

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.

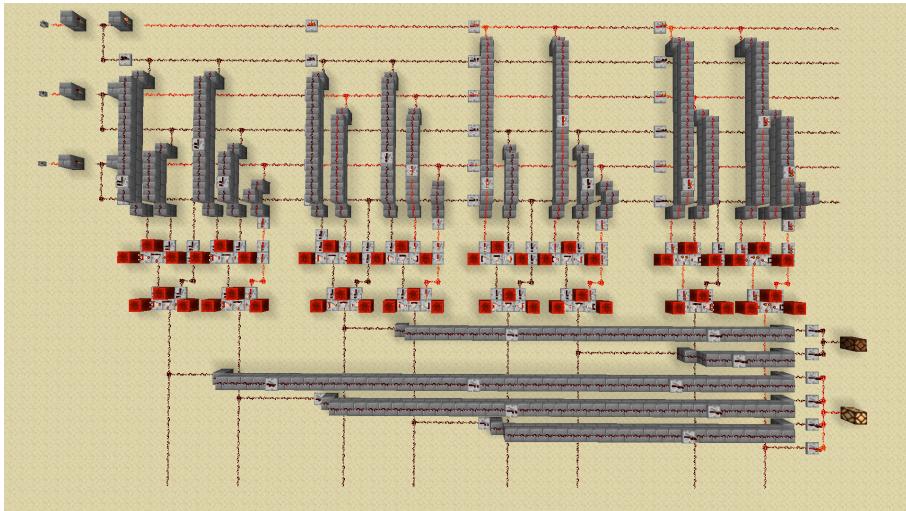
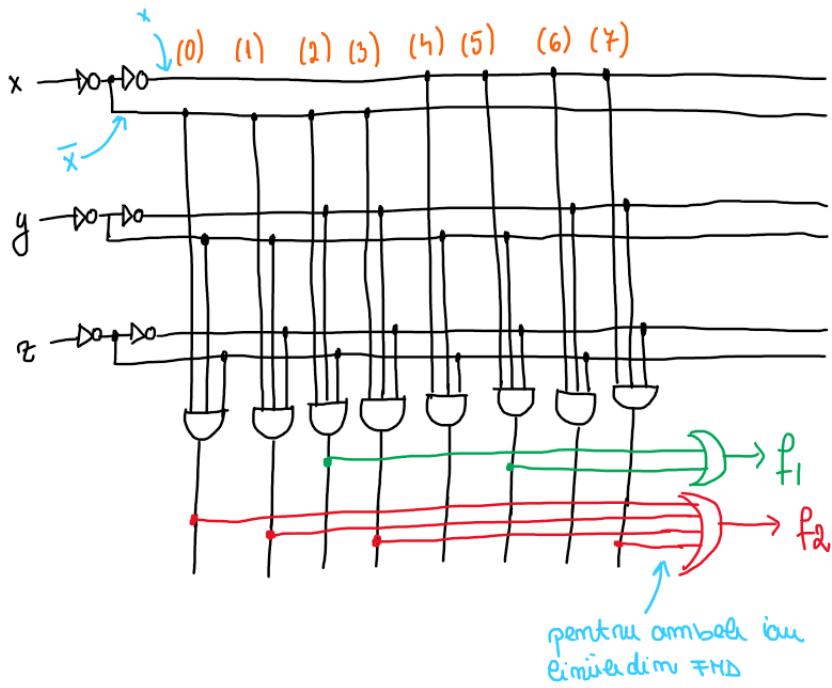
Exemplul 1 : Vom lua functia din tutoriatul 4, de la Exemplul 1, pentru ca ei i-am facut deja tabelul (inainte sa faceti PLA mereu va trebui sa faceti tabelul functiei): $f : B_2^3 \rightarrow B_2^2$, $f(x, y, z) = (f_1(x, y, z), f_2(x, y, z))$ cu

- $f_1(x, y, z) = (x \oplus y)(y \oplus z)$
- $f_2(x, y, z) = 1$ 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.

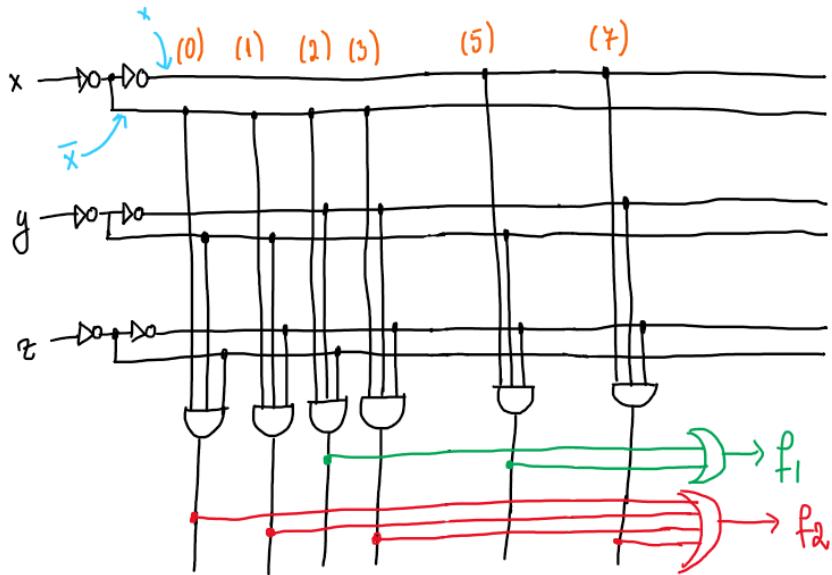


Observatii:

- 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.

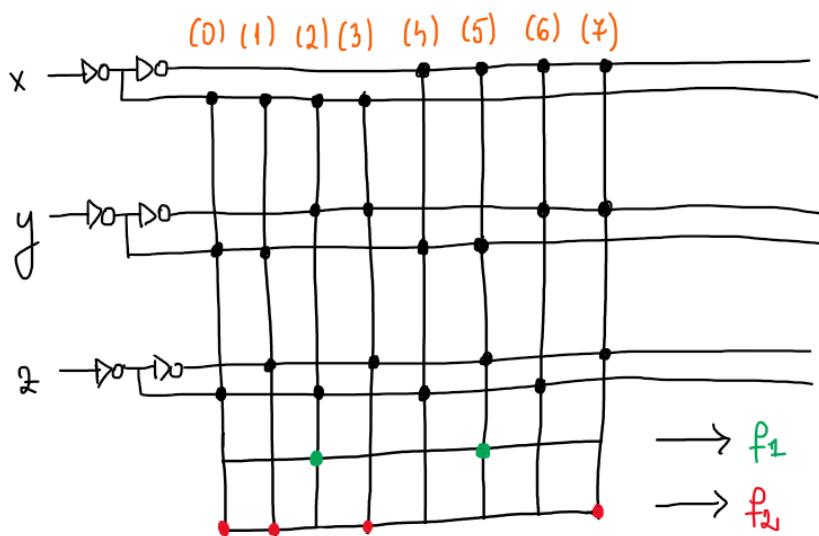
Daca va cere sa scrieti forma PLA cu numar minim de sume/produse, eliminati "liniile" care nu se afla nici in FND-ul lui f_1 , nici in FND-ul lui f_2 . In cazul nostru, eliminam (4) si (6):



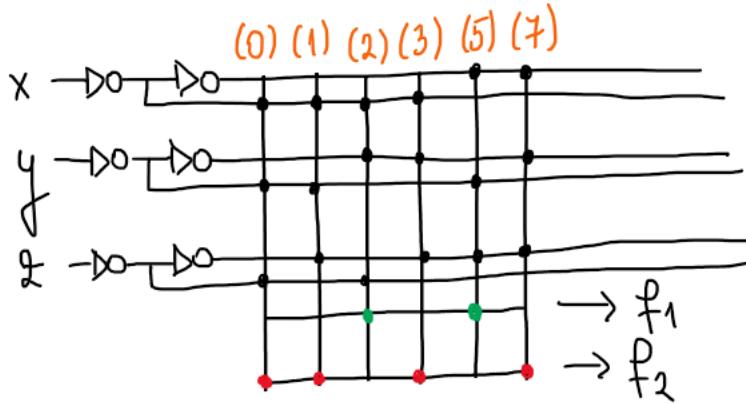
1.2 PROM

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.

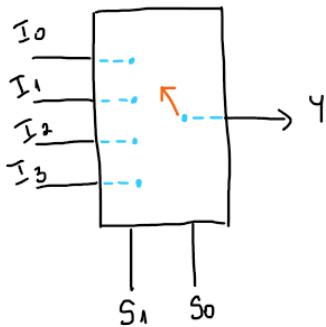


La fel, daca va cere sa scrieti un PROM cu numar nimic de sume/produse, eliminati "liniile" care nu sunt nici un FND-ul lui f_1 nici in FND-ul lui f_2 .



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:

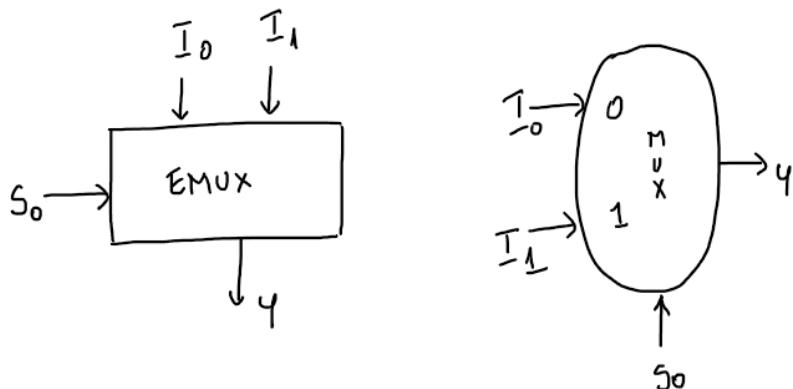


In functie de S_0 si S_1 , el alege una dintre intrari, mutand "comutatorul" pe acea intrare. Daca $S_0 = 0$ si $S_1 = 0$ va alege I_0 . . Daca $S_0 = 1$ si $S_1 = 0$ va alege I_2 .

Observatii:

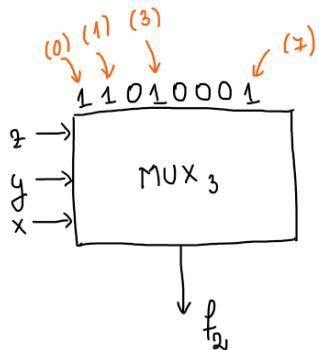
- n din MUX_n este dat de numarul de S -uri care intra in multiplexor.
- Numarul de I -uri care intra in multiplexor este 2^n

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



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

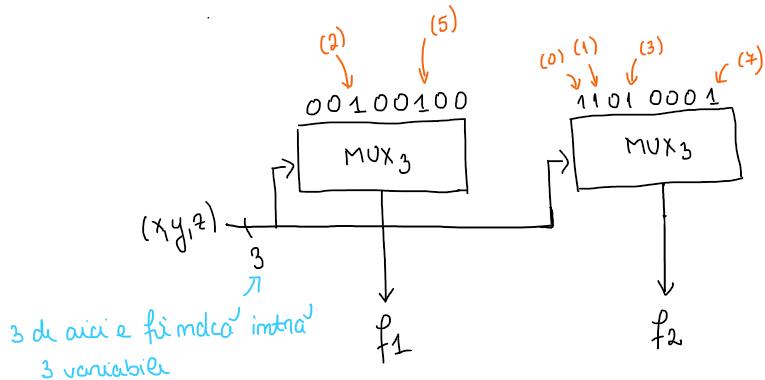
Reamintim ca functia f_2 avea valoarea 1 pentru (0), (1), (3) si (7). Asadar, reprezentarea cu un multiplexor este:



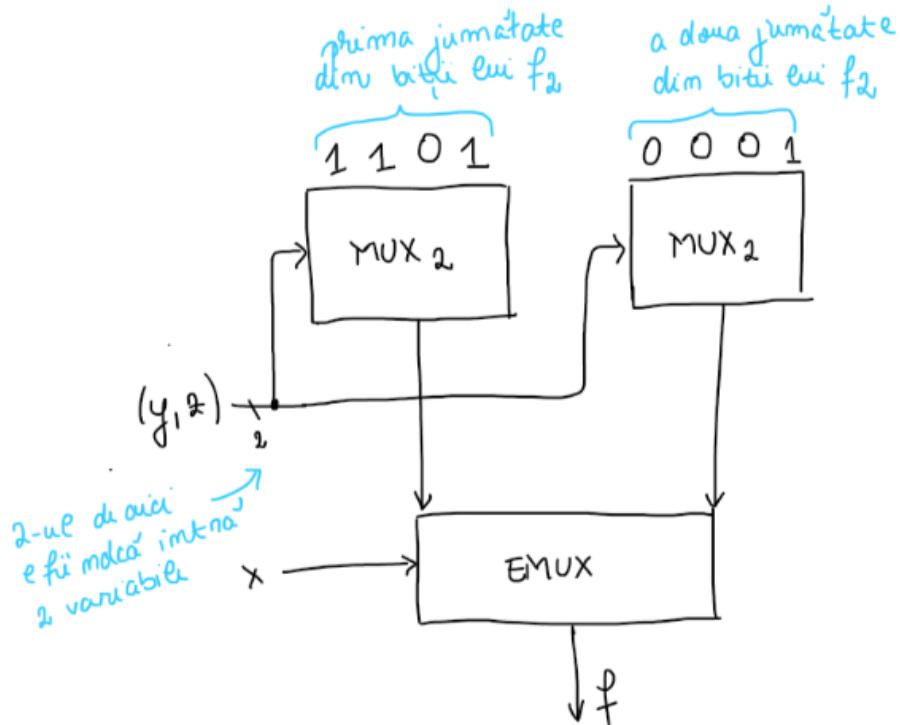
Exemplul 4 [RESTANTA SEPTEMBRIE 2020]

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

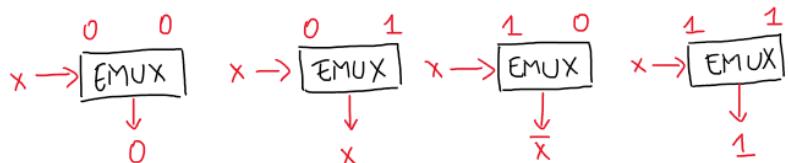
Reamintim ca f_1 avea valoarea 1 pentru (2) si (5).



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. se cunosc xyz este reprezentarea in baza 2 a unui numar natural care nu este divizibil cu 3.

