

Tutoriat 5

Stan Bianca-Mihaela, Stăncioiu Silviu

November 30, 2020



Contents

1 0-DS	2
1.1 PLA	2
1.2 PROM	4
1.3 Multiplexori	5
1.4 Multiplexori elementari	7
1.5 Decodificatori si codificatori	10
1.6 Sumatori	12
1.6.1 Half adder	13
1.6.2 Full adder	13
1.6.3 Sumator serial	15
2 Proceduri MIPS (conform standardelor MIPS și C)	16
2.1 Regiștri	17
2.2 Instrucțiuni folositoare	17
2.3 Operații pe stivă	18
2.4 Convenții	19
2.5 Exerciții	19

2.6	Mai multe exerciții	22
2.7	Apeluri imbricate	22
2.8	Exerciții	22
2.9	Proceduri recursive	24
2.10	Exerciții	24
2.11	Mai multe exerciții	25
2.12	Proceduri pentru array-uri	25
2.13	Exerciții	25
2.14	Mai multe exerciții	26
2.15	Array-uri de proceduri	27
2.16	Exerciții	27
2.17	Mai multe exerciții	30

1 0-DS

Sistemele 0-DS sunt circuite logice fara cicluri.

1.1 PLA

PLA-ul este un mod de a reprezenta o functie printr-un circuit.

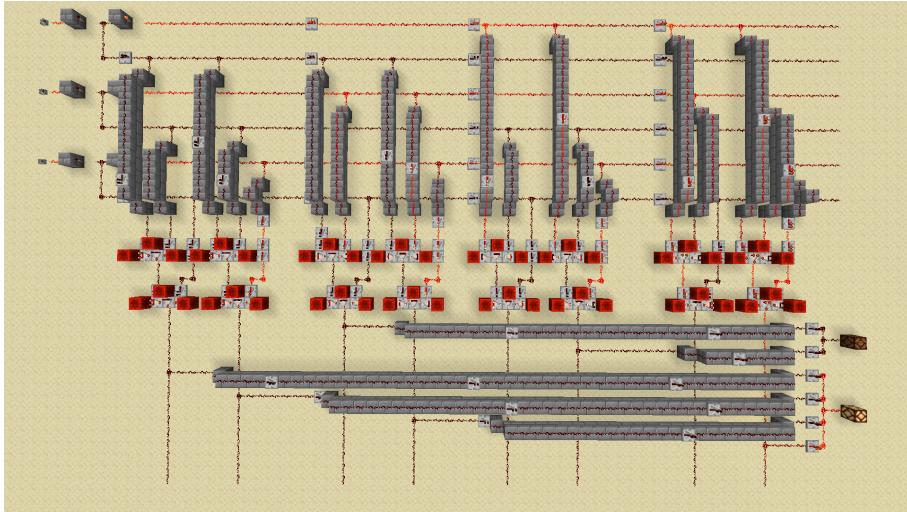
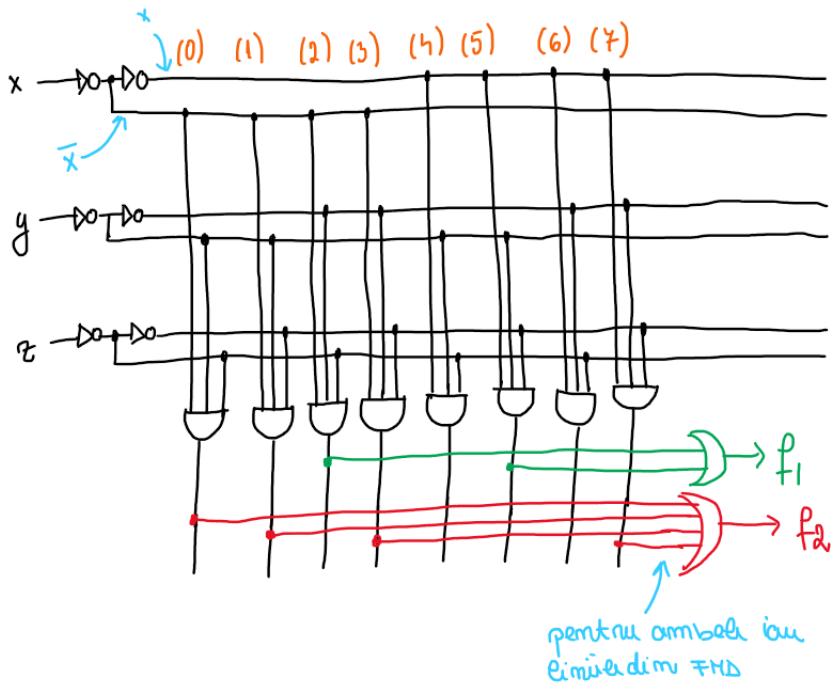
Exemplul 1 : Vom lua functia din tutoriatul 4, de la Exemplul 1, pentru ca ei i-am facut deja tabelul (inainte sa faceti PLA mereu va trebui sa faceti tabelul functiei): $f : B_2^3 \rightarrow B_2^2$, $f(x, y, z) = (f_1(x, y, z), f_2(x, y, z))$ cu

- $f_1(x, y, z) = (x \oplus y)(y \oplus z)$
- $f_2(x, y, z) = 1$ d.d. secventa x, y, z este crescatoare (adica $x \leq y \leq z$).

Tabelul pentru aceasta functie era:

index	x	y	z	$x \oplus y$	$y \oplus z$	$f_1(x, y, z)$	$f_2(x, y, z)$
(0)	0	0	0	0	0	0	1
(1)	0	0	1	0	1	0	1
(2)	0	1	0	1	1	1	0
(3)	0	1	1	1	0	0	1
(4)	1	0	0	1	0	0	0
(5)	1	0	1	1	1	1	0
(6)	1	1	0	0	1	0	0
(7)	1	1	1	0	0	0	1

Reprezentati prin PLA functia f.

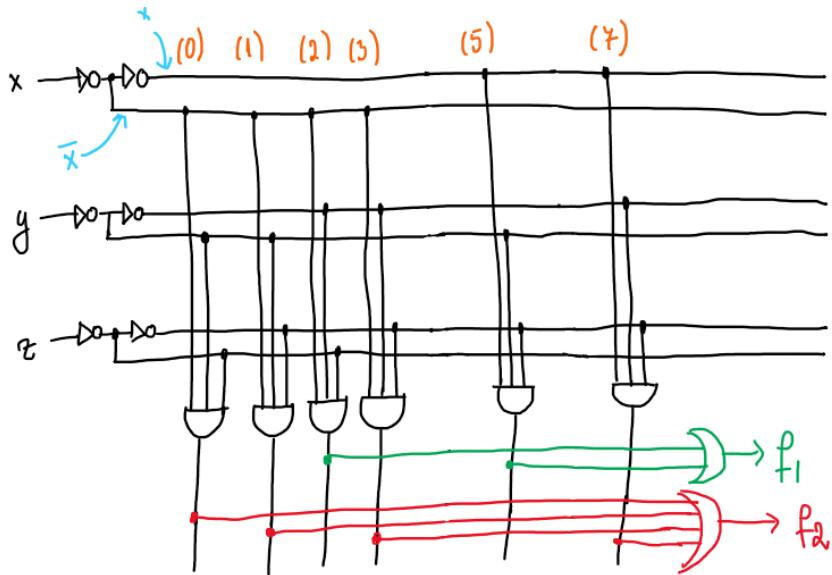


Observatii:

- Pentru fiecare dintre variabile avem o linie negata si una negata de 2 ori \Leftrightarrow valoarea initiala
- Avem grupari de cate 3 variabile, fiecare reprezentand o linie din tabel. Le-am notat exact ca in tabel, indexate de la 0.
- O functie se formeaza din "liniile" corespunzatoare FND-ului sau. De aceea avem nevoie de tabel dinainte.
- La fel cum in tabel, pentru x puneam 4 de 0 urmati de 4 de 1, pentru y 2 de 0, 2 de 1, 2 de 0 si 2 de 1 iar pentru z alternam: 0,1,0,1..., asa facem si in PLA: pentru (0), (1), (2), (3) luam valorile lui x de pe linia de jos, iar pentru restul de pe linia de sus; pentru y alternam liniile din 2 in 2, iar pentru z din 1 in 1.

- Forma FND-ului era $(.) + (.) + \dots + (.) \Rightarrow$ cand formam functia finala o formam cu SAU-uri, iar intre variabile avem SI.

