

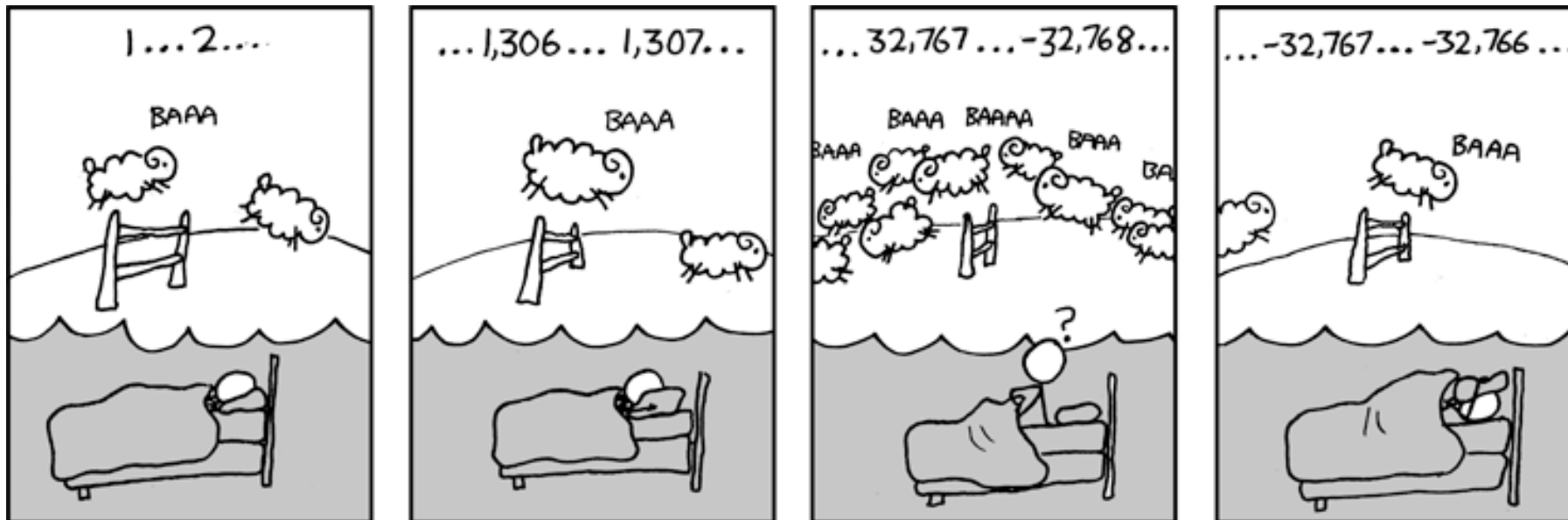
Calculatoare Numerice

– Cursul 4 –

Reprezentarea numerelor

Facultatea de Automatică și Calculatoare
Universitatea Politehnica București

Comic of the day



If androids someday DO dream of electric sheep, don't forget to declare `sheepCount` as a long int.

<http://xkcd.com/571>

Numere

- Este $x^2 \geq 0$?
 - floats: Da!
 - ints:
 - $40000 * 40000 \rightarrow 1600000000$
 - $50000 * 50000 \rightarrow ??$
- Este $(x + y) + z = x + (y + z)$?
 - unsigned & signed ints: Da!
 - floats:
 - $(1e20 + -1e20) + 3.14 \rightarrow 3.14$
 - $1e20 + (-1e20 + 3.14) \rightarrow ??$

ints are not integers,
floats are not reals

Exemplu Code Security

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

- Similar cu implementarea **getpeername** dintr-o versiune de FreeBSD
- Sunt armate întregi de oameni care încearcă să găsească vulnerabilități în cod

Utilizare tipică

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

```
void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, MSIZE);
    printf("%s\n", mybuf);
}
```

Utilizare malițioasă

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

```
void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, -MSIZE);
    . . .
}
```

Operațiile matematice

- Nu generează valori aleatoare
 - Operațiile aritmetice au proprietăți matematice importante
- Nu putem să acoperim toate proprietățile matematice " uzuale"
 - Reprezentarea numerelor este finită
 - Operațiile cu întregi satisfac proprietățile unui inel
 - Comutativitate, asociativitate, distributivitate
 - Operațiile în virgulă mobilă satisfac proprietățile de ordonare
 - Monotonie
- Observație
 - Trebuie să înțelegem care abstractizări funcționează în anumite contexte
 - Problemă importantă pentru cei care scriu compilatoare și pentru dezvoltatorii de aplicații

Hard Truth: Trebuie să știți assembly

- Sunt foarte multe șanse că nu veți scrie niciodată un program în assembly
 - Compilatoarele sunt mai bune și au mai multă răbdare decât voi
- DAR, cunoașterea assembly este instrumentală pentru înțelegerea modului în care funcționează mașina de calcul
 - Comportamentul programelor în prezența erorilor
 - Dificil de înțeles (de multe ori imposibil) într-un limbaj de nivel înalt
 - Îmbunătățirea performanței programelor
 - Ce optimizări sunt făcute/nu pot fi făcute de compilator
 - Care sunt sursele ineficienței unui program
 - Implementarea software de sistem
 - Compilatorul generează cod mașină
 - Sistemele de operare trebuie să gestioneze starea proceselor
 - Crearea / lupta împotriva malware
 - x86 assembly este limbajul de bază!

