d_{-m}$$

- La notazione con *n+m* cifre è detta a **virgola fissa (fixed point)**
- Conversione da base 10 a base 2
 - $10,5_{\text{dieci}} \rightarrow 1010,1_{\text{due}}$

Arch. Elab. - S. Orlando 29

Numeri razionali (a virgola mobile)

- La notazione a virgola fissa (es.: *n=8* e *m=8*) non permette di rappresentare numeri molto grandi o molto piccoli
- per numeri grandi
 - utile spostare la virgola a **destra** e usare la maggior parte dei bit della rappresentazione per la parte intera
 - $10000000000,0100 \rightarrow$ parte intera non rappresentabile su *n=8* bit
- per numeri piccoli
 - utile spostare la virgola a **sinistra** e usare la maggior parte dei bit della rappresentazione per la parte frazionaria
 - $0,000000000000001 \rightarrow$ parte frazionaria non rappresentabile su *m=8* bit
- Notazione in **virgola mobile**, o FP (Floating Point)
 - si usa la notazione scientifica, con l'esponente per far fluttuare la virgola
 - Segno, Esponente, Mantissa $\Rightarrow (-1)^s * 10^E * M$
$$\begin{array}{r} 0,121 \\ +10^0 * 0,121 \\ 14,1 \\ +10^2 * 0,141 \\ -911 \\ -10^3 * 0,911 \end{array}$$
- Standard \Rightarrow Mantissa rappresentata come numero frazionario, con parte intera uguale a 0

Arch. Elab. - S. Orlando 31


Conversione della parte frazionaria

- Vogliamo convertire in base 2 a partire da una base B
- La parte frazionaria in base 2 che vorremmo ottenere sarà:
 - $0, d_{-1} d_{-2} \dots d_{-m}$ dove $d_{-i} \in \{0,1\}$ con significatività
 $2^{-1} * d_{-1} + 2^{-2} * d_{-2} + \dots + 2^{-m} * d_{-m}$
 - se **moltiplichiamo per 2**, la **virgola si sposta a destra**
 $2^0 * d_{-1} + 2^{-1} * d_{-2} + \dots + 2^{-m+1} * d_{-m}$
 - dopo aver moltiplicato per 2, la parte intera diventa del numero diventa d_{-1}
 $d_{-1}, d_{-2} \dots d_{-m}$
 - il processo di moltiplicazione deve essere **iterato** con la nuova parte frazionaria (fino a quando la parte frazionaria diventa nulla)

Processo di conversione di $0,43_{\text{dieci}}$		
	*2	Cifre frazionarie
0,43	0,86	0
0,86	1,72	1
0,72	1,44	1
0,44	0,88	0
0,88	1,76	1
0,76	1,52	1
0,52	1,04	1
0,04	0,08	0
0,08	0,16	0
0,16
0,011011100...due		

Arch. Elab. - S. Orlando 30

Numeri razionali (a virgola mobile)

- In base 2, l'esponente E si riferisce ad una potenza di 2
 - Segno, Esponente, Mantissa $\Rightarrow (-1)^s * 2^E * M$
 - Dati i bit disponibili per la rappresentazione FP, si suddividono in
 - 1 bit per il segno
 - gruppo di bit per E
 - gruppo di bit per M
- 
- Torneremo alla rappresentazione FP, e allo standard IEEE 754 di rappresentazione, quando parleremo delle operazioni FP e dei circuiti corrispondenti

Arch. Elab. - S. Orlando 32

Rappresentazione informazione alfanumerica

- Per rappresentare le lettere (maiuscole, minuscole, punteggiature, ecc.) è necessario fissare uno standard
- L'esistenza di uno standard permette la comunicazione di documenti elettronici (testi, programmi, ecc.), anche tra tra computer differenti
- ASCII (American Standard Code for Information)
 - in origine ogni carattere una stringhe 7 bit
 - 128 caratteri, da 0 a 7F
 - codici da 0 a 1F usati per *caratteri non stampabili (caratteri di controllo)*