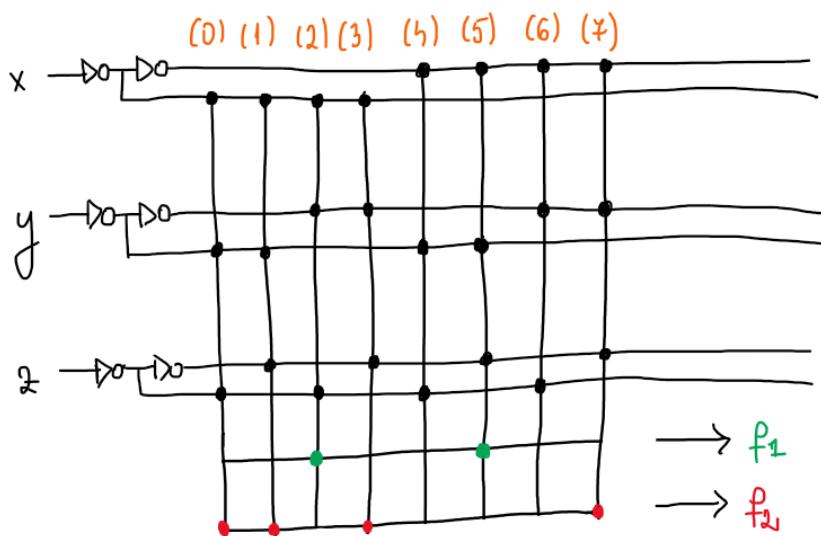
Daca va cere sa scrieti forma PLA cu numar minim de sume/produse, eliminati "liniile" care nu se afla nici in FND-ul lui f_1 , nici in FND-ul lui f_2 . In cazul nostru, eliminam (4) si (6):



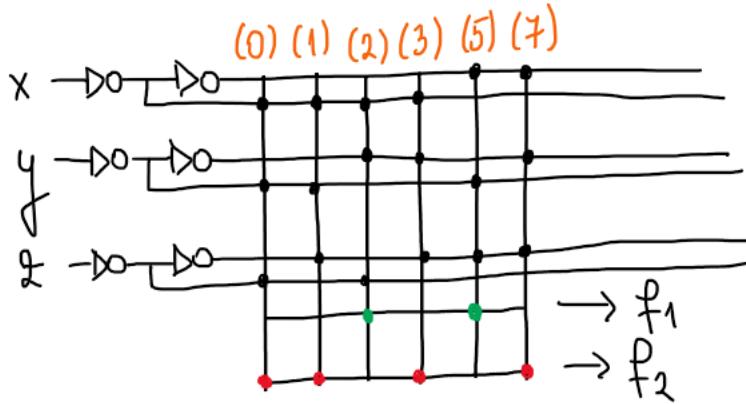
1.2 PROM

PROM-ul este un caz particular de PLA, deci o alta modalitate de reprezentare a functiei cu circuite. Aceasta reprezentare este insa simplificata.

Exemplul 2 Vom lua aceeasi functie de la Exemplul 1 si ii vom face PROM-ul.

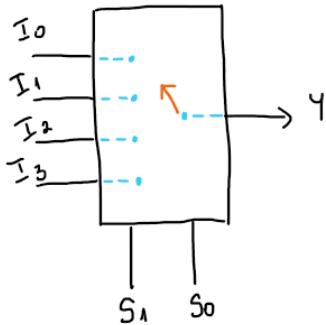


La fel, daca va cere sa scrieti un PROM cu numar nimic de sume/produse, eliminati "liniile" care nu sunt nici un FND-ul lui f_1 nici in FND-ul lui f_2 .



1.3 Multiplexori

Un multiplexor $MUX_n, n \geq 1$ cu selector pe n biti este un comutator de tip "many into one" care poate conecta o intrare selectabila printr-un cod numeric la o iesire unica. De exemplu, un MUX_2 arata cam asa:

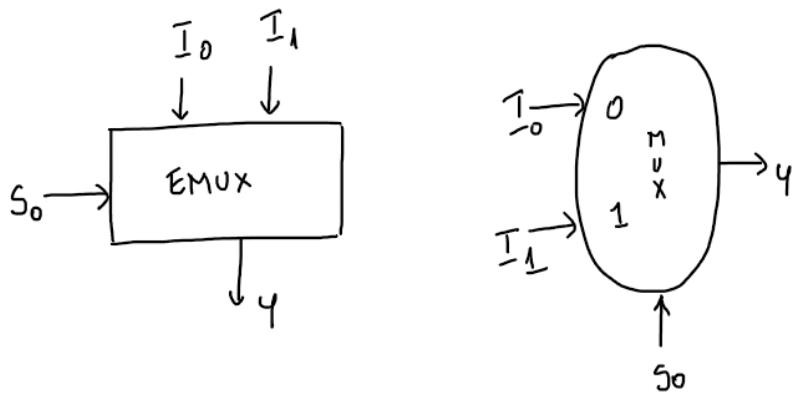


In functie de S_0 si S_1 , el alege una dintre intrari, mutand "comutatorul" pe acea intrare. Daca $S_0 = 0$ si $S_1 = 0$ va alege I_0 . . Daca $S_0 = 1$ si $S_1 = 0$ va alege I_2 .

Observatii:

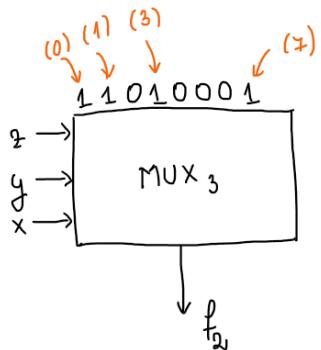
- n din MUX_n este dat de numarul de S -uri care intra in multiplexor.
- Numarul de I -uri care intra in multiplexor este 2^n

Pentru $n=1$ obtinem MULTIPLEXORUL ELEMENTAR care are reprezentarile:



Exemplul 3 Reprezentati functia f_2 de la Exemplul 1 cu un multiplexor.

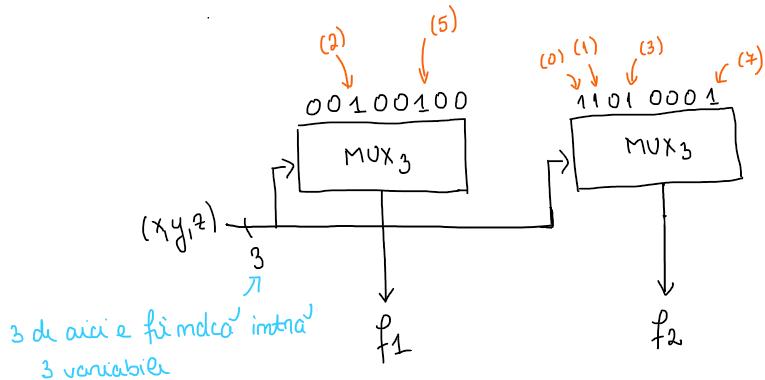
Reamintim ca functia f_2 avea valoarea 1 pentru (0), (1), (3) si (7). Asadar, reprezentarea cu un multiplexor este:



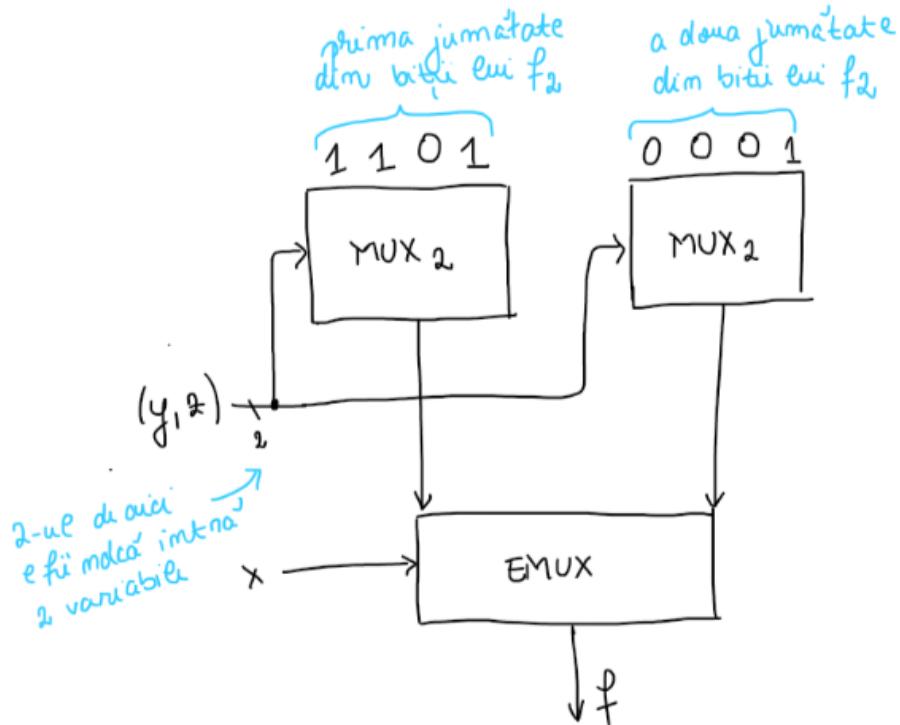
Exemplul 4 [RESTANTA SEPTEMBRIE 2020]

Implementati f printr-un circuit cu doua multiplexoare (cate unul pentru f_1 , f_2) cu aceiasi selectori x, y,z.

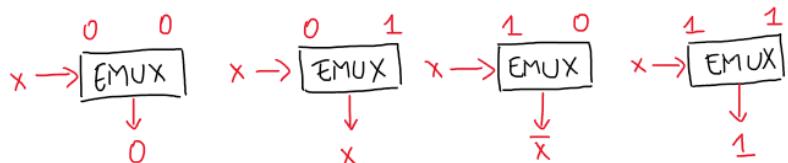
Reamintim ca f_1 avea valoarea 1 pentru (2) si (5).



Exemplul 5 Implementati functia f_2 de la Exemplul 1 cu 3 multiplexoare.



1.4 Multiplexori elementari



Exemplul 6 [RESTANTA MAI 2020]

Reprezentati functia $f : B_2^3 -> B_2^2$, $f(x, y, z) = (f_1(x, y, z), f_2(x, y, z))$ unde:

- $f_1(x, y, z) = x \cdot (y + z)$
- $f_2(x, y, z) = 1$ d.d. secventa xyz este reprezentarea in baza 2 a unui numar natural care nu este divizibil cu 3.

