Lezione 12 - Standard IEEE-754 per le operazioni floating-point

- **Nelle operazioni floating point dobbiamo rappresentare**
  - **numeri con decimali, e.g., 3.1416**
  - **numeri molto piccoli, e.g., .000000001**
  - **numeri molto grandi, e.g., 3.15576×10^9**
- **Rappresentazione nel calcolatore:**
  - **Partendo dall'idea di memorizzare tali numeri usando cifre binarie, scriveremo il numero floating point x da rappresentare nella forma:**

    $x = (-1)^{segno} \times mantissa \times 2^{esponente}$

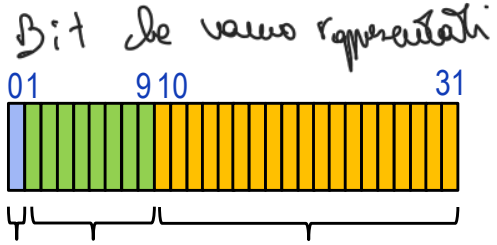
    - **dove viene scelto 1 <= 'mantissa' < 2**
    - **più cifre binarie (bit) di 'mantissa' danno più accuratezza nelle operazioni**
    - **più cifre binarie (bit) di 'esponente' permettono numeri molto grandi e molto piccoli**

IEEE-754

- **IEEE 754 floating point standard:**
  - **singola precisione ("float" del *C*) → uso 32 bit per memorizzare:**
    1 bit per S, 8 bit per E, 23 bit per M**
  - **doppia precisione ("double" del *C*) → uso 64 bit per memorizzare:**
    1 bit per S, 11 bit per E, 52 bit per M**
  - **quadrupla precisione ("__float128" del *C*) → uso 128 bit:**
    1 bit per S, 15 bit per E, 112 bit per M**

:= esize := msize

Esempio:



IEEE-754 Singola Precisione (32 bit totali)

Se x = 0.6, la sua rappresentazione floating point IEEE-754 è:
F = 00111111100110011001100110011010 (=0x3f19999a)

non è il numero, va ricostruito segno esp mantissa

**Nota: utilizzando il "trucco" della normalizzazione della mantissa nella forma 1.xxxxx
le cifre (binarie) effettive per la mantissa sono 24 (singola prec.), 53 (doppia prec.), 113 (quad. prec.)

RELAZIONE FRA NUMERO e SUA RAPPRESENTAZIONE IEEE-754 <S,E,M>

- Sia

  $x = (-1)^{segno} \times mantissa \times 2^{esponente}$

- Nel formato IEEE-754 la rappresentazione F di x è:

  $(x)_{IEEE-754} = F = <S,E,M>$

  dove

  S = segno

  E = esponente + polarizzazione

  M = int[(mantissa-1) × 2^{msize}]

RELAZIONE FRA NUMERO x
E SUA RAPPRESENTAZIONE IEEE-754

(S,E,M sono numeri interi rappresentati su un certo numero di bit ovvero
S = 0 o 1, $M < 2^{msize}$, $E < 2^{esize}-1$, polarizzazione=$2^{esize-1}-1$)

Per es. in «IEEE-754 singola precisione» si ha: polarizzazione=127, msize=23, esize=8 quindi:

$x=0.375 = (-1)^0 \times \boxed{1.5 \times 2^{-2}}$

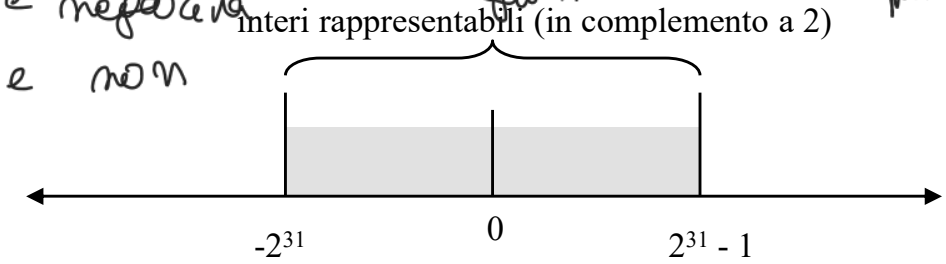
→ segno=0, esponente=-2, mantissa=1.5 → S=0, E=-2+127=125, M=(1.5-1)* $2^{23} = 2^{22}$