0A	(Line Feed)	20	(Space)	41	A	61	a
0D	(Carriage Return)	2C	,
1B	(Escape)	2E	.	5A	Z	7A	z

30	0
...
39	9

Arch. Elab. - S. Orlando 33

ASCII e evoluzioni

- Codici ASCII esteso a 8 bit
- 256 codici diversi non bastano a coprire i set di caratteri usati, ad esempio, nelle lingue latine, slave, turche, ecc.
- IS (International Standard) con concetto di *code page*
 - /S 8859-1 è il codice ASCII a 8 bit per Latin-1 (esempio l'inglese o l'italiano con le lettere accentate ecc.)
 - /S 8859-2 è il codice ASCII a 8 bit per Latin-2 (lingue latine slave cecoslovacco, polacco, e ungherese)
 - ecc.
- UNICODE
 - ulteriore estensione (IS 10646) con codici a 16 bit (65536 codici diversi)
 - standard creato da un consorzio di gruppi industriali
 - i codici che vanno da 0000 a 00FF corrispondono a /S 8859-1
 - per rendere più facile la conversione di documenti da ASCII a UNICODE

Arch. Elab. - S. Orlando 34

UNICODE

- Gruppi di codici (*code points*) consecutivi associati ai più importanti alfabeti
 - 336 al Latino, 256 al cirillico, ecc.
- Molti gruppi di codici assegnati a cinese, giapponese e coreano
 - 1024 per i simboli fonetici
 - 20992 per gli ideogrammi (Han) usati in cinese e giapponese
 - 11156 per le sillabe Hangul usate in coreano
 - cinesi e giapponesi richiedono nuovi ideogrammi per le parole nuove (modem, laser, smileys) e quindi nuovi codici
 - molti problemi ancora aperti ...

Arch. Elab. - S. Orlando 35

Istruzioni macchina e codifica binaria

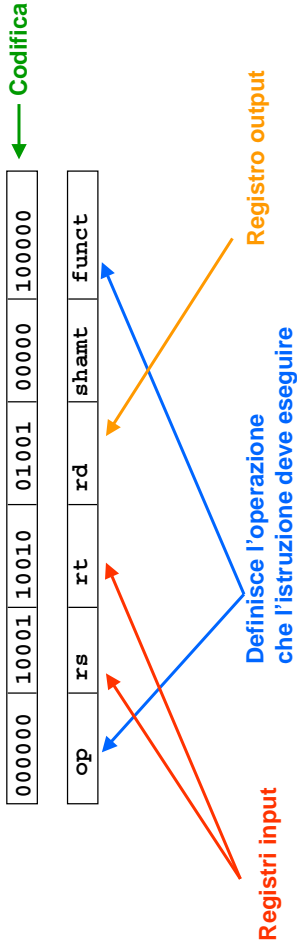
- Le istruzioni macchina, ovvero il linguaggio che la macchina (processore) comprende, hanno bisogno anch'esse di essere codificate in binario
 - devono essere *rappresentate* in binario in accordo ad un **formato** ben definito
- Il linguaggio macchina è molto restrittivo
 - il processore che studieremo sarà il **MIPS**, usato da Nintendo, Silicon Graphics, Sony
 - l'ISA del MIPS è simile a altre architetture RISC sviluppate dal 1980
 - le istruzioni aritmetiche del MIPS permettono solo operazioni elementari (add, sub, mult, div) tra coppie di operandi a 32 bit
 - le istruzioni MIPS operano su particolari supporti di memoria denominati **registri**, la cui lunghezza è di 32 bit = 4 Byte

Arch. Elab. - S. Orlando 36

Formato (codifica) delle istruzioni macchina

- Esempio:
`add $9, $17, $18` (semantica: $\$9 = \$17 + \$18$)
dove i registri sono identificati dai numeri 9, 17, 18

