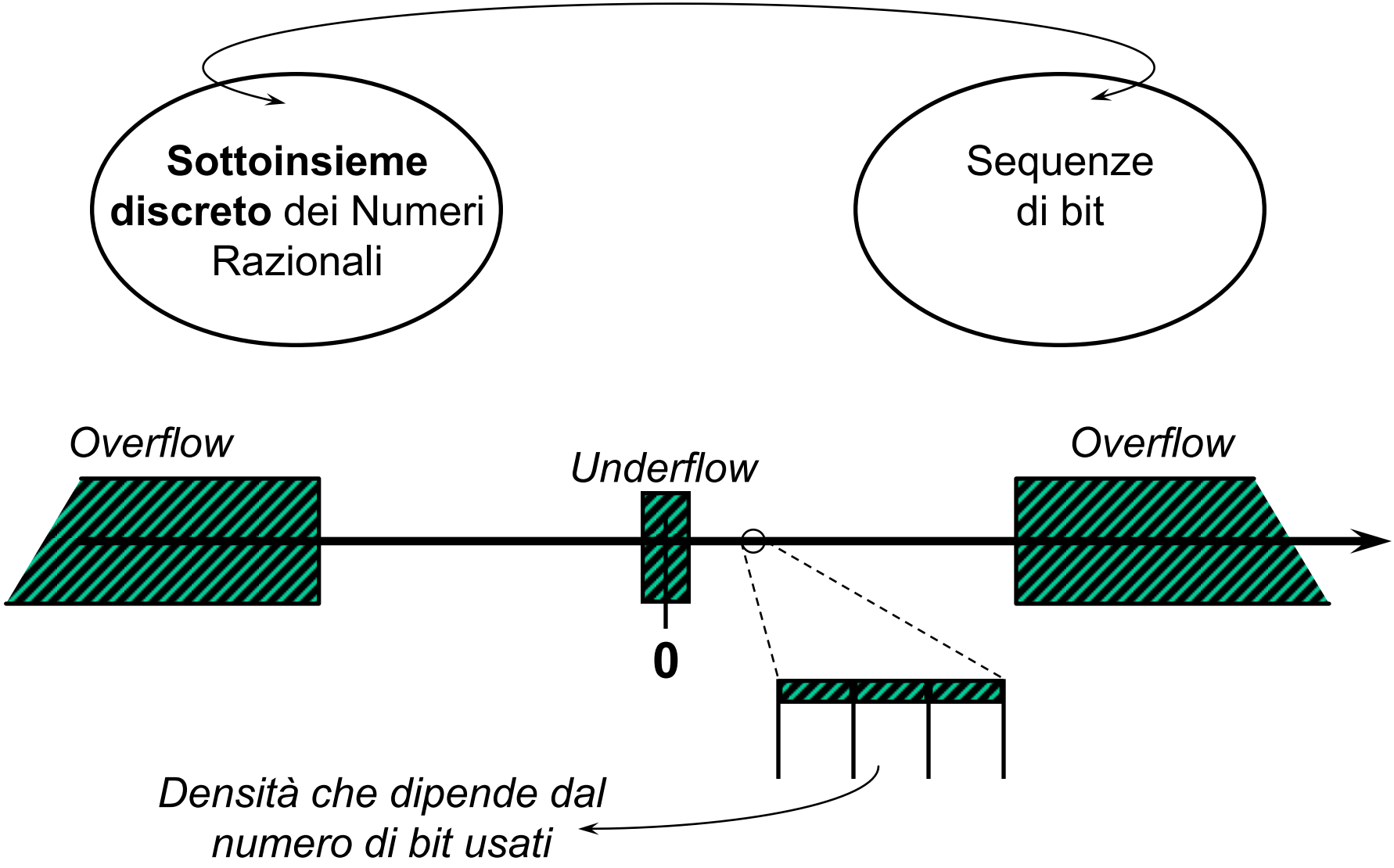
Numeri Reali



Numeri Reali – Virgola fissa (1/5)

- Si usa un numero fisso di bit per la parte intera ed un numero fisso di bit per la parte frazionaria.
- Sia *r* un numero reale da rappresentare. Di seguito, indichiamo con I(r) la parte intera e con F(r) la parte frazionaria di *r*
- Siano *p* i bit per rappresentare *r*: *f* per la parte frazionaria e (p-f) i bit la parte intera:

$$R = a_{p-f-1}\ldots a_0\, a_{-1}\ldots a_{-f} \qquad r \cong \sum_{i=-f}^{p-f-1} a_i \beta^i = a_{p-f-1}\beta^{p-f-1} + \ldots + a_0\beta^0 + \underbrace{a_{-1}\beta^{-1} + \ldots + a_{-f}\beta^{-f}}_{\textit{parte frazionaria}}$$

Esempio: +1110.01 in base *due* vale +14.25 in base *dieci*

- NB: La virgola non si rappresenta
- La parte intera I(r) si rappresenta con le tecniche note
- Per la parte frazionaria si usa il procedimento seguente:

Numeri Reali – Virgola fissa(2/5)

Si usa la così detta procedura *parte frazionaria-parte intera:*

$f_0 = F(r)$
Se $f_0 \neq 0$ eseguire la seguente procedura iterativa:

$f_{-1} = F(f_0 * 2)$ $a_{-1} = I(f_0 * 2)$
$f_{-2} = F(f_{-1} * 2)$ $a_{-2} = I(f_{-1} * 2)$
...
fino a che f_{-j} è uguale a zero oppure si è raggiunta la precisione desiderata

Esempio:

***p* = 16 e *f* = 5**
r = +331,6875

Numeri Reali – Virgola fissa(3/5)

QUOZIENTE	RESTO
331	-
165	1
82	1
41	0
20	1
10	0
5	0
2	1
1	0
0	1

- **Dalla rappresentazione dei numeri naturali: 331 ⇔ 101001011;**

Numeri Reali – Virgola fissa(4/5)

F	I
f_{-1}=F(0.6875*2=**1.375**)=0.375	a_{-1}=I(1.375)=**1**
f_{-2}=F(0.375*2=**0.75**)=0.75	a_{-2}=I(0.75)=**0**
f_{-2}=F(0.75*2=**1.5**)=0.5	a_{-3}=I(1.5)=**1**
f_{-3}=F(0.5*2=**1.0**)=0	a_{-4}=I(1.0)=**1**

- r = +10100101110110 $\Rightarrow$ R = 0010100101110110
- Parte frazionaria ->

$$1*\ 2^{-1} + 0*2^{-2} + 1*2^{-3} + 1*2^{--4} \text{ -> } 0{,}5 + 0{,}125 + 0{,}0625 \text{ -> } 0{,}6875$$

- Dall'algoritmo precedente: 0,6875 $\Leftrightarrow$ 0,1011
- r = (+101001011,1011) base *due*

Numeri Reali – Virgola fissa(5/5)

ERRORE DI TRONCAMENTO

Rappresentare $r = 2.3$ con f = 6.