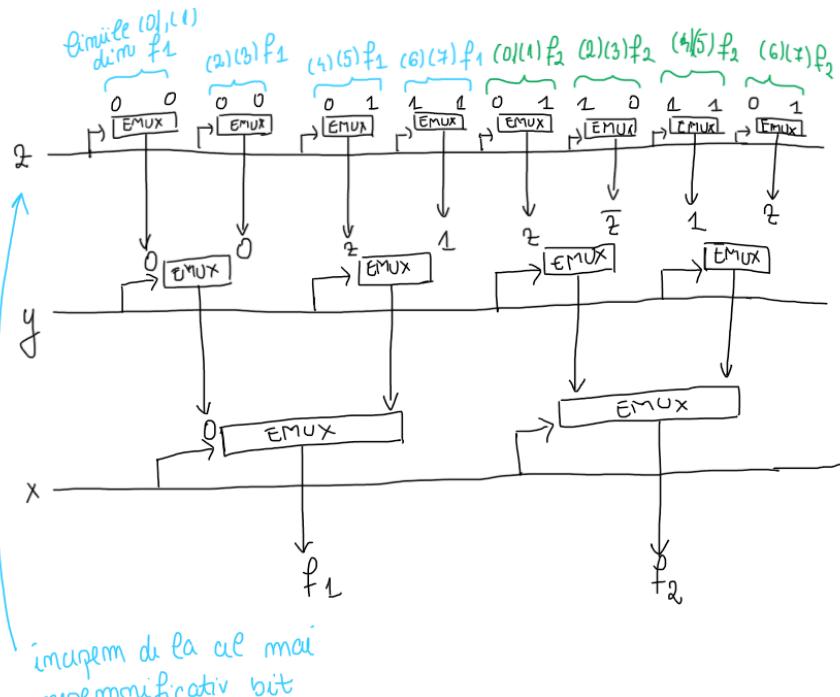
Implementati f printr-un circuit care contine doar multiplexoari elementari; apoi, reduceti la maximum numarul multiplexorilor elementari (nu este permisa adaugarea de porti NOT).

In primul rand vrem tabelul acestei functii.

Pentru f_2 avem 0 la liniile (0),(3),(6) (intuitiv, pentru ca stim ca linia reprezinta de fapt numarul \bar{xyz} in baza 10).

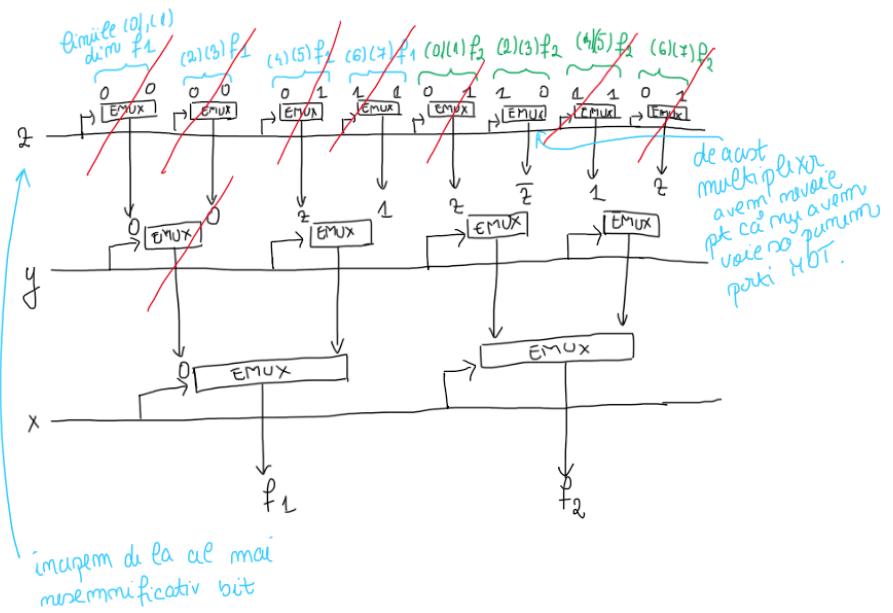
index	x	y	z	$y + z$	$f_1(x, y, z)$	$f_2(x, y, z)$
(0)	0	0	0	0	0	0
(1)	0	0	1	1	0	1
(2)	0	1	0	1	0	1
(3)	0	1	1	1	0	0
(4)	1	0	0	0	0	1
(5)	1	0	1	1	1	1
(6)	1	1	0	1	1	0
(7)	1	1	1	1	1	1

Acum ca avem tabelul, putem sa scriem reprezentarea:

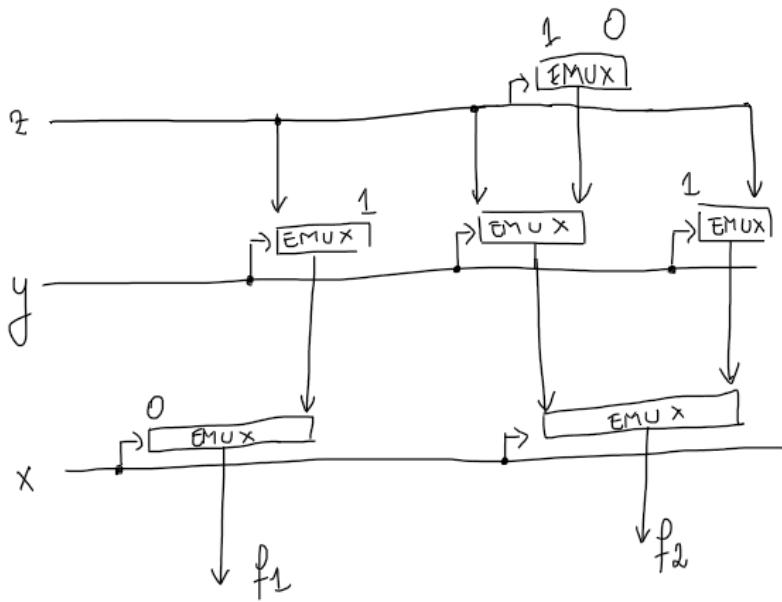


Pentru partea a doua a problemei, reducerea la minim a multiplexorilor elementari, vom elibera toti multiplexorii din careiese un 0, un 1 sau un x, y sau z. Daca aveam voie cu porti NOT putem sa eliminam si multiplexorii din careiese un \bar{x} , \bar{y} sau \bar{z} .

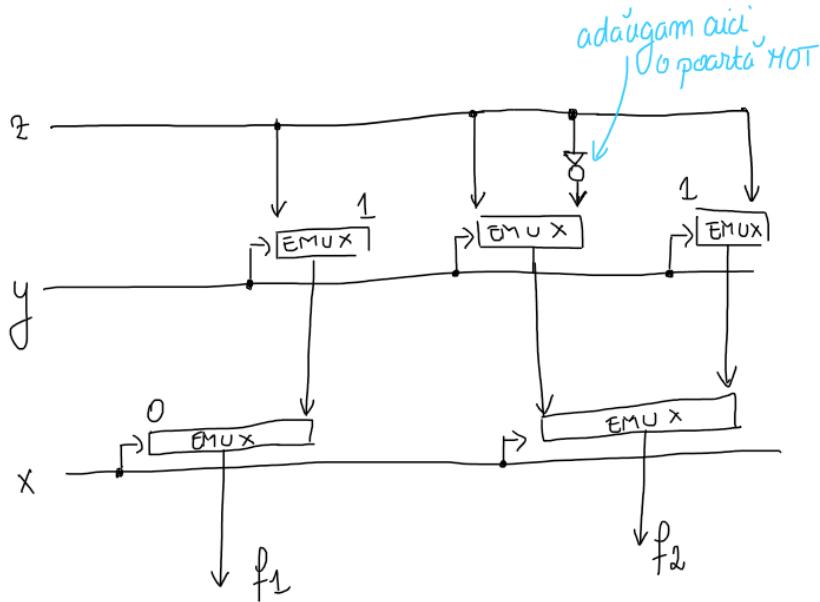
Cand scriem forma finala, e bine sa incepem cu multiplexorii de la x si sa urcam catre z. Asa ne dam seama foarte bine de ce avem nevoie si de ce nu. Taiem cu rosu ce vom elibera:



In final, minimizarea va fi:



Daca enuntul nu mentiona ca nu avem voie sa include porti NOT, diagrama minimizata ar fi fost:

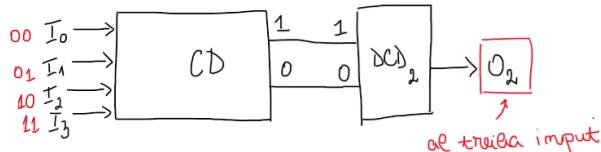


1.5 Decodificatori si codificatori

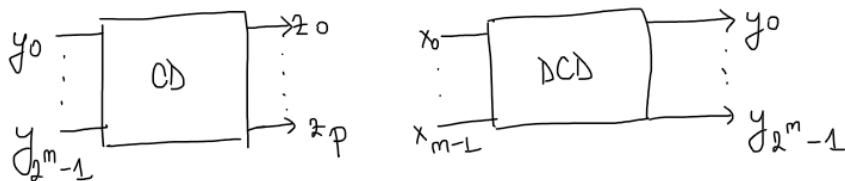
Un decodificator cu selector pe n biti $DCD_n, n \geq 1$ este un circuit care transforma un cod numeric k de n biti intr-o alegere fizica a liniei de iesire cu numarul k .

Un $(2^n, p)$ codificator este un circuit cu 2^n intrari dintre care la fiecare moment doar una este activa (are valoarea 1) si care genereaza la iesire o configuratie binara oarecare de lungime p .