→ → $(x)_{IEEE-754} = F = <0,125,2^{22}>_{base10} = <0,0111\ 1101,1000\ 0000\ 0000\ 0000\ 0000\ 000>_{base2}$

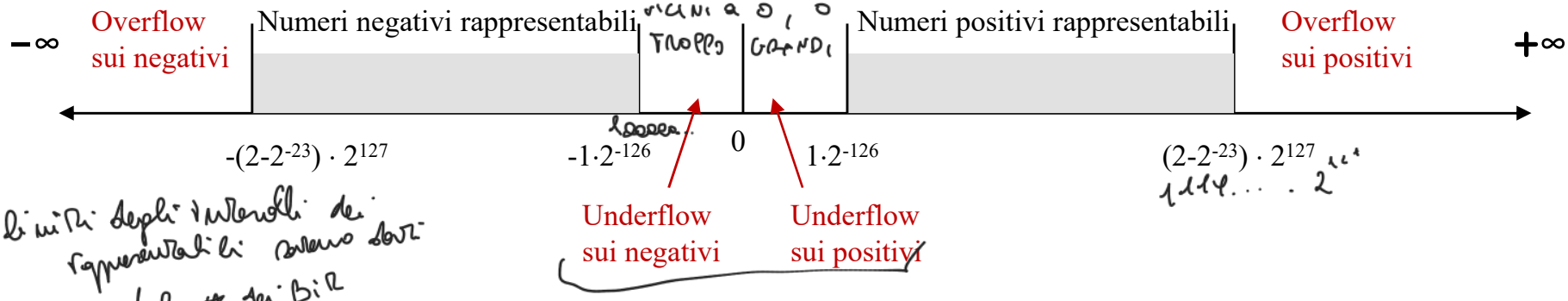
Nota: l'operazione int[.] tronca il numero di bit → quindi la rappresentazione IEEE-754 **spesso approssima** il numero x

Come utilizzare i bit

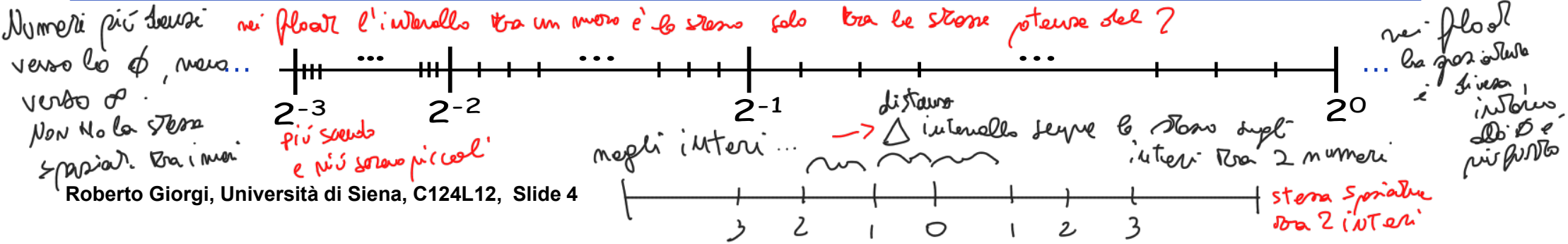
- **Nel caso di interi a 32 bit:**



- **Nel caso di floating-point:**

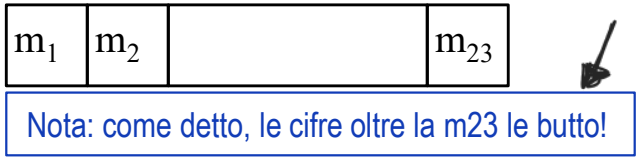


Nota: il numero di valori rappresentati con 32 bit è circa lo stesso nel caso INT e in quello FP... la densità dei punti non è costante, quindi.