F	I
f_{-1}=F(0.3*2=**0.6**)=0.6	a_{-1}=I(**0.6**)=**0**
f_{-2}=F(0.6*2=**1.2**)=0.2	a_{-2}=I(1.2)=**1**
f_{-3}=F(0.2*2=**0.4**)=0.4	a_{-3}=I(0.4)=**0**
f_{-4}=F(0.4*2=**0.8**)=0.8	a_{-4}=I(0.8)=**0**
f_{-5}=F(0.8*2=**1.6**)=0.6	a_{-5}=I(0.6)=**0**
f_{-6}=F(0.6*2=**1.2**)=0.2	a_{-6}=I(0.8)=**1**

Per esprimere r sono necessarie un numero infinito di cifre!

(10.0 1001 1001 1001… = 10.01<u>1001</u> dove <u>1001</u> è la parte periodica)

Ne abbiamo a disposizione solo 6. Quindi si opera un <u>*troncamento*</u> alla 6ª cifra dopo la virgola.

Quindi, in realtà si rappresenta non r ma il numero r'

r' = 10.010011

$r' = 2 + 2^{-2} + 2^{-5} + 2^{-6} = 2 + 0.25 + 0.03125 + 0.015625 = 2.296875$

L'errore di troncamento è (2.3-2.296875)=0,00312499 ($< 2^{-6} = 0.015625$)

CALCOLO DI FISICA ASTRONOMICA

$m_e = 9 \times 10^{-28}$ gr; $m_{sole} = 2 \times 10^{+33}$ gr $\Rightarrow$ Intervallo di variazione $\approx 10^{+60}$.

Sarebbero quindi necessarie almeno 62 cifre, di cui 28 per la parte frazionaria.

Si hanno tante cifre perché l'intervallo da rappresentare è grande $\Rightarrow$ si separa l'intervallo di rappresentabilità dalla precisione, cioè dal numero di cifre.

Notazione Scientifica

$r = \pm m \cdot \beta^e$ m = mantissa; e = esponente

L'intervallo di rappresentabilità è fissato dal numero di cifre dell'esponente.

La precisione è fissata dal numero di cifre della mantissa.

Diverse rappresentazioni

Si fissa una forma standard

3.14 $0.314\ 10^{+1}$ $31.4\ 10^{-1}$

Numeri Reali – Virgola mobile(2/10)

Rappresentazione di un numero binario in virgola mobile:

+1110.01 numero reale binario in virgola fissa

esponenti espressi in binario

$+1.11001 \cdot due^{+11}$ … … $+111001 \cdot due^{-10}$

(alcune possibili rappresentazioni in virgola mobile)

Numeri reali *normalizzati*:

- mantissa con parte intera costituita da un solo bit di valore 1
- rappresentazione è una tripla costituita da tre numeri naturali

$$r \leftrightarrow R = \langle s, \quad E, \quad F \rangle$$

Standard IEEE 754-1985

La rappresentazione R è composta da tre naturali (s, E ed F), dove:

s = codifica del segno (1 bit)
F = codifica della parte frazionaria della mantissa su G bit
E = codifica dell'esponente su K bit

$$r = (s == 0)? \; [+(1+f) \cdot due^e)] : [-(1+f) \cdot due^e)]$$

$f = F/2^G$ è la parte frazionaria della mantissa (m=1+f= $1+F/2^G$)

$e = +E - (2^{K-1} - 1)$ è l'esponente rappresentato dal numero naturale E (*rappresentazione con polarizzazione*)

Conseguenza dell' "uno implicito" $\Rightarrow$ *lo zero non è rappresentabile!*

Numeri Reali – Virgola mobile(4/10)

Half precision: 16 bit, $K = 5$ e $G = 10$. Esempi di trasformazione R => r

Esempio 1

R = {1,10011,1110100101} rappresenta il numero reale negativo con:

$f = F/2^G = F/2^{dieci} = 0.1110100101$

$e = E - (2^{K-1} - 1) = +10011 - (+01111) = +100$ (*bias*=*quindici*)

r = $-1.1110100101 \cdot due^{+quattro} = -11110.100101$

r = -30.578125 in base *dieci*

Esempio 2

R = {0,01111,0000000001} rappresenta il numero reale positivo:

$f = F/2^M = 1/2^G = due^{-dieci} = 0.0009765625$

$e = E - (2^{K-1} - 1) = 01111-01111 = 15-15=0$

r = $+1.0000000001 \cdot due^{+zero} = 2.0009765625 \cdot due^{+zero} = 2.0009765625$

r = + 2.0009765625 in base *dieci*

Half precision, esempi di trasformazione r => R

Esempio A – Rappresentare **r=2** *in half precision*
La rappresentazione di **r**= *2 è* R = {0,10000,0000000000}. Infatti, decodificando:
$f = F/2^G = 0$
$e = E - (2^{K-1} - 1) = 10000\text{-}01111 = 1$
r = $+1.0000000000 \cdot due^{+1}$ = 2 in base *dieci*

Esempio B – Rappresentare **r=0.5** *in half precision*
La rappresentazione di **r** = 0.5 *è* R = {0,01110,0000000000}. Infatti, decodificando:
$f = F/2^G = 0$
$e = E - (2^{K-1} - 1) = 01110\text{-}01111 = -1$
r = $+1.0000000000 \cdot due^{-1}$ = 0.5 in base *dieci*

Half precision, esempi di trasformazione r => R

Esempio C – Rappresentare **r=2.3** *in half precision*
Sapendo che **r** ha rappresentazione in virgola fissa 10.01001,
iniziamo a portarlo in forma normalizzata: 1.001001 $\cdot$ *due*$^{+1}$

Ora ci servono 10 cifre per la parte frazionaria della mantissa, le restanti verranno scartate, provocando il consueto *errore di troncamento*:

1.00 1001 1001 ~~*1001 1001*~~*...* $\cdot$ *due*$^{+1}$

F (prime 10 cifre a destra della virgola. Le altre vengono ignorate)

A questo punto la rappresentazione è immediata (per trovare *E* si faccia riferimento all'esempio A: **R**={0,10000,0010011001}

Numeri Reali – Virgola mobile(7/10)

Esempio (numeri con modulo massimo, i cosiddetti ± ∞):

R={*0*,11111,1111111111} (massimo numero rappresentabile, +∞)
R={*1*,11111,1111111111} (minimo numero rappresentabile, -∞)

$f = F/2^{G} = F/2^{dieci} = 0.1111111111$
$e = + E -(2^{K-1} - 1) = +11111 - (+01111) = +10000$ (+*sedici*)

$|\mathbf{r}| = 1.1111111111 \cdot due^{+10000} < 2^{(2^{(K-1)}+1)} = \mathbf{2^{bias+2}} = 2^{17} = 131072$
$|\mathbf{r}| = 131008 \cong 1.3 \cdot 10^{+5}$ in base *dieci*

Riassumendo, nella rappresentazione *half precision*:
il «+∞» corrisponde a $\cong 2^{+17}$
il « -∞» corrisponde a $\cong -2^{+17}$

NB: L'intervallo di rappresentabilità [-∞, +∞] è approssimabile dal seguente: [$\cong -2^{bias+2}$, $\cong 2^{bias+2}$]

Numeri Reali – Virgola mobile(8/10)