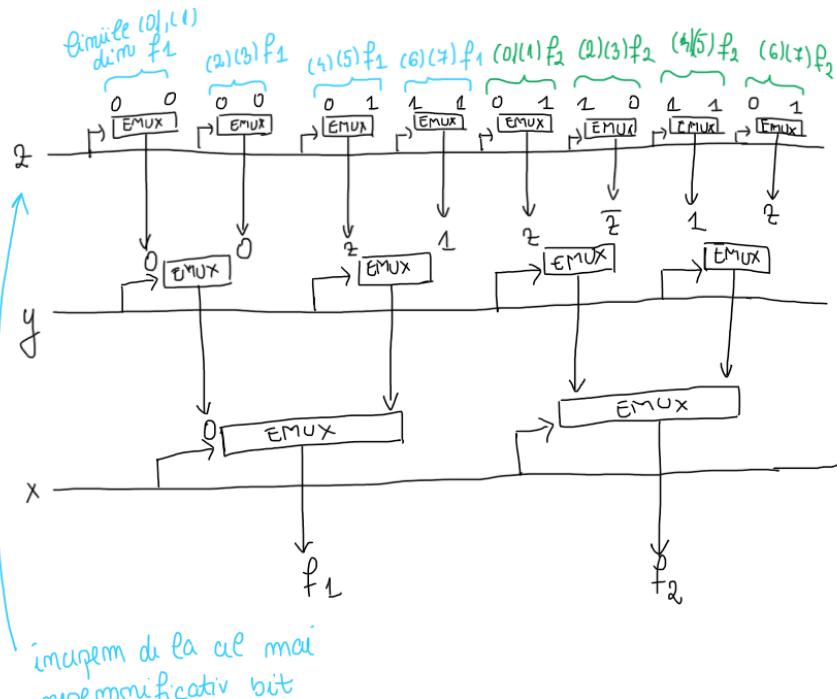
Implementati f printr-un circuit care contine doar multiplexoari elementari; apoi, reduceti la maximum numarul multiplexorilor elementari (nu este permisa adaugarea de porti NOT).

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).

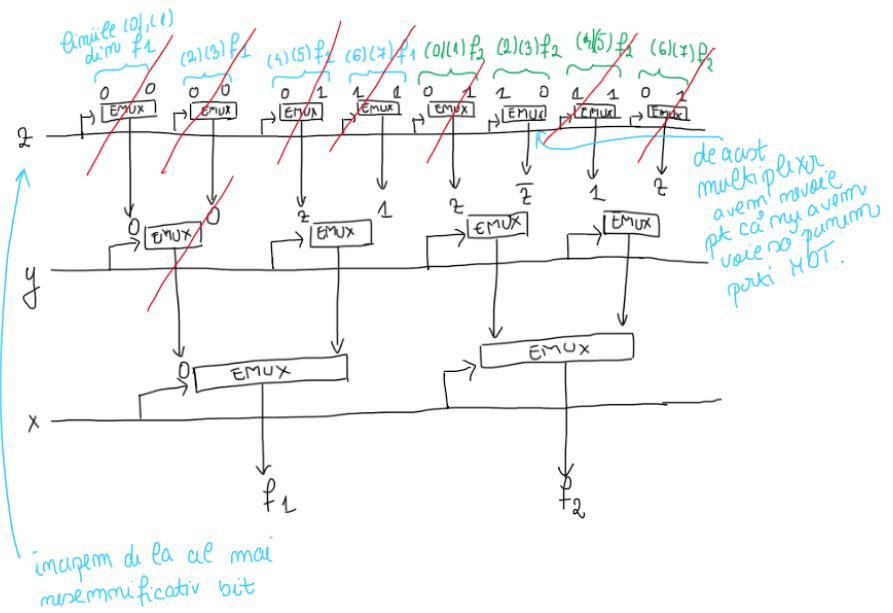
index	x	y	z	$y + z$	$f_1(x, y, z)$	$f_2(x, y, z)$
(0)	0	0	0	0	0	0
(1)	0	0	1	1	0	1
(2)	0	1	0	1	0	1
(3)	0	1	1	1	0	0
(4)	1	0	0	0	0	1
(5)	1	0	1	1	1	1
(6)	1	1	0	1	1	0
(7)	1	1	1	1	1	1

Acum ca avem tabelul, putem sa scriem reprezentarea:

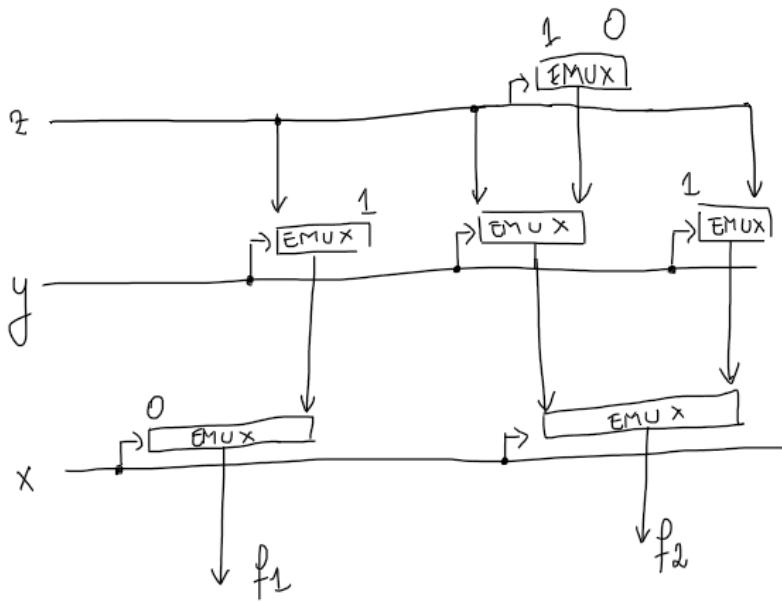


Pentru partea a doua a problemei, reducerea la minim a multiplexorilor elementari, vom elibera toti multiplexorii din careiese un 0, un 1 sau un x, y sau z. Daca aveam voie cu porti NOT puteam sa eliminam si multiplexorii din careiese un \bar{x} , \bar{y} sau \bar{z} .

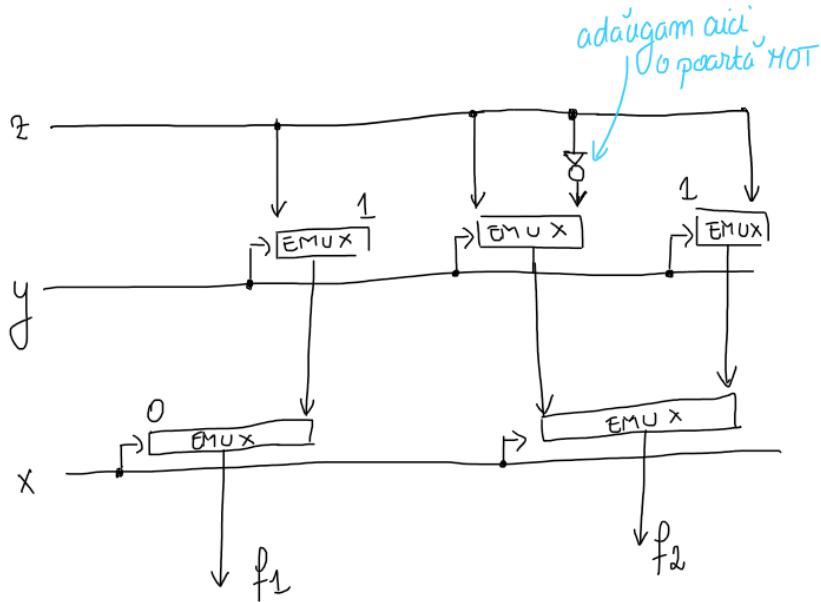
Cand scriem forma finala, e bine sa incepem cu multiplexorii de la x si sa urcam catre z. Asa ne dam seama foarte bine de ce avem nevoie si de ce nu. Taiem cu rosu ce vom elibera:



In final, minimizarea va fi:



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

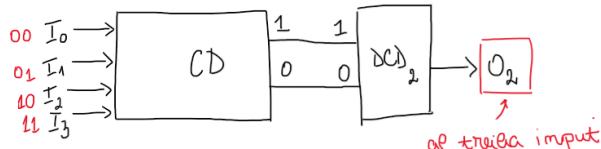


1.5 Decodificatori si codificatori

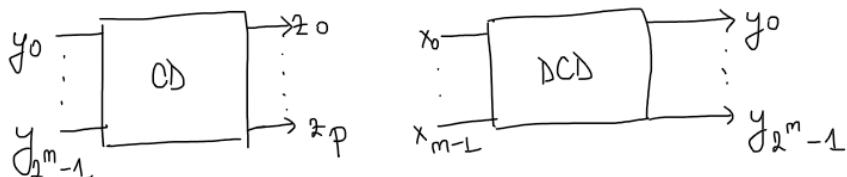
Un decodificator cu selector pe n biti $DCD_n, n \geq 1$ este un circuit care transforma un cod numeric k de n biti intr-o alegere fizica a liniei de iesire cu numarul k .

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:



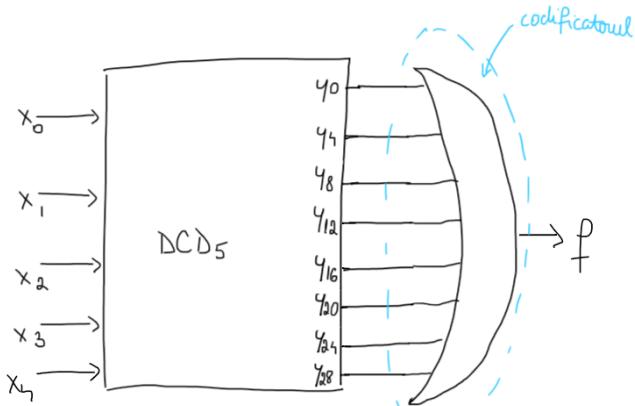
Asadar, diagrama pentru un codificator si diagrama pentru un decodificator arata:



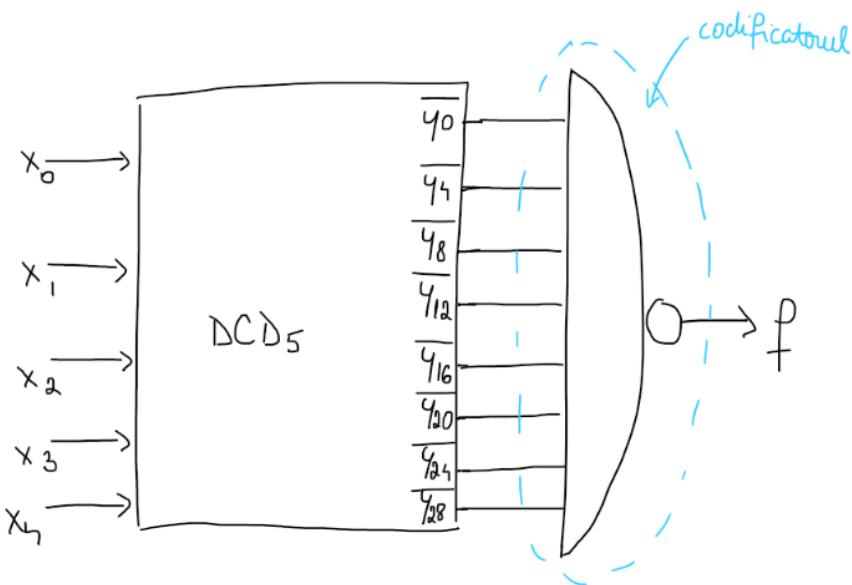
Exemplul 7 Scrieti un codificator care pentru 5 biti scoate "1" daca x divizibil cu 4.

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:

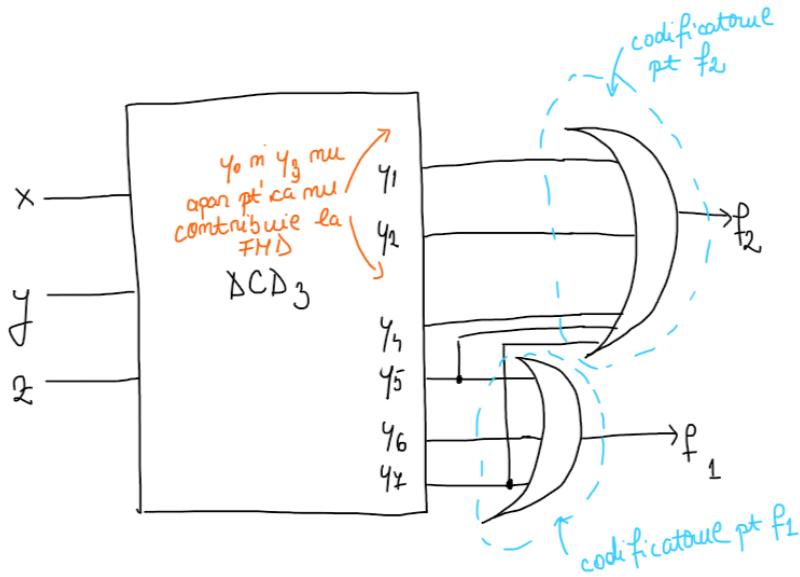


Exemplul 8 Pentru functia de la Exemplul 6, implementati f printr-un codificator.

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.

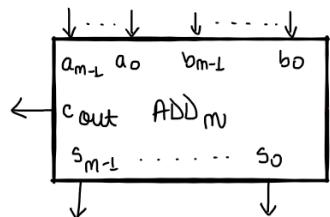
Din tabelul facut la Exemplul 6 vedem ca in FND-ul lui f_1 sunt liniile (5), (6) si (7), iar in FND-ul lui f_2 sunt liniile (1), (2), (4), (5) si (7). Asadar, implementarea cu codificator este:



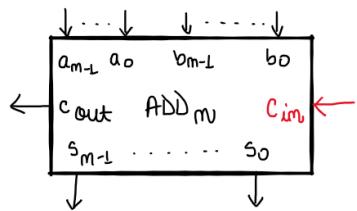
1.6 Sumatori

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:

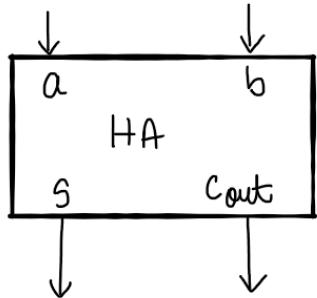
- Half adder: sumatorul pe care il folosim in adunarea celui mai nesemnificativ bit. Automat, acest sumator nu are transport de intrare.
- Full adder: sumotorul pe care il folosim pentru adunarea bitilor cu rang ≥ 1 . Poate avea transport de intrare.