IEEE 754 floating-point mantissa ed esponente

- **Mantissa**
  - Rappresentazione in base due con Normalizzazione (e modifica dell'esponente) in modo tale da riportare un 1 nella cifra più significativa
    - (tale unico) 1 prima della virgola non compare nella codifica
  - I bit dalla mantissa rappresentano un numero decimale (in base 2) >=1 e <2
    - M=1-mantissa= $m_1 \times 2^{-1} + m_2 \times 2^{-2} + \ldots + m_{23} \times 2^{-23}$



Nota: come detto, le cifre oltre la m23 le butto!

- **Esponente**
  - Rappresentazione "polarizzata" (biased) per rendere più facile il sort
    - polarizzazione=127 per la singola precisione, 1023 per la doppia precisione
    - i valori vanno da 1 a 2^e-2 (e=#bit per l'esponente) e codificano gli esponenti da $-2^{e-1}+2$ a $+2^{e-1}-1$ (e.g. -126..+127)
  - Codifiche riservate
    - esponente con tutti 1 e mantissa != 0
      codifica "Not a Number" (NAN) (risultato di 0/0)
    - esponente con tutti 1 e mantissa == 0
      codifica "Infinity" (INF) (risultato di un numero diverso da zero diviso 0)
    - esponente con tutti 0 e mantissa != 0
      codifica "underflow" (numero denormalizzato)

- **Ricapitolando: per $(-1)^S \times (1+M) \times 2^{E-polarizzazione}$ → memorizzo <S,M,E>**
  - S=0 o 1, M=mantissa_normalizzata senza "1." iniziale, E=esponente+polarizzazione

Esercizio

- Rappresentare -0.75 in formato IEEE single precision
  - rappresentazione decimale: $-0.75 = -3/4 = -3 \times 2^{-2} = -1.5 \cdot 2^{-1}$
  - rappresentazione binaria: $-11 \times 2^{-2} = -1.1 \times 2^{-1}$ (-3 → -1) moltiplico per 2
  - rappresentazione binaria normalizzata: -1.1×2^{-1}

  - segno=negativo
    → S=1
  - mantissa_normalizzata=1.1000 0000 0000 0000 0000 000
    → M = 1000 0000 0000 0000 0000 000
  - esponente=-1
    → E=esponente+polarizzazione = -1 + 127 =126 → 01111110

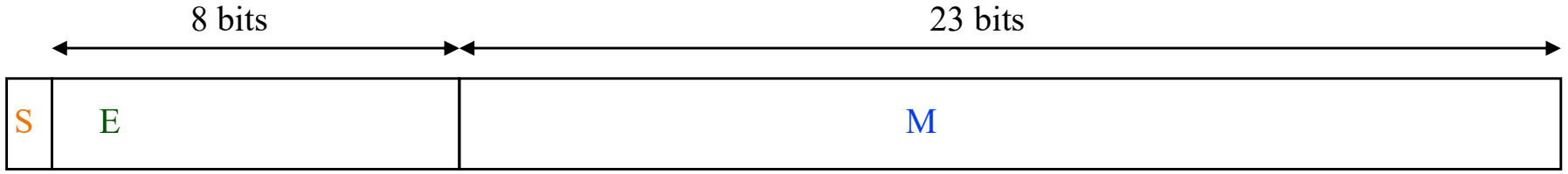
$M = 0,5 \cdot 2^{23} = 2^{22}$

$Exp = -1 + 127 = 126$

  - IEEE single precision: 10111111010000000000000000000000

  - Nota: la mantissa è stata normalizzata. Per questo tutti i bit più significativi risultano spostati verso SINISTRA.