Esempio (numeri con modulo minimo, i cosiddetti ± 0):

R={*0*, 00000,0000000000} (minimo numero positivo, +0)
R= {*1*,00000,0000000000} (massimo numero negativo, -0)

$f = F/2^{G} = F/2^{dieci} = 0.0000000000$
$e = + E -(2^{K-1} - 1) = +00000 - (+01111) = -01111 =$ *-quindici*

$|\mathbf{r}| = 1.0000000000 \cdot due^{-01111} = \mathbf{2^{-bias}}$
$|\mathbf{r}| = 2^{-15} \cong 0.31 \cdot 10^{-4}$

Riassumendo, nella rappresentazione *half precision*:
il «+0» corrisponde a $2^{-15} \cong 0.31 \cdot 10^{-4}$
il « -0» corrisponde a $-2^{-15} \cong -0.31 \cdot 10^{-4}$

Numeri Reali – Virgola mobile(9/10)

Standard IEEE 754-1985 prevede:

Rappresentazione in ***single precision*** su **32 bit** con ***K* = 8** e ***G* = 23**
Intervallo di rappresentabilità:

massimo modulo $\cong 2^{(bias+2)} = 2^{+129} \cong 6.8 \cdot 10^{+38}$
minimo modulo $= 2^{-bias} = 2^{-127} \cong 0.58 \cdot 10^{-38}$

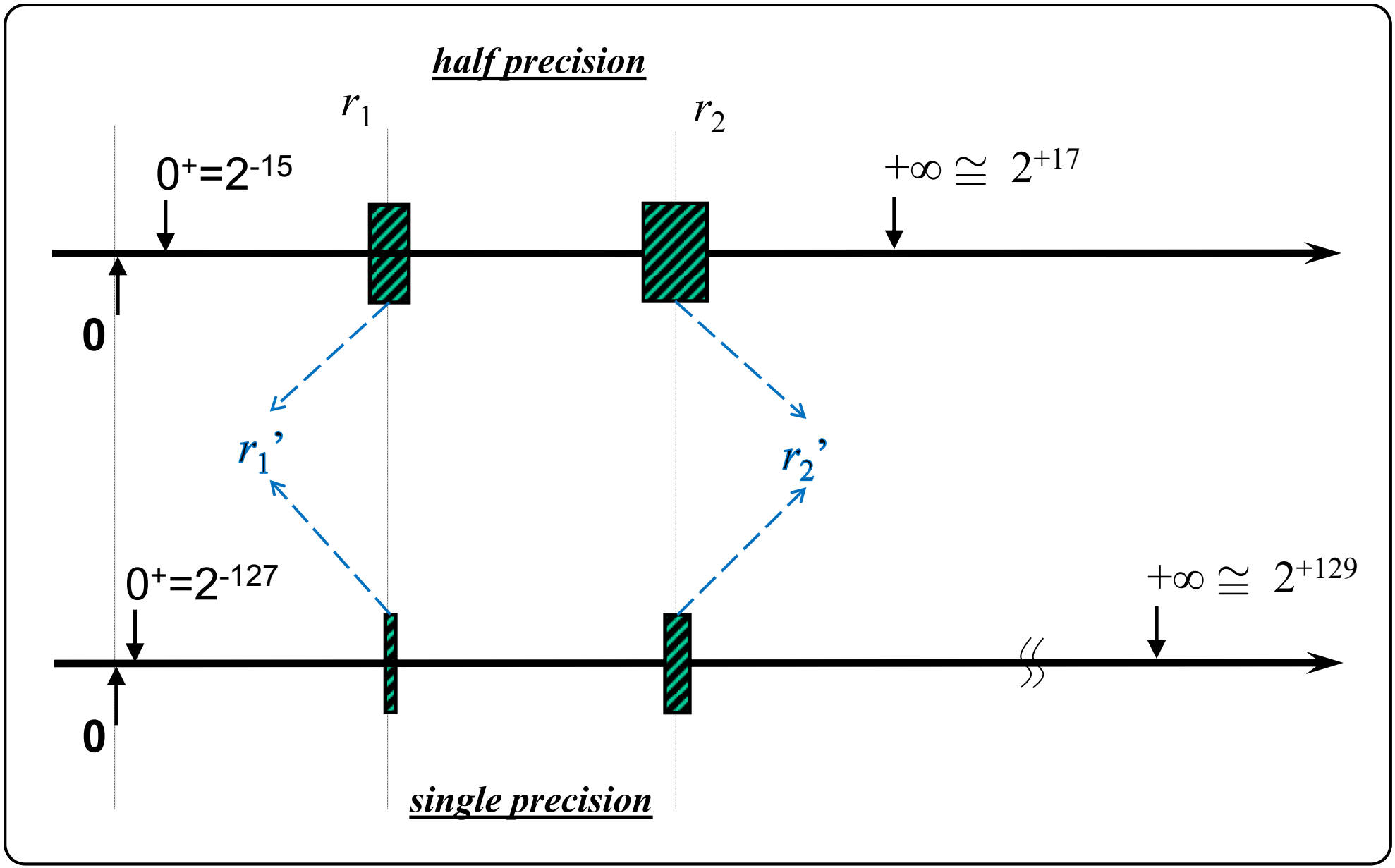
Rappresentazione in ***double precision*** su **64 bit** con ***K* = 11** e ***G* = 52**
Intervallo di rappresentabilità:

massimo modulo $\cong 2^{(bias+2)} = 2^{+1025} \cong 3.6 \cdot 10^{+308}$
minimo modulo $= 2^{-bias} = 2^{-1023} \cong 1.1 \cdot 10^{-308}$

IEEE 754-2008 aggiunge:
rappresentazione in ***quadruple precision*** su **128 bit** con ***K* = 15** e ***G* = 112**