- Formato delle istruzioni:



Numeri binari magici

- $2^3 = 8$
 - $2^5 = 32$
 - $2^{10} = 1024$
 - 2^{20}
 - 2^{30}
 - 2^{40}
 - 2^{50}
- (8 bit = 1 Byte B)
(32 bit = 1 Word)
La dimensione della word dipende dal processore.
Esistono processori dove la Word è di
 $2^6 = 64$ bit (oppure di $2^4 = 16$ bit)
(K Kilo Migliaia - KB (kilobytes) - Kb (kilobits))
(M Mega Milioni - MB)
(G Giga Miliardi - GB)
(T Tera Migliaia di Miliardi - TB)
(P Peta Milioni di Miliardi - PB)

- 8 bit (1 B) è un'unità fondamentale:
- è l'unità di allocazione della memoria
 - codici ASCII e UNICODE hanno dimensione, rispettivamente, 1 B e 2 B

Informazione e memoria

- L'informazione, opportunamente codificata, ha bisogno di essere memorizzata nel calcolatore per essere utilizzata.
- In particolare i programmi (e i dati) devono essere trasferiti nella **memoria principale** del computer per l'esecuzione
- Organizzazione della memoria
 - sequenza di **celle** (o **locazioni**) di lunghezza prefissata
 - ogni cella è associata con un numero (chiamato **indirizzo**)
 - se un indirizzo è espresso come numero binario di m bit
 - sono indirizzabili 2^m celle diverse (da 0 a $2^m - 1$)
 - indirizzi consecutivi \Rightarrow celle contigue
 - nelle memorie attuali, ogni cella di memoria è lunga
 - $2^3 = 8$ bit = 1 Byte (memoria indirizzabile al Byte)
- I Byte consecutivi sono organizzate in gruppi
 - ogni gruppo è una **Word**
 - processori a **64 bit** (Word di 8 Bytes) e a **32 bit** (Word di 4 Bytes)
 - le istruzioni aritmetiche operano su Word
 - la dimensione della Word stabilisce qual è il massimo intero rappresentabile

0	8 bits of data
1	8 bits of data
2	8 bits of data
3	8 bits of data
4	8 bits of data
5	8 bits of data
6	8 bits of data
...	

Codici per correggere o scoprire errori

- Le memorie elettroniche (o magnetiche come i dischi) memorizzano bit usando meccanismi che possono occasionalmente generare errori
 - es.: un bit settato ad 1 viene poi letto uguale a 0, o viceversa
- Formalizziamo il concetto di errore in una codifica a n bit
 - C codifica corretta, C' codifica letta
 - Distanza di Hamming** tra le codifiche
 - $H(C, C')$: numero di cifre binarie differenti a parità di posizione
 - Possibili situazioni
 - $H(C, C') = 0$: significa che C e C' sono uguali (OK)
 - $H(C, C') = 1$: significa che C e C' differiscono per 1 solo bit
 - $H(C, C') = 2$: significa che C e C' differiscono per 2 soli bit
 - ecc.

Parità

- Per scoprire gli errori *singoli*, ovvero per accorgersi se $H(C, C')=1$
 - **aggiungiamo** bit di **parità** alla codifica
 - bit aggiuntivo posto a 1 o a 0
 - affinché il numero di bit 1 nelle varie codifiche sia pari (dispari)
 - se si verifica un errore **singolo** (un numero di errori **dispari**) in C' , allora il numero di bit 1 non sarà più pari
 - purtroppo, con un singolo bit di parità, non scopriremo mai un numero di errori **doppio**, o in generale **pari**
- In verità, usare un bit di parità significa usare una **codifica non minimale** nella rappresentazione dell'informazione
 - una **codifica minimale** usa tutte stringhe possibili
 - in questo si usano solo la **metà** delle stringhe permesse su **$n+1$ bit**
 - la **distanza “minima” di Hamming** tra coppie di codifiche permesse è **2**
 - es.: $n=2$ con 1 bit di parità (no. bit pari)
 - Stringhe (codifiche) permesse
 - 000 011 101 110
 - Stringhe (codifiche) **non** permesse
 - 001 010 100 111