Exemplu Assembly

- Time Stamp Counter
 - Registru special 64-biți pe mașinile compatibile Intel
 - Incrementat la fiecare ciclu de ceas
 - Citit cu instrucțiunea *rdtsc*
- Aplicație
 - Măsoară timpul (în cicli de ceas) necesar pentru execuția unei bucăți de cod/proceduri/funcții

```
double t;  
start_counter();  
P();  
t = get_counter();  
printf("P required %f clock cycles\n", t);
```

Codul pentru citirea contorului

- Scriem un mic program folosind directiva *asm* a GCC
- Inserează cod assembly în codul mașină generat de compilator

```
static unsigned cyc_hi = 0;
static unsigned cyc_lo = 0;

/* Set *hi and *lo to the high and low order bits
   of the cycle counter.
*/
void access_counter(unsigned *hi, unsigned *lo){
    asm("rdtsc; movl %%edx,%0; movl %%eax,%1"
        : "=r" (*hi), "=r" (*lo)
        : "%edx", "%eax");
}
```

Hard Truth #2: Memory Matters

- Memoria nu este nelimitată
 - Trebuie să fie alocată și administrată
 - Majoritatea aplicațiilor lucrează extensiv cu memoria
- Erorile de referențiere la memorie sunt foarte dese
 - Efectele uneori nu se fac simțite imediat în spațiu și timp
- Performanța memoriei nu este uniformă
 - Folosirea incorectă a memoriei cache și a celei virtuale pot afecta semnificativ performanțele unui program
 - Adaptarea unui program la caracteristicile sistemului de memorie poate duce la îmbunătățiri semnificative de viteză

Exemplu eroare

```
double fun(int i)
{
    volatile double d[1] = {3.14};
    volatile long int a[2];
    a[i] = 1073741824; /* Possibly out of bounds */
    return d[0];
}
```

fun(0) → 3.14
fun(1) → 3.14
fun(2) → 3.13999998664856
fun(3) → 2.00000061035156
fun(4) → 3.14, then segmentation fault

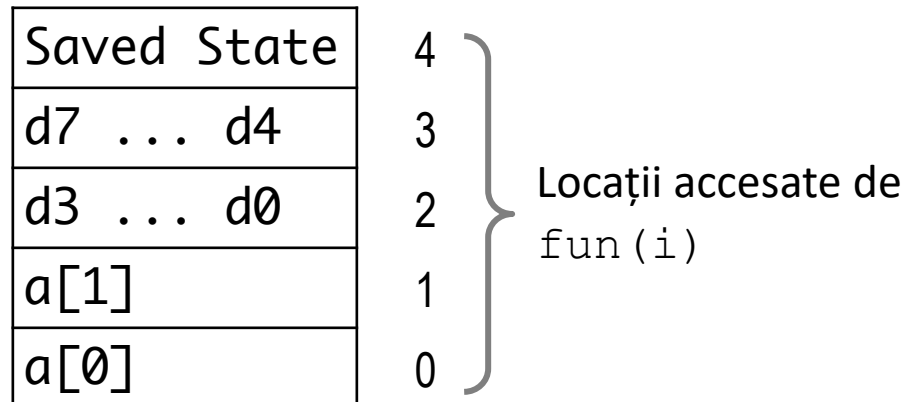
- Rezultatele sunt specifice arhitecturii de calcul!

Hard Truth #2: Memory Matters

```
double fun(int i)
{
    volatile double d[1] = {3.14};
    volatile long int a[2];
    a[i] = 1073741824; /* Possibly out of bounds */
    return d[0];
}
```

```
fun(0) → 3.14
fun(1) → 3.14
fun(2) → 3.1399998664856
fun(3) → 2.00000061035156
fun(4) → 3.14, apoi segfault
```

Explicație:



Erori de referențiere la memorie

- C și C++ nu au niciun mecanism de protecție a memoriei
 - Referințe la vectori out-of-bounds
 - Valori invalide pentru pointeri
 - Abuzarea malloc/free
- Poate să ducă la erori urâte
 - Dacă bug-ul are sau nu vreun efect depinde de sistem și de compilator
 - Acționează la distanță
 - E posibil ca obiectul corupt să nu fie deloc legat de cel accesat
 - Efectul bug-ului poate fi observat la mult timp după ce a fost generat
- Cum pot să scap de toate astea?
 - Programează în Java, Ruby, ML etc.
 - Înțelege care sunt interacțiunile ce pot să apară
 - Folosește sau dezvoltă unelte care să detecteze erorile de referențiere (de ex. Valgrind)

Codificarea binară

- Byte = 8 bits

- Binar 00000000_2 to 11111111_2

- Zecimal: 0_{10} to 255_{10}

- Prima cifră **diferită de 0** în C

- Hexazecimal 00_{16} to FF_{16}

- Reprezentare în baza 16

- Folosim caracterele '0' ..'9' și 'A' ..'F'

- Scriem $FA1D37B_{16}$ în C: **0xFA1D37B**

- Sau **0xfa1d37b**

Hex	Dec	Binar
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Machine words

- Calculatoarele au “word size”
- Dimensiunea nominală a datelor pe întregi
 - Inclusiv adresele
- Multe calculatoare folosesc cuvinte de 32-biți (4 byte)
 - Limitează adresele la 4GB
 - Devine prea mic pentru aplicațiile memory-intensive
- Mașinile moderne folosesc cuvinte de 64-biți (8 byte)
 - Spațiu potențial de adrese $\sim 1.8 \times 10^{19}$ bytes
 - x86-64 au adresare pe 48-biți: 256 Terabytes
- Mașinile suportă formate multiple de date
 - Frații sau multipli de word size
 - Întotdeauna un număr întreg de biți

Reprezentarea datelor

Dimensiunea tipurilor de date C (în octeți)

C data type	Typical 32-bit	ia32	Intel x86-64
char	1	1	1
short	2	2	2
int	4	4	4
long	4	4	8
long long	8	8	8
float	4	4	4
double	8	8	8
long double	8	10/12	10/16
char *	4	4	8

Ordonarea octeților

- Cum ar trebui să fie ordonați în memorie octeții unui cuvânt de mai mulți octeți?
 - Big Endian: Sun, PPC, Internet
 - Cel mai puțin semnificativ octet are adresa cea mai mare
 - Little Endian: x86
 - Cel mai puțin semnificativ octet are adresa cea mai mică

- Origine: Călătoriile lui Gulliver (Gulliver's Travels)
- La ce capăt trebuie să spargi un ou fiert?

