Numele si prenumele (cu MAJUSCULE): _____ Grupa: _____

Test: _____ Activitate: _____ Colocviu: _____ FINAL: _____

Test de laborator - Arhitectura Sistemelor de Calcul

Grupele 141, 142, 143, 144, 152
17 ianuarie 2026
Varianta B

- Nota maximă pe care o puteţi obţine este 10.
- Nota obţinută trebuie să fie minim 5 pentru a promova, fără nicio rotunjire superioară.
- Orice tentativă de fraudă este considerată o încălcare a Regulamentului de Etică!

1 Partea 0x00: x86 (3 puncte)

1.1 Exerciţiul 0x00 (1.5 puncte)

Presupunem că aveţi acces la un executabil `exec`, pe care îl inspectaţi cu `objdump -d exec`. În momentul în care rulaţi această comandă, vă opriţi asupra următorului cod. Analizaţi-l şi răspundeţi întrebărilor de mai jos. Pentru fiecare răspuns în parte, veţi preciza şi instrucţiunile (indicând numărul liniilor) care v-au ajutat în rezolvare. Formatul de mai jos este *număr linie: adresă: instrucţiune*.

```
08049166 <f>:
 0x00: 8049166:       push   %ebp
 0x01: 8049167:       mov    %esp,%ebp
 0x02: 8049169:       sub    $0x14,%esp
 0x03: 8049176:       mov    0x8(%ebp),%eax
 0x04: 8049179:       mov    %al,-0x14(%ebp)
 0x05: 804917c:       movl   $0x0,-0x8(%ebp)
 0x06: 8049183:       movl   $0x0,-0x4(%ebp)
 0x07: 804918a:       jmp    80491a4 <f+0x3e>
 0x08: 804918c:       mov    -0x8(%ebp),%edx
 0x09: 804918f:       mov    0xc(%ebp),%eax
 0x0a: 8049192:       add    %edx,%eax
 0x0b: 8049194:       movzbl (%eax),%eax
 0x0c: 8049197:       cmp    %al,-0x14(%ebp)
 0x0d: 804919a:       jl     80491a0 <f+0x3a>
 0x0e: 804919c:       addl   $0x1,-0x4(%ebp)
 0x0f: 80491a0:       addl   $0x1,-0x8(%ebp)
 0x10: 80491a4:       mov    -0x8(%ebp),%edx
 0x11: 80491a7:       mov    0xc(%ebp),%eax
 0x12: 80491aa:       add    %edx,%eax
 0x13: 80491ac:       movzbl (%eax),%eax
 0x14: 80491af:       cmpb   $0,%al
 0x15: 80491b1:       jne    804918c <f+0x26>
 0x16: 80491b3:       mov    -0x4(%ebp),%edx
 0x17: 80491b6:       mov    0x10(%ebp),%eax
 0x18: 80491b9:       add    %edx,%eax
 0x19: 80491bc:       ret
```

a. (0.5p) Câte argumente primeşte procedura `f`?

Solution: trei argumente, aflate la 0x8(ebp), 0xc(ebp) si 0x10(ebp) (vedem indexul maxim la linia 0x17)

b. (0.5p) Care este tipul argumentelor identificate şi cum aţi identificat acest lucru? Instrucţiunea `movzbl` reprezintă o instrucţiune care face `mov` cu o conversie de la `byte` la `long`.

Solution: identificam tipul argumentelor in functie de instructiunile in care sunt utilizate. urmarim pentru 0x8(ebp) si vedem ca este utilizat la 0x03, unde e copiat in eax, iar din eax se foloseste byte-ul cel mai putin nesemnificativ, adica al, care este mutat intr-o variabila locala, de unde conchidem ca este un byte (un char). pentru 0xc(ebp) observam calupul 0x09-0x0b, unde argumentul este pus in eax, i se adauga indexul curent (tinut in edx, luat din -0x8(ebp), initial facut 0 si incrementat la fiecare pas la 0x0f), iar apoi este dereferentiat prin conversie de la byte la long, deci este adresa unui sir de caractere. in ceea ce priveste 0x10(ebp), apare doar pe instructiuni pentru long, si este si utilizat in constructia valorii de retur care este un long, de unde conchidem ca este un long. Se acorda 0.15p pentru fiecare tip identificat corect, si inca 0.05p daca cel putin un tip este identificat corect.

c. (0.5p) Procedura `f` conţine o structură repetitivă. Identificaţi condiţia de ieşire din structură (oferiţi o explicaţie a condiţiei de ieşire, nu este suficient doar să precizaţi instrucţiunile în limbaj de asamblare).