Floating-Point: single precision (32 bits)



Polarizz. è stata implementata perché risultano più veloci gli algoritmi di sort

Esempi con mantissa ed esponente positivi/negativi:

+ 127

$+1.1010001 \cdot 2^{+10100}$ → 0 10010011 10100010000000000000000

$-1.1010001 \cdot 2^{+10100}$ → 1 10010011 10100010000000000000000

$+1.1010001 \cdot 2^{-10100}$ → 0 01101011 10100010000000000000000

$-1.1010001 \cdot 2^{-10100}$ → 1 01101011 10100010000000000000000

Esempi di numeri "notevoli":

0 → 0 00000000 00000000000000000000000

0+127 = POLARIZZ

MANTISSA

1 → 0 01111111 00000000000000000000000 ← 1,0000 ... 0

esponente = polariz

minore rappresentabile (2^{-126}) → 0 00000001 00000000000000000000000

1 · 2^-126

numero più piccolo rappres.

maggiore rappres. $(2-2^{-23}) \cdot 2^{127}$ → 0 11111110 11111111111111111111111

−127 e 128, codifiche riservate

∞ → 0 11111111 00000000000000000000000

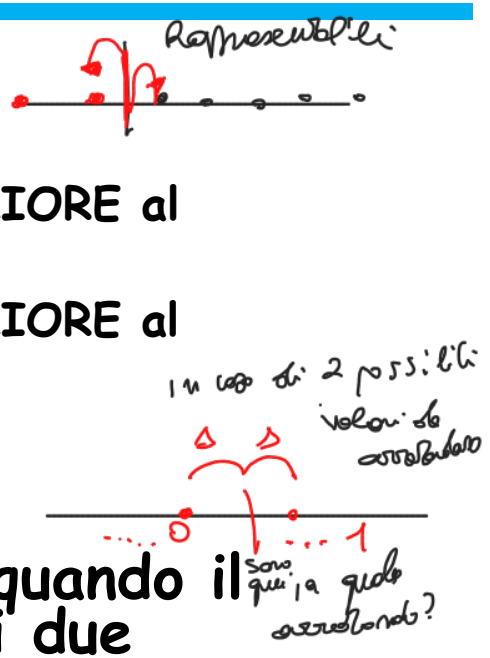
NaN → 0 11111111 XXXXXXXXXXXXXXXXXXXXXXX

≠0

Nota: $2^{-126} \approx 1.18 \cdot 10^{-38}$, $(2-2^{-23}) \approx 3.4 \cdot 10^{38}$

Modalità di arrotondamento IEEE-754

- IEEE-754 prevede 4 modalità di arrotondamento
  - round to nearest: arrotonda al più vicino rappresentabile
  - round to+∞: arrotonda al primo numero rappresentabile NON INFERIORE al valore da arrotondare (eventualmente arrotonda a +inf)
  - round to-∞: arrotonda al primo numero rappresentabile NON SUPERIORE al valore da arrotondare (eventualmente arrotonda a -inf)
  - round to 0: arrotonda al più vicino rappresentabile con modulo NON SUPERIORE al valore da arrotondare

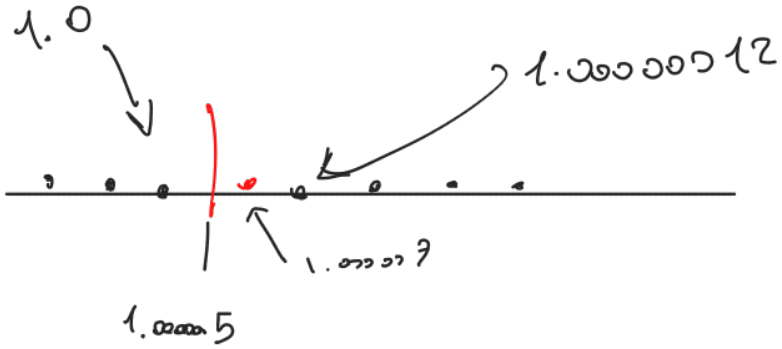


- Nel caso di "round to nearest" ci può essere ambiguità quando il numero da arrotondare sia esattamente equidistante dai due numeri più vicini rappresentabili
  - Lo standard adotta la regola "round ties to even", ovvero in caso di pareggio va scelto il più vicino rappresentabile PARI
  - Il vantaggio rispetto alla modalità "round ties to away (from 0)" (supportata nella revisione dello standard IEEE-754-2008) è che in media l'errore di arrotondamento si annulla**
  - L'implementazione di "round ties to even" è più complessa