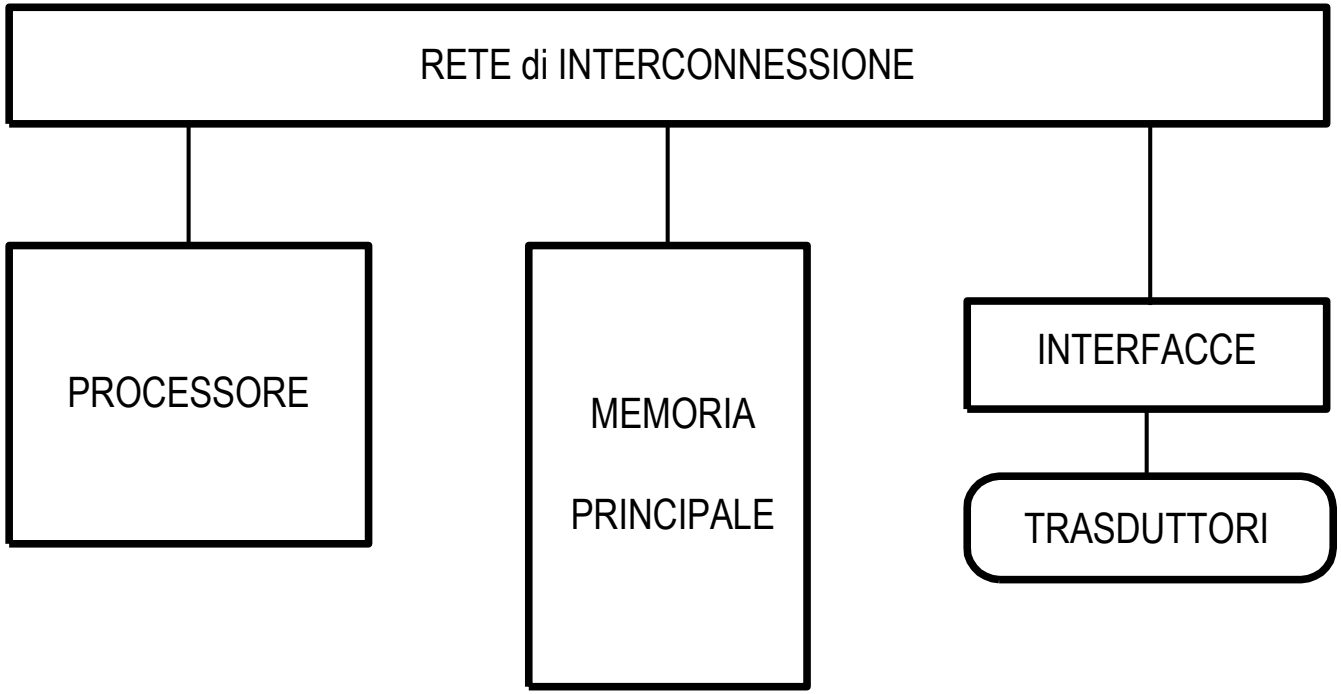
massimo modulo $\cong 2^{(bias+2)} = 2^{+16385} \cong 1.2 \cdot 10^{+4932}$
minimo modulo $= 2^{-bias} = 2^{-16383} \cong 1.6 \cdot 10^{-4932}$

Numeri Reali - Virgola mobile (10/10)



Schema a blocchi di un semplice calcolatore

Architettura di von Neumann (1946)



Funzionamento

La memoria contiene dati e programmi (istruzioni) codificati in forma binaria

Il processore ripete all'infinito le azioni seguenti :

- preleva una nuova istruzione dalla memoria
- la decodifica
- la esegue

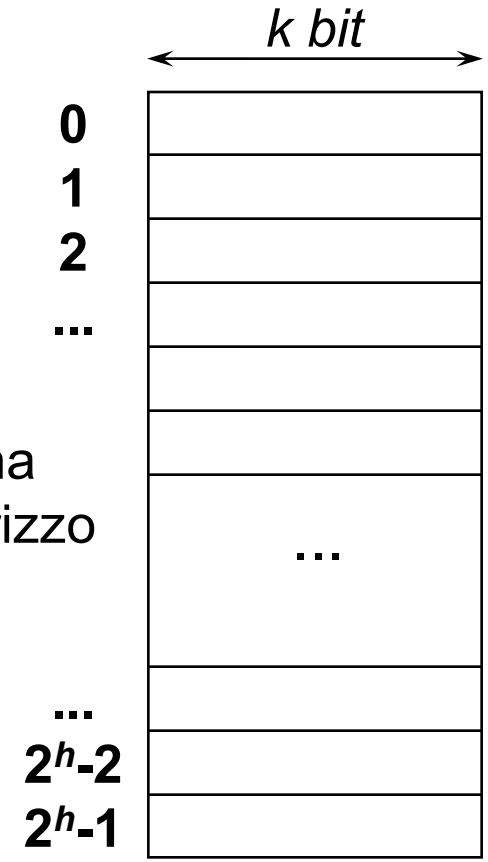
  L'esecuzione di un'istruzione può comportare

  » Elaborazione e/o Trasferimento (memoria $\leftrightarrow$ processore, I/O $\leftrightarrow$ processore)

Le periferiche permettono al calcolatore di interagire con il mondo esterno

Struttura logica della memoria

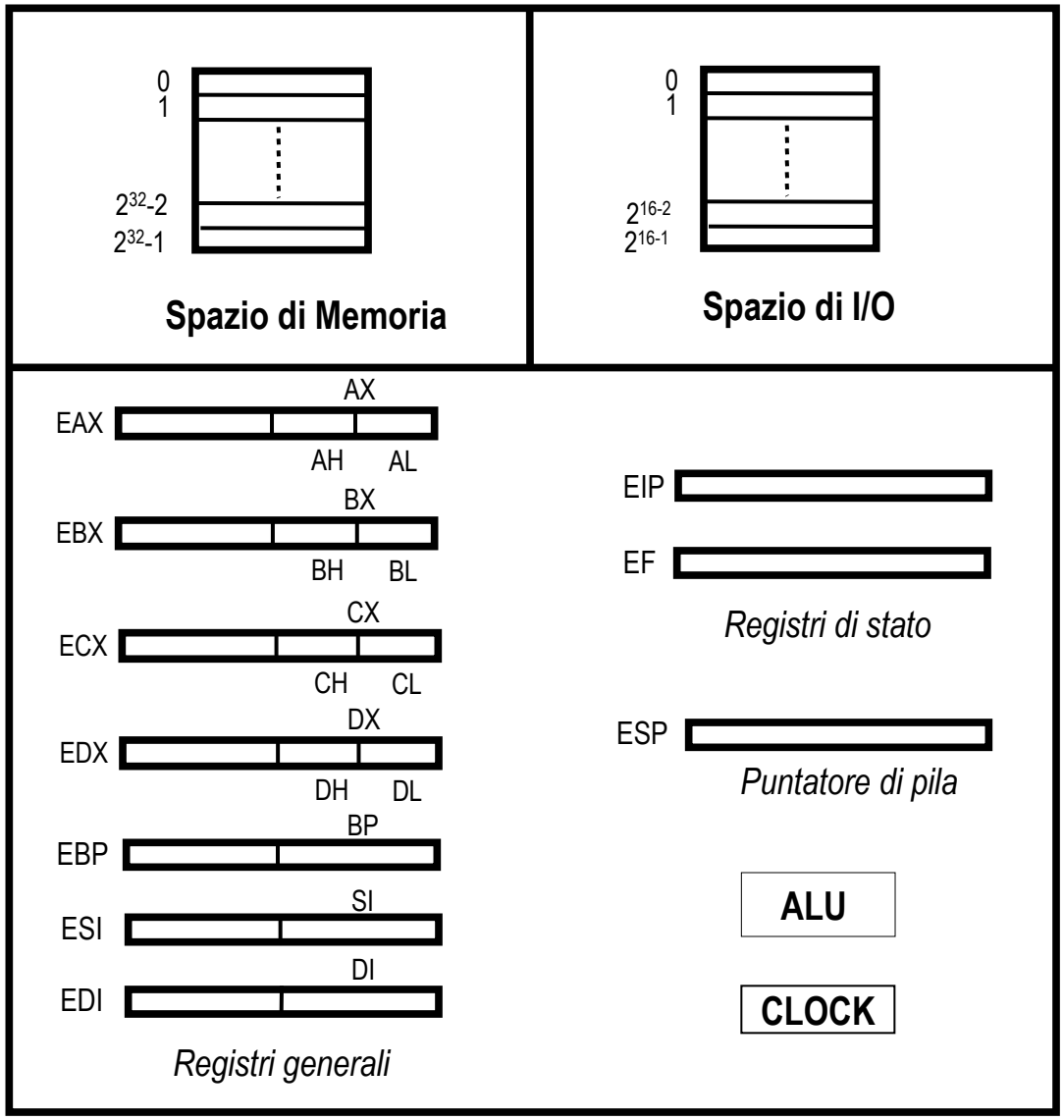
- 2^h *celle* di *k* bit ognuna
- Ogni cella ha un indirizzo