Spre exemplu, din 4 intrari vrem sa o selectam pe a 3-a cu un codificator si un decodificator:



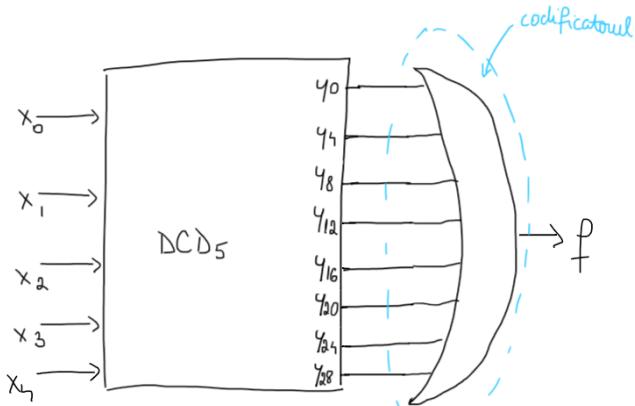
Asadar, diagrama pentru un codificator si diagrama pentru un decodificator arata:



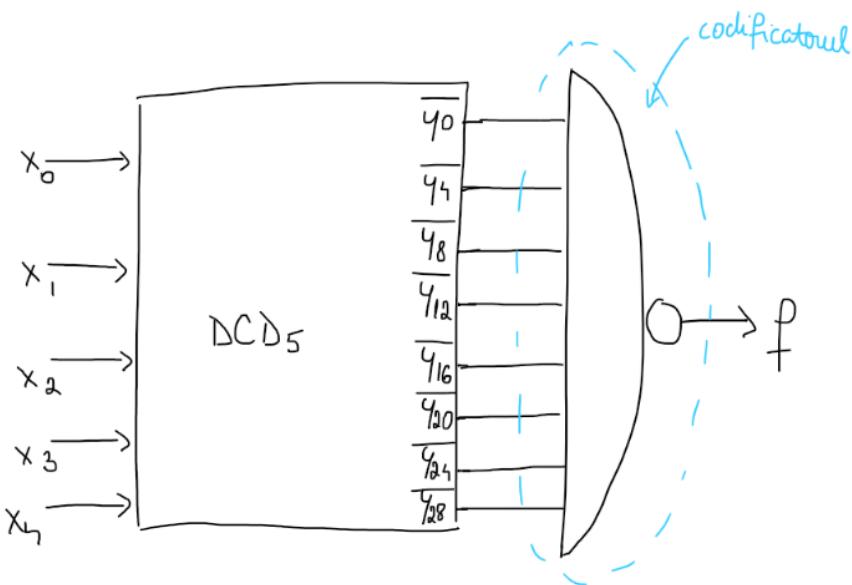
Exemplul 7 Scrieti un codificator care pentru 5 biti scoate "1" daca x divizibil cu 4.

Ce interval de numere pot reprezenta cu 5 biti? $0, \dots, 2^5 - 1 = 0, \dots, 31$. Numerele divizibile cu 4 din acest interval sunt: 0, 4, 8, 12, 16, 20, 24, 28.

Asadar:



Observatie! Daca folosim un decodificator cu iesiri negate, poarta OR se transforma in NAND:

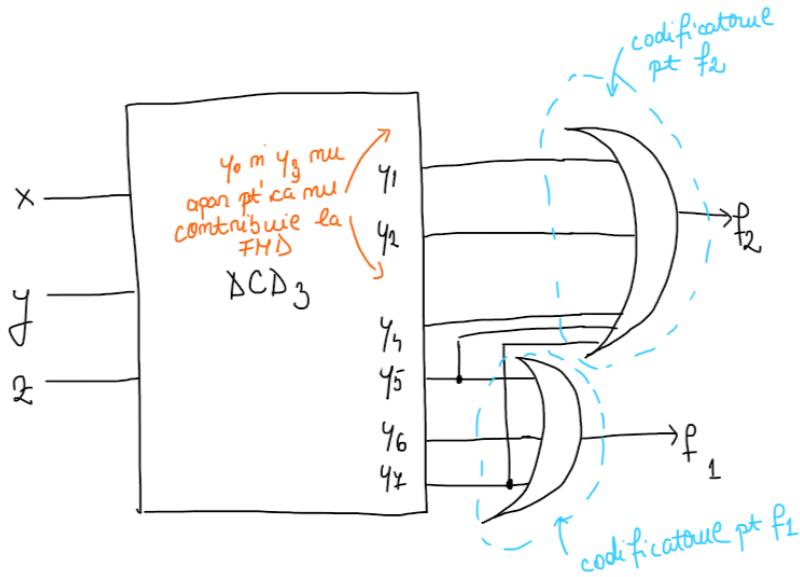


Exemplul 8 Pentru functia de la Exemplul 6, implementati f printr-un codificator.

Reamintim ca functia era: $f : B_2^3 \rightarrow B_2^2$, $f(x, y, z) = (f_1(x, y, z), f_2(x, y, z))$ unde:

- $f_1(x, y, z) = x \cdot (y + z)$
- $f_2(x, y, z) = 1$ d.d. secventa xyz este reprezentarea in baza 2 a unui numar natural care nu este divizibil cu 3.

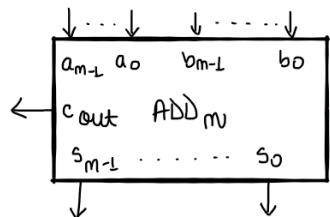
Din tabelul facut la Exemplul 6 vedem ca in FND-ul lui f_1 sunt liniile (5), (6) si (7), iar in FND-ul lui f_2 sunt liniile (1), (2), (4), (5) si (7). Asadar, implementarea cu codificator este:



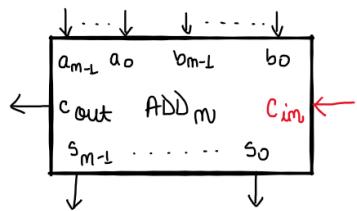
1.6 Sumatori

Un sumator pe n biti $ADD_n, n \geq 1$ este un circuit care implementeaza operatia de adunare pe biti. El primeste ca intrare doua siruri de biti a_{n-1}, \dots, a_0 si b_{n-1}, \dots, b_0 si eventual un transport de intrare c_{in} pentru pozitia 0. Pe aceste input-uri aplica algoritmul de adunare pe n biti. Reamintim ca transportul de pe pozitia $n-1$ se pierde.

Reprezentare:



Iar daca avem si un transport de intrare reprezentarea va fi:



Pentru a face adunari pe mai multi biti folosim sumatori pe un bit:

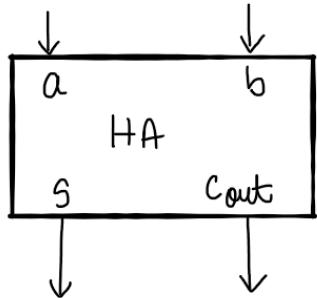
- Half adder: sumatorul pe care il folosim in adunarea celui mai nesemnificativ bit. Automat, acest sumator nu are transport de intrare.
- Full adder: sumotorul pe care il folosim pentru adunarea bitilor cu rang ≥ 1 . Poate avea transport de intrare.

1.6.1 Half adder

Intrare: a, b (doi operanzi de un bit)

Iesire: $s = (a+b) \bmod 2 \Leftrightarrow s = a \oplus b$ si $c_{out} = (a+b) \text{div } 2 \Leftrightarrow c = a \cdot b$

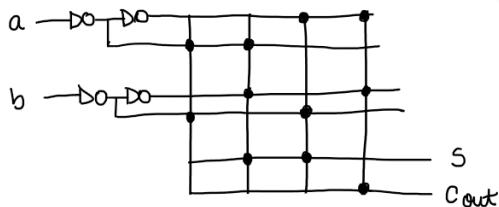
Reprezentarea sa este:



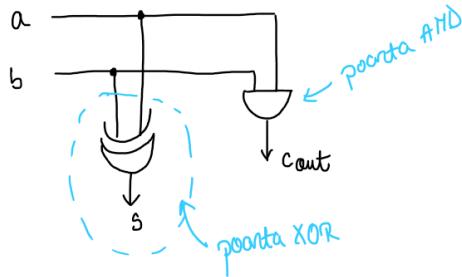
Iar tabelul:

a	b	s	c_{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

PROM-ul unui Half adder arata asa:



Avand in vedere observatia de mai sus ($s = (a+b) \bmod 2 \Leftrightarrow s = a \oplus b$ si $c_{out} = (a+b) \text{div } 2 \Leftrightarrow c = a \cdot b$) putem construi o schema cu numar minim de porti pentru s si c_{out} astfel:

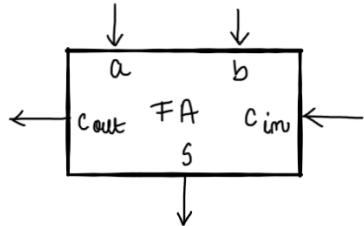


1.6.2 Full adder

Intrare: a, b (operanzi pe un bit), c_{in} (tot pe un bit)

Iesire: $s = (a+b+c_{in}) \bmod 2$, $c_{out} = (a+b+c_{in}) \text{div } 2$.