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 $c_{out} = (a+b) \text{div } 2 \Leftrightarrow c = a \cdot b$

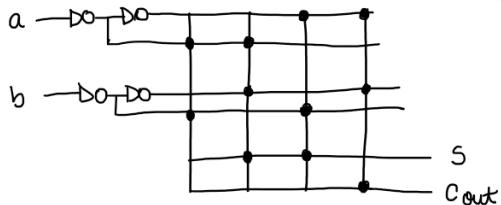
Reprezentarea sa este:



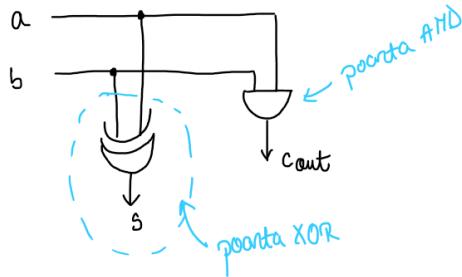
Iar tabelul:

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:

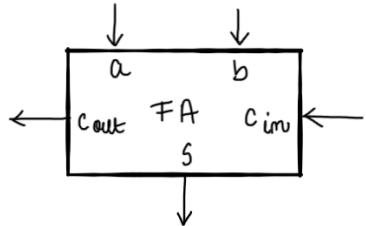


1.6.2 Full adder

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

$$= (\bar{a} \cdot \bar{b}) \cdot (\bar{b} \cdot \bar{a}) =$$

$$= (\bar{a} + b) \cdot (\bar{b} + a) =$$

$$= \bar{a} \cdot \bar{b} + \bar{a} \cdot a + b \cdot \bar{b} + ab = A$$

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

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

Iar pentru c cand scriem FND-ul avem:

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

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

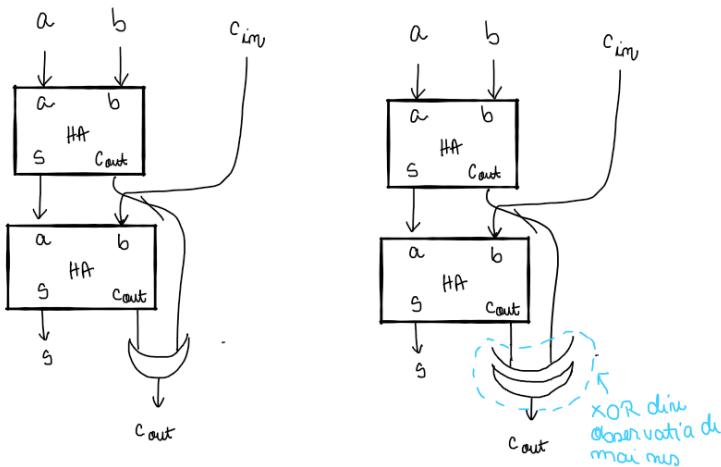
Stim ca $HA_S(a,b) = a \oplus b$
 $HA_c(a,b) = a \cdot b$

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

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

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

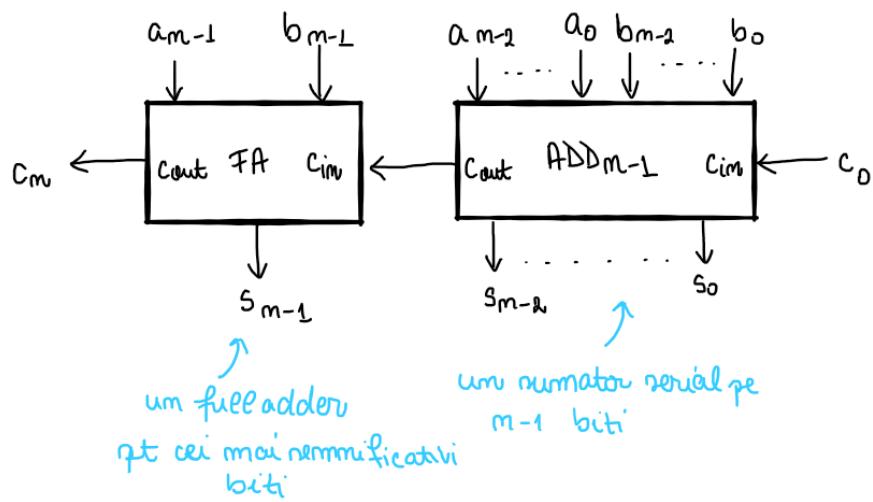
Dragulici spune in materialul lui ca $HA_c(HA_S(a,b),c)$ si $HA_c(a,b)$ nu pot fi simultan 1 (si o sa il credem pe cuvant) deci acel plus dintre ele poate fi inlocuit cu un XOR. Astfel, putem folosi 2 Half addere pentru a construi un Full adder:



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)

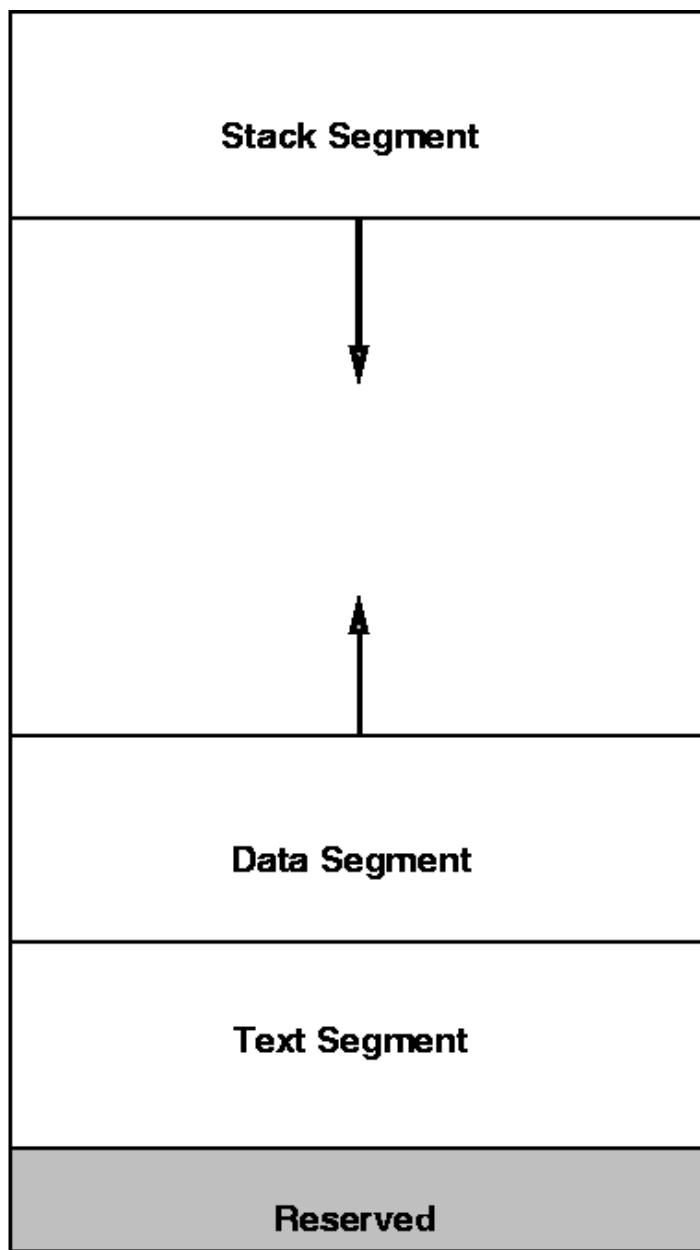
2.2 Instrucțiuni folosite

- 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:

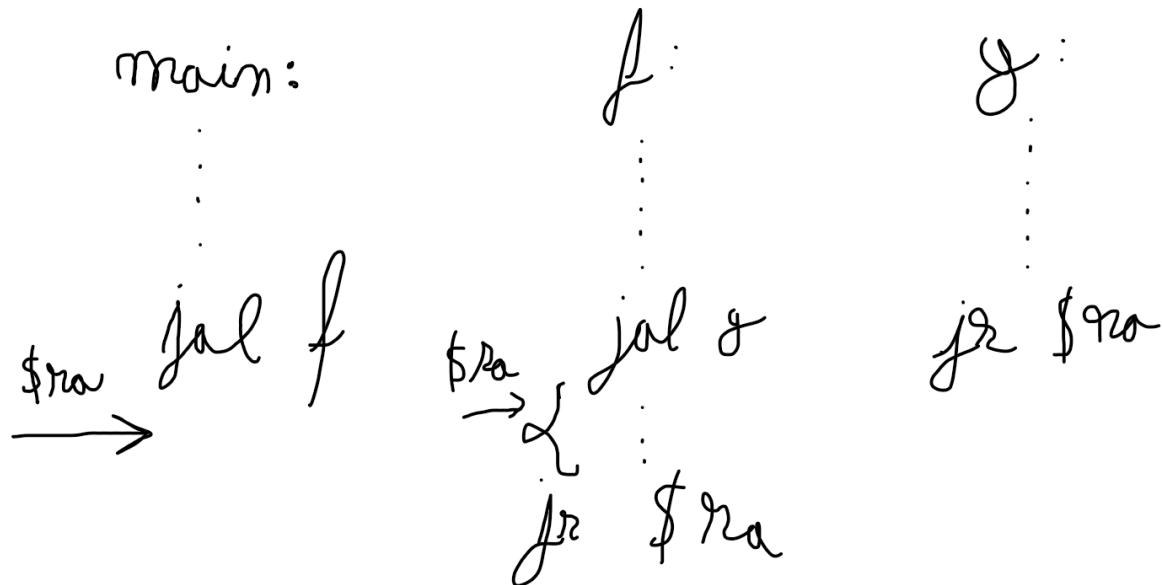
6 este perfect deoarece $6 = 1 + 2 + 3$

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

```

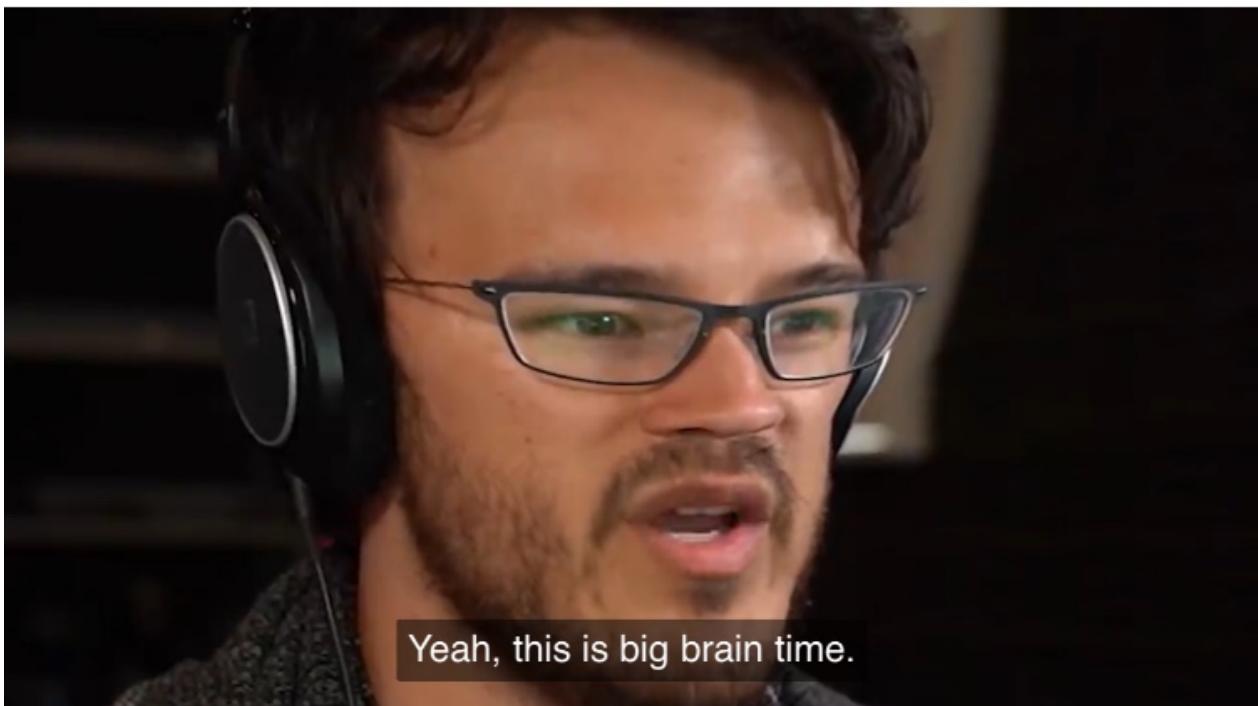
problema1_3.s

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/ vizualizere).



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*

Tutoriat 6

Stan Bianca-Mihaela, Stăncioiu Silviu

November 2020

**PROFI DE MATE DUPA CE SCRII PE
FOAIA DE EXAMEN CUM COVRIGII SUNT
PRODUSE DE PANIFICATIE, IAR APOI
DESCRII METAFIZIC NATURA
PARADOXALA A UNIVERSULUI PENTRU
CA AI UITAT CA DAI EXAMEN LA MATE,
NU LA ASC.**





TUTORIATE SCRISE FOARTE
FORMAL, CU BIBLIOGRAFIE
ADEVARATA SI VERIFICATE
DE MULTE ORI



TUTORIATE NEVERIFICATE,
PLINE DE TYPO-URI,
DEZACORDURI SI
MEME-URI

Contents

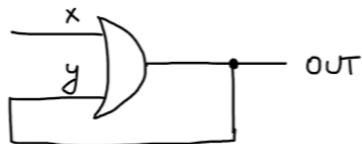
1	1-DS (Memorii)	3
1.1	Zavoare	3
1.1.1	Zavorul elementar	3
1.1.2	Zavorul elementar eterogen (SR latch)	4
1.2	Delay Flip-Flop (DFF)	6
2	Arhitectura MIPS	11
2.1	Reprezentarea internă a instrucțiunilor procesorului MIPS	12
2.2	Procesorul MIPS cu un ciclu	19
2.2.1	PC	20
2.2.2	Citirea instructiunii din memorie	21
2.2.3	Identificarea tipului de instructiune	22
2.2.4	Citirea registrilor	23
2.2.5	Scrierea in memorie / citirea din memorie	24
2.2.6	Executarea instructiunii	25
2.2.7	Unitatea de control	26

1 1-DS (Memorii)

Sistemele 1-DS sunt sisteme 0-DS inchise printr-un ciclu.

1.1 Zavoare

1.1.1 Zavorul elementar



x	y	$x+y$
0	0	0
0	1	1
1	0	1
1	1	1

Mai sus avem reprezentarea unui zavor elementar, impreuna cu tabelul de adevar pentru OR.

Sa spunem ca avem un circuit prin care, la apasarea unui switch, eu pot sa ii schimb valoarea de adevar a lui x.

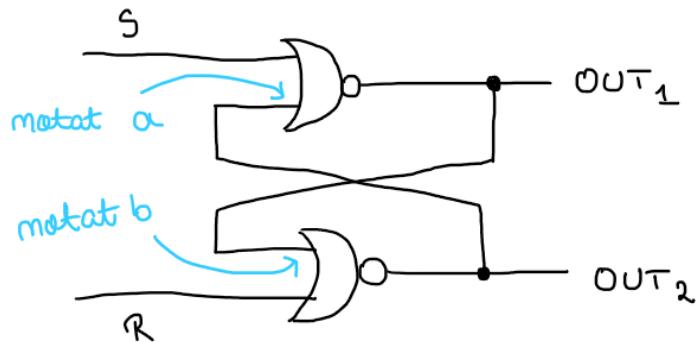
- Conectez circuitul la curent. Ambele valori, x si y, sunt 0. \Rightarrow OUT are valoarea 0.
- Ce observam? OUT isi transmite valoarea inapoi catre y. Deci avem o bucla infinita: atata timp cat x ramane 0, y il face pe OUT sa fie 0, iar OUT il face la randul lui pe y sa fie 0.
- Apas pe switch. Ce am zis ca face switch-ul? Schimba valoarea de adevar a lui x. \Rightarrow Acum valoarea de adevar a lui x este 1. Ce se intampla cu valoarea lui OUT? Valorile lui x si y trec prin poarta OR. Stim ca $1 \text{ OR } 0 = 1$. \Rightarrow valoarea lui OUT va fi 1.
- Stim ca OUT isi transmite valoarea sa inapoi catre y. Deci ce se intampla? y devine 1. \Rightarrow poarta OR primeste 1 si 1 \Rightarrow OUT e in continuare 1. Deci avem din nou o bucla infinita.
- Apas din nou pe switch. \Rightarrow valoarea lui x devine 0. \Rightarrow Poarta OR primeste un 0 si un 1 deci OUT=1. OUT isi transmite valoarea catre y si rezultatul ramane acelasi. Din nou am o bucla infinita. Mai mult decat atat, observ ca ori de cate ori as apasa eu switch-ul de acum incolo, OUT va fi tot 1.
- Singura solutie sa il fac pe OUT 0 este sa deconectez circuitul de la curent.

Am vazut deci cum functioneaza un zavor elementar.

Pentru demonstratia fizica vezi: <https://www.youtube.com/watch?v=KM0DdEaY5sY>

Am vrea acum sa avem o modalitate sa il facem pe OUT 0 oricand vrem noi. Cu acest scop in minte, introducem zavorul elementar eterogen.

1.1.2 Zavorul elementar eterogen (SR latch)



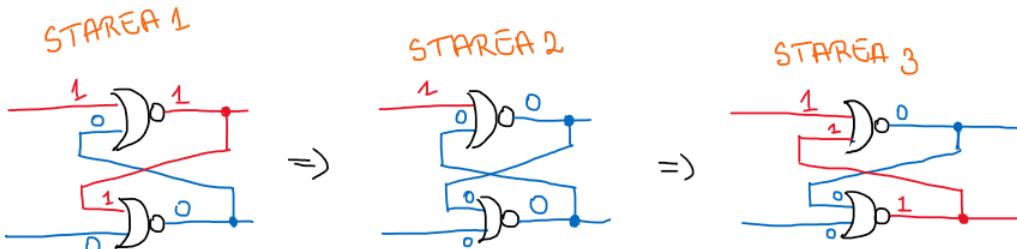
x	y	x NOR y
0	0	1
0	1	0
1	0	0
1	1	0

Avem mai sus schema unui zavor elementar eterogen, impreuna cu tabelul de adevar pentru NOR.

Acum avem 2 switch-uri: unul pentru S si unul pentru R.

Sa o luam din nou pe pasi:

- Conectam circuitul la curent. Pentru prima poarta NOR, ambele input-uri vor fi 0 \Rightarrow OUT₁ va fi 1.
- Pentru ca OUT₁ este 1, intrarea notata de mine cu b va fi tot 1 (OUT₂ isi propaga valoarea in b). R va fi 0 (inca nu am apasat pe switch). \Rightarrow OUT₂ = 0. Acest OUT₂ este propagat catre a, care era tot 0 inainte deci nu isi schimba valoarea. Avem din nou o bucla infinita.
- Apas pe switch-ul lui R. \Rightarrow valoarea lui R devine 1. Din tabel vedem ca 1 NOR 1 = 0 deci nu se schimba nimic pentru OUT₁ si OUT₂. Asa ca pot sa apas switch-ul lui R de oricate ori fara a schimba rezultatele. Mai apas o data ca sa il fac pe R din nou 0.
- Apas pe switch-ul lui S. Ce se intampla?



STAREA 1: S devine 1.

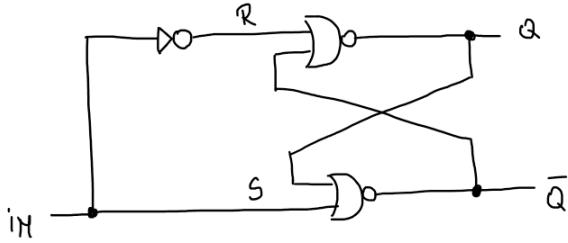
STAREA 2: 1 NOR 0 = 0. \Rightarrow OUT₁ = 0, OUT₁ isi transmite valoarea catre b, deci si b=0.

STAREA 3: In a doua poarta NOR: 0 NOR 0 = 1 \Rightarrow OUT₂ = 1, OUT₂ isi transmite valoarea catre a, deci a=1. \Rightarrow In prima poarta NOR: 1 NOR 1 = 0. \Rightarrow OUT₁ nu isi schimba valoarea si ne-m intors la o bucla infinita.

- In final, vedem ca am schimbat valorile de output. Initial aveam OUT₁ = 1 si OUT₂ = 0, acum avem OUT₁ = 0 si OUT₂ = 1
- Analog, ca sa schimbam din nou outputurile, acum trebuie sa apasam pe switch-ul lui R.

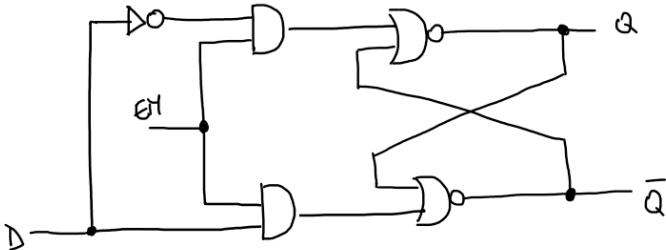
Observatie ! Mereu OUT₁ si OUT₂ au valori complementare. De aceea, ele se noteaza cu OUT₁ = Q si OUT₂ = \bar{Q} .

Putem acum sa facem o mica simplificare. Vrem sa avem un singut input, nu doua. Am vazut ca, desi aveam doua input-uri, la orice moment de timp unul dintre ele nu facea nimic, ori de cate ori ii schimbam valoarea. Asa ca putem transforma circuitul in:



Ce am reusit sa facem acum? Daca avem un switch prin care putem schimba valoarea lui IN, de fiecare data cand apasam switch-ul, Q si \bar{Q} isi schimba valorile. Daca initial $Q=0$ si $\bar{Q}=1$, dupa apasarea switch-ului $Q=1$ si $\bar{Q}=0$.

Acum, nu ne place ca noi putem schimba valoarea lui Q (si implicit si a lui \bar{Q}) oricand. Vrem sa avem un enabler care sa imi spuna cand am voie sa schimb valorile si cand nu. Schema care rezolva aceasta problema este:



Acum putem sa schimbam valoarea lui Q doar daca $EN=1$. Acest circuit obtinut poarta numele de D-latch. Pentru demonstratia cu circuite vezi: <https://www.youtube.com/watch?v=peCh859q7Qt> = 26s

Me: Mom, can I have



?

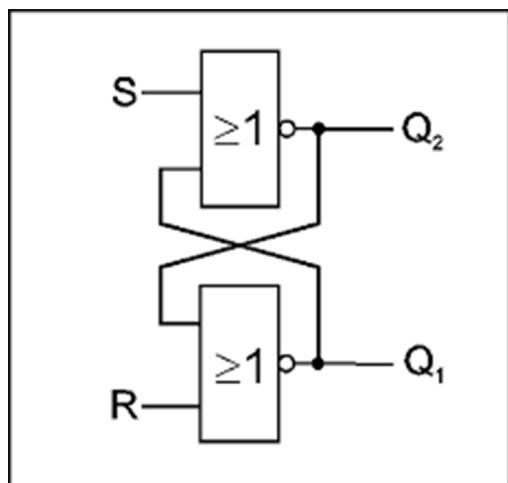
Mom: No we have



at home

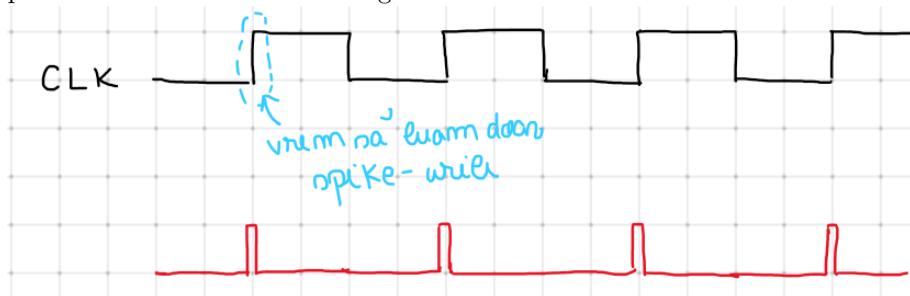


at home:

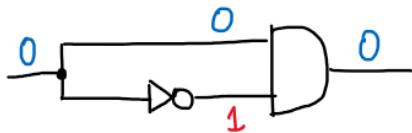


1.2 Delay Flip-Flop (DFF)

Vrem sa inlocuim enable-ul de mai devreme cu un clock CLK. Un clock este un circuit care isi schimba periodic valoarea de la low la high.

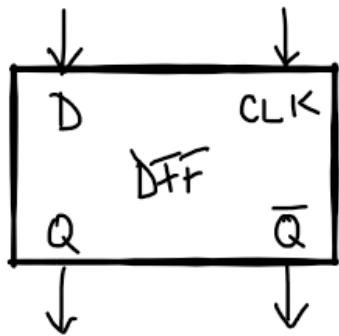


Ne intereseaza sa obtinem din acest clock un circuit care sa ne izoleze acele spike-uri de la high la low. Acest circuit pe care il cautam noi se numeste edge detector. Un exemplu de un astfel de circuit este:



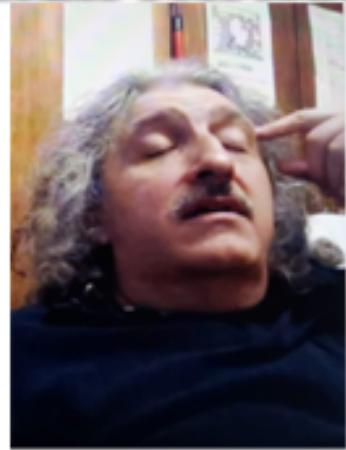
Observam starea in care este acum edge detector-ul, cu valorile scrise pe fiecare ramura. Daca input-ul isi schimba valoarea, pentru un timp foarte scurt, ramura de sus o sa fie 1, iar ramura de jos isi face nagatia putin mai greu si va ramane tot 1. Deci, pentru un timp foarte foarte scurt, ambele vor avea valoarea 1 deci output-ul va fi 1. Am obtinut astfel un edge detector.

D Flip-Flop este un D-latch ca mai sus, doar ca un loc de EN are un edge detector conectat la CL. Simbolul lui este:



Pentru mai multe detalii: <https://www.youtube.com/watch?v=YW-GkUguMM>

YOUR CRUSH **HER EX** **HER FATHER**



HER BROTHER



HER MOTHER



YOU

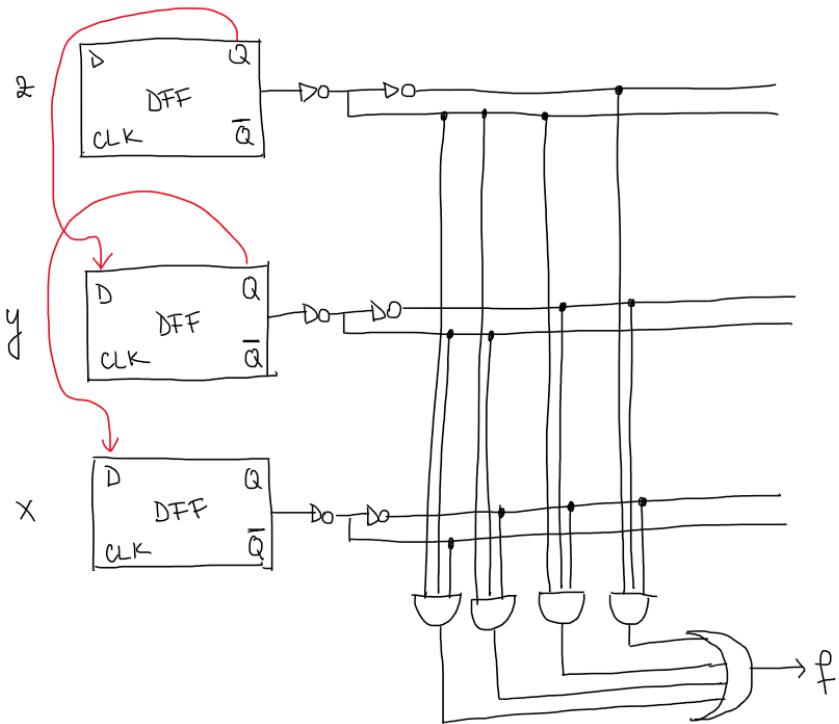


CAND ESTI IN MINECRAFT SI
CONSTRUIESTI UN D FLIP
FLOP IN LOC SA PUI UN
SIMPLU REPEATER



Exemplul 1 Construiti un circuit 1-DS care citeste unul cate unul o secventa de biti si de fiecare data scoate 1 daca $x \leq y \leq z$.

Ne amintim ca am facut deja tabelul acestei functii in tutoriatul 4 si stim ca in FND-ul sau apar liniile (0), (1), (3) si (7).



2 Arhitectura MIPS



2.1 Reprezentarea internă a instrucțiunilor procesorului MIPS

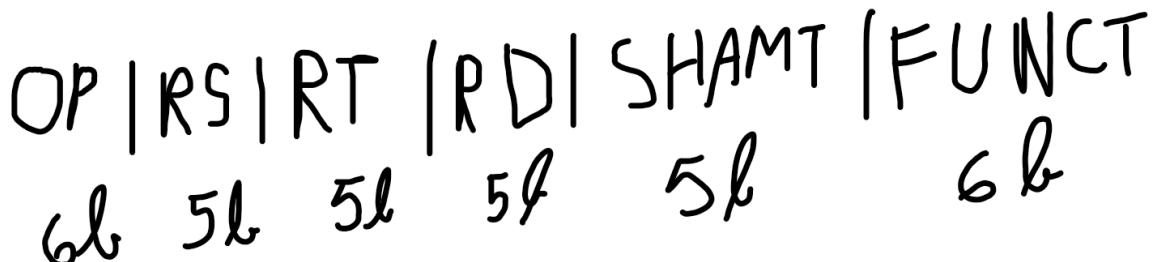
În MIPS avem 3 clase de instrucțiuni, și anume:

- R
 - Operații aritmetice, care nu utilizează valori imediate (de exemplu: add \$t0, \$t1, \$t2. Contraexemple: addi \$t0, 1 nu este de tip R deoarece se folosesc valori imediate)
 - Set: slt, seq, sre, ...
 - Shiftări logice: sll, slr
- I
 - Operații cu valori constante indicate: load, store, branch conditions (ble, blt, ...)
- J
 - jumps (j, jal)

Fiecare instrucțiune poate fi reprezentată în binar pe 32 de biți. Biții din reprezentarea binară a instrucțiunii se deduc din clasa de instrucțiuni din care face parte, regiștrii implicați și valorile constante implicate. Astfel, avem următoarele reprezentări pentru:

Clasa de instrucțiuni R:

Reprezentarea binară a unei instrucțiuni de tip R este compusă din mai multe secțiuni, și anume: op, rs, rt, rd, shamt și func. op și func ocupă 6 biți, iar restul ocupă 5 biți.



Când vrem să reprezentăm o instrucțiune în binar, putem deduce fiecare secțiune din ea în felul următor:

- op - În clasa R este mereu 000000.
- rs - Registrul sursă 1 (adică codul registrului reprezentat în binar. Fiecare registru are un număr. De exemplu registrul: \$t0 este echivalent cu registrul \$8. Găsiți în tabel codurile fiecărui registru).
- rt - Registrul sursă 2.
- rd - Registrul destinație.
- shamt - Shift amount. Se completează ≠ 0 când se face shiftare.
- func - În pereche cu op se decide ce instrucțiune MIPS se aplică. (găsim în tabel valorile pentru func care ne trebuie)

Exemplul 2

add \$t0, \$t1, \$t2

op → 000000 (pentru că instrucțiunea ∈ R)

rs → \$t1 = \$9 = 01001 (registru \$t1 are valoarea 9, again, găsim în tabel valorile)

rt → \$t2 = \$10 = 01010

rd → \$t0 = \$t8 = 01000

func → 100000 (este dat)

Acum vom pune la un loc toate valorile obținute, deci vom avea (le-am grupat direct în bucăți de câte 4 pentru a ne fi ușor după să le transformăm în baza 16):

0000 0001 0010 1010 0100 0000 0010 0000

Acum vom transofma rezultatul obținut în baza 16 (exact cum făceam și în primele 2 tutoriate), deci vom avea:

0x012A4020

Exemplul 3

sll \$s1, \$v0, 2 (este instrucțiune de tip R, iar $func = 000000$)

op → 000000 (instrucțiunea ∈ R)

rs → 00000 (nu se completează aici deoarece avem un singur regisztru sursă)

rt → 00010 (codul pentru regisztrul \$v0)

rd → \$s1 = \$17 = 10001

shamt → 2 = 00010

func → 000000

Reprezentarea în baza 2 este:

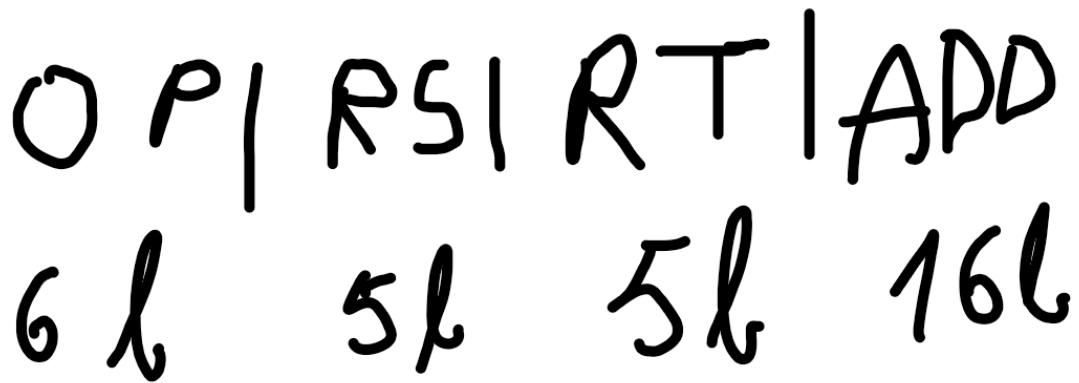
0000 0000 0000 0010 1000 1000 1000 0000

Iar în baza 16 avem:

0x00028880

Clasa de instrucțiuni I:

Secțiunile pentru aceste instrucțiuni sunt: op, rs, rt și add/ imm. Op ocupă 6 biți, rs și rt ocupă 5 biți, iar add/ imm restul de 16 biți.



Avem:

- op - Operația, avem în tabel codificarea binară.
- rs - Registru sursă.
- rt - Registru sursă/ destinație după caz.
- add/ imm - Câmp de adresă/ valoare imm, după caz.

Exemplul 4

lw \$t0, 4(\$t2)

$\text{op} \rightarrow 100011$
 $\text{rs} \rightarrow \$t2 = \$10 = 01010$
 $\text{rt} \rightarrow \$t0 = \$8 = 01000$
 $\text{imm} \rightarrow = 0000000000000100$

Exemplul 5

beq \$t1, \$s0, 28

$\text{op} \rightarrow 000100$
 $\text{rs} \rightarrow \$t1 = \$9 = 01001$
 $\text{rt} \rightarrow \$s0 = \$16 = 10000$
 $\text{add} \rightarrow 28 \sim \frac{28}{4} = 7 = 0000000000000111$

Clasa de instrucțiuni J:

Aici secțiunile sunt doar op (6 biți) și add (26 de biți).

OP | ADA
6l 26l

Exemplul 6

j 28

Tabelul cu valorile regiștrilor:

\$zero:0	\$t7:15	\$gp:28
\$at:1	\$s0:16	\$sp:29
\$v0:2	\$s1:17	\$fp:30
\$v1:3	\$s2:18	
\$a0:4	\$s3:19	\$ra:31
\$a1:5	\$s4:20	
\$a2:6	\$s5:21	
\$a3:7	\$s6:22	
\$t0:8	\$s7:23	
\$t1:9	\$t8:24	
\$t2:10	\$t9:25	
\$t3:11	\$k0:26	
\$t4:12	\$k1:27	
\$t5:13		
\$t6:14		

Imaginea de mai sus este extrasă din tutoriatul de ASC tinut anul trecut de Larisa Dumitrashe.

La examen, pentru a găsi *op* și *func* pentru anumite instrucțiuni, folosiți acest link:
<http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html>

Folosiți-vă de secțiunea encoding a instrucțiunilor pentru a deduce cum să completați reprezentarea binară a instrucțiunilor.

Alternativ, folosiți-vă de acest link unde este scris direct codul pentru op/ func:
<http://alumni.cs.ucr.edu/~vladimir/cs161/mips.html>

Exemple:

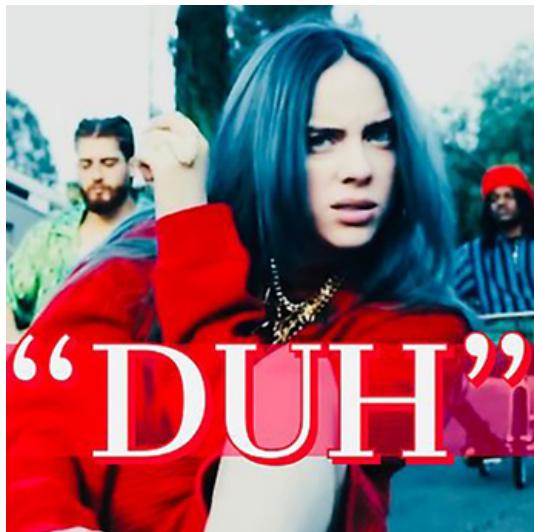
Exemplul 7 [Restanță MAI 2020]

Considerăm implementarea procesorului MIPS cu 1 ciclu / instrucțiune (vezi verso). Fie fragmentul de program:

```
1 li $t1 , 2
2 li $t2 , 3
3 li $t3 , 2
4 et :
5 add $t1 , $t2 , $t1
6 sub $t3 , $t1 , $t3
7 beq $t3 , $t2 , et
```

prog.mips.s

side note, la examen nu veți primi un program cu syntax-highlighting, duuh



Presupunem că în memorie instrucțiunea *sub* din program are adresa α .

a) Pentru instrucțiunile *sub* și *beq* din program scrieți câmpurile din reprezentarea lor internă (ex: op/rs/rt/imm, valorile se scriu hexa); pentru *beq* din program scrieți reprezentările ei binară (32 biți) și hexa (8 cifre hexa).

Rezolvare:

Observăm că *sub* face parte din clasa R de instrucțiuni, deci avem:

op \rightarrow 000000, adică 0 în hexa. (cum ne cere exercițiul)

rs \rightarrow \$t1 = \\$9 = 01001, adică 9 în hexa.

rt \rightarrow \$t3 = \$11 = 01011, adică B în hexa.

rd \rightarrow \$t3 = \$11 = 01011, adică B în hexa.

func \rightarrow 100010, adică 22 în hexa.

Observăm că *beq* face parte din clasa I de instrucțiuni, deci avem:

op \rightarrow 000100, adică 4 în hexa.

$rs \rightarrow \$t3 = \$11 = 01011$, adică B în hexa.

$rt \rightarrow \$t2 = \$10 = 01010$, adică A în hexa.

CUM FACEM SĂ AFLĂM VALOAREA LUI et ? Noi mai devreme am văzut cum se codifică beq când avem o constantă multiplu de 4, dar acum cum facem...?

Trebuie să stim că beq nu aşteaptă o adresă de memorie absolută din program la care să facă jumpul, ci mai degrabă, un offset de la poziția lui către poziția unde vrem să facă jump-ul. Trebuie să ținem minte că atunci când se execută instrucțiunea beq , PC (un fel de registru care ține poziția curentă a instrucțiunii care se execută) are valoarea adresei instrucțiunii următoare lui beq . De aceea pentru a calcula poziția relativă trebuie să facem: $\frac{(adresă_etichetă - adresă_instructiune_branch - 4)}{4}$. Observăm că această formulă ne dă numărul de instrucțiuni dintre etichetă și beq (inclusiv) în cazul în care label-ul este înaintea lui et . Observăm că rezultatul este negativ în acest caz, deci îl vom codifica ca număr signed. În cazul în care eticheta se află după beq , formula ne va da offset-ul dintre instrucțiunea de după beq și instrucțiunea din dreptul label-ului. Iar în cazul în care label-ul este pus fix peste beq , atunci am avea -1.

Ne vom imagina că înainte de fiecare instrucțiune din următorul program avem un label, deci relativ la beq vom avea valorile din comentarii:

```
1 .data
2 .text
3 main :
4
5 li $t0 , 0          # -5
6 li $t1 , 0          # -4
7
8 li $a0 , 100        # -3
9 li $v0 , 1          # -2
10
11 beq $t0 , $t1 , et # -1
12 et :
13 syscall             # 0
14
15 li $v0 , 10         # 1
16 syscall             # 2
```

test_instr.s

Acum fiind spuse, observăm că et este înaintea lui beq în problema noastră, deci vom lua numărul de instrucțiuni dintre et și beq (inclusiv), adică 3 instrucțiuni. Eticheta fiind înainte de beq , avem -3 pentru add, deci:

$add \rightarrow -3 = 1111111111111101$, adică FFFD în hexa

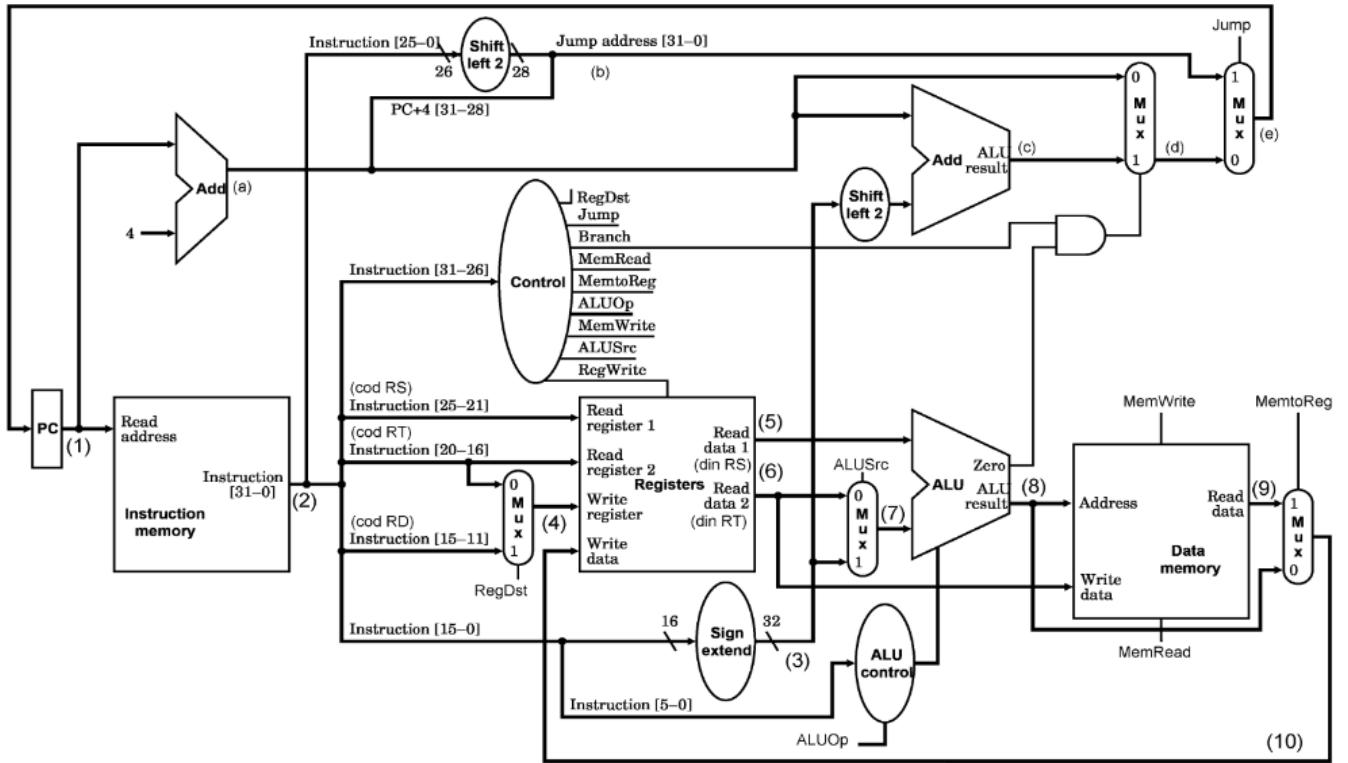
Pentru beq problema ne cere să scriem și reprezentarea binară și reprezentarea în hexa, deci vom avea:

0001 0001 0110 1010 1111 1111 1111 1101 (în binar)

0x116AFFFD (în hexa)

2.2 Procesorul MIPS cu un ciclu

Aşa arată procesorul MIPS cu un ciclu:



Inainte sa aruncati laptopul pe fereastra, sa ne gandim putin ce face acest procesor:

- Face FETCH pentru o instructiune din memoria de instructiuni.
- Decodifica instructiunea: vream sa stim daca e ADD sau SUB, daca e de tip I sau R, etc.
- Citeste operanzii din registrii (\$rs, \$rd, op, etc).
- Executa instructiunea.
- Scrie inapoi in memorie.

Voi mentiona acum si tabelele de ajutor, de care ne vom folosi pe tot parcursul subiectului 3:

*no ignorează complet
jucările și
cărora*

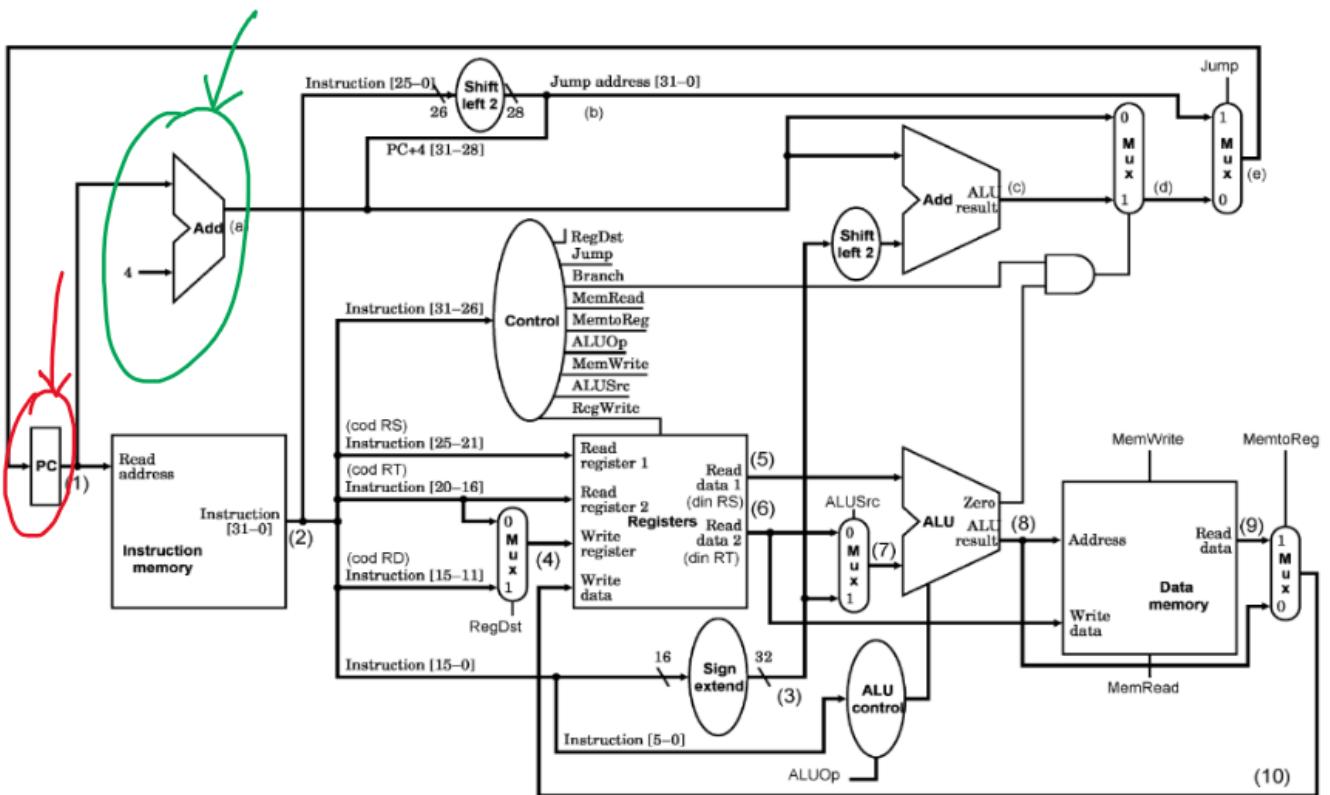
Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch
R-format	1	0	0	1	0	0	0
lw	0	1	1	1	0	0	
sw	X	1	X	0	0	1	0
beq	X	0	X	0	0	0	1

ALU Control (slide 7.36)

ALUOp ₁	ALUOp ₀	Camp functie						Operatie	
		F5	F4	F3	F2	F1	F0		
lw/sw	0	X	X	X	X	X	X	010 (+)	
beq	X	1	X	X	X	X	X	110 (-)	
add	1	X	X	X	0	0	0	010 (+)	
sub	1	X	X	X	0	0	1	110 (-)	
and	1	X	X	X	0	1	0	000 (and)	
or	1	X	X	X	0	1	0	001 (or)	
R-format	1	X	X	X	1	0	1	0	111 (slt)

2.2.1 PC

Acum ca avem asta in minte, sa o luam cu prima chestie de la stanga la dreapta: PC.

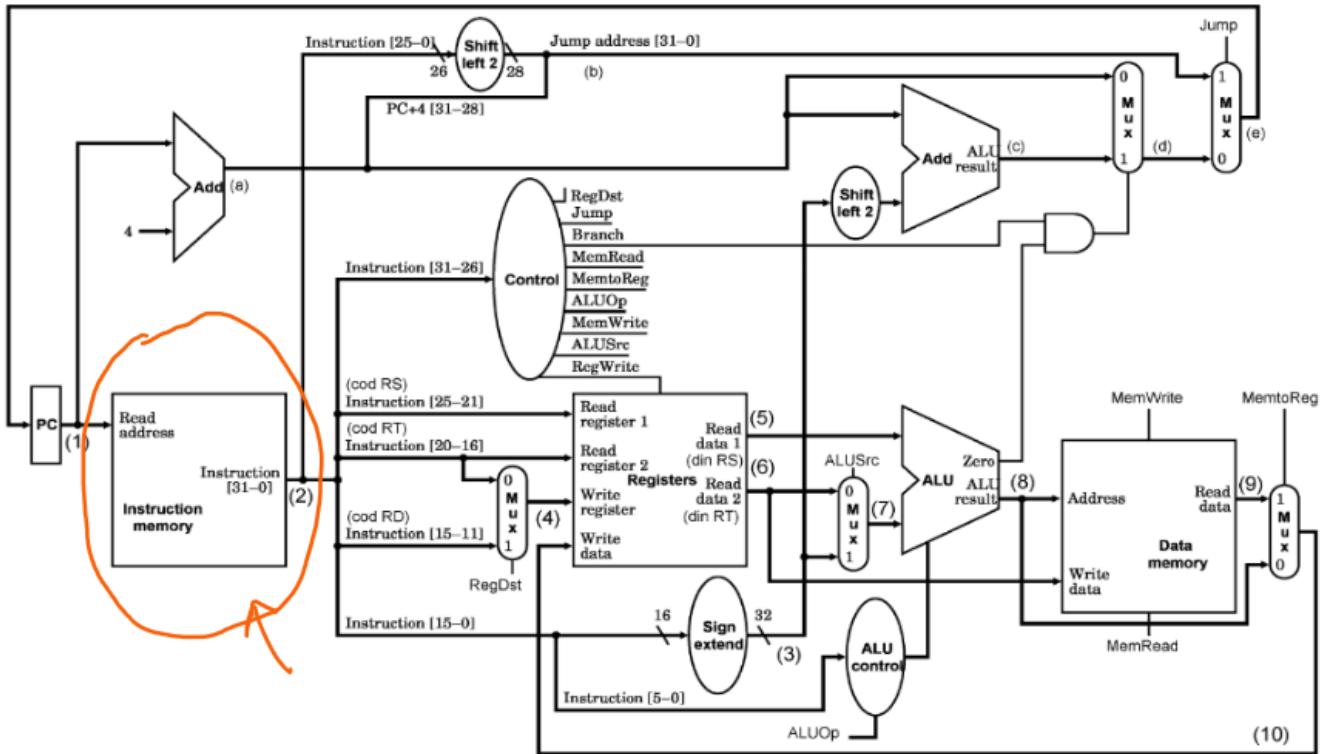


Ne putem gandi la PC ca la un pointer catre urmatoarea instructiune. De fiecare data cand se executa o instructiune, sa "incrementeaza" acest "pointer" pentru a arata catre urmatoarea instructiune. Incrementarea se face cu 4. De ce 4? Reprezentarea instructiunilor MIPS in calculator are 32 biti. Un byte are 8 biti. \Rightarrow Reprezentarea instructiunilor MIPS in calculator are 4 bytes.

Asadar, in schema, PC este cel incercuit cu rosu, iar incrementarea printr-un adder(sumator) este incercuita cu verde.

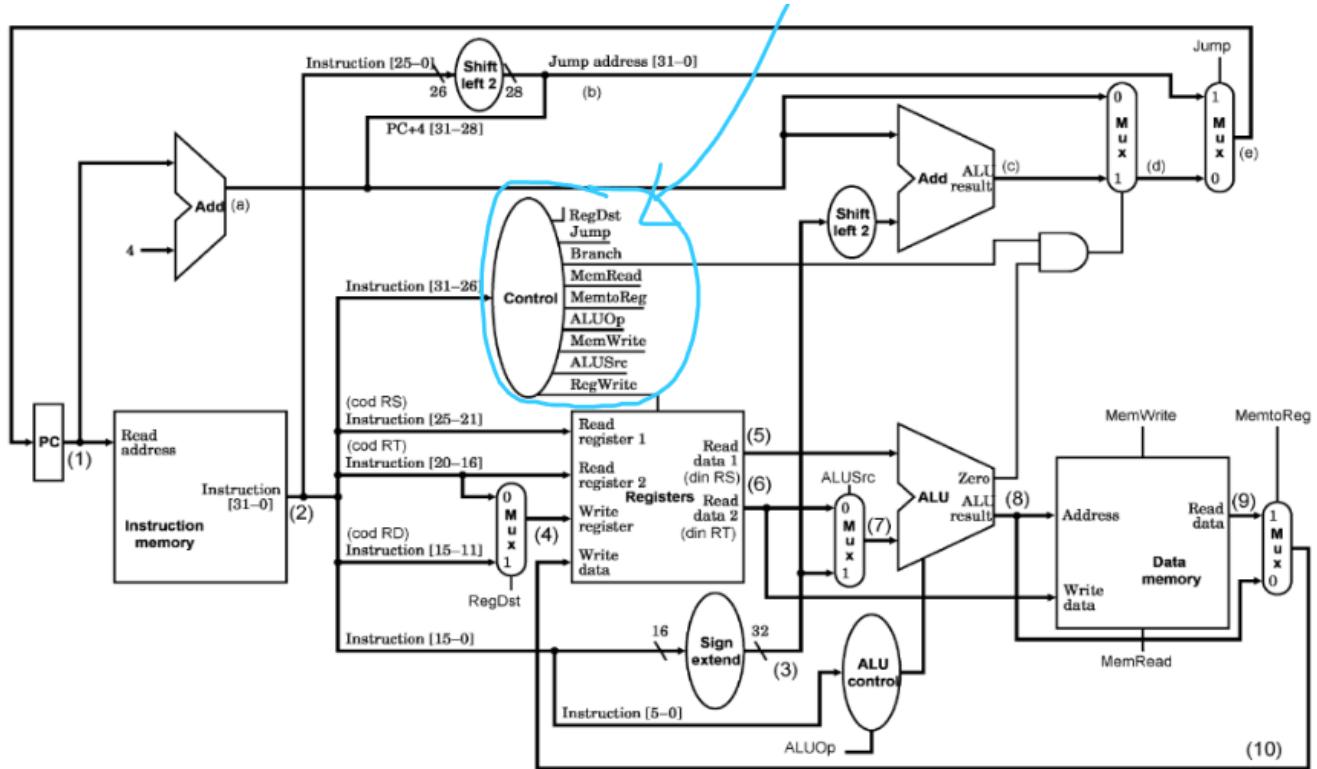
Observatie! In timpul executarii instructiunii, PC "arata" catre urmatoarea instructiune. De aceea, la 2.1 Reprezentarea interna a instructiunilor procesorului MIPS, trebuie sa adaugam acel -1.

2.2.2 Citirea instructiunii din memorie

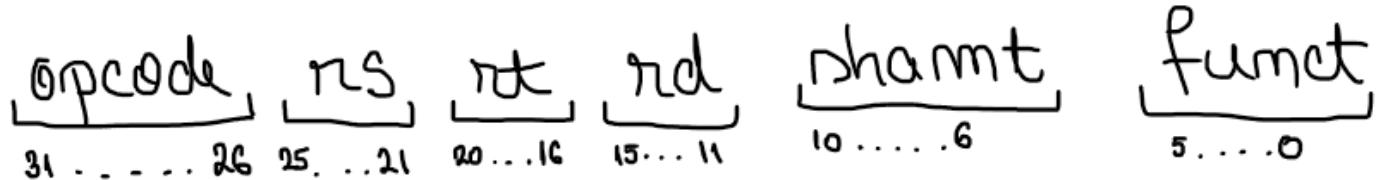


Citirea instructiunii din memorie este incercuita cu portocaliu. Vedem ca avem un patrat care se numeste chiar "Instruction memory" in care intra "Read address" (deci PC ii spune acestui bloc de la ce adresa sa citeasca instructiunea). Blocul are ca output o instructiune, pe 32 de biti.

2.2.3 Identificarea tipului de instructiune

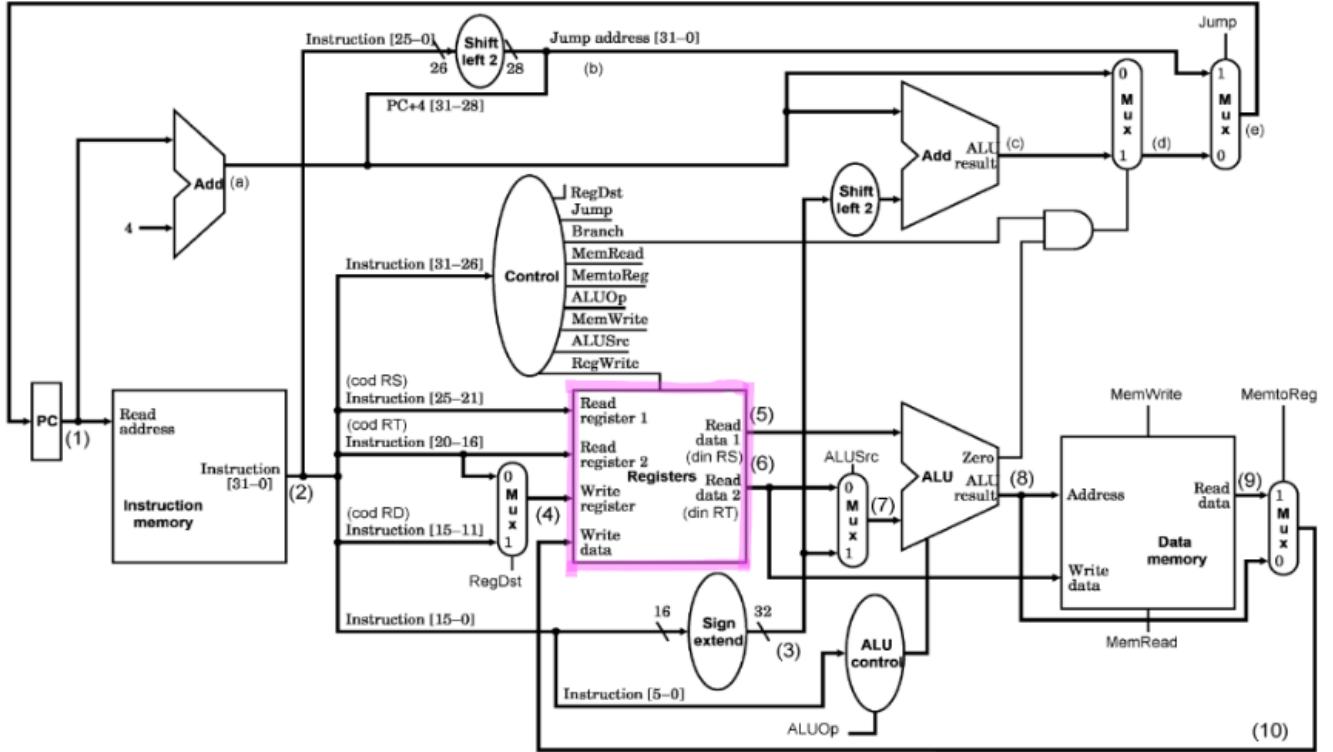


Identificarea tipului de instructiune se face prin structura incircuita cu albastru. Aceasta se numeste Control Unit. Ce parte din cei 32 de biti ai unei instructiuni ne spune noua ce fel de instructiune avem? Opcode. Acesta e comun tuturor tipurilor de instructiuni (R, I, J) si se afla mereu in primii 6 biti. Observam ca in Control Unit intra bitii de la 31 la 26. Dar noi am spus ca opcode este mereu in primii 6 biti. Ne dam seama deci, ca asta este doar o chetiere de notatie. In memorie, fiecare bit este indexat de la 31 la 0.(putem sa ne gandim ca ultimul bit este cel mai nesemnificativ bit si de aceea are index 0).



Prin Opcode identificam ce fel de instructiune avem, iar prin datele pe care le stim noi despre instructiuni (pe care le vom identifica din tabelele de ajutor) vom afla valorile tuturor acelor variabile: RegDst, Jump, Branch, MemRead, MemtoReg, ALUOp, MemWrite, ALUSrc, RegWrite.

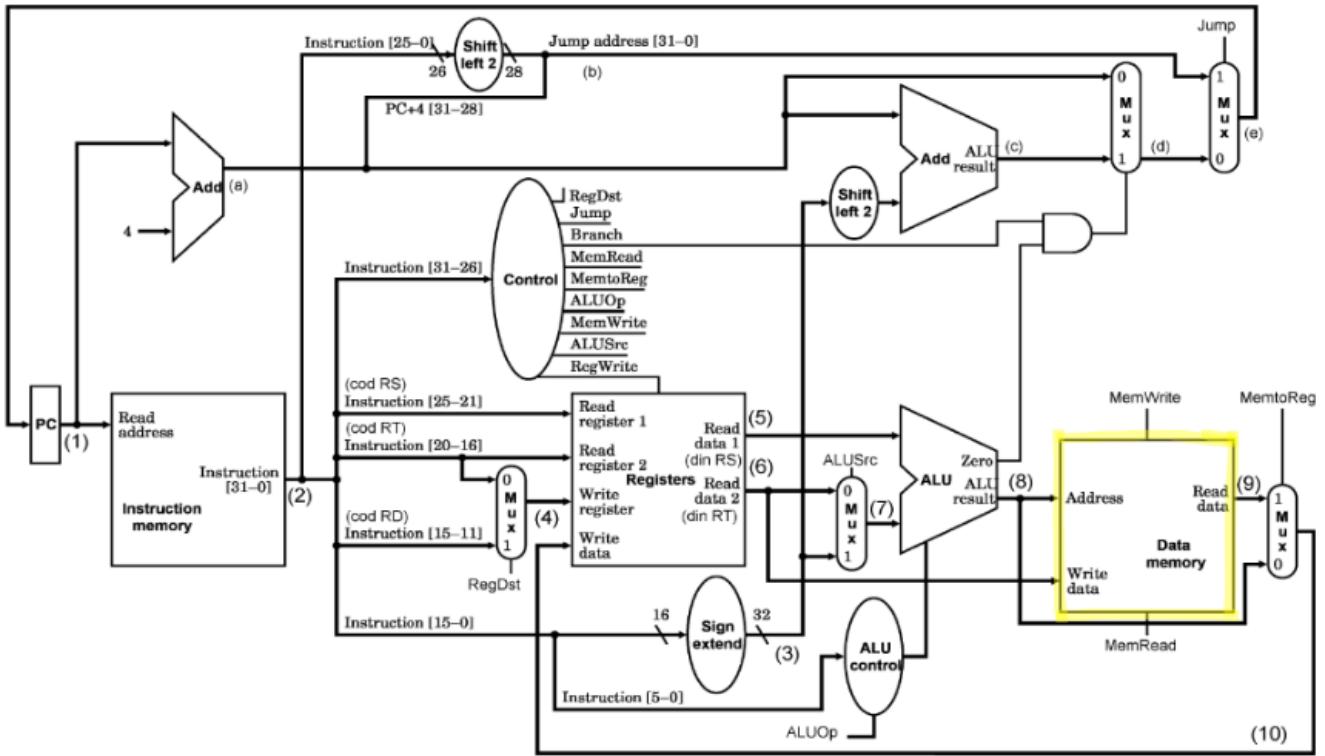
2.2.4 Citirea registrilor



Trecem la citirea registrilor.

- RegWrite: care, după cum vedem din tabel, are valoarea 1 pentru instrucțiunile de tip R ($\$d = \$s + \$t$) și pentru lw (evident, se „scrise” în registrul $\$t$)
- Read register 1: citeste care este registrul $\$s$ (poate să fie $\$t0$, $\$v2$, etc.).
- Read register 2: citeste care este registrul $\$t$ (poate să fie $\$t0$, $\$v2$, etc.).