OPERAZIONI

1. LETTURA di UNA cella
2. SCRITTURA di UNA cella

Struttura logica del processore (1)



Livelli di astrazione dei Linguaggi

Esistono linguaggi a vari livelli di astrazione

Linguaggio macchina — sequenze di bit (difficile da leggere e capire)

```
0001001000110100
```

Linguaggio Assembler — istruzioni macchina espresse con nomi simbolici (dipendente dalla macchina)

```
MOV $35, %AL
```

Linguaggi ad Alto livello indipendenti dalla macchina

```
int main()
{
    …
}
```

Istruzioni MOV e JMP

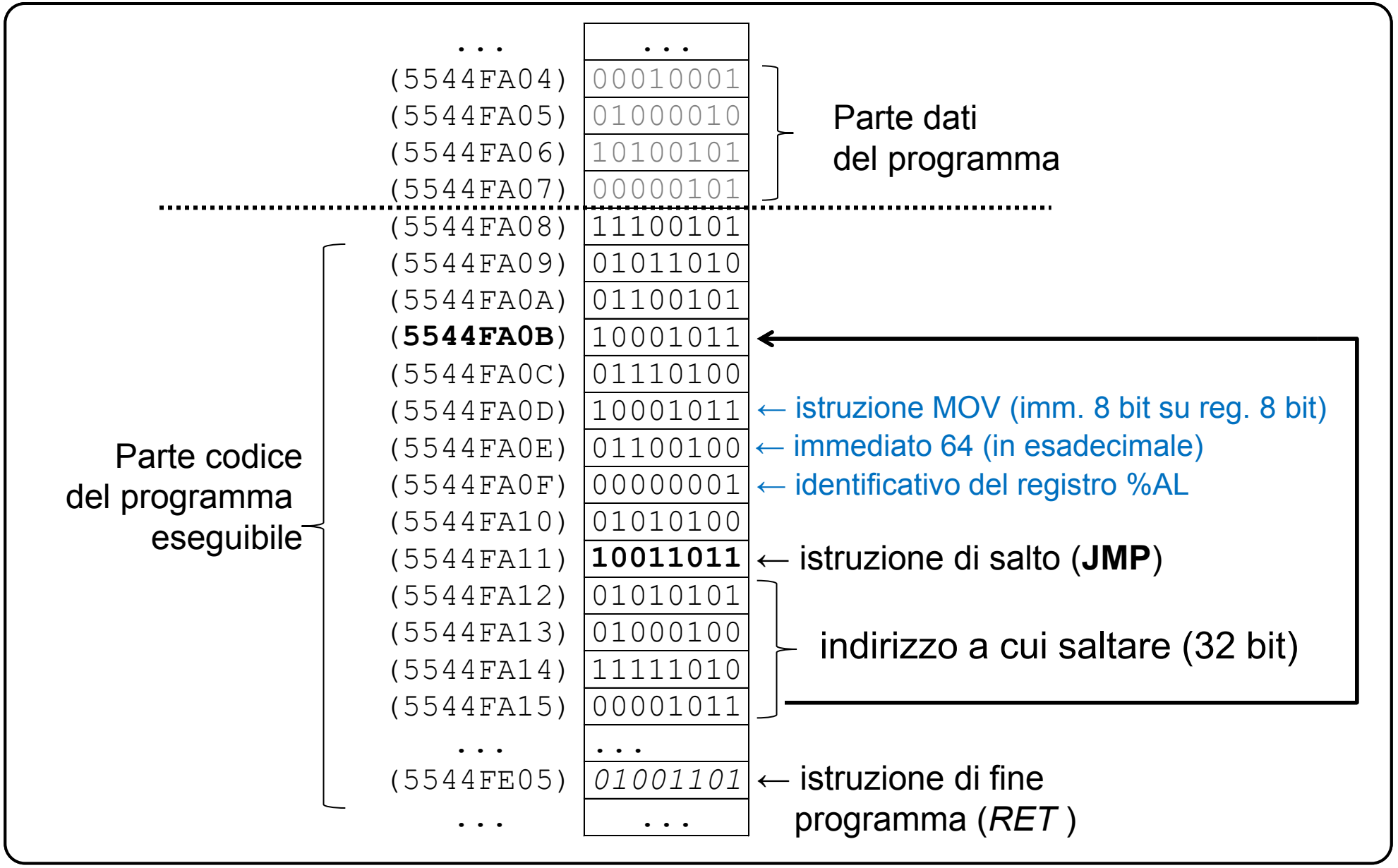
L'istruzione MOV del processore DP.86.32 permette di:

- Copiare un valore (immediato) in un registro
  Es: MOV $0x64, %AL
- Copiare il contenuto di una cella di memoria in un registro:
  Es: MOV (5544FA04), %BH
- Copiare il contenuto della cella puntata da un registro a 32 bit in un altro registro a 8 bit:
  Es: MOV (%EBX), %CH
- Copiare il contenuto delle **2 celle consecutive** di cui la prima è puntata da un registro a 32 bit in un altro registro a 16 bit:
  Es: MOV (%EDX), %DX
- Copiare il contenuto delle **4 celle consecutive** di cui la prima è puntata da un registro a 32 bit in un altro registro a **32 bit**:
  Es: MOV (%ECX), %EAX

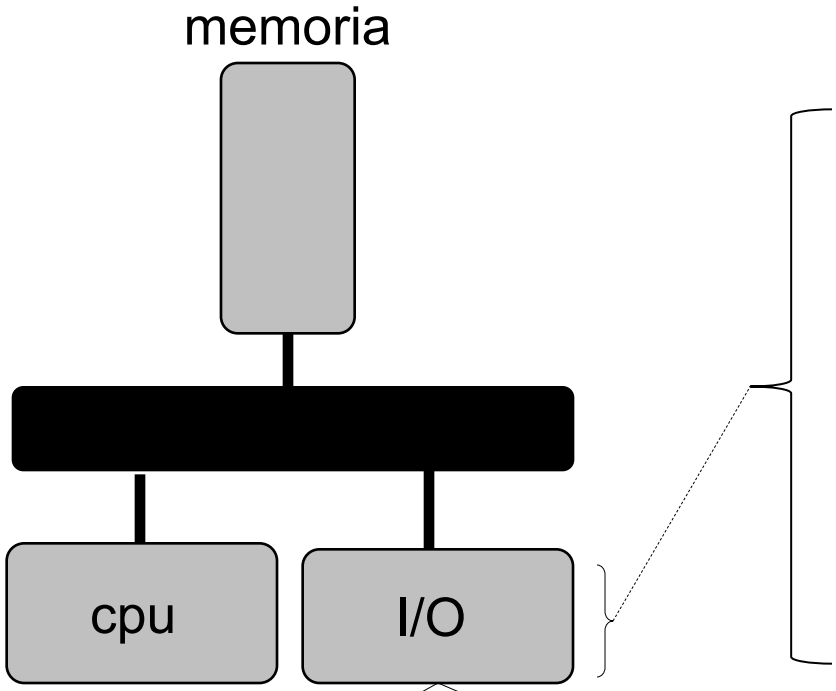
L'istruzione JMP del processore DP.86.32 permette di saltare all'indirizzo (sempre a 32 bit) specificato:

Es: JMP 5544FA0B

Esempio di programma in memoria



Lo spazio di ingresso/uscita

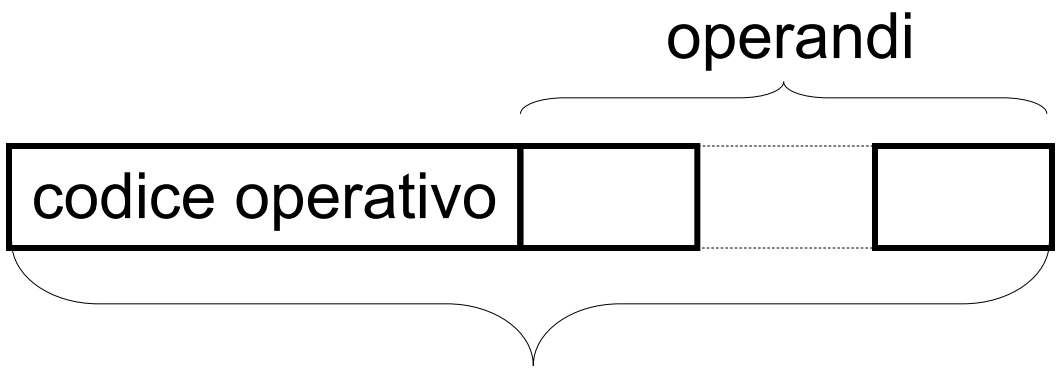


	. . .	. . .	
	(C1CF)	10010001	← Porta di IN
video	(C1D0)	*01110000*	← Porta di OUT
	(C1D1)	10100101	← Porta di OUT
	(C1D2)	00100101	← Porta di OUT
tastiera	(C1D3)	***01000001***	← Porta di IN
	(C1D4)	01011010	← Porta di OUT
	. . .	. . .	

```
IN (C1D3), %AL          # carica in AL l'esad. 41, ossia 'A' (cfr cod. ASCII)
OUT $0x70, (C1D0)       # stampa a video 0x70, ossia il carattere 'p'
```

Formato delle istruzioni (1/2)

Le istruzioni sono codificate come sequenze di bit.



Esempio

Codice Operativo	**Operandi**	
`10001011`	`01100100`	`00000001`
MOV	immediato 0x64	%AL

Formato delle istruzioni (2/2)

OPCODEsize source destination

OPCODEsize source

OPCODEsize destination

Dove size può essere:

B (byte): 8 bit
W (word): 16 bit
L (Long): 32 bit

Esempi:

```
MOVB  (%EBX), %AL
MOVW (%EBX), %AX
MOVL  (%EBX), %EAX
```

Esempio di programma in lunguaggio assembler

```
3F0D0100    00000000                Dato da elaborare (una WORD)
3F0D0102    00000000                Risultato (un BYTE)
3F0D0103
...
...
40B1A200    MOV   $0, %AL           Azzera  il contatore AL
40B1A203    MOV   (3F0D0100), %DX   Copia in DX il dato da elaborare
40B1A207    CMP   %DX, $0           Confronta il contenuto di DX con 0
40B1A211    JE    40B1A232          Salta se uguale
40B1A215    SHL   %DX               Trasla a sinistra il contenuto di DX
40B1A217    JC    40B1A225          Salta se CF è settato all'indirizzo 225
40B1A221    JMP   40B1A207          Salta incondizionatamente a 207
40B1A225    ADD   $1, %AL           Incrementa il contatore AL
40B1A228    JMP   40B1A207          Salta incondizionatamente
40B1A232    MOV   %AL, (3F0D0102)   Memorizza il risultato
40B1A236    HLT                     ALT
…
```

Istruzioni operative (1)

Istruzioni per il trasferimento di dati

MOV	Movimento (ricopiamento)
IN	Ingresso dati
OUT	Uscita dati
PUSH	Immissione di una word nella pila
POP	Estrazione di una word dalla pila

Istruzioni aritmetiche

ADD	Somma (fra interi oppure naturali)
SUB	Sottrazione (fra interi oppure naturali)
CMP	Confronto (fra interi oppure naturali)
MUL	Moltiplicazione (fra naturali)
DIV	Divisione (fra naturali)
IMUL	Moltiplicazione (fra interi)
IDIV	Divisione (fra interi)

Istruzioni operative (2)

Istruzioni logiche

NOT	Not logico bit a bit
AND	And logico bit a bit
OR	Or logico bit a bit

Istruzioni di traslazione/rotazione

SHL	Traslazione a sinistra
SHR	Traslazione a destra
ROL	Rotazione a sinistra
ROR	Rotazione a destra

Istruzioni di controllo

Istruzioni di salto

JMP	Salto incondizionato
Jcond	Salto sotto condizione

Istruzioni per la gestione dei sottoprogrammi

CALL	Chiamata di sottoprogramma
RET	Ritorno da sottoprogramma

Istruzione di alt

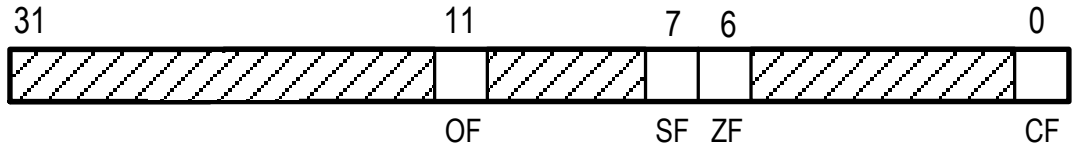
HLT	Alt

Controllo del flusso: salto condizionato e registro dei flag

Registri di stato:

Extended Instruction Pointer EIP: registro che contiene l'indirizzo della prossima istruzione da eseguire

Extended Flag register EF



Carry Flag CF, Zero Flag ZF

SignFlag SF, Overflow Flag OF

Esempio di test del contenuto dei flag:

Es1: CMP $0, %AL

JZ *indirizzo_di_memoria*

Es2: ADD %AL,%BL

JO *indirizzo_di_memoria*

NB:

Alcune istruzioni influenzano i flag, altre no

Ad esempio: la ADD influenza sia CF che OF, ma

- CF va testato nel caso si lavori sui naturali
- OF va testato nel caso si lavori sugli interi

Labels

Ogni istruzione assembler inizia ad un certo indirizzo in memoria (l'indirizzo della cella contenente il primo byte dell'istruzione)

L'indirizzo assoluto di una istruzione sarà noto solo a tempo di esecuzione, quando il programma assembler, reso eseguibile dall'assemblatore e dal linker, verrà caricato in memoria.

