Arhitectura Sistemelor de Calcul (ASC)
Examinarea finală
Varianta 1
(2024 - 2025)

Anul I, Semestrul I
7 februarie 2025
Cristian Rusu

Nume: _____

Prenume: _____

Grupa: _____

Completați aici totul cu majuscule.

Toate răspunsurile sunt în albastru.

Înainte de a începe, citiți cu atenție indicațiile următoare:

- Testul și rezolvarea sa vor fi disponbile online în zilele următoare.
- *Nu aveți voie cu laptop-uri sau alte dispozitive de comunicație.*
- *Nici calculatoarele de buzunar nu sunt permise.*
- *Vă rugăm să vă opriți telefoanele mobile.*
- Pentru întrebările cu răspunsuri multiple/simple folosiți tabelele puse la dispoziție.
- Acest test are 6 enunțuri totalizând 100 de puncte.
- Aveți la dispoziție 120 de minute pentru a completa examinarea.
- Mult succes!

Întrebarea 1. (22 puncte)

Completați tabelul de mai jos cu valorile numerice corecte. Toate numerele sunt naturale pe 16 biți. (Fiecare răspuns corect valorează 1 punct)

binar	octal	zecimal	hexazecimal
b0000110001000001	o06101	3137	0x0C41
b0000100101010110	o04526	2390	0x0956
b0000100101110011	o04563	2419	0x0973
b0000010111000111	o02707	1479	0x05C7

a binar	a hexa	b binar	b hexa	a+b zecimal	a+b binar	a+b hexa
b0000001001	0x09	b0000111011	0x3B	68	b0001000100	0x44

a binar	a hexa	b binar	b hexa	axb zecimal	axb binar	axb hexa
b0000100101	0x25	b0001100101	0x65	3737	111010011001	0xE99

Răspunsurile care țin cont de overflow sunt de asemenea corecte.

Întrebarea 2. (13 puncte)

Completați tabelul de mai jos cu valorile numerice corecte. Toate numerele sunt întregi pe 12 biți. (Fiecare răspuns corect valorează 1 punct)

binar	octal	zecimal	hexazecimal
b1100011110	o7436	-226	0xF1E
b001110110100	o1664	948	0x3B4

a zecimal	a binar	a hexa	b zecimal	b binar	b hexa
-330	b1010110110	0xDB6	-24	b1111101000	0xFE8

produs axb zecimal	produs axb binar	produs axb hexa
7920	b1111011110000	0x1EF0

Răspunsurile care țin cont de overflow sunt de asemenea corecte. S-au punctat și cazurile în care octal si hexazecimal s-au considerat doar 10 biți (1 în loc de F sau 1 în loc de 7).

3.1.	Conrad Zuse
3.3.	Pascal
3.5	20 biți
3.7.	$\frac{16^n-1}{16-1}$
3.9.	NOT(b_1 XOR a_1) AND NOT(b_0 XOR a_0)
3.11.	A OR B
3.13.	$\min(a, b) - 1$
3.15.	0xC1800000
3.17.	0xC0780000
3.19.	verifică dacă au același exponent
3.21.	RAW
3.23.	1 ns (am găsit în cache L1 ce căutăm)
3.25.	suma = suma + (x[i] & 1)
3.27.	Single Instruction Multiple Data
3.29.	RISC
3.31.	bytecode
3.33.	plăcile grafice, este memoria

Întrebarea 3. (17 puncte)

Răspundeți la următoarele întrebări scurte. Completați în tabelul de pe pagina anterioară.

3.1. Cine construiește primele sisteme de calcul care reprezintă numere în virgulă mobila?

3.3. Matematician francez în onoarea căruia a fost numit un limbaj de programare în anii 80-90, este printre primii europeni care construiește un calculator mecanic.

3.5. În tabelul lui Mendeleev sunt 118 elemente. Fiecare element poate să aibă maxim 7 legături cu alți atomi. Câți biți are codificarea formulei H_2O (codificare cu dimensiune fixă).

3.7. Dacă un număr scris în binar are reprezentarea un șir de n biți de 1 atunci valorea lui în zecimal este $2^n - 1$? Analog, care este valoarea în zecimal a unui număr care în hexazecimal este un șir de n cifre 1?

3.9. Avem două numere naturale a, b pe 2 biți (a este a_1a_0 și b este b_1b_0). Scrieți o expresie logică (folosind AND, OR, NOT, XOR) care verifică dacă a este egal cu b?