Ou în configurație "Little Endian" (Wikipedia)



Exemplu

- Big Endian
 - Least significant byte has highest address
- Little Endian
 - Least significant byte has lowest address
- Exemplu
 - Variabila **x** de 4 octeți are reprezentarea **0x01234567**
 - Adresa dată de **&x** este **0x100**

Big Endian

0x100 0x101 0x102 0x103



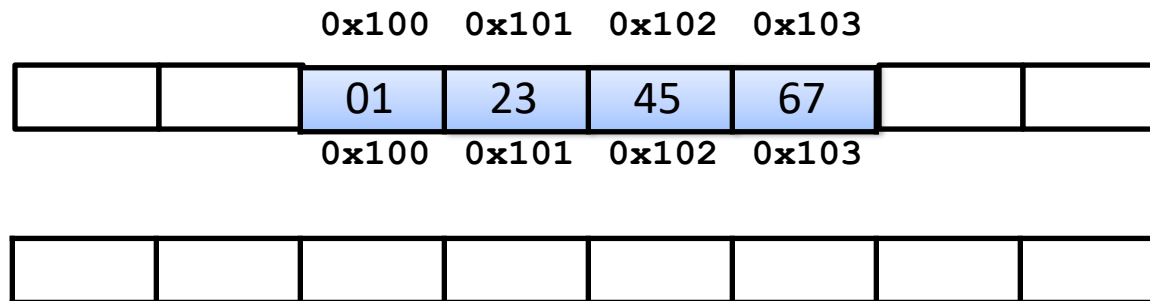
Little Endian

0x100 0x101 0x102 0x103



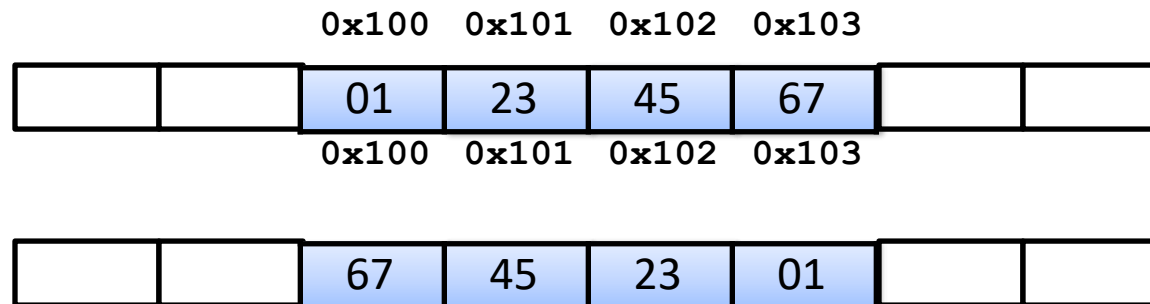
Exemplu

- Big Endian
 - Least significant byte has highest address
- Little Endian
 - Least significant byte has lowest address
- Exemplu
 - Variabila **x** de 4 octeți are reprezentarea **0x01234567**
 - Adresa dată de **&x** este **0x100**



Exemplu

- Big Endian
 - Least significant byte has highest address
- Little Endian
 - Least significant byte has lowest address
- Exemplu
 - Variabila **x** de 4 octeți are reprezentarea **0x01234567**
 - Adresa dată de **&x** este **0x100**



Citirea listing-ului

- Dezasamblare
 - Reprezentarea text a codului mașină
 - Generat de programul care citește codul mașină
- Exemplu de fragment:

Adresă	Instruction Code	Cod Assembly
8048365:	5b 00 00	pop %ebx
8048366:	81 c3 ab 12	add \$0x12ab, %ebx
804836c:	83 bb 28 00 00 00 00	cmpl \$0x0, 0x28(%ebx)

- Descifrarea numerelor:

Valoare:

0x12ab

Pad la 4 octeți:

0x000012ab

Despărțire in octeți:

00 00 12 ab

Oglindire (endian):

ab 12 00 00

Examinarea reprezentărilor

Cod care printează reprezentarea datelor

- Cast pointer la `unsigned char*` creează un vector de octeți

```
typedef unsigned char *pointer;

void show_bytes(pointer start, int len)
{
    int i;
    for (i = 0; i < len; i++) {
        printf("0x%p\t0x%.2x\n", start+i, start[i]);
    }
}
```

`printf` directives:

`%p`: print pointer

`%x`: print hexadecimal

Exemplu show_bytes()

```
int a = 15213;  
printf("int a = 15213;\n");  
show_bytes((pointer) &a, sizeof(int));
```

Rezultat folosind Linux pe Intel x86:

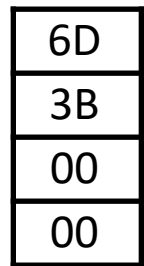
```
int a = 15213;  
0x11ffffcb8  0x6d  
0x11ffffcb9  0x3b  
0x11ffffcba  0x00  
0x11ffffcbb  0x00
```


Reprezentarea întregilor

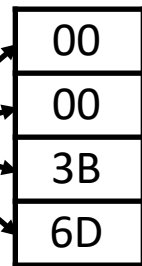
```
int A = 15213;  
int B = -15213;  
long int C = 15213;
```