Solution: Pentru a gasi conditia de iesire, cautam saltul inapoi, si il identificam la linia 0x15. Conditia vine din comparatia cu 0 a registrului al - daca al este diferit de 0, se face salt inapoi, deci se iese cand al este 0. Urmarim cine este al in acest context, si din calupul 0x11-0x13 conchidem ca este vorba despre elementul curent al unui sir de caractere, deci se face o parcurgere pana la finalul sirului (pana cand str[i] eq 0).

1.2 Exerciţiul 0x01 (1.5 puncte)

În acest exerciţiu vreţi să calculaţi şi să afişaţi valoarea lui $erf\left(\frac{x}{4}\right)$ (definită prin $erf(z) := \frac{2}{\sqrt{\pi}} \int_0^z e^{-t^2} dt$). Consideraţi că aveţi următoarea secţiune `.data`:

```
x: .float 5
y: .space 4
formatPrintf: .asciz "erf(5/4) = %f\n"
```

Pentru a calcula valoarea $erf\left(\frac{x}{4}\right)$, veţi apela procedura `erf(x/4)` din `libmath`, care returnează valoarea conform convenţiilor `FPU`. Pentru calculul $\frac{x}{4}$ veţi utiliza instrucţiunea `divss` din `SIMD`. Scrieţi secvenţa de instrucţiuni din `main` care calculează $\frac{x}{4}$, aplică funcţia erf, respectiv afişează rezultatul la `stdout`, printr-un apel al procedurii `printf`.

Solution:

```
movl $4, %eax
cvtsi2ss %eax, %xmm1
movss x, %xmm0
divss %xmm1, %xmm0      # 0.5p

sub $4, %esp
movss %xmm0, 0(%esp)
call erff
add $4, %esp
fstps y                 # 0.5p

sub $12, %esp
movl $formatPrintf, 0(%esp)
movss y, %xmm0
cvtss2sd %xmm0, %xmm0
movsd %xmm0, 4(%esp)
call printf
add $12, %esp           # 0.5p
```

2 Partea 0x01: RISC-V (3 puncte)

2.1 Exerciţiul 0x00 (1 punct)

Scrieţi o procedură în RISC-V care să calculeze produsul cifrelor impare ale unui număr. În această procedură veţi utiliza cel puţin **un registru** dintre `s1-s11` pentru valorile suplimentare necesare, şi nu doar `t0-t6`. Reprezentaţi stiva în momentul în care are adâncimea maximă (pentru programul vostru).

Solution:

```
productOddDigits:
  addi sp, sp, -8
  sw ra, 4(sp)
  sw s0, 0(sp)
  add s0, sp, x0
  addi sp, sp, -4
  sw s1, 0(sp)

  li s1, 1
  li t0, 10
  li t1, 2
  li t4, 1
productOddDigitsLoop:
```

```
  beqz a0, productOddDigitsLoopExit
  rem t2, a0, t0
  rem t3, t2, t1
  beq t3, t4, isOddDigit

productOddDigitsLoopCont:
  div a0, a0, t0
  j productOddDigitsLoop

isOddDigit:
  mul s1, s1, t2
  j productOddDigitsLoopCont

productOddDigitsLoopExit:
  add a0, s1, x0
  lw ra, 4(s0)
  lw s1, -4(s0)
  lw s0, 0(s0)
  addi sp, sp, 12
  jr ra
```

2.2 Exerciţiul 0x01 (1 punct)

Un procesor pe 32b execută în 25% dintre cazuri instrucţiuni aritmetice, în 30% dintre cazuri doar apeluri de memorie, iar în rest instrucţiuni care folosesc `floating point`. O instrucţiune aritmetică necesită 2 cicli, o accesare de memorie necesită 4 cicli, iar o instrucţiune pe `floating point` necesită tot 4 cicli, dar efectuează o accesare suplimentară de memorie. Determinaţi CPI şi IPC pentru acest procesor. Ar fi mai eficient dacă instrucţiunile aritmetice ar necesita 6 cicli, dar am introduce un nou circuit pentru `floating point`, încât să necesite o astfel de instrucţiune doar 5 cicli, fără alte accesări de memorie? Justificaţi numeric, prin calculul CPI şi IPC pentru această specificaţie.