Arch. Elab. - S. Orlando 41

Codici correttori

- In generale, possiamo avere più di un bit per **correggere** o **scoprire** possibili errori multipli
 - n sono i bit della **codifica minimale**
 - m sono i bit della codifica estesa, $m > n$
 - $r = m - n$ sono i **bit ridondanti** che estendono la codifica minimale
- solo 2^n delle 2^m codifiche possibili sono valide
 - per ogni codifica su n bit, i rimanenti r **bit ridondanti** possono essere codificati soltanto in modo fisso affinché la codifica sia valida e corretta
 - se singolo bit di parità
 - $m = n + 1$
 - solo 2^n delle 2^{n+1} codifiche possibili sono valide (solo metà)

Arch. Elab. - S. Orlando 42

Distanza di Hamming e codici correttori

- E' possibile definire una codifica **non** minimale su m bit per **correggere** o **scoprire** errori
- La **distanza di Hamming della codifica** è definita come:
 - la distanza di Hamming “minima” tra le varie coppie di codici validi
- Nota che la distanza di Hamming in una codifiche minimale è 1
- Codifica non minimale su 6 bit con **distanza di Hamming** uguale a 3
 - solo 4 codici validi: 000000 000111 111000 111111
 - $H(000000, 000111)=3$ $H(000000, 111000)=3$
 - $H(000000, 111111)=6$ $H(000111, 111000)=6$
 - $H(000111, 111111)=3$ $H(111000, 111111)=3$
 - la codifica di sopra permette di
 - **scoprire** fino a 2 errori
 - es.: se siamo sicuri che ci possono essere al massimo 2 errori, la distanza tra una stringa corretta C e la stringa erronea C' è $2 < 3$. C' non può essere scambiato per un codice corretto perché la distanza di Hamming del codice è 3.
 - **correggere** fino a 1 errore
 - es.: 010000 è più vicino a 000000

Arch. Elab. - S. Orlando 43

Distanza di Hamming e codici correttori

- Per **scoprire** fino a d errori su singoli bit
 - è necessario che la distanza di Hamming della codifica sia **$d+1$**
 - supponiamo di avere una codifica siffatta
 - supponiamo che C' sia una codifica erronea di C tale che:
 $1 < H(C, C') \leq d$
 - C' non può essere scambiato per una codifica valida, perché in questo caso dovrebbe essere vero che: $H(C, C') \geq d+1$
- Per **correggere** fino a d errori su singoli bit
 - è necessario che la distanza di Hamming della codifica sia **$2d+1$**
 - supponiamo di avere una codifica siffatta
 - supponiamo che C' sia una codifica erronea di C tale che:
 $1 < H(C, C') \leq d$
 - C' non può essere scambiato per una codifica valida, perché in questo caso dovrebbe essere vero che: $H(C, C') \geq 2d+1$
 - poiché C è la codifica valida più vicina a C' , possiamo pensare che C sia la codifica corretta e correggere l'errore

Arch. Elab. - S. Orlando 44

Codici correttori

- Esiste un algoritmo dovuto a Hamming (1950) che permette di determinare una codifica con un numero minimo di *bit di ridondanza* per la correzione degli errori
- Esempio:
 - Numero minimo di check bit (bit ridondanti) per correggere *errori singoli* ($d=1$)
 - I check bit devono essere configurati in modo che la distanza di Hamming tra le codifiche valide sia $2d+1=3$

Word size	Check bits	Total size	Percent overhead
8	4	12	50
16	5	21	31
32	6	38	19
64	7	71	11
128	8	136	6
256	9	265	4
512	10	522	2