Zecimal: 15213
Binar: 0011 1011 0110 1101
Hex: 3 B 6 D

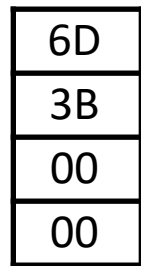
A on ia32, x86-64



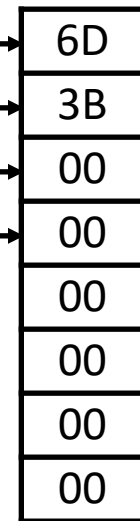
A on SPARC



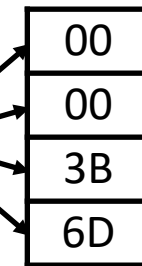
C on ia32



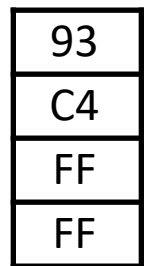
C on x86-64



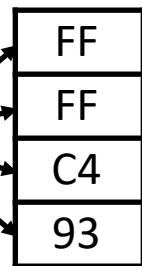
C on SPARC



B on ia32, x86-64



B on SPARC



Two's complement representation

Reprezentarea pointer-ilor

```
int B = -15213;  
int *P = &B;
```

SPARC P

EF
FF
FB
2C

IA32 P

D4
F8
FF
BF

x86-64 P

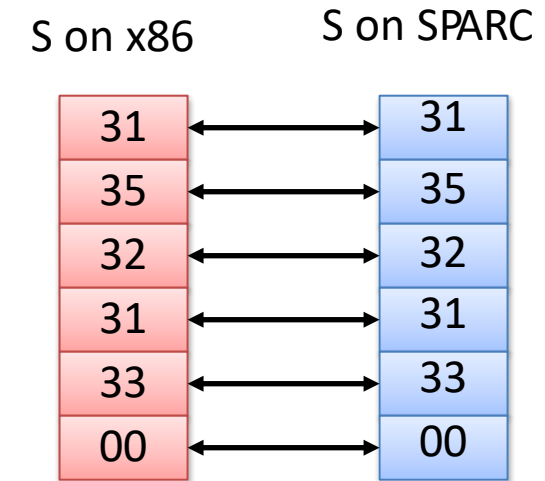
0C
89
EC
FF
FF
7F
00
00

Compilatoare și mașini diferite vor atribui locații diferite obiectelor

Reprezentarea string-urilor

- String în C
 - Reprezentate ca un vector de caractere
 - Fiecare caracter codificat în format ASCII
 - Codificare standard pe 7 biți
 - Caracterul "0" are codul 0x30
 - Cifra i are codul 0x30+i
 - Orice string trebuie să fie null-terminated
 - Caracterul final = 0
- Compatibilitate
 - Ordonarea octeților nu e o problemă

```
char S[6] = "15213";
```



Sisteme de numerație

- Numerele sunt reprezentate în calculator în format binar
 - Numere pozitive
 - Unsigned binary
 - Numere negative
 - Two's complement
 - Sign/magnitude numbers

- Cum reprezentăm **fracțiile**?

Numere fracționare

- Două notații consacrate:
 - **Fixed-point (Virgulă fixă)**: poziția virgulei este fixă
 - **Floating-point (Virgulă mobilă)**: virgula se poate afla oriunde și “plutește” către dreapta celui mai semnificativ 1

Numere în virgulă fixă

- 6.75 folosind 4 biți pentru întreg și pentru fracție:
 - Virgula nu este specificată explicit

01101100

0110.1100

$$2^2 + 2^1 + 2^{-1} + 2^{-2} = 6.75$$

- Numărul biților pt. partea întreagă și fracționară trebuie stabilit dinainte

Exemplu de calcul în v.f.

Reprezentați 7.5_{10} folosind câte 4 biți pentru partea întreagă și cea fracționară

01111000

Numere cu semn în virgulă fixă

- **Reprezentări:**

- Semn/magnitudine
- Cod complement al lui 2

- **Exemplu:** Reprezentați -7.5_{10} folosind câte 4 biți pentru partea întreagă și cea fracționară

- **Semn/magnitudine:** 11111000

- **Complement al lui 2:**

1. +7.5:	01111000
2. Invert bits:	10000111
3. Add 1 to lsb: +	1
	<u>10001000</u>

Numere în virgulă mobilă

- Virgula “plutește” către dreapta celui mai semnificativ 1
- Similar cu notația științifică zecimală
- De exemplu, scrieți 273_{10} în notație științifică:

$$273 = 2.73 \times 10^2$$

- În general, un număr este scris în notație științifică ca și:

$$\pm M \times B^E$$

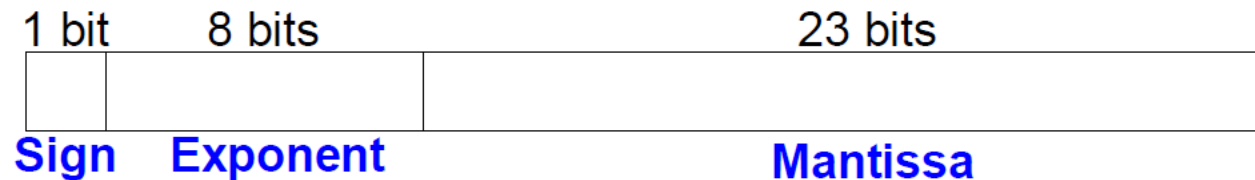
– **M** = mantisă

– **B** = bază

– **E** = exponent

– În exemplul de mai sus, $M = 2.73$, $B = 10$, and $E = 2$

Numere în virgulă mobilă



- **Exemplu:** reprezentați valoarea 228_{10} folosind notația în v.m. 32-biți

Arătăm trei versiuni – ultima este numită **IEEE 754 floating-point standard**

Reprezentarea v.m. 1

1. Convertim din zecimal în binar:

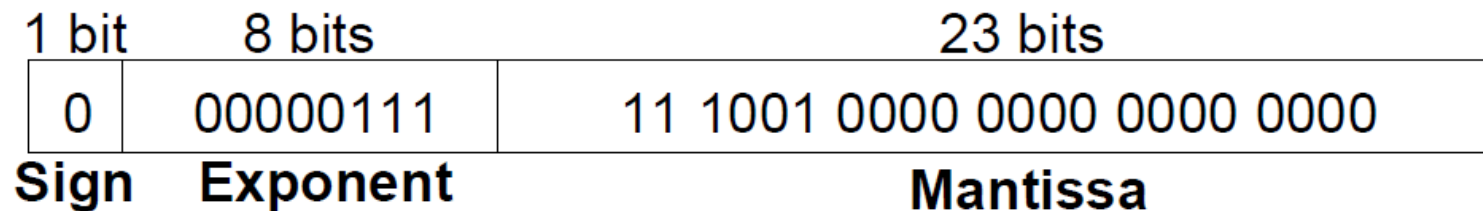
$$228_{10} = 11100100_2$$

2. Scriem numărul binar în notația științifică:

$$11100100_2 = 1.11001_2 \times 2^7$$

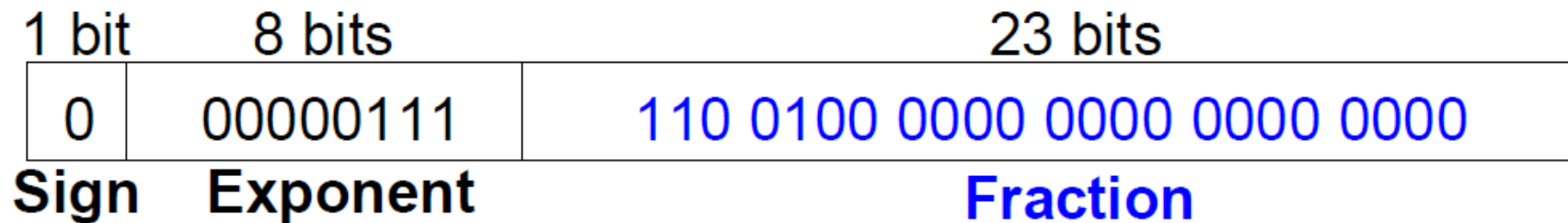
3. Completăm fiecare câmp al numărului în virgulă mobilă:

- Semnul e pozitiv (0)
- Exponentul are valoarea 7
- Ceilalți 23 de biți reprezintă mantisa



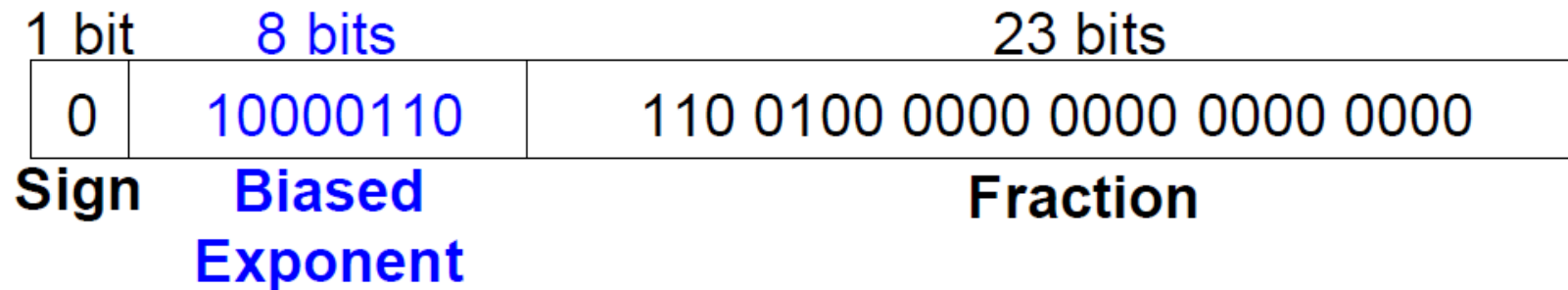
Reprezentarea v.m. 2

- Primul bit al mantisei este întotdeauna 1:
 - $228_{10} = 11100100_2 = 1.11001 \times 2^7$
- Deci, nu e nevoie să îl stocăm: *1 implicit*
- Stocăm doar biții de fracție în câmpul pt mantisă



Reprezentarea în v.m. 3

- *Exponent deplasat*: deplasament = 127 (01111111_2)
 - Exponentul deplasat = deplasament + exponent
 - Exponentul 7 este stocat ca:
 $127 + 7 = 134 = 0x10000110_2$
- **IEEE 754 32-bit floating-point representation a 228_{10}**



în hexazecimal: **0x43640000**

IEEE Floating Point

- IEEE Standard 754
 - Stabilit în 1985 ca standard uniform pentru aritmetica în virgulă mobilă
 - Înainte de asta, existau mai multe forme ne-standard
 - Suportat de majoritatea CPU-urilor

- Motivat de probleme numerice
 - Standarde bune pentru rotunjire, overflow, underflow
 - Greu de optimizat în hardware
 - La scrierea standardului au contribuit mai mulți analiști numerici și puțini designeri hardware

Exemplu în v.m.