Reprezentarea unui Full adder este:



Iar tabelul:

a	b	c_{in}	s	c_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Daca scriem FND-ul lui s avem:

$$\begin{aligned}
 FA_s(a, b, c) &= \bar{a} \cdot \bar{b} \cdot c + \bar{a} \cdot b \cdot \bar{c} + a \cdot \bar{b} \cdot \bar{c} + a \cdot b \cdot c = \\
 &= c(\bar{a} \cdot \bar{b} + a \cdot b) + \bar{c}(\bar{a} \cdot b + a \cdot \bar{b}) =
 \end{aligned}$$

Stim că $a \cdot \bar{a} = 0$ și $b \cdot \bar{b} = 0 \Rightarrow$

putem să adăugăm la două numără $\bar{a} \cdot a, \bar{b} \cdot b$ fără probleme

$$= c(\underbrace{\bar{a} \cdot \bar{b} + a \cdot \bar{a} + \bar{b} \cdot b + ab}_A) + \bar{c}(\underbrace{\bar{a} \cdot b + a \cdot \bar{b}}_B) =$$

Stim că $a \oplus b = a \cdot \bar{b} + b \cdot \bar{a} = B$ și $\overline{a \oplus b} = \overline{a \cdot \bar{b} + b \cdot \bar{a}} = D$ de mernită

$$\begin{aligned}
 &= \overline{(a \cdot \bar{b}) + (b \cdot \bar{a})} = \\
 &= (\bar{a} + b) \cdot (\bar{b} + a) = \\
 &= \bar{a} \cdot \bar{b} + \bar{a} \cdot a + b \cdot \bar{b} + ab = A
 \end{aligned}$$

$$\Rightarrow FA_s(a, b, c) = c(\underbrace{\overline{a \oplus b}}_D) + \bar{c}(\underbrace{a \oplus b}_D) =$$

$$\begin{aligned}
 &= c \cdot \bar{D} + \bar{c} \cdot D = c \oplus D = \\
 &= c \oplus (a \oplus b)
 \end{aligned}$$

Iar pentru c cand scriem FND-ul avem:

$$\begin{aligned} FA_C &= (\bar{a} \cdot b \cdot c) + (a \cdot \bar{b} \cdot c) + (a \cdot b \cdot \bar{c}) + (a \cdot b \cdot c) = \\ &= (\underbrace{\bar{a} \cdot b + a \cdot \bar{b}}_{a \oplus b}) \cdot c + a \cdot b (\underbrace{c + \bar{c}}_1) = \end{aligned}$$

$$= (\underbrace{a \oplus b}_{A} \cdot c) + \underbrace{a \cdot b}_{B}$$

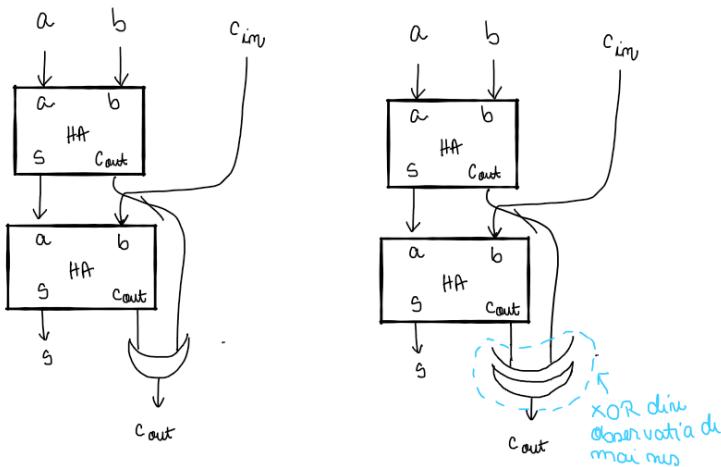
Stim ca $HA_S(a,b) = a \oplus b$
 $HA_c(a,b) = a \cdot b$

$$\begin{aligned} A &= (a \oplus b) \cdot c = HA_S(a,b) \cdot c = \\ &= HA_c(HA_S(a,b) \cdot c) \end{aligned}$$

$$B = a \cdot b = HA_c(a,b)$$

$$FA_C(a,b,c) = HA_c(HA_S(a,b),c) + HA_c(a,b)$$

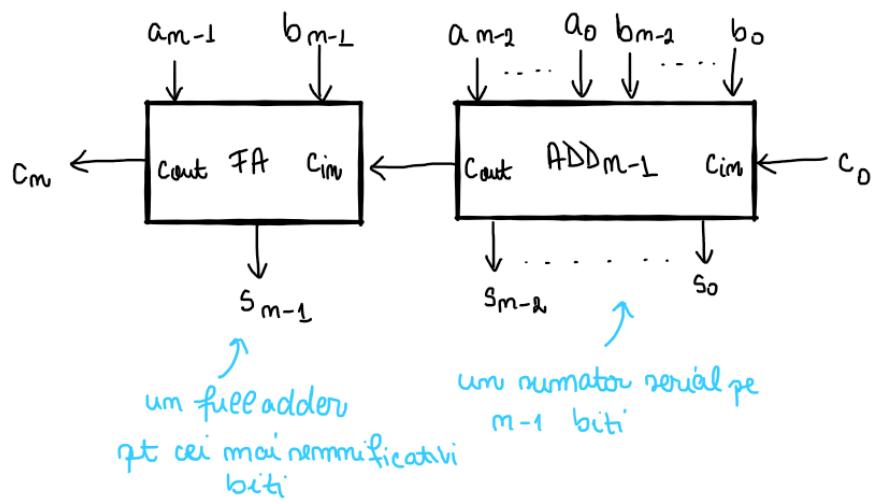
Dragulici spune in materialul lui ca $HA_c(HA_S(a,b),c)$ si $HA_c(a,b)$ nu pot fi simultan 1 (si o sa il credem pe cuvant) deci acel plus dintre ele poate fi inlocuit cu un XOR. Astfel, putem folosi 2 Half addere pentru a construi un Full adder:



1.6.3 Sumator serial

Un sumator serial pe n biti ADD_n calculeaza bitii sumei succesiv, de la cel de rang minim la cel de rang maxim, folosind la calculul fiecarui nou bit transportul obtinut la bitul anterior.

Recursiv, putem spune ca ADD_1 este un FA, iar schema pentru ADD_n este:



2 Proceduri MIPS (conform standardelor MIPS și C)

PROCEDURI IN MIPS



AI CONTROL DEPLIN ASUPRA STIVEI

PROCEDURI IN PYTHON



NU POTI AVEA VALORI
IMPLICITE PENTRU
ARGUMENTELE CARE SUNT IN
FATA ARGUMENTELOR CARE NU
AU VALORI IMPLICITE

2.1 Regiștri

- \$s0-\$s7 - Regiștri salvați (îi vom folosi ca pe niște variable locale. Dacă o procedură folosește un registru s, ea trebuie prima oară să pună pe stivă valoarea inițială din registru, apoi să atribuie regisrului valoarea cu care se vrea să se lucreze, iar apoi la sfârșit să se restituie valoarea inițială a regisrului pentru a putea simula proprietățile unei "variable locale")
- \$sp - Stack pointer, este un registru care ține adresa de memorie a vârfului stivei.
- \$fp - Frame pointer (este un registru care ține un pointer către partea de început a stivei în cadrul nostru de apel)
- \$ra - Return address (acest registru ne va ajuta să ieșim din cadrul de apel direct direct la linia de unde a fost "apelată" procedura)

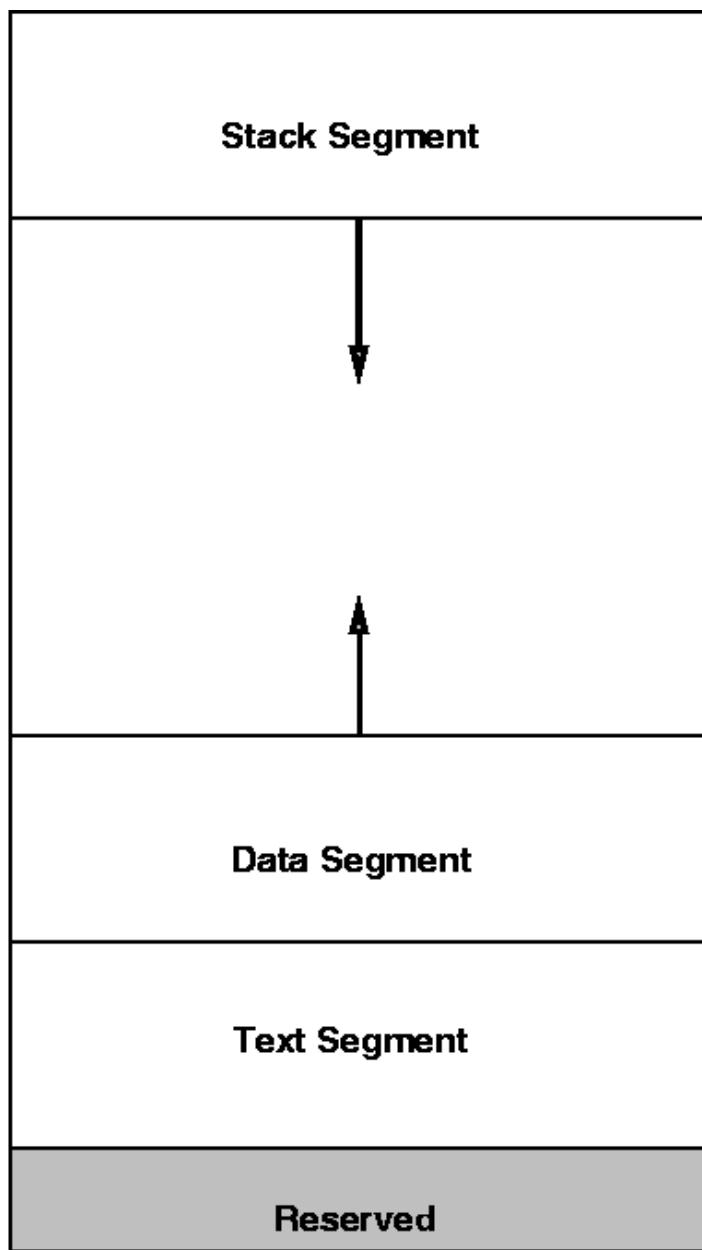
2.2 Instrucțiuni folosite

- jal et - jump and link, va face un jump către eticheta *et* din cod și va pune în \$ra un pointer către adresa de memorie a liniei imediat următoare (adică linia care urmează după instrucțiunea jal).
- jr - jump to register, având o adresă de memorie a unei poziții din program ținută într-un registru, putem da jump la linia respectiva folosind această instrucțiune. De regulă vom folosi construcția *jr \$ra* pentru a ieși din proceduri.