Solution: $CPI_1 = 0.25 * 2 + 0.3 * 4 + 0.45 * (4 + 4) = 0.5 + 1.2 + 3.6 = 5.3$, deci $IPC_1 = 1/5.3 \approx 0.188$. Cu modificarile precizate, $CPI_2 = 0.25 * 6 + 0.3 * 4 + 0.45 * 5 = 4.95$, deci $IPC_2 = 1/4.95 \approx 0.202$. Avem ca $CPI_2 < CPI_1$, deci da, ar fi mai eficient sa facem aceste inlocuiri.

2.3 Exerciţiul 0x02 (1 punct)

Un sistem de calcul pe 32b are o memorie principală de 2^{22}`B` şi o memorie cache de `16 KiB`, împărţite pe blocuri de dimensiune `512 B`. Determinaţi numărul de blocuri din memoria principală, respectiv numărul de blocuri din memoria cache. Determinaţi unde se mapează, în cache, variabila de la adresa de memorie `0xFEAB`.

Solution: Mem. principala: 2**(22) : 2**9 = 2**(13) blocuri; Cache: 2**10 * 2**4 : 2**9 = 2**5 blocuri in cache. Adresa 0xFEAB = 1111 1110 1010 1011, de unde extragem tag, index, offset. Avem offset = ultimii 9b, adica 0 1010 1011 (0x0AB), index urmatorii 5, adica 1 1111 (0x1F), iar tag = 11 (0x3).

3 Partea 0x02: Întrebări generale (3 puncte)

Alegeţi varianta corectă sau răspundeţi pe scurt.

0x00 Regiştrii `s0-s11` sunt o alegere bună pentru stocarea unor valori pe care urmează să le folosim în cadrul mai multor proceduri distincte. (de exemplu, să avem valoarea constantei π în registrul `s0` şi să accesăm mereu această constantă astfel în interiorul procedurilor)

a. Adevărat b. Fals

0x01 (0.50p) De ce, în codificarea unei instrucţiuni din clasa **J**, nu scriem şi valoarea pentru `imm[0]`?

Solution: Mereu este 0 valoarea respectiva, pentru ca aliniem mereu la multiplu de 2, obtinem codificari de forma ...0, sau ...1 + 1 = ...0 (complementul fata de 2) (offset-urile sunt mereu pare)

0x02 (0.50p) Fie următoarea secţiune `.data`:

```
.data:
    .ascii "sir"
    .byte 5
```

Scrieţi o instrucţiune prin intermediul căreia să încărcaţi valoarea 5 din `.data` în registrul `a0`.

Solution: `lb s1, 3(gp)`

0x03 (0.75p) În analiza unui executabil, identificaţi următoarea valoare pe formatul `single`, `x = 0xC5000C80`. Ce număr îi corespunde în baza 10?

Solution: 0xC5000C80 are codificarea binara 1 10001010 00000000000110010000000. Este un numar negativ. Identificam exponentul, 10001010 = 2**7 + 2**3 + 2**1 = 138. 138 - 127 = 11, deci exponentul este 11. Avem ca numarul in forma stiintifica este 1.00000000000110010000000 * 2**11, deci forma initiala era 100000000000.110010000000. Transformam partile intreaga/fractionara: 100000000000 este 2**11, deci 2048, iar 11001 este 2**(-1) + 2**(-2) + 2**(-5) = 1/2 + 1/4 + 1/32 = (16 + 8 + 1)/32 = 25/32 = 0.78125. Astfel, numarul este -2048.78125.

0x04 (0.25p) Consideraţi instrucţiunea `beq` din `RISC-V` (instrucţiune a clasei `B`). Care este intervalul de valori `immediate` pe care le poate primi ca argument această instrucţiune?

a. [-2047, 2048] b. $[-2^{11}, 2^{12}]$ c. $[-2^{11}, 2^{12} - 1]$ d. [-4094, 4095] e. [-4096, 4095]

Solution: e

0x05 (0.75p) Precizaţi de ce decodificarea instrucţiunilor în `RISC-V` poate fi paralelizată, spre deosebire de decodificarea în `x86`.

Solution: In RISC-V, instructiunile au codificare pe lungime fixa, astfel ca stim mereu unde incepe si unde se termina o instructiune, incat putem paraleliza.

Important! Puteţi folosi spaţiul rămas mai jos pentru a scrie rezolvările altor subiecte, pentru care nu v-a ajuns spaţiul alocat anterior. Marcaţi fiecare completare prin textul *Continuarea părţii . . . , exerciţiul . . . .*