Il programmatore può creare delle etichette (dette *labels*), che possono essere utilizzate dal programmatore per indicare l'indirizzo di una istruzione. Sarà compito dell'assemblatore e del linker sostituire l'etichetta con l'indirizzo fisico assoluto.

Esempio di un programma che stampa 5 asterischi a video (il sottoprogramma *outchar* stampa a video il carattere la cui codifica ASCII si trova in %AL):

```
_main:    MOV   $5, %AH
          MOV   $'*', %AL
label:    CALL  outchar        # label conterrà l'indirizzo dell'istruzione
          DEC   %AH            # CALL outchar
          JNZ   label
          RET
.INCLUDE "utility"
```

Le direttive .BYTE, .FILL e .ASCII

In assembler si può allocare memoria «statica» per delle «variabili». Si tratta dell'equivalente C/C++ delle variabili globali.

Una «variabile» assembler è una etichetta (che corrisponde all'indirizzo della prima cella di memoria in cui inizia la «variabile») seguita da una direttiva assembler che riserva area di memoria.

```
v1:    .BYTE 0x6F,'k' # alloca un vettore di 2 byte ad "ok"
v2:    .FILL   12, 1  # alloca un vettore di 12 byte (1=byte)
v3:    .FILL   25, 2  # alloca un vettore di 25 word (2=word = 2byte)
v4:    .FILL   10, 4  # alloca un vettore di 10 long (4=long = 4byte)
str:   .ASCII "Ciao Mondo"

_main:
       MOV    $str, %EBX  # copia l'indirizzo di str in EBX
       MOV    $10,  %CX
       CALL   outmess
       RET
```

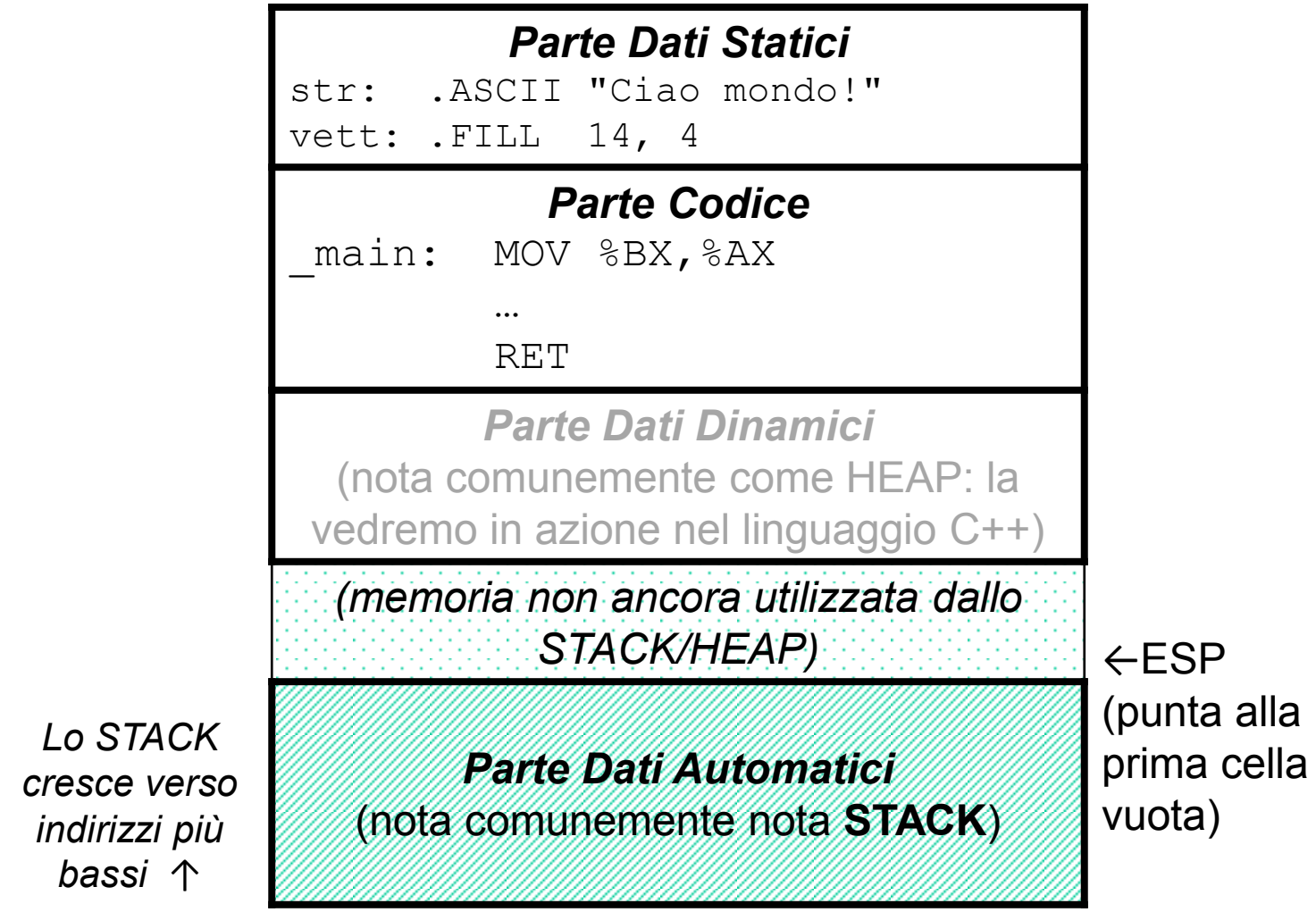
NB: Le direttive .BYTE e .ASCII calcolano il numero di celle di memoria da allocare ed inoltre provvedono anche alla loro inizializzazione. La .FILL non inizializza.

Lo Stack (1/7)

- Lo STACK è un'area di memoria a disposizione dei programmi (sia Assembler che C/C++)
- Viene gestita mediante strategia LIFO (last in-first out): un nuovo dato può essere immesso oppure prelevato dalla cima allo STACK
- Sfrutta una registro speciale del processore, l'Extended Stack Pointer (ESP)
- L'ESP punta sempre alla prima locazione libera dello STACK

Lo Stack (2/7)

Struttura di un programma eseguibile in memoria



Le istruzioni PUSH e POP

- L'istruzione PUSH salva il contenuto di un registro a 32 bit nelle 4 celle in cima allo STACK e decrementa ESP di 4

  Esempio:

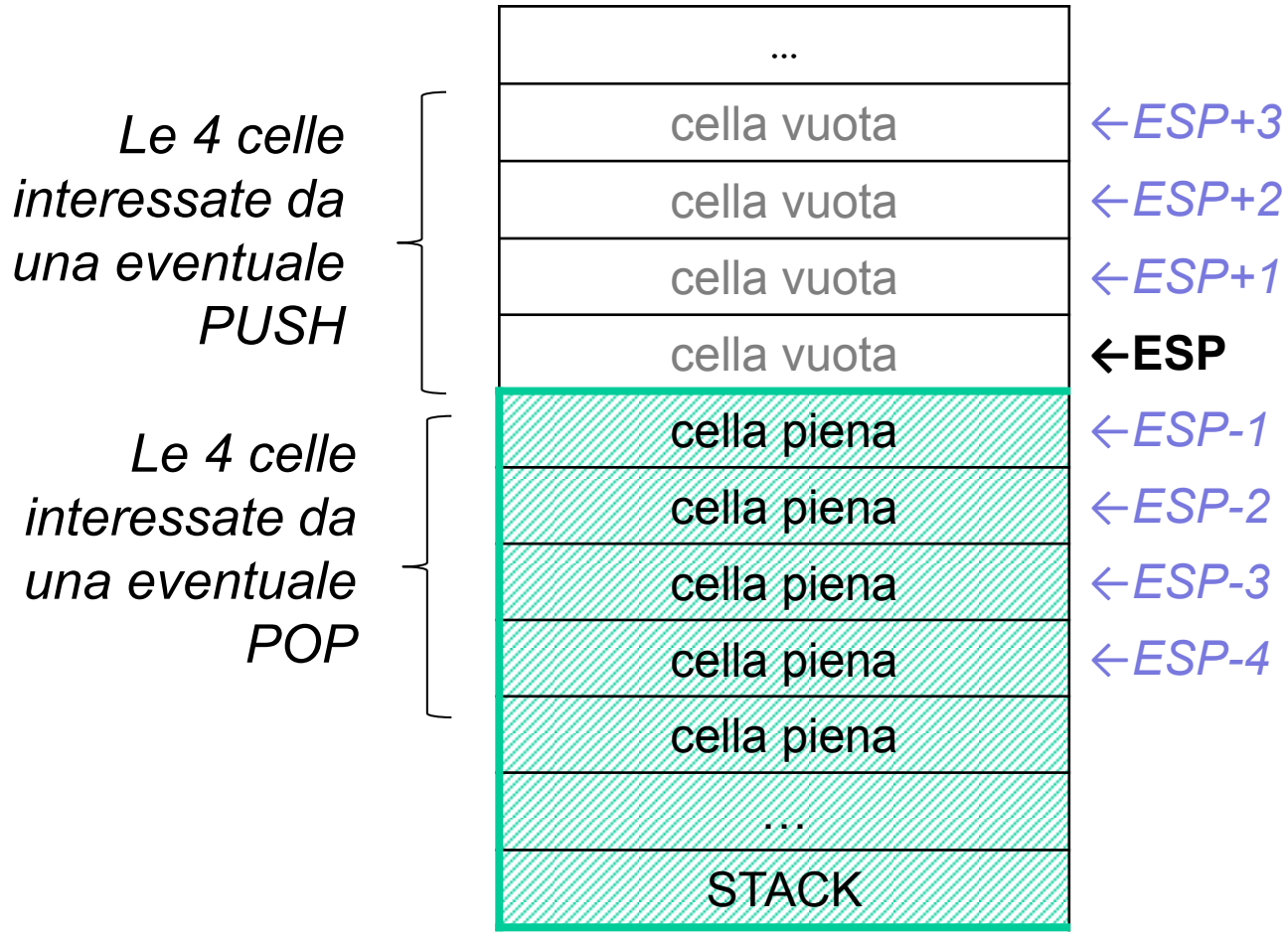
  ```
  PUSH %EAX
  ```

- L'istruzione POP recupera il contenuto delle 4 locazioni in cima allo STACK, lo salva in un registro a 32 bit e incrementa ESP di 4

  Esempio

  ```
  POP %EBX
  ```

Funzionamento delle istruzioni PUSH e POP



L'istruzione CALL

- L'istruzione CALL permette di andare ad eseguire un sottoprogramma
- Esempio di chiamata del sottoprogramma `inchar`:

```
04AAFFB3  CALL inchar
04AAFFB8  INC  %EAX
          ...
```

← *quando si arriva ad eseguire la CALL in EIP ci sarà indirizzo dell'istruzione INC (ad esempio, 04AAFFB8)*

- Concettualmente la CALL equivale alle seguenti istruzioni:

```
PUSH %EIP      # salva 04AAFFB8 sullo STACK
JMP 1E22AF0C   # se inchar inizia a 1E22AF0C
```

NB: la `PUSH %EIP` non si può fare esplicitamente in quanto non fa parte del set delle istruzioni del processore!

L'istruzione RET

- L'istruzione RET permette di tornare ad eseguire il sottoprogramma chiamante a partire dall'istruzione successiva alla CALL

Esempio:

```
1E22AF0C  inchar: IN (0AFB), %AL
                  ...
                  RET
```

- Concettualmente, effettua le seguenti operazioni:

```
POP %EIP
JMP %EIP
```

Lo Stack (7/7)

Sintetizzando, le istruzioni che utilizzano lo STACK sono:
CALL, PUSH, POP, e RET

NB:

- Tutte modificano l'ESP
- Alcune modificano anche l'EIP (CALL e RET)

Sottoprogrammi (1/2)

Qualunque blocco di istruzioni assembler consecutive che inizia con una label e termina con una RET.

Esempio di un sottoprogramma che stampa a video un asterisco:

```
subprog:  PUSHAD          # salva il contenuto di tutti i registri gen.
          MOV '*', %AL
          CALL outchar
          POPAD           # ripristina il contenuto di tutti i reg. gen.
          RET
```

Le informazioni fra il sottoprogramma chiamante (ad esempio il `_main`) e quello chiamato avviene tramite registri.

```
subprog:  CALL outchar # stampa il carattere messo in %AL dal chiamante
          CALL inchar  # legge un nuovo carattere e lo restituisce in %AL
          RET
```

Il sottoprogramma principale: _main

Ogni programma Assembler deve prevedere un sottoprogramma principale, denominato `_main`.

Quando il file ascii (con estensione .s) viene compilato per generare il file oggetto file.o e poi linkato in un .exe, la prima istruzione assembler che verrà eseguita sarà quella che si trova all'indirizzo dell'etichetta _main

Sottoprogrammi (3/3)

Alcuni sottoprogrammi di utilità, contenuti nel file assembler `"utility"`

Input ed output di caratteri e stringhe

`inchar` aspetta un carattere dalla tastiera e mette il suo codice ASCII in AL

`outchar` stampa a video il carattere contenuto in %AL

`outmess` stampa a video gli N caratteri che si trovano in memoria a partire dall'indirizzo contenuto in EBX. N è il naturale contenuto in CX.

Input ed output di naturali e interi su 8 bit

`inbyte` aspetta che vengano inseriti due caratteri (entrambi fra 0-9 o A-F) e poi, interpretandoli come cifre esadecimali, assegna gli 8 bit di %AL. Esempio 'E3' => in AL ci finisce 11100011

`outbyte` stampa a video i due caratteri ASCII corrispondenti alle due cifre esadecimali associate al corrente contenuto di %AL. Esempio: se il contenuto di AL è 01101010 a video viene stampata la stringa `"6A"`

`newline` Porta il cursore all'inizio della linea seguente (`CALL newline` equivale al `cout<<endl`)