**Goldberg, D. 1991. What every computer scientist should know about floating-point arithmetic. *ACM Comp. Surveys* 23, 1 (Mar. 1991), 5-48. DOI= http://doi.acm.org/10.1145/103162.103163

Esempi di arrotondamento

numero	r.to 0	r.to nearest	r.to +inf	r.to -inf
1.00000003	1.0	1.0	1.00000012	1.0
1.00000007	1.0	1.00000012	1.00000012	1.0
-1.00000003	-1.0	-1.0	-1.0	-1.00000012
-1.00000007	-1.0	-1.00000012	-1.0	-1.00000012



Ricapitolazione

- L'aritmetica del calcolatore è limitata da precisione finita
- I pattern di bit non hanno particolare significato ma esistono standard per: interpretare i bit
  - complemento a due
  - IEEE 754 floating point
- Le istruzioni determinano il "significato" dei pattern di bit

a seconda delle istruzioni interpreto i bit

ho riepilogo delle istruzioni per i float

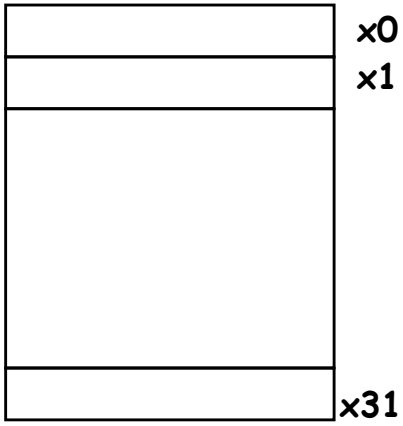
- Le prestazioni e l'accuratezza sono importanti: ci sono diverse complessità nelle macchine reali
- Lo standard è stato rivisto nel Giugno 2008 come IEEE 754-2008: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=4610935&isnumber=4610934
  - Fra le novità il supporto per floating point a 128-bit
  - Aritmetica floating point decimale (oltre che binaria)

OPERAZIONI FLOATING-POINT NEL PROCESSORE RISC-V

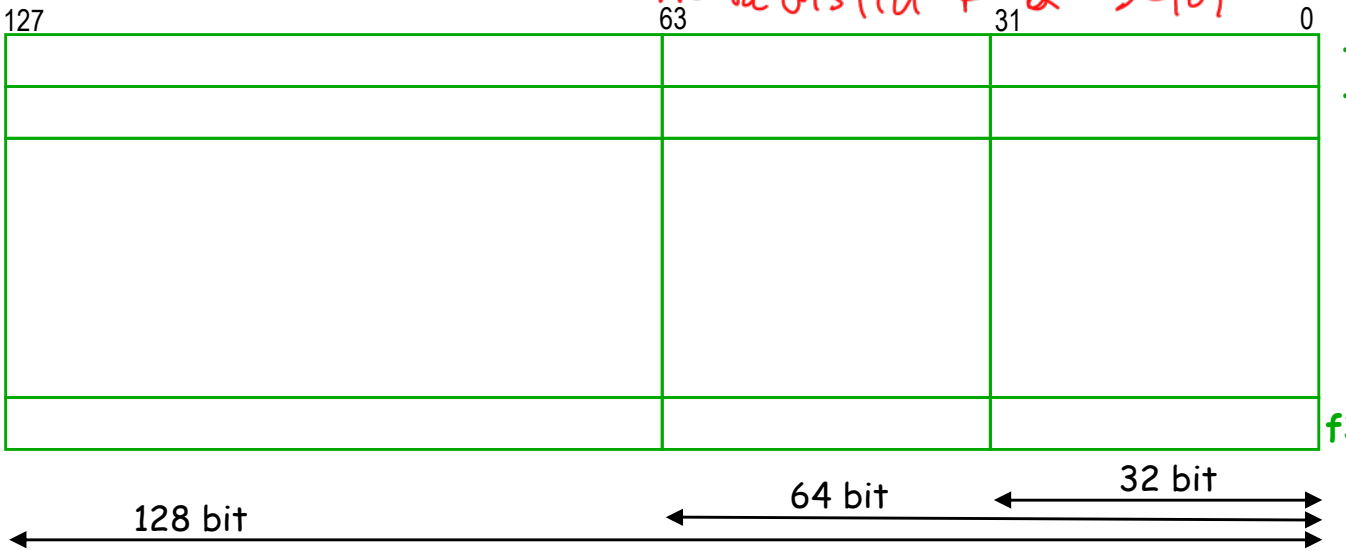
meno 64 bit, rappresentato a 32, troncato,
rimane sempre rappresentabile, e meno
di salvare la info del segno