Scrieți -58.25_{10} în floating point (IEEE 754)

1. Convertim în binar:

$$58.25_{10} = 111010.01_2$$

2. Scriem în notația științifică:

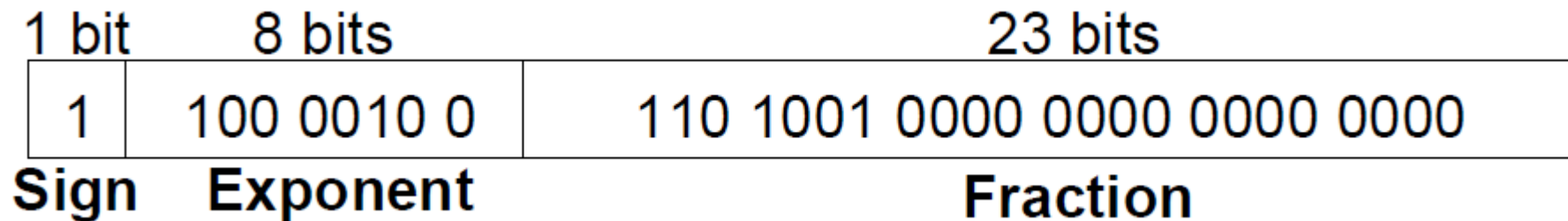
$$1.1101001 \times 2^5$$

3. Completăm câmpurile:

Bit de semn: 1 (negativ)

8 biți exponent: $(127 + 5) = 132 = 10000100_2$

23 biți fracție: 110 1001 0000 0000 0000 0000



în hexazecimal: **0xC269000**

Virgula mobilă: Cazuri speciale

Number	Sign	Exponent	Fraction
0	X	00000000	00000000000000000000000000000000
∞	0	11111111	00000000000000000000000000000000
$-\infty$	1	11111111	00000000000000000000000000000000
NaN	X	11111111	non-zero

Precizia în v.m.

- **Single-Precision:**
 - 32-bit
 - 1 bit semn, 8 biți exponent, 23 biți fracție
 - deplasament = 127
- **Double-Precision:**
 - 64-bit
 - 1 bit semn, 11 biți exponent, 52 biți fracție
 - deplasament = 1023

Floating-Point: Rounding

- **Overflow:** number too large to be represented
- **Underflow:** number too small to be represented
- **Rounding modes:**
 - Down
 - Up
 - Toward zero
 - To nearest
- **Exemplu:** rotunjiți 1.100101 (1.578125) la doar 3 biți parte fracționară
 - Down: 1.100
 - Up: 1.101
 - Toward zero: 1.100
 - To nearest: 1.101 (1.625 is closer to 1.578125 than 1.5 is)

Adunarea în v.m.

1. Extragem biții pentru exponent și mantisă
2. Adăugăm un bit de 1 în fața mantisei
3. Comparăm exponenții
4. Shiftăm mantisa mai mică dacă e necesar
5. Adunăm mantisele
6. Normalizăm mantisele și ajustăm exponentul dacă e necesar
7. Rotunjim rezultatul
8. Asamblăm exponentul și fracția înapoi în formatul pentru virgulă mobilă

Exemplu de adunare în v.m.

Adunați următoarele numere în v.m.:

0x3FC00000

0x40500000

Exemplu de adunare în v.m.

1. Extragem biții pentru exponent și fracție

Pentru primul număr (N1): $S = 0$, $E = 127$, $F = .1$

Pentru al doilea număr (N2): $S = 0$, $E = 128$, $F = .101$

1 bit	8 bits	23 bits
0	01111111	100 0000 0000 0000 0000 0000
Sign	Exponent	Fraction

1 bit	8 bits	23 bits
0	10000000	101 0000 0000 0000 0000 0000
Sign	Exponent	Fraction

2. Anexăm 1 pentru a forma mantisa

N1: 1.1

N2: 1.101

Exemplu de adunare în v.m.

3. Comparăm exponenții

$127 - 128 = -1$, așa că shiftăm N1 dreapta cu 1 bit

4. Shiftăm mantisa mai mică dacă e necesar

shift mantisa lui N1: $1.1 \gg 1 = 0.11 (\times 2^1)$

5. Adunăm mantisele

$$\begin{array}{r} 0.11 \times 2^1 \\ + 1.101 \times 2^1 \\ \hline 10.011 \times 2^1 \end{array}$$

Exemplu de adunare în v.m.

6. Normalizăm mantisa și ajustăm exponentul, dacă e necesar

$$10.011 \times 2^1 = 1.0011 \times 2^2$$

7. Rotunjim rezultatul

Nu e nevoie (intră în 23 biți)

8. Asamblăm exponentul și fracția înapoi în formatul v.m.

$$S = 0, E = 2 + 127 = 129 = 10000001_2, F = 001100..$$

1 bit	8 bits	23 bits
0	10000001	001 1000 0000 0000 0000 0000
Sign	Exponent	Fraction

în hexazecimal: **0x40980000**

Floating Point în C

- C garantează două niveluri
 - float** single precision
 - double** double precision
- Conversii/Casting
 - Cast între **int**, **float** și **double** schimbă reprezentarea binară
 - **double/float** → **int**
 - Trunchiază partea fracționară
 - Similar cu rotunjirea spre zero
 - Nu e definită când out of range sau NaN: în general setează la min
 - **int** → **double**
 - Conversie exactă, cât timp **int** are ≤ 53 biți word size
 - **int** → **float**
 - Va rotunji conform modului de rotunjire

Floating Point Puzzles

- Pentru fiecare din expresiile de mai jos:
 - Ce rezultat este adevărat pentru orice argument al expresiilor?
 - Explicați de ce anumite expresii nu sunt adevărate

```
int x = ...;  
float f = ...;  
double d = ...;
```