2.3 Operații pe stivă

0x7fffff

0x400000



Priviți imaginea de mai sus. Acel ”Stack Segment” este zona de memorie unde avem noi stiva. În acea zonă de memorie putem pune date, și le putem scoate. Observați cum în figură stiva arată de parcă se continuă în jos. Așa și este în realitate. Pe o stivă avem doar două operații: push și pop. Push pune o valoare în vârful din partea de jos a stivei și o mărește în jos, iar pop scoate cea mai de jos valoare și micșorează stiva.

Inițial în program, stiva noastră va fi goală deci noi vom avea doar un pointer care ”duce” spre vârful din sus al stivei, iar noi pentru a o umple, vom scădea din acel pointer mărimea datelor pe care vrem să le punem și vom pune valori la pointerul curent către vârful stivei. Pointerul pe care îl vom folosi este \$sp, acesta fiind initializat implicit cu vârful stivei (valoare foarte mare)

Exemple:

Pentru a pune valoarea din registrul \$t0 pe stivă putem face:

```
subu $sp, 4 # decrementează vârful stivei  
sw $t0, 0($sp) # pune valoarea lui $t0 în vârful stivei
```

După această operație, registrul \$sp va pointa către stiva noastră, care va arăta aşa: (\$t0). Adică pe stivă se va afla doar valoarea din registrul \$t0.

Dacă acum dormi să punem și valoarea din registrul \$t1 pe stivă, putem face:

```
subu $sp, 4  
sw $t1, 0($sp)
```

Acum stiva noastră va arăta aşa:
(\$t1), (\$t0)

Adică pe stivă se află valorile din \$t1 și \$t0.

Pentru a face pop este suficient să scoatem elementul din vârful stivei și să-l salvăm într-un registru (dacă avem nevoie de el, dacă nu, putem sări acest pas), iar apoi să incrementăm \$sp cu dimensiunea elementului scos.

Pentru a scoate valorile de pe stiva anterioară putem face:

```
lw $t0, 0($sp) # scoate elementul din stivă și îl copiază în registrul $t0.  
addu $sp, 4 # incrementează vârful stivei
```

Observație:

Dacă nu ne dorim să păstrăm valorile din stivă când dăm pop, este suficient să incrementăm \$sp cu dimensiunea dorită.

2.4 Convenții

- O procedură primește argumentele prin stivă.
- O procedură returnează rezultatele fie prin regiștri \$v0, \$v1, fie prin vârful stivei. Noi vom vedea ambele moduri de lucru întrucât la unele grupe la laborator se returnează prin regiștri, iar la alte grupe se returnează prin vârful stivei.

2.5 Exerciții

Problema 1: Să se implementeze suma a două numere date în memorie utilizând o procedură Suma(x, y). Procedura va returna rezultatul prin registrul \$v0.

```
1 .data  
2     x:.word 3  
3     y:.word 6  
4 .text  
5  
6 suma:  
7     subu $sp, 4      # pun frame pointerul pe stiva  
8     sw $fp, 0($sp)    # aceste două linii de la inceputul procedurii
```

```

10      # trebuie sa le scrieti la inceputul fiecarei proceduri
11      # aceasta este voia maestrului
12      # acum stiva arata asa: ($fp), (x), (y)
13
14      addi $fp, $sp, 4 # face ca $fp sa pointeze la inceputul cadrului de apel
15      # adica ($fp), <fp_pointeaza_aici> (x), (y)
16
17      subu $sp, 4 # pun s0 pe stiv
18      sw $s0, 0($sp) # acum stiva arata asa ($s0), ($fp), (x), (y)
19
20      subu $sp, 4 # pun $s1 pe stiv
21      sw $s1, 0($sp) # acum stiv arata asa ($s1), ($s0), ($fp), (x), (y)
22
23          # Motivul pentru care punem fp, s0 i s1 pe stiv este
24          # deoarece noi le vom considera ca pe nite variabile locale,
25          # dar ideea este ca si alte proceduri le consider tot ca pe niste
26          # variabile locale, deci in cadrul nostru de apel trebuie sa aiba
27          # unele
28          # valori, iar in alte cadre de apel trebuie sa aib alte valori.
29          # De aceea noi cand intram in procedura salvam pe stiva valorile cu
30          # care au venit
31          # iar cand terminam proocedura, rstituim valorile bune pentru ca
32          # ace ti registri sa aiba valorile bune in cadrul
33          # celorlalte proceduri, ci nu valorile pe care l-am folosit
34          # in procedura noastra
35
36      lw $s0, 0($fp) # incarc in s0 prima valoare catre care pointeaza
37      # frame pointer-ul nostru, adica x
38      lw $s1, 4($fp) # incarc in s1 a doua valoare catre care pointeaza
39      # frame pointer-ul nostru, adica y
40      # tinem minte ca fp pointeaza catre (x), (y)
41
42      add $v0, $s0, $s1 # facem adunarea si punem rezultatul
43      # in registrul $v0
44
45      lw $s1, -12 ($fp) # restitui $s1, asa cum am vazut mai devreme
46      # ca trebuie sa facem
47      lw $s0, -8 ($fp) # analog, restitui $s0
48      lw $fp, -4 ($fp) # analog, restitui $fp
49
50      addu $sp, 12 # acum ca am restituit $s1, $s1 si $fp
51      # trebuie sa le dau pop de pe stiva
52
53      jr $ra # acum ma intor in main
54      # la linia urmatoare apelului
55      # procedurii
56
57 main:
58
59      lw $t0, y
60      subu $sp, 4
61      sw $t0, 0($sp) # il pun pe y pe stiva
62      # acum stiva arata asa: (y)
63
64      lw $t0, x
65      subu $sp, 4
66      sw $t0, 0($sp) # il pun pe x pe stiva
67      # acum stiva arata asa (x), (y)
68
69      jal suma # apelez procedura care face suma
70
71      addu $sp, 8 # dau pop la x si y de pe stiva
72      # din moment ce nu imi trebuie valorile lor
73      # nu le mai salvez
74
75      move $a0, $v0 # afisez pe ecran rezultatul
76      # (pe care procedura) mi l-a returnat
77      # prin registrul $v0

```

```

76      li $v0, 1
77      syscall
78
79      li $v0, 10
80      syscall

```

problema1.s

Problema 2: Să se implementeze suma a două numere date în memorie utilizând o procedură Suma(x, y). Procedura va returna rezultatul vârfului stivei.

```

1 .data
2     x:.word 3
3     y:.word 6
4 .text
5
6 suma:
7
8     subu $sp, 4
9     sw $fp, 0($sp)
10
11    addi $fp, $sp, 4
12
13    subu $sp, 4
14    sw $s0, 0($sp)
15
16    subu $sp, 4
17    sw $s1, 0($sp)
18
19    lw $s0, 0($fp)
20    lw $s1, 4($fp)
21
22    add $s0, $s0, $s1 # acum în loc să salvez rezultatul în $v0
23          # îl salvez tot în $s0
24
25    sw $s0, 0($fp)    # pun valoarea returnată în capatul stivei
26          # deci stiva mea de acum se va transforma
27          # din (s1), (s0), (fp), (x), (y)
28          # în (s1), (s0), (fp), (rezultat), (y)
29          # iar după ce vom scoate (s1), (s0) și (fp)
30          # vor ramane doar (rezultat), (y)
31
32    lw $s1, -12 ($fp)
33    lw $s0, -8 ($fp)
34    lw $fp, -4 ($fp)
35
36    addu $sp, 12
37          # acum stiva noastră este (rezultat) (y)
38
39    jr $ra
40
41 main:
42
43    lw $t0, y
44    subu $sp, 4
45    sw $t0, 0($sp)
46
47    lw $t0, x
48    subu $sp, 4
49    sw $t0, 0($sp)
50
51    jal suma
52
53    lw $t0, 0($sp)    # luăm rezultatul din vârful stivei
54    addu $sp, 8        # dam pop și rezultatului, și lui y de pe stiva
55
56    move $a0, $t0      # afisam pe ecran rezultatul

```

```

57      li $v0, 1
58      syscall
59
60      li $v0, 10
61      syscall

```

problema2.s

2.6 Mai multe exerciții

Problema 1: Să se scrie o procedură care decide dacă un număr este perfect. Numim că un număr este perfect, dacă el este egal cu suma tuturor divizorilor lui (mai puțin el însuși).