3.11. Simplificați maxim expresia logică (A AND B) OR (A XOR B).

3.13. Avem un sistem de calcul pe care cineva a uitat să implementeze instrucțiunea de înmulțire pentru două numere naturale. Dar avem instrucțiunea de adunare pentru două numere naturale. Vrem să realizăm produsul $a \times b$ folosind doar instrucțiunea de adunare $a + b$. De câte ori este apelată această instrucțiune pentru a realiza înmulțireaa? (vrem să apelăm de cât mai puține ori adunarea)

3.15. în formatul single float IEEE, care este reprezentarea hexazecimală a numărului -16?

3.17. în formatul single float IEEE, care este reprezentarea hexazecimală a numărului -3.875?

3.19. Fie A și B două numere în format IEEE 754 FP pe 32 de biți. Care este rolul secvenței următoare? NOT(((A$\gg$23) AND 0x000000FF) XOR (B AND 0x7F800000)$\gg$23)

3.21. Pe un sistem cu arhitectura de calcul RISC-V avem următoarele două instrucțiuni: add x5, x1, x2 și sub x6, x5, x3. Ce fel de hazard este prezent în această secvență?

3.23. Avem un sistem de calcul cu memorie ierarhică: memorie cache L1 de 1MB cu viteza de acces 1ns, memorie cache L2 de 5MB cu viteza de acces 5ns, memorie cache L3 de 10MB cu viteza de acces 10ns și memoria principală RAM de 32GB cu viteza de acces 50ns. Care este timpul minim de acces în memorie pe un astfel de sistem?

3.25. Într-o variabilă *suma* contorizăm câte numere impare sunt în vectorul x. Care este expresia logică/aritmetică fără *dacă* care calculează corect suma?

3.27. Acronimul SIMD înseamnă

3.29. Arhitectura de calcul care este recunoscută pentru performanța per Watt este

3.31. Codul sursă Java nu se compilează direct în cod assembly, ci în

3.33. GDDR este folosită în

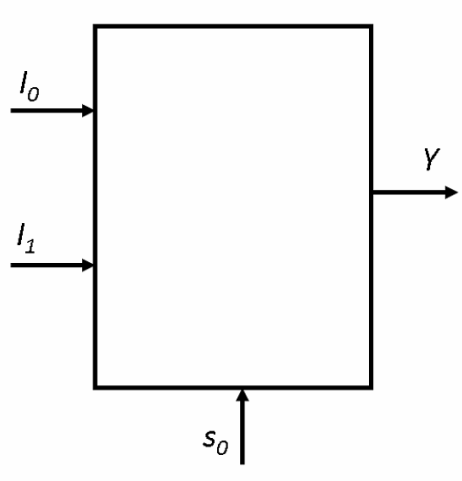
Întrebarea 4. (12 puncte)

Am văzut în timpul cursului cât de important este un multiplexor (MUX). Răspundeți la următoarele întrebări și realizați următoarele cerințe:

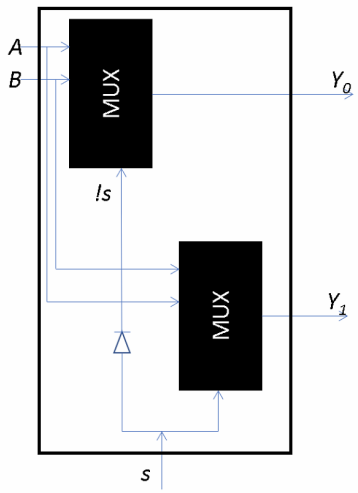
4.1. (2 puncte) Desenați în schema de mai jos funcția logică pentru un MUX cu două intrări și o ieșire.

4.2. (4 puncte) Scrieți funcția logică pentru un MUX cu 4 intrări (I_0, I_1, I_2 și I_3) și două semnale de selecție (s_0 și s_1).

4.3. (6 puncte) În x86/x87 am văzut instrucțiunea *xchg* (care interschimbă două valori). Implementați această instrucțiune pentru două variabile digitale A și B. Desenați un circuit care are două ieșiri: fie A și B, fie B și A în funcție de un semnal de selecție s (dacă $s = 0$ atunci se schimbă ordinea, altfel nu). Folosiți cât mai mult MUX-uri. Pentru acest circuit scrieți și funcțiile logice.



4.1. Trebuie desenat circuitul pentru $Y = I_0!s_0 + I_1 s_0$.
4.2. $Y = I_0!s_0!s_1 + I_1 s_0!s_1 + I_2!s_0 s_1 + I_3 s_0 s_1$.
4.3. E nevoie de două MUX-uri. Funcțiile logice sunt $Y_0 = As + B!s$ și $Y_1 = A!s + Bs$ iar circuitul este mai jos.