Presupuneți că nici
d nici **f** nu este NaN

- $x == (\text{int})(\text{float}) x$
- $x == (\text{int})(\text{double}) x$
- $f == (\text{float})(\text{double}) f$
- $d == (\text{float}) d$
- $f == -(-f);$
- $2/3 == 2/3.0$
- $d < 0.0 \Rightarrow ((d*2) < 0.0)$
- $d > f \Rightarrow -f > -d$
- $d * d \geq 0.0$
- $(d+f)-d == f$

Computation Cost

- La implementarea unui algoritm, folosiți operațiile de cost computațional minim
- Ierarhia operațiilor în majoritatea proc (fast to slow):
 - comparații
 - (u)int add, subtract, bitops, shift
 - floating point add, sub (separate unit!)
 - indexed array access (caveat: cache effects)
 - (u)int32 mul
 - FP mul
 - FP division, remainder
 - (u)int division, remainder

Number of Digits

```
uint32_t digits10(uint64_t v) {  
    uint32_t result = 0;  
    do {  
        ++result;  
        v /= 10;  
    } while (v);  
    return result;  
}
```

- Basic algorithm using division

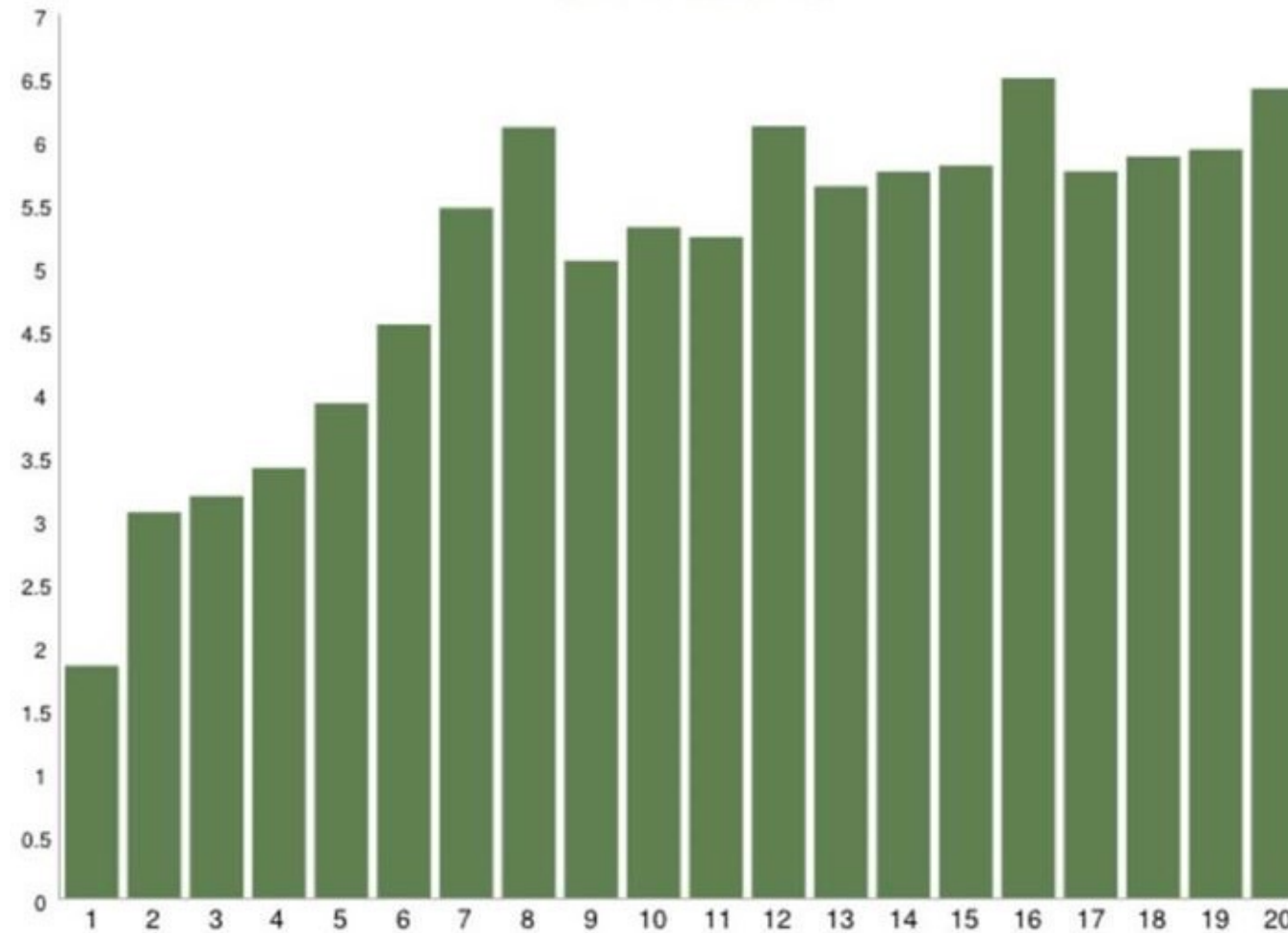
Number of Digits

```
uint32_t digits10(uint64_t v) {
    uint32_t result = 1;
    for (;;) {
        if (v < 10) return result;
        if (v < 100) return result + 1;
        if (v < 1000) return result + 2;
        if (v < 10000) return result + 3;

        v /= 10000U;
        result += 4;
    }
}
```

Reduced strength algorithm using
comparison with division fallback

digits10() Relative Speedup



Data from Andrei Alexandrescu (Facebook)

String to integer

```
unsigned atoi(const char* b, const char* e) {  
    enforce(b < e);  
    unsigned result = 0;  
    for (; b != e; ++b) {  
        enforce(*b >= '0' && *b <= '9');  
        result = result * 10 + (*b - '0');  
    }  
    return result;  
}
```

How Does it Work?

$$523924 = ((((((0 * 10 + 5) * 10 + 2) * 10 + 3) * 10 + 9) * 10 + 2) * 10 + 4$$

Divide & Conquer?