Exemple:

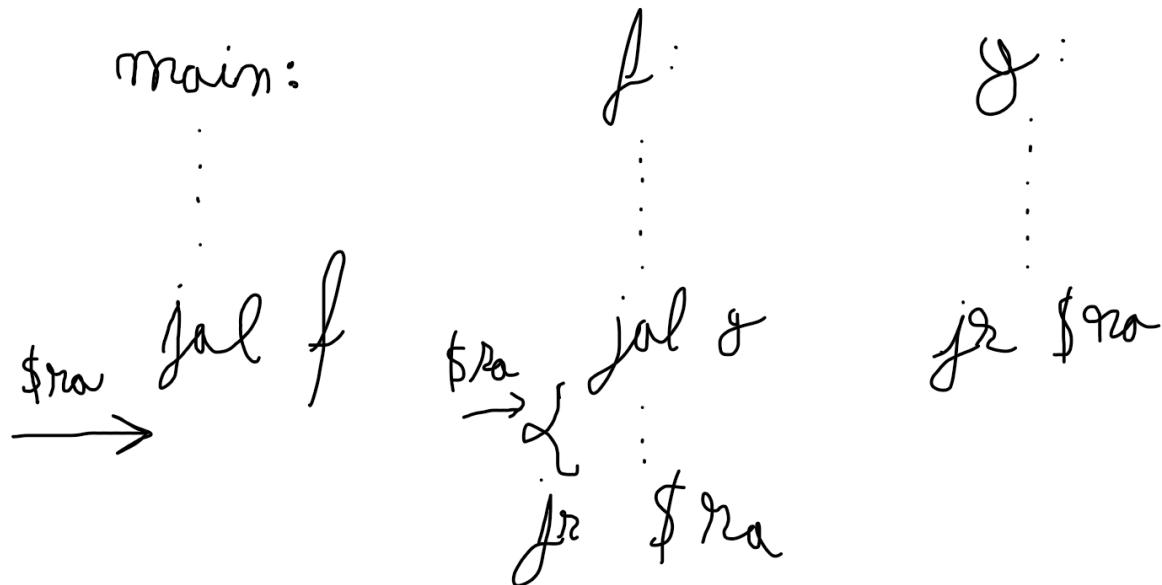
6 este perfect deoarece $6 = 1 + 2 + 3$

28 este perfect deoarece $28 = 1 + 2 + 4 + 7 + 14$

2.7 Apeluri imbricate

Dacă avem o procedură f care apelează o procedură g, numim acesta apel imbricat. Dacă am face acest lucru, folosind convențiile de mai sus, atunci, când procedura f ar apela procedura g, ar face ca \$ra-ul din f să să poarte tot în f, în loc să poarte către main, unde am vrea să continuăm executarea programului după ce se termină procedura f.

Ca să rezolvăm asta, pur și simplu vom pune și vom scoate \$ra de pe stivă la începutul/ sfârșitul procedurii, exact cum facem și cu \$fp.



2.8 Exerciții

Problema 1: Se dă un număr x în memorie și două funcții f și g astfel: $g(x) = x + 1$, $f(x) = 2g(x)$. Să se interpreteze ca proceduri funcțiile f și g și să se afișeze pe ecran f(x).

```

1 .data
2     x:.word 7
3 .text
4

```

```

5   g:
6     subu $sp, 4
7     sw $fp, 0($sp)
8
9     addi $fp, $sp, 4
10
11    subu $sp, 4
12    sw $ra, 0($sp)  # punem $ra pe stiva
13
14    subu $sp, 4
15    sw $s0, 0($sp)
16
17    lw $s0, 0($fp)
18
19    addi $v0, $s0, 1
20
21    lw $s0, -12($fp)
22    lw $ra, -8($fp)  # scoatem $ra de pe stiva
23    lw $fp, -4($fp)
24
25    addu $sp, 12
26
27    jr $ra
28
29 f:
30   subu $sp, 4
31   sw $fp, 0($sp)
32
33   addi $fp, $sp, 4
34
35   subu $sp, 4      # punem $ra pe stiva
36   sw $ra, 0($sp)  # teoretic, doar in procedura f
37           # trebuie sa punem si sa scoatem
38           # $ra de pe stiva, dar pentru consistenta
39           # facem asta si in procedura g
40
41   subu $sp, 4
42   sw $s0, 0($sp)
43
44   lw $s0, 0($fp)
45
46   subu $sp, 4
47   sw $s0, 0($sp)
48
49   jal g
50
51   addu $sp, 4
52
53   add $v0, $v0, $v0
54
55   lw $s0, -12($fp)
56   lw $ra, -8($fp)  # scoatem $ra de pe stiva, pentru
57           # a ne putea intoarce inapoi in main
58           # daca nu faceam asta, atunci la jr $ra
59           # ne-am fi intors inapoi dupa jal g
60   lw $fp, -4($fp)
61
62   addu $sp, 12
63
64   jr $ra
65
66 main:
67   lw $t0, x
68   subu $sp, 4
69   sw $t0, 0($sp)
70
71   jal f
72

```

```

73      addu $sp, 4
74
75      move $a0, $v0
76      li $v0, 1
77      syscall
78
79      li $v0, 10
80      syscall

```

problema1_1.s

2.9 Proceduri recursive

Acum că am văzut cum putem apela proceduri din alte proceduri, putem face în același mod și pentru a avea proceduri recursive. Adică proceduri care se apelează pe ele însese.

2.10 Exerciții

Problema 1: Se dă un număr natural n în memorie. Să se calculeze $n!$ folosind o procedură recursivă.

```

1 .data
2     n:.word 5
3 .text
4
5 fact:
6
7     subu $sp, 4
8     sw $fp, 0($sp)
9
10    addi $fp, $sp, 4
11
12    subu $sp, 4
13    sw $ra, 0($sp)
14
15    subu $sp, 4
16    sw $s0, 0($sp)
17
18    lw $s0, 0($fp)
19
20    ble $s0, 1, cond # daca numarul pentru care a fost
21                      # apelata procedura este mai mic sau egal cu 1
22                      # va returna 1
23
24    subu $s0, 1
25
26    subu $sp, 4      # calculeaza fact(n-1)
27    sw $s0, 0($sp)
28
29    jal fact
30    addu $sp, 4
31
32    addu $s0, 1
33    mul $v0, $v0, $s0 # returneaza in v0
34                      # valoarea n * fact(n-1)
35
36    j exit           # nu returnam 1 pe cazul general
37                      # doar daca avem parametru mai mic sau egal cu 1
38 cond:
39     li $v0, 1
40 exit:
41
42     lw $s0, -12($fp)
43     lw $ra, -8($fp)
44     lw $fp, -4($fp)
45
46     addu $sp, 12

```

```

47      jr  $ra
48
49 main:
50
51     lw  $t0 , n
52     subu $sp , 4
53     sw  $t0 , 0($sp)
54
55     jal fact
56
57     addu $sp , 4
58
59     move $a0 , $v0
60     li   $v0 , 1
61     syscall
62
63     li   $v0 , 10
64     syscall

```

problema1_2.s

2.11 Mai multe exerciții

Problema 1: Se citește de la tastatură un număr $n \in N^*$. Să se găsească al n -ulea număr din sirul lui Fibonacci folosind o procedură recursivă.

2.12 Proceduri pentru array-uri

Pentru a transmite un array la o procedură, de regulă se transmite adresa de memorie unde începe vectorul și lungimea acestuia. Având adresa lui de memorie și lungimea lui, poate fi parcurs cu ușurință.

2.13 Exerciții

Problema 1: Se dă un array stocat în memorie și lungimea acestuia, să se afișeze array-ul pe ecran folosind o procedură.

```

1 .data
2     v:.word 5, 13, 27, 3, 11, 29
3     n:.word 6
4     ch:.byte ' '
5 .text
6
7 afis:
8
9     subu $sp , 4
10    sw  $fp , 0($sp)
11
12    addi $fp , $sp , 4
13
14    subu $sp , 4
15    sw  $s0 , 0($sp)
16
17    subu $sp , 4
18    sw  $s1 , 0($sp)
19
20    lw  $s0 , 0($fp) # iau adresa de memorie a vectorului
21          # de pe stiva
22    lw  $s1 , 4($fp) # iau si lungimea lui de pe stiva
23    li   $t0 , 0       # vom folosi $t0 aici
24          # puteam foarte bine sa folosim si $s2 in loc
25          # dar pentru asta trebuie sa il punem pe stiva
26          # si sa il scoatem la sfarsit
27          # pe $t-uri nu avem astfel de conventii
28

```

```

29      loop:           # parcurg vectorul normal
30          bge $t0, $s1, exit
31
32          lw $a0, 0($s0)
33          li $v0, 1
34          syscall
35
36          lb $a0, ch
37          li $v0, 11
38          syscall
39
40          addu $s0, 4 # incrementez adresa de memorie
41              # curenta a vectorului
42              # pentru a trece la urmatorul
43              # element
44          addu $t0, 1
45
46          j loop
47
48      exit:
49
50      lw $s1, -12($fp)
51      lw $s0, -8($fp)
52      lw $fp, -4($fp)
53
54      addu $sp, 12
55
56      j $ra
57
58 main:
59
60      lw $t0, n      # pun pe stiva lungimea array-ului
61      subu $sp, 4
62      sw $t0, 0($sp)
63
64      la $t0, v      # pun pe stiva adresa de memorie
65      subu $sp, 4    # de unde incepe array-ul
66      sw $t0, 0($sp)
67
68      jal afis       # fac afis(v, n), unde v
69          # este adresa de memorie unde incepe v
70
71      addu $sp, 8
72
73      li $v0, 10
74      syscall