Supporto architetturale per le op. floating-point nel RISC-V

REGISTRI PER
OPERAZIONI SU INTERI:



REGISTRI **AGGIUNTIVI**
PER OPERAZIONI SU FLOATING-POINT (s=32 bit, d=64bit, q=128bit):



- **fadd.s f0, f1, f2** — **lavora sui registri fp a 32 bit (estensione 'F' del RISC-V)**
- **fadd.d f0, f2, f4** — **lavora su registri fp a 64 bit (estensione 'D ' del RISC-V)**
- **fadd.q f0, f2, f4** — **lavora su registri fp a 128 bit (estensione 'Q' del RISC-V)**

→ **Sta al programmatore di utilizzare correttamente i contenuti dei registri**

- **Ad es., se uso fadd.d f0,f1,f2 e subito dopo pretendo di usare fadd.s f4,f3,f0 sicuramente commetto un errore dato che i bit meno significativi di f0 codificano il valore a singola precisione in maniera diversa di quello a singola precisione→attenzione!**
- **Per usare correttamente operandi di precisione diversa ci sono istruzioni di conversione, es. fra double e single**

Istruzioni floating point nel processore RISC-V

- **ARITMETICHE** (dove 'z' può essere s per single, d per double, q per quad)

```
fadd.z  f1, f2, f3   z = s, d, q   # somma
fsub.z               # sottrazione
fmul.z               # moltiplicazione
fdiv.z               # divisione
fsqrt.z  f1,f2       # radice quadrata
fmv.z    f1,f2       # spostamento fra reg. fp   (qui non è una pseudo inst)
fabs.z   f1,f2       # valore assoluto
fneg.z   f1,f2       # valore contrario (negato)   non è banale
```

In blu-chiaro: pseudoistruzioni basate su FSIGNJ

- **MEMORIA**

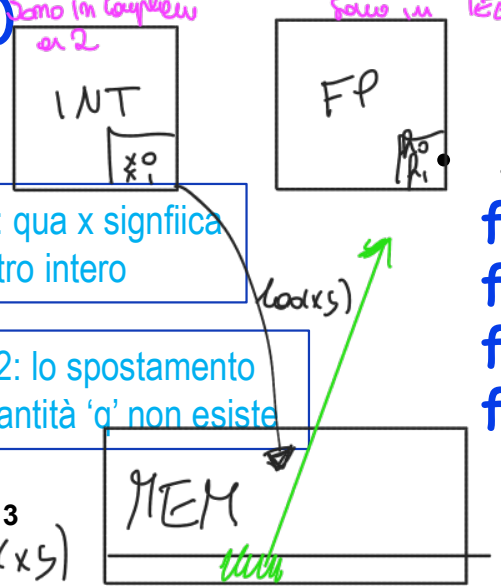
```
flw / fld / flq   f1, 100(x5)   # flw:32bit, fld:64bit, flq:128bit
fsw / fsd / fsq   f1, 100(x5)
```

- **SPOSTAMENTO int←→fp**

```
fmv.x.w   x5, f1
fmv.w.x   f1, x5
fmv.x.d   x5, f1
fmv.d.x   f1, x5
```

