Laborator 0x09 + 0x0A

1. Reprezentarea nr. în virgulă mobilă
2. Float în x86. Registrii
   Instrucțiuni
   Apeluri în libm
3. FPU - stack : fldz și fstps
4. Calculul lui log(x)
5. Registrii xmm
6. Media aritmetică a unui array de floats
7. Introducere în RISC V
8. Registrii în RISC V
9. Tipuri de date

Reprezentarea în virgulă mobilă

Formatul single

- pe 32 de biți
  - 1 bit semn
  - 8 biți exponent
  - 23 biți mantisă (partea fracționară)

Exemplu

Fie $n = -1313.3125$

În reprezentarea pe single IEEE FP 754

Algoritm

1. Se consideră numărul în modul

$n = 1313.3125$

$1313_{10} \Rightarrow 2^{10} + 2^9 + 2^5 + 2^0 = 10100100001_2$

1313 -
1024
289 -
256
33
32
1
1
=

$0.3125_{10} = 0101_2$

$0,3125 \cdot 2 = 0,625 \quad 0$

$0,625 \cdot 2 = 1,250 \quad 1$

$0,25 \cdot 2 = 0,5 \quad 0$

$0,5 \cdot 2 = 1 \quad 1$

$(1313.3125)_2 = 10100100001.0101$

normalizare

=> mut virgula după primul 1

$1.01001\,0001\,0101 \cdot 2^{10}$

mantisă

-1313.3125

bit semn : 1

exponent : $10 + 127 = 137 = 2^7 + 2^3 + 1 = 10001001$ (127 din format)

mantisa : 01001 0000 1010 1 0000 0000

(completăm cu 0 până la 23)

Reprezentarea pe 32 de biți

1 10001001 01001 00001 0101 00000000

Float în x86

Registrii, instrucțiuni, apeluri în libm

două abordări:

- utilizarea registrilor %xmm0 → %xmm7 (sunt pe 128 b) + instrucțiuni specifice calculez date în progr. fără apeluri de fct. matematice
- FPU (floating point unit) se utilizează FPU-stack st(0) ... st(7) când fac apeluri în libm (lib math)

FPU - stack

- stivă cu 8 spații pt. floats
- avem două operații

flds op = float load single

fstps op = float store & pop single

↑ op este FLOAT !

Convenții de apel în libm

- construim cadrul de apel ca în cazul procedurilor deja studiate
- funcțiile din libm returnează în st(0)

gcc -m32 file.s -o file -no-pie -lm ← linkm explicit pt. libm

Exemplu : Calculul ln(x) logf

```
.data
    x: .float 2.718218
    logResult: .space 4
.text
.global main
main:
    push x
    call logf
    # la revenire din apel, rezultatul este in st(0)
    fstps logResult
    add $4, %esp
et_exit:
    mov $1, %eax
    mov $0, %ebx
    int $0x80
```

Debug

```
b et_exit
run
print (float) logResult
```

Registrii %xmm

Instrucțiuni specifice

addss src, dest => dest += src

subss src, dest dest -= src

mulss src, dest dest *= src

divss src, dest dest /= src

src, dest ∈ FLOAT

movss src, dest

cvtsi2ss src, dest = instrucțiune de conversie

long ↙ ↘ float

Exemplu:

Să se calculeze media aritmetică a unui array de floaturi

Sol:

```
.data
    v: .float 11, 2, 3, 7
    n: .long 4
    avg: .space 4
.text
.global main
```

```
main :
    lea v, %edi
    xor %ecx, %ecx
    // initializăm xmm cu 0 pentru sumă
    movl $0, %eax
    cvtsi2ss %eax, %xmm0

et_loop :
    cmp n, %ecx
    je et_exit_loop
    movss (%edi, %ecx, 4), %xmm1
    addss %xmm1, %xmm0

    inc %ecx
    jmp et_loop

et_exit_loop :
    cvtsi2ss %ecx, %xmm1    # xmm1 ← n
    divss %xmm1, %xmm0      # xmm0 ← sum / n
    movss %xmm0, avg

et_exit :
    mov $1, %eax
    mov $0, %ebx
    int $0x80
```

□

Introducere în RISC-V

două clase de procesoare

- CISC (complex Instruction-Set Computer)
  - Intel x86, AT&T, AMD
- RISC (Reduced Instruction-Set Computer)
  - ARM = advanced risc machine
  - MIPS, RISC-V (Apple M1, M2, ...)

Obs 1. În clasa CISC avem codificare variabilă

RISC avem codificare fixă pe 32 biți

2. RISC are o arhitectură LOAD & STORE = instr. nu operează direct asupra memoriei

Regiștrii în RISC-V = 32 reg pe 32 biți

- zero are valoarea zero
- gp global pointer = reține adresa din mem. a secțiunii .data
- sp stack pointer
- s0 frame pointer (/ ebp)
- tp thread pointer
- t0 - t7 reg. temporari (reg. ce NU trebuie restaurați
- s1 - s11 reg. salvați (reg ce trebuie restaurați)
- a0 - a7 reg. pt argumente → a0, a1 pentru return
  - a7 pt codul apelurilor de sistem
- pc program counter (/ eip)

Tipuri de date:

. byte : 1 B = 8 biți

. halfword : tip pe 16 b (2 B)

. word : tip pe 32 b (4 B)

. double word : tip pe 64 b (8 B)

+ .ascii, .asciiz, .space - la fel ca pe x86

Exemplu:

```
gp:     .data
            x:  .word 15
            y:  .doubleword 255
            z:  .byte 55
            t:  .halfword 16
```

```
x ← 0(gp)
y ← 4(gp)
z ← 12(gp)
t ← 13(gp)
```