```

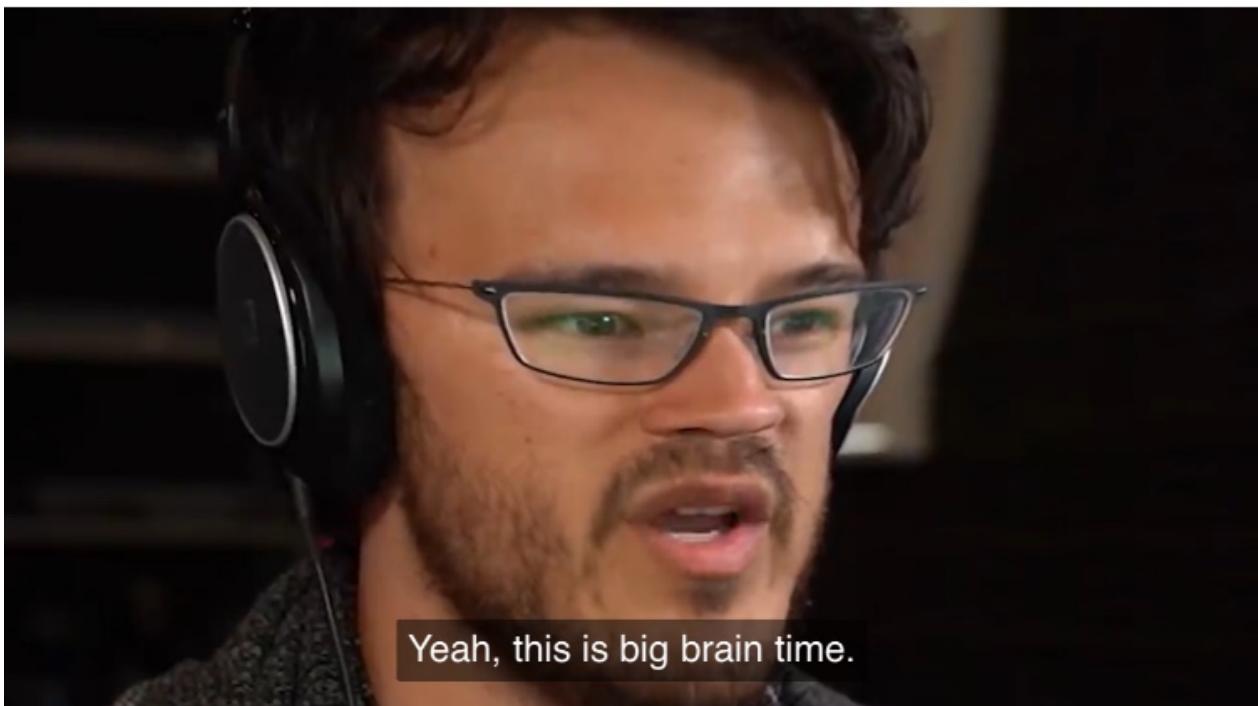
problema1_3.s

2.14 Mai multe exerciții

Problema 1: Se dă un array stocat în memorie și lungimea acestuia, să se afișeze array-ul pe ecran folosind o procedură recursivă.

2.15 Array-uri de proceduri

TUTORELE DE MIPS CAND
ESTE NEVOIT SA PREDEA
CEVA CE EL NU A INVATAT
LA LABORATOR.



După cum am văzut, pentru a apela o procedură este nevoie de label-ul ei. Acest label de fapt, este un fel de macro pentru adresa de memorie a instrucțiunii imediat următoare. Am văzut și că folosind jr putem ” sări” la o adresă de memorie ținută într-un registru.

Deci am putea să tine procedurile ca pe niște ”variabile” în registri, iar apoi să le ”apelăm”. Singura problemă în acest caz, este că \$ra nu va pointa către adresa de memorie următoare apelului. Deci înainte de a sări cu jr, este nevoie să punem în \$ra adresa la care vrem să ne întoarcem după apelul procedurii.

2.16 Exerciții

Problema 1: Translația în MIPS următorul program C:

```
1 int aplica(int (*f)(int), int x) {  
2     return (*f)(x);  
3 }  
4 int f1(int y) {return y+y;}
```

```

6 int f2( int y) {return y*y;}
7 int f3( int y) {return -y;}
8
9 int (*vf [ ]) = {f1 , f2 , f3 }, v[3];
10
11 void main() {
12     register int i;
13     for ( i=0;i<3;++i)    v[ i]=aplica(vf[ i],1+i);
14 }
15
16 /* in final v[0]=2 , v[1]=4 , v[2]=-3 */

```

```

1 .data
2
3     vf:.space 12
4     v:.space 12
5
6 .text
7
8 applica:
9     subu $sp, 4
10    sw $fp, 0($sp)
11    addi $fp, $sp, 4
12    subu $sp, 4
13    sw $ra, 0($sp)
14    subu $sp, 4
15    sw $s0, 0($sp)
16    subu $sp, 4
17    sw $s1, 0($sp)
18
19    lw $s0, 0($fp)      # adresa de memorie a procedurii
20    lw $s1, 4($fp)       # argumentul pe care il vom da procedurii
21
22    subu $sp, 4          # punem argumentul pe stiva
23    sw $s1, 0($sp)
24    la $ra, cont_apl    # lui $ra ii punem ca valoare adresa de memorie
25    # din program unde vor reveni procedurile f1 , f2 , f3
26    # dupa ce se executa
27    jr $s0                # apelam procedura daca ca argument
28
29 cont_apl:             # aici ajungem dupa ce se executa procedura
30
31    addu $sp, 4
32
33    lw $s1, -16($fp)
34    lw $s0, -12($fp)
35    lw $ra, -8($fp)
36    lw $fp, -4($fp)
37
38    addu $sp, 16
39
40    j $ra
41
42 f1:
43    subu $sp, 4
44    sw $fp, 0($sp)
45    addi $fp, $sp, 4
46    subu $sp, 4
47    sw $ra, 0($sp)
48    subu $sp, 4
49    sw $s0, 0($sp)
50    lw $s0, 0($fp)
51    add $v0, $s0, $s0
52    lw $s0, -12($fp)
53    lw $ra, -8 ($fp)
54    lw $fp, -4 ($fp)

```

```

55      addu $sp, 12
56      jr $ra
57
58 f2 :
59      subu $sp, 4
60      sw $fp, 0($sp)
61      addi $fp, $sp, 4
62      subu $sp, 4
63      sw $ra, 0($sp)
64      subu $sp, 4
65      sw $s0, 0($sp)
66      lw $s0, 0($fp)
67      mul $v0, $s0, $s0
68      lw $s0, -12($fp)
69      lw $ra, -8 ($fp)
70      lw $fp, -4 ($fp)
71      addu $sp, 12
72      jr $ra
73
74 f3 :
75      subu $sp, 4
76      sw $fp, 0($sp)
77      addi $fp, $sp, 4
78      subu $sp, 4
79      sw $ra, 0($sp)
80      subu $sp, 4
81      sw $s0, 0($sp)
82      lw $s0, 0($fp)
83      subu $v0, $zero, $s0
84      lw $s0, -12($fp)
85      lw $ra, -8 ($fp)
86      lw $fp, -4 ($fp)
87      addu $sp, 12
88      jr $ra
89
90 main :
91
92     la $t0, vf
93
94     la $t1, f1
95     sw $t1, 0($t0)      # punem procedura f1 in array
96
97     la $t1, f2
98     sw $t1, 4($t0)      # punem procedura f2 in array
99
100    la $t1, f3
101    sw $t1, 8($t0)      # punem procedura f3 in array
102
103    li $t1, 0
104    li $t2, 3
105
106 loop:
107
108     bge $t1, $t2, exit
109
110     move $t3, $t1
111     add $t3, $t3, $t3
112     add $t3, $t3, $t3
113
114     lw $t3, vf($t3)  # luam procedura de la pozitia i din array
115     move $t4, $t1
116     addi $t4, 1        # calculam valoarea argumentului curent dat procedurii
117
118     subu $sp, 4        # punem argumentul pe stiva
119     sw $t4, 0($sp)
120
121     subu $sp, 4        # punem si adresa procedurii pe stiva
122     sw $t3, 0($sp)

```

```

123
124     jal aplica
125
126     addu $sp, 8
127
128     move $t3, $t1
129     add $t3, $t3, $t3
130     add $t3, $t3, $t3
131
132     sw $v0, v($t3)    # punem in array-ul v valoarea rezultata
133
134             # am comentat afisarea deoarece
135             # cerinta nu cere sa si afisam rezultatele
136             # pe ecran
137     # move $a0, $v0
138     # li $v0, 1
139     # syscall
140
141     # li $a0, ,
142     # li $v0, 11
143     # syscall
144
145     addu $t1, 1
146
147     j loop
148
149 exit:
150
151     li $v0, 10
152     syscall

```

problema1_4.s

2.17 Mai multe exerciții

Problema 1: Rezolvați prima problemă din secțiunea anterioară returnând valorile prin capătul stivei. (așa cum vrea maestrul)

Problema 2: Rezolvați prima problemă de la Advent of Code 2020 în MIPS și postați pe subredditul /adventofcode (la momentul scrierii acestui material, încă nu a început Advent of Code 2020)

Problema 3: Rezolvați restul problemelor de la Advent of Code 2020 în Python (și postați pe reddit dacă aveți soluții interesante ale problemelor/ visualizere).



Problema 4: Dați wishlist și follow pe Steam la PalmRide dacă nu ați făcut-o deja.
<https://store.steampowered.com/app/1415320/PalmRide/>

References

- [1] Dumitru Daniel Drăgulici. *Curs Arhitectura Sistemelor de Calcul*.
- [2] Larisa Dumitrache. *Tutoriat 2019*
- [3] Bogdan Macovei. *Laboratoare ASC 2019/ 2020*
- [4] Advent Of Code. *2020*