Întrebarea 5. (16 puncte)

Aveți un sistem de calcul care suportă instrucțiunile Assembly x86 de bază (aritmetice, logice, salturi). Din păcate, din cauza limitărilor sistemului acesta nu are implementate instrucțiuni de împărțire (adunare/scădere, înmulțire sunt implementate). Răspundeți la următoarele întrebări:

5.1. (4 puncte) Se dau două numere naturale $a > 1$, $b > 1$ și $a \geq b$. Scrieți pseudo-cod pentru o secvență aritmetică/logică care implementează operația de împărțire pentru a și b (avem nevoie de împărțirea întreagă) folosind repetat operația de adunare. De câte ori este utilizată operația de adunare?

5.2. (12 puncte) Acum, știm că $a \gg b$, a este mult mai mare decât b. Idem cu punctul 5.1., dar folosiți repetat și operația de înmulțire pentru a realiza împărțirea mai repede. De câte ori este utilizată operația de înmulțire?

5.1. result $= -1$, while $a \geq 0$ $a = a - b$ și result $=$ result $+ 1$. Se realizează $\lfloor a/b \rfloor$ operații de adunare (este în regulă și dacă ați numărat incrementarea rezultatului).

5.2. Ideea este să folosiți înmulțirea pentru a reduce numărul mare de adunări de la punctul anterior: porniți de la găsirea celei mai mari puteri p a lui b astfel încât $b^p \leq a$.
Pseudo-codul pentru o primă variantă:
result $= 0$
while $a - b \geq 0$
... $i_3 = 1$, $i_2 = 1$, $i_1 = b$
... while $i_1 \leq a$
... $\rightarrow i_3 = i_2$, $i_2 = i_1$, $i_1 = i_1 \times b$
... result $=$ result $+ i_3$, $a = a - i_2$

Numărul de înmulțiri este $\sum_{i=1} B_i \times i$ unde B_i este cifra i din reprezentarea numărului a în baza b. B_0 nu este folosit, B_0 este restul operației de împărțire. Numărul de adunări este proporțional cu numărul total de cifre B_i. Asta dacă nu păstrați în memorie puterile deja calculate. E nevoie de toate puterile, pentru că defapt aveți nevoie de reprezentarea numărului a în baza b.

Se punctează (parțial sau complet) orice algoritm corect care pornește de la ideea de mai sus atinge performanța: $\log_2(a)$, $\log_b(a)$ sau chiar $\log_b^2(a)$. Orice altă metodă care merge în acestă direcție este de asemenea punctată.

Întrebarea 6. (20 puncte)

Avem o implementare a unei liste simplu înlănțuite într-un limbaj de programare. Elementele listei sunt numere întregi pe 4 octeți iar adresele în memorie sunt numere naturale tot pe 4 octeți. Implementarea folosește o funcție *adaugă_nod* pentru a adăuga câte un element în listă, la finalul acesteia. Acestă funcție alocă memorie HEAP (nu pe stivă) la fiecare apel (iar alocarea se face prin syscall la sistemul de operare). Presupunem că avem o situație în care avem nevoie să adăugăm în listă un număr mare de noduri N. Răspundeți la următoarele întrebări:

6.1. (4 puncte) Descrieți două probleme de performanță care apar la utilizarea funcției *adaugă_nod*.

6.2. (8 puncte) Propuneți soluții pentru problemele identificate la punctul 6.1. Hint: folosiți operații de blocuri de memorie, nu pe elemente individuale.

6.3. (2 puncte) Pentru soluțiile propuse la punctul 6.2. analizați eficiența de stocare (date versus adrese la noduri următoare).

6.4. (6 puncte) Cu soluțiile propose anterior explicați cum accesați elementul $0 \leq i \leq N-1$ din lista și cum maximizați utilizarea cache-ului. Presupunem că o linie de cache are 512 octeți.

6.1. Elementele listei nu respectă principiu localizării (pentru cache) pentru că nodurile nu sunt contigue în memorie (fiecare nod este alocat undeva, și nu avem control unde anume). Apeluri repetate duc la syscall-uri repetate, iar acestea sunt lente.

6.2. Nu vom adăuga elemente unul câte unul, ci vom aloca un număr fix $M \gg 1$. Vom crea practic o lista de vectori de dimensiune M. Vom avea următoarea adresă doar la sfârșitul vectorului. Astfel vom avea N/M syscall-uri.

6.3. În loc de N adrese de salt pentru N noduri, vom avea N/M adrese de salt. Pentru M suficient de mare, aproape dublăm eficiența de stocare.

6.4. Blocul în care se află elementul i este i div M, iar în acel bloc vector indexul este i mod M. Ca toate aceste operații să fie eficiente, alegem $M = 2^m$ (div și mod se rezumă la operații pe biți). Valoarea m este aleasă astfel încât 2^m să fie cât mai aproape ca valoare de lungimea unei linii de cache (depinde de sistemul de calcul), $m = 10$. După un cache miss avem 128 hit-uri.