"523924" : "523" then "924"

$$523924 = 523 * 1000 + 924$$

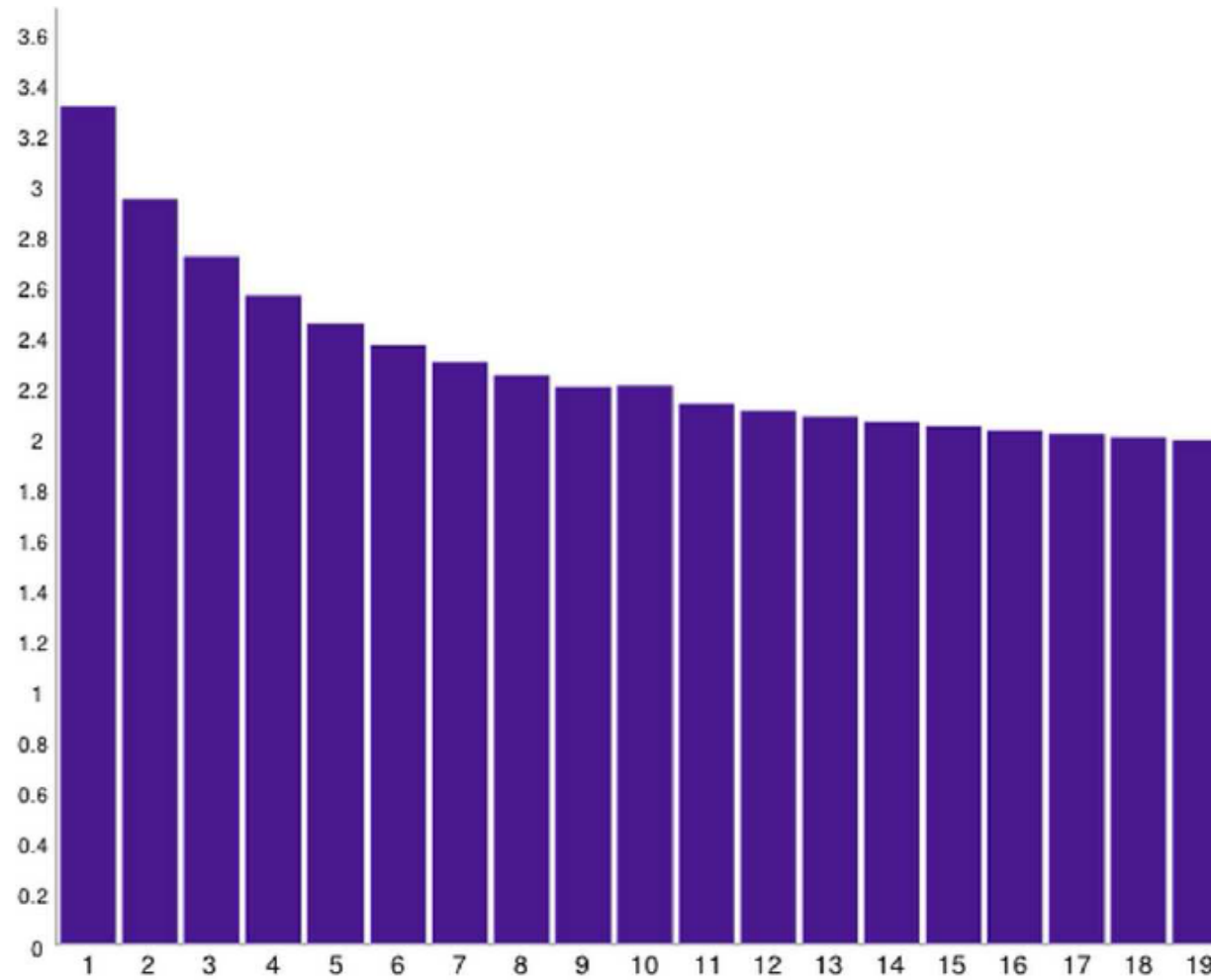
Key: Multiply then Add

$$\begin{aligned} 523924 &= 5 * 100,000 \\ &+ 2 * 10,000 \\ &+ 3 * 1,000 \\ &+ 9 * 100 \\ &+ 2 * 10 \\ &+ 4 \end{aligned}$$

Reducing Dependencies

```
unsigned atoui(const char* b, const char* e) {
    static const unsigned pow10[20] = {
        10000000000000000000UL,
        ...
        1
    };
    enforce(b < e);
    unsigned result = 0;
    auto i = sizeof(pow10) / sizeof(*pow10) - (e - b);
    for (; b != e; ++b) {
        enforce(*b >= '0' && *b <= '9');
        result += pow10[i++] * (*b - '0');
    }
    return result;
}
```

Performance



Data from Andrei Alexandrescu (Facebook)

Acknowledgements

- Aceste slide-uri conțin materiale aparținând:
 - Arvind (MIT)
 - Krste Asanovic (MIT/UCB)
 - Joel Emer (Intel/MIT)
 - James Hoe (CMU)
 - John Kubiatowicz (UCB)
 - David Patterson (UCB)
 - Behrooz Parhami (UCSB)
 - Andrei Alexandrescu
- MIT material derived from course 6.823
- UCB material derived from course CS252