Nota: qua x signfiica registro intero

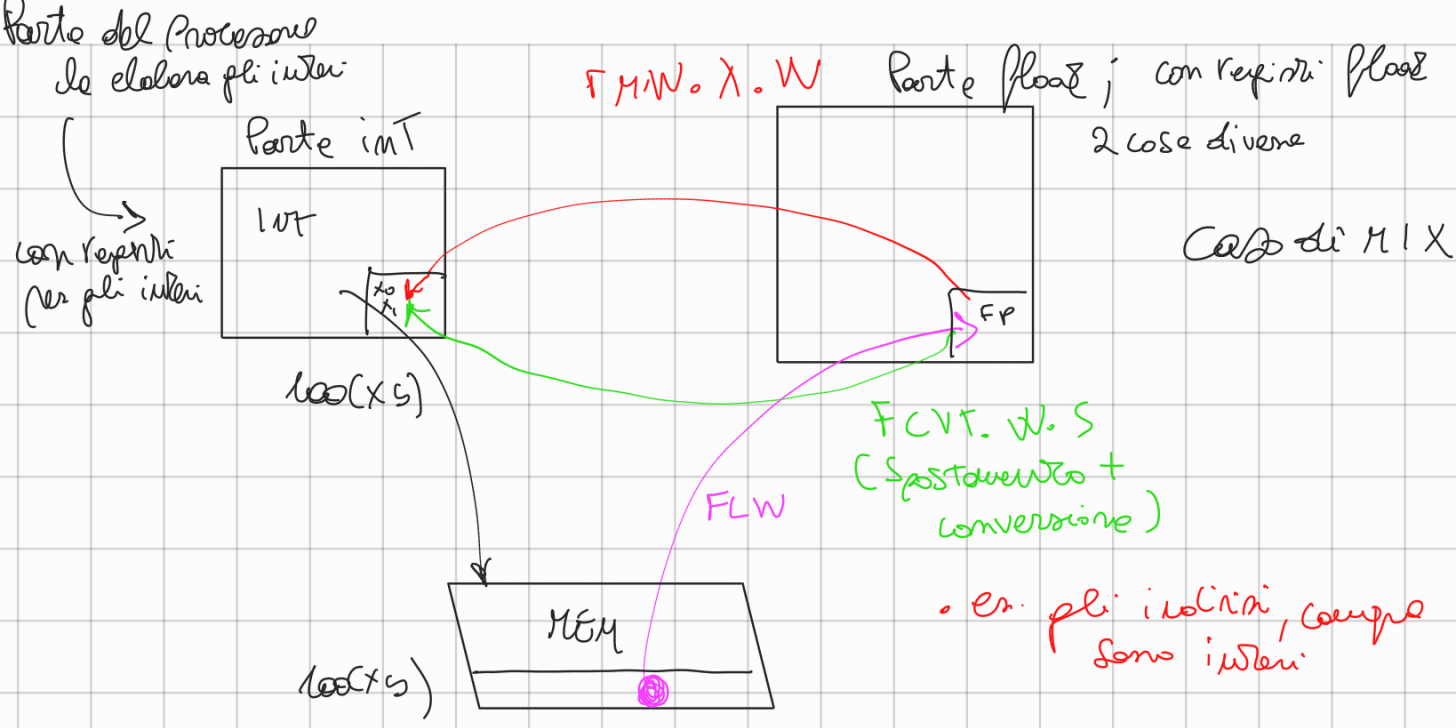
Nota2: lo spostamento di quantità 'q' non esiste



SPOST. CON CONVERSIONE

```
fcvt.s.w  / fcvt.s.wu   f1, x5
fcvt.w.s  / fcvt.wu.s   x5, f1
fcvt.d.w  / fcvt.d.wu   f1, x5
fcvt.w.d  / fcvt.wu.d   x5, f1
```

W = intero con segno (word)
WU=intero senza segno (word-unsigned)



FMV.X.W

- es. gli indirizzi, comunque sono interi

- Nel caso dell'accesso alla memoria uso sia float che interi, registri X per gli indirizzi della memoria ma carico un valore che è un float

Dalla memoria posso prendere con load e store elementi a 32, 64, 128 bit

Istruzioni condizionali floating-point

- Sono nella forma

```
fcc.s  x5, f2, f4
```

dove 'cc' indica una condizione di confronto e può essere:

'lt' → less than

'le' → less than or equal

'eq' → equal

- Perché non ci sono i casi (gt, ne, ge)?
  Nel RISC-V si è scelto di non inserire istruzioni per tali casi in quanto basta… scambiare gli operandi* per trattarli!
  - Il risultato va su un registro intero
  - Esempio:

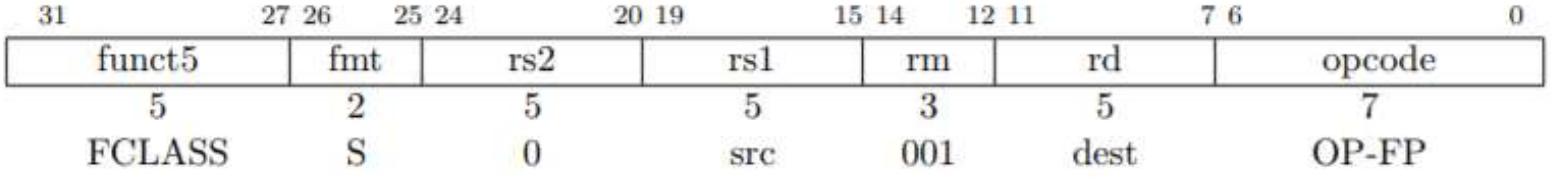
```
flt.s  x5, f2, f4
beq    x5, x0, L1
```

* oppure, es., usare bne invece beq nell'uso del valore booleano della condizione

Istruzione FCLASS.x non la vedremo

- **FCLASS.z permette di sapere se il numero FP è normalizzato o meno, +/-INF, +/-0, NaN**
  - **dal manuale "RISC-V User-Level ISA V2.2"**

The FCLASS.S instruction examines the value in floating-point register *rs1* and writes to integer register *rd* a 10-bit mask that indicates the class of the floating-point number. The format of the mask is described in Table 8.5. The corresponding bit in *rd* will be set if the property is true and clear otherwise. All other bits in *rd* are cleared. Note that exactly one bit in *rd* will be set.



rd bit	Meaning
0	*rs1* is $-\infty$.
1	*rs1* is a negative normal number.
2	*rs1* is a negative subnormal number.
3	*rs1* is -0.
4	*rs1* is $+0$.
5	*rs1* is a positive subnormal number.
6	*rs1* is a positive normal number.
7	*rs1* is $+\infty$.
8	*rs1* is a signaling NaN.
9	*rs1* is a quiet NaN.

RISC-V Floating-Point ABI (Application Binary Interface)

- Definisce la convenzione di uso dei registri nel software
- Valido nel caso single e in quello double precision

  - f10,f11 (chiamati fa0,fa1) → argomenti delle funzioni o valori restituiti
  - f12-f17 (chiamati fa2,fa7) → argomenti ulteriori di funzioni
  - f8,f9,f18-f27 (chiamati fs0-fs11) → registri "saved"
  - f0-f7,f28-f31 (chiamati ft0-ft11) → registri temporanei

Fø generico ($X_0 \equiv \emptyset$)

per i registri FP. non ho registri speciali

devono salvarli se li uso

f0 è un registro generico R1 ecc. uguale

non c'è dello che FØ è Ø, non ho registri speciali, non mi serve

X_0 è Ø

Alcune syscall che coinvolgono i numeri floating point

- **'ecall' serve per chiedere un servizio al sistema operativo (chiamata anche 'system call')**
  - **PARAMETRI DI INGRESSO/USCITA**
    - **a7 deve contenere il numero del servizio**
      - 2: stampa un single-precision float a video (fa0 contiene il float da stampare)
      - 6: leggi un single-precision float dalla tastiera (fa0 in uscita, contiene il float letto in binario)
      - 3: stampa un double-precision float a video (fa0 contiene il double da stampare)*
      - 7: leggi un double-precision float dalla tastiera (fa0 contiene il float letto in binario)*

*Nota: non disponibile in RARS 1.3