- Write register: Aici se complica puțin lucrurile. Avem mai multe cazuri, în funcție de tip:
 - Pentru instrucțiuni de tip R, registrul $\$d$ este cel în care se scrie (ex: pt add $\$d = \$s + \$t$) de aceea el se numește Write register. Ce intra în Write register este determinat de un multiplexor. Rezultatul multiplexorului este determinat de RegDst. Dacă ne uitam în tabel, RegDst este 1 doar pentru instrucțiunile de tip R (evident, pentru că celelalte tipuri de instrucțiuni nu au un registru $\$d$). => Din multiplexor o să iasa codul lui $\$d$, și acolo va fi scris rezultatul instrucțiunii.
 - Pentru instrucțiuni de tip I: nu vom vorbi despre toate instrucțiunile de tip I pentru că unele dintre ele rulează 2 instrucțiuni în loc de una. Pentru mai multe detalii vezi Exemplul 10. În tabele avem menționate că instrucțiuni de tip I doar: lw, sw și beq. Pentru lw și sw, rezultatul este stocat în $\$t$.
- Read data 1 ca output: citeste **valoarea** din registrul $\$s$.
- Read data 2 ca output: citeste **valoarea** din registrul $\$t$.
- Write data: valoarea care trebuie scrisă în Write Register.

2.2.5 Scrierea in memorie / citirea din memorie

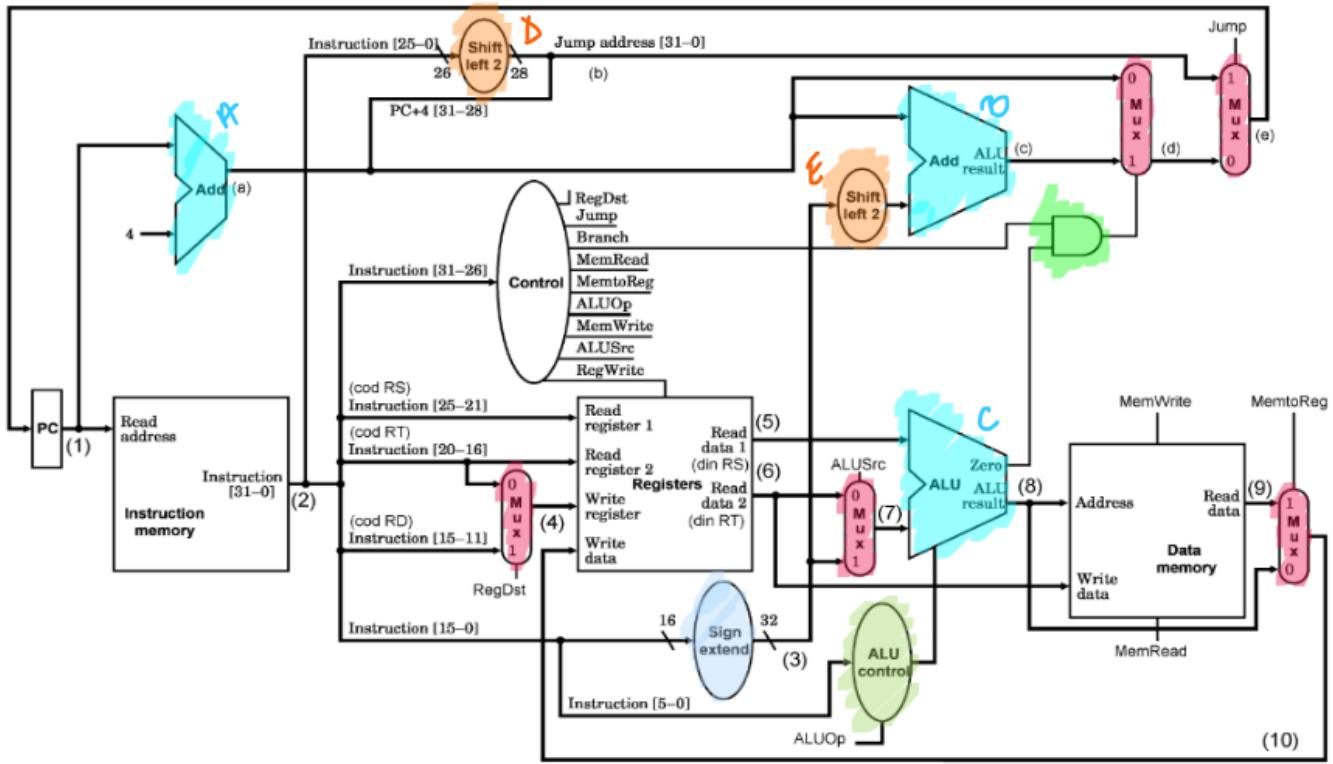


Pentru citirea/scrierea in memorie (aka lw si sw).

2.2.6 Executarea instructiunii

Inainte sa trecem la exemple, sa identificam restul de structuri din procesor:

- ALU control
- multiplexor
- shiftare pe biti la stanga
- extindere de semn
- unitate aritmetica si logica
- portata AMD



- ALU control: Primeste codul ALUOp si il transforma intr-o functie (operatie):

<u>ALU control input</u>	<u>Function</u>
000	AND
001	OR
010	add
110	subtract
111	set on less than

- Multiplexor elementar: primeste 2 input-uri X si Y notate cu 0 si 1 si in functie de valoarea de adevar input-ului de decizie Z scoate valoarea lui X sau a lui Y (pentru mai multe detalii vezi tutoriat 5)
- Shiftare pe biti la stanga cu 2: exemplu - 1010010 shiftat la stanga cu 2 este 10100100. Observam ca

initial avea 7 biti, acum are 9. De aceea, putem observa si la D-ul de pe desen(cu portocaliu), ca intra 26 de biti si ies 28.

- Extindere de semn: Din cardinalitatea care intra si iese din Sign extend observam ca: intra un cod pe 16 biti si iese unul pe 32 de biti.

Eu aveam un numar pe 16 biti si vreau sa il scriu pe 32. Cum fac? Ii extind bitul de semn in fata lui. Care era bitul de semn? Primul.

Deci pentru 10010001 pe 8 biti, extinderea sa la 16 biti este 111111110010001.

- Unitatea aritmetica si logica (ALU): este un circuit care aplica unor operanzi numere intregi pe n biti o operatie aritmetica sau logica selectata printr-un cod numeric.

Este exact ce vedem ca se intampla la C. Avem 2 input-uri, unul de la (5) si unul de la (7). Pe ele se aplica o operatie selectata printr-un cod numeric. Care cod numeric? ALUOp, care intra in ALU control. ALU control transforma acest cod intr-o functie, pe care o trimit ca input "de decizie" catre Unitatea aritmetica si logica.

exemplu: Vedem ca in tabel add are ALUOp=010. Functia in care il transforma ALU control este adunarea. Deci Unitatea aritmetica si logica va scoate ca rezultat (5)+(7).

Dar si B (cu albastru deschis) este este un ALU. Doar ca acum nu mai intra un input de decizie in el, ci scrie deja pe el Add. S-a decis deja ca acest ALU va face o adunare a celor 2 input-uri.

- Poarta AND: daca nu suntem familiarizati, vezi tutoriat 4.

2.2.7 Unitatea de control

Activity	Signal	Purpose
PC Update	Branch	Combined with a condition test boolean to enable loading the branch target address into the PC.
	Jump	Enables loading the jump target address into the PC (only appears in Figure 4.24 in Patterson and Hennessey).
Source Operand Fetch	ALUSrc	Selects the second source operand for the ALU (rt or sign-extended immediate field in Patterson and Hennessey).
ALU Operation	ALUOp	Either specifies the ALU operation to be performed or specifies that the operation should be determined from the function bits.
Memory Access	MemRead	Enables a memory read for load instructions.
	MemWrite	Enables a memory write for store instructions.
Register Write	RegWrite	Enables a write to one of the registers.
	RegDst	Determines how the destination register is specified (rt or rd in Patterson and Hennessey).
	MemtoReg	Determines where the value to be written comes from (ALU result or memory in Patterson and Hennessey).

Acum ca stim ce fac toate aceste structuri, trecem la exemple:

Exemplul 8

Considerăm implementarea procesorului MIPS cu 1 ciclu / instrucțiune (vezi verso). Fie fragmentul de program:

```
.data           1a $t3, x
x: .word 5          et:
.text          lw $t4, 0($t3)
main:          sub $t2, $t2, $t4
               beq $t2, $0, et
               li $t2, 5
```

b) Completă tabelul următor cu valorile obținute la prima executare a instrucțiunilor lw și beq din program; valorile se vor scrie hexa/formulă, iar dacă valoarea este necunoscută/nedefinită se va nota "?"; în coloanele PC și \$t2 se vor trece valorile noi, de la sfârșitul executării instrucțiunilor respective:

	1	4	5	7	8	ALU zero	10	(d)	(e)	Branch	Mem To Reg	ALU Op (2b)	ALU Ctrl (3b)	Reg Write	PC	\$t2	
Initial	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	α	5
lw	α																
beq																	

Incepem sa completam tabelul in certinta, folosindu-ne de datele din tabelele de ajutor.

	1	4	5	7	8	ALU zero	10	(d)	(e)	Branch	Mem To Reg	ALU Op (2b)	ALU Ctrl (3b)	Reg Write	PC	\$t2	
Initial	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	α	5
lw	α										0	1	00	010	1		
beq																	

Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

	ALUOp		Camp functie						Operatie
	ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
lw/sw	0	0	X	X	X	X	X	010	(+)
beq	X	1	X	X	X	X	X	110	(-)
add	1	X	X	X	0	0	0	010	(+)
sub	1	X	X	X	0	1	0	110	(-)
and	1	X	X	X	0	1	0	000	(and)
or	1	X	X	X	0	0	1	001	(or)
slt	1	X	X	X	1	0	1	111	(slt)

Am colorat cu aceeasi culoare perechile (valoare completata in tabelul din cerinta, de unde am luat acea valoare).

Acum ca am completat campurile pe care le putem lua direct din tabelele de ajutor, incepem "executia". Identificam valorile registrilor.

Reamintim ca lw are forma: lw \$t, offset(\$s)

- \$s=11

- \$t=12

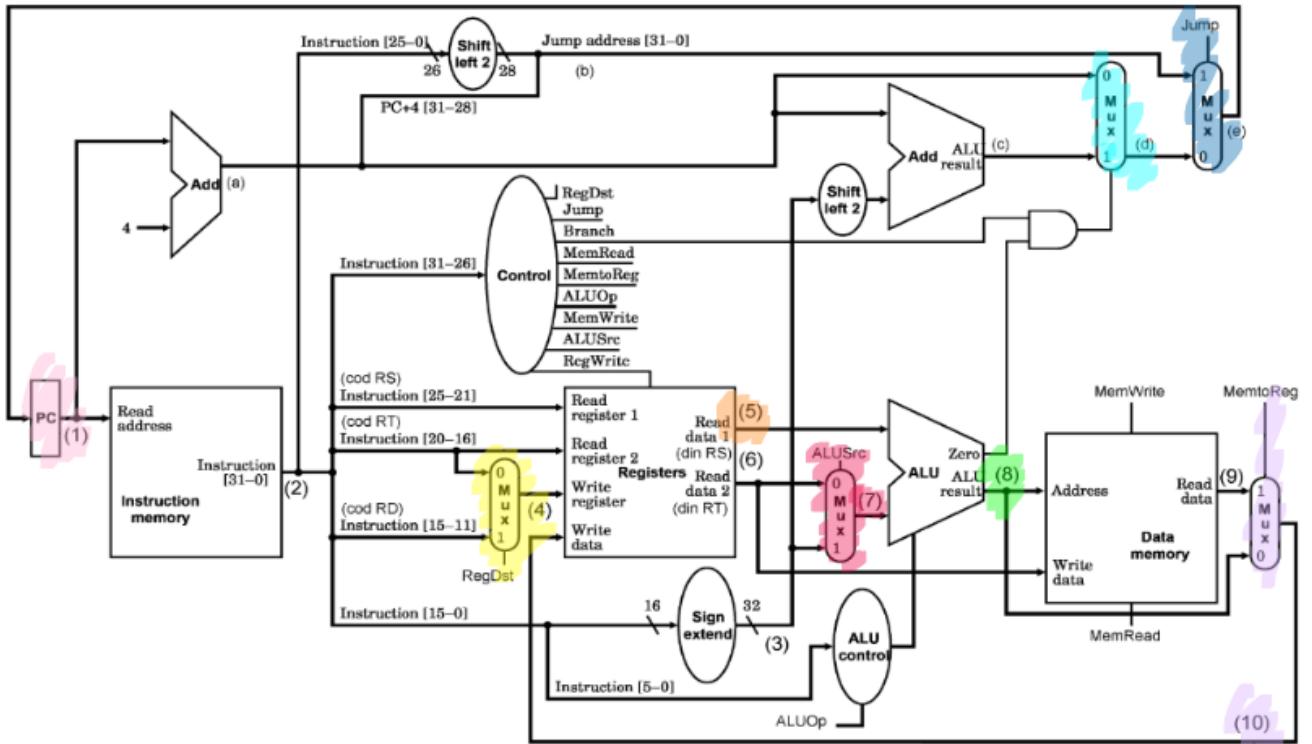
- offset=0

Mai departe, identificam ce valori se află în acești registri.

Aveți un x care initial este 5, \$t2 primește valoarea 5. \$t3 primește ca valoarea adresa lui x . \$t4 primește valoarea de la "punctul 0" din \$t3, adică valoarea lui $x \Rightarrow$

- valoarea din \$s este adresa de memorie a lui x pe care o notăm cu β
- valoarea din \$t este 5

	1	4	5	7	8	ALU zero	(d)	(e)	Branch	Mem To Reg	ALU Op (2b)	ALU Ctrl (3b)	Reg Write	PC	\$t2
Initial	—	—	—	—	—	—	—	—	—	—	—	—	—	α	5
lw	α	12	β	0	β	?	5	$\alpha + \beta$	$\alpha + \beta$	0	1	00	010	1	$\alpha + \beta$ 5
beq															



Să parcurgem ce am completat și de ce:

- (1): era deja completată cu α . 1 este de fapt valoarea din PC care indică cadrul instrucțiunii curente. Nu stim la ce adresa de memorie se află instrucțiunea curentă, astăzi este notată cu α .

- (4): Valoarea din 4iese dintr-un multiplexor. Input-ul de decizie este RegDst. Ne uitam in tabelele de ajutor sa vedem care este valoarea lui RegDst pentru lw. Observam ca valoarea este 0. \Rightarrow (4)=ce intra in dreptul lui 0 in multiplexor. Deasupra la "Instruction [20-16]" scrie "(cod RT)" care ne indica ca valoarea de la Instruction [20-16] este chiar codul lui \$t, adica 12. \Rightarrow valoarea lui (4) este 12.
- (5): Ne amintim de la sectiunea Identificarea tipului de instructiune ca Read data 1 ne citeste **valoarea** stocata in registrul \$s (nu codul registrului, valoarea din el). Asa cum am vazut mai sus, valoarea din \$s este adresa de memorie a lui x pe care nu avem de unde sa o stim, asa ca o notam cu β .
- (7): Avem din nou un multiplexor care depinde de data aceasta de ALUSrc. Ne uitam in tabelele de ajutor care este valoarea lui ALUSrc pentru lw. ALUSrc=1 \Rightarrow Iese din multiplexor de a intrat pe la 1.
Ce a intrat pe la 1? [Instruction 15-0], care este chiar offset in instructiunile de tip I, caruia i s-a aplicat un sign extend. offset avea valoarea 0 (pe 16 biti) \Rightarrow in urma lui sign extend avem valoarea 0 pe 32 de biti.
- (8): Avem ALU (Arithmetic and Logical Unit). Ce operatie efectueaza? Trebuie sa ne uitam la ce iese din ALUcontrol, adica ne uitam in tabela de ajutor la ALUCtrl pentru lw. Vedem ca operatia este adunare. Deci se face adunarea intre valorile din (5) si (7). Deci rezultatul este β .
- ALU zero = 0, daca valoarea din (8) \neq 0
1, daca valoarea din (8) = 0
Noi nu stim exact valoarea din (8), sim doar notatia ei, β , deci nu putem sa stim care este valoarea lui ALU zero.
- (10): Ce iese din multiplexorul a carui valoare este determinata de MemToReg. Pentru lw MemToReg are valoarea 1 \Rightarrow Iese ce intra la Read data. Ce intra la Read data? Daca ne amintim ca lw citeste ceva de la o adresa de memorie si pune acea valoarea intr-un registru, lucrurile sunt destul de clare. Deci in (10) vom avea valoarea citita la 0(\$t3), adica valoarea lui x, adica 5.
- (d): Rezultatul este determinat de o poarta AND intre Branch si ALUzero. Branch stim sigur ca are valoarea 0 pentru lw, deci si rezultatul portii va fi 0. Deci in (d) avem valoarea care intra in multiplexor la 0, care este α la care se adauga un 4 prin adder. \Rightarrow (d)= $\alpha+4$
- (e): Un multiplexor determinat de valoarea lui Jump. Jump pentru lw e 0 \Rightarrow iese ce intra din (d) \Rightarrow $\alpha + 4$
- PC: Observam ca im PC intra fix ce a iesit din (e) \Rightarrow PC= $\alpha+4$
- \$t2: Trebuie sa ne intoarcem putin la codul de MIPS si sa ne amintim ce se intampla acolo. \$t2 primește valoarea 5. Apoi avem peratia de lw pe \$t3 si \$t4, deci nu au treabă cu \$t2. Deci valoarea lui \$t2 ramane aceeași.

Incepem sa completam si randul pentru beq, incepand cu valorile pe care le luam direct din tabela de adevar.

	1	4	5	7	8	ALU zero	10	(d)	(e)	Branch	Mem To Reg	ALU Op (2b)	ALU Ctrl (3b)	Reg Write	PC	St2
Initial	—	—	—	—	—	—	—	—	—	—	—	—	—	—	α	5
lw	α									0	1	00 010	1			
beq										1	X	X 1 110	0			

Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp	ALUCtrl	RegWrite	PC	St2
R-format	1	0	0	1	0	0	0	010	000	0		
lw	0	1	1	1	1	0	0	010	000	1		
sw	X	1	X	0	0	1	0	010	000	1		
beq	X	0	X	0	0	0	1	010	000	1		

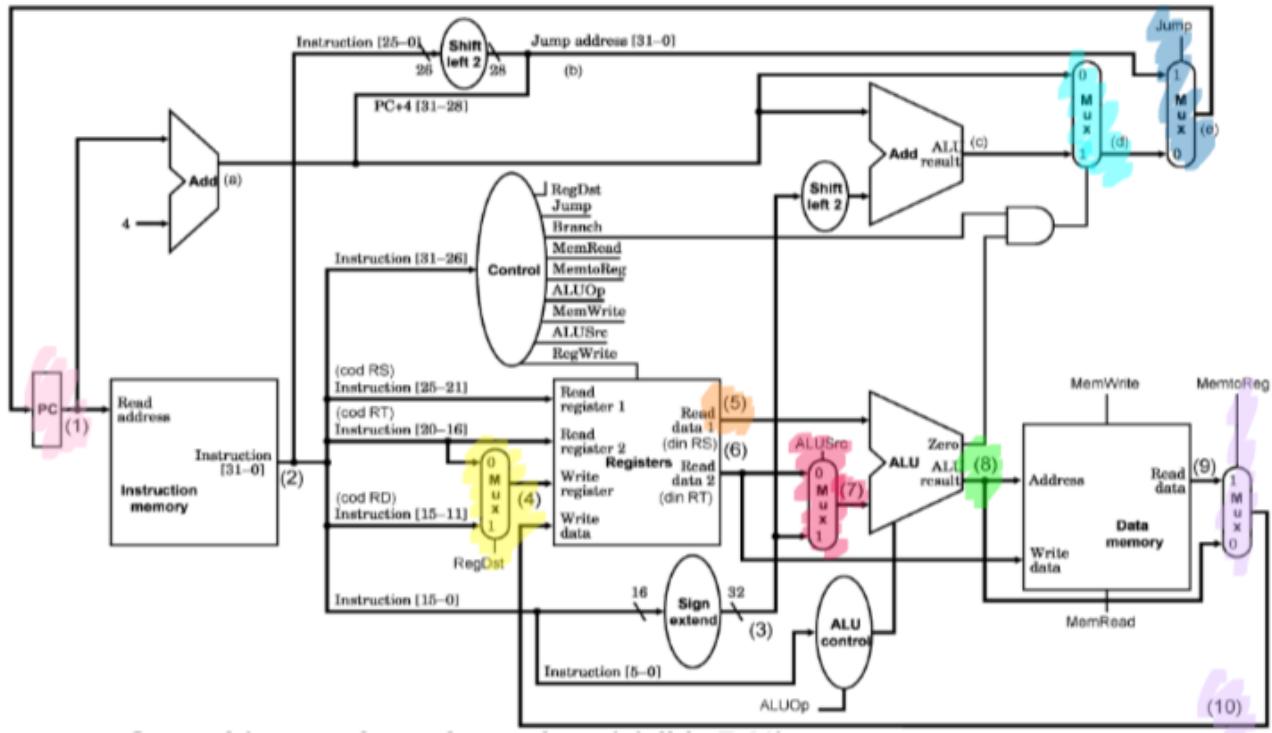
ALU Control (slide 7.36)

lw/sw beq add sub and R- format slt	ALUOp		Camp functie						Operatie	
	ALUOp ₁	ALUOp ₀	F5	F4	F3	F2	F1	F0		
	0	0	X	X	X	X	X	X	010	(+)
beq	X	1	X	X	X	X	X	X	110	(-)
add	1	X	X	X	0	0	0	0	010	(+)
sub	1	X	X	X	0	0	1	0	110	(-)
and	1	X	X	X	0	1	0	0	000	(and)
or	1	X	X	X	0	1	0	1	001	(or)
R-format	1	X	X	X	1	0	1	0	111	(slt)

Continuam cu restul valorilor:

$\{0, \text{daca } (8) \neq 0\}$
 $\{1, \text{daca } (8) = 0\}$

	1	4	5	7	8	ALU zero	10	(d)	(e)	Branch	Mem To Reg	ALU Op (2b)	ALU Ctrl (3b)	Reg Write	PC	\$t2
Initial	—	—	—	—	—	—	—	—	—	—	—	—	—	—	α	5
lw	α	12	7	0	7	?	5	$\alpha+4$	$\alpha+4$	0	1	00	010	1	$\alpha+4$	5
beq	$\alpha+4$?	0	0	1	1	?	$\alpha+4$	$\alpha+4$	1	X	X1	110	0	$\alpha+4$	0



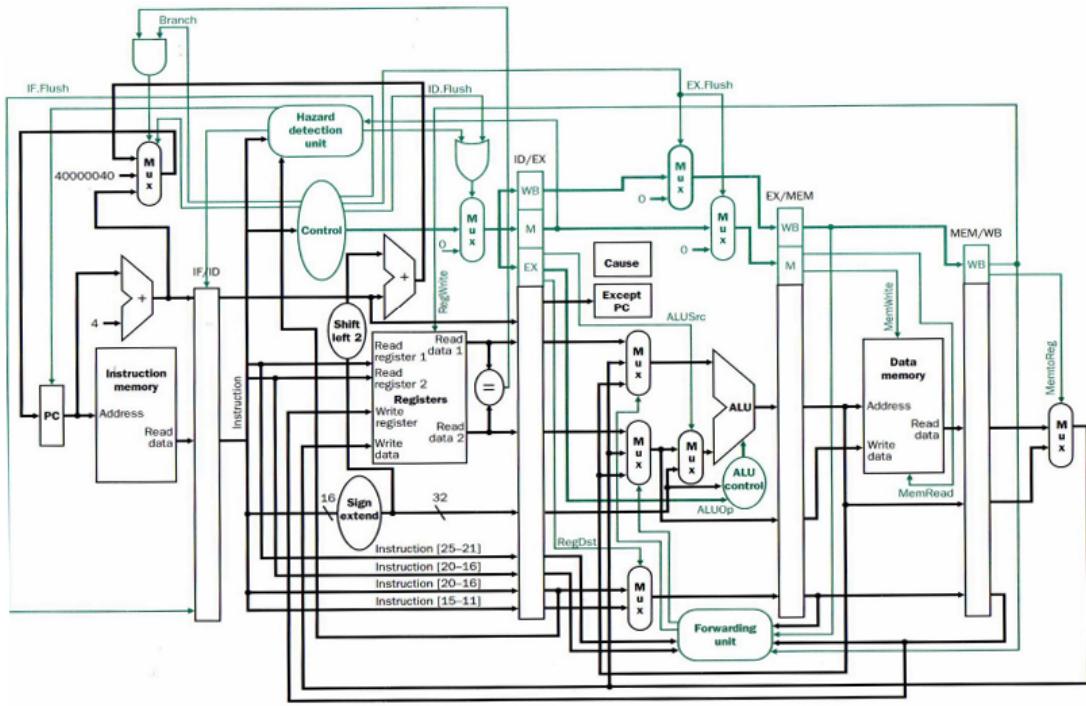
Exemplul 10 De ce spunem ca instructiunile de tip I pot sa ruleze 2 instructiuni in loc de una?
 Luam ca exemplu:

li \$t0, 1234567890

Transformarea lui 1234567890 in baza 2 este: 1001001100101100000001011010010. Stim ca instructiunile de tip I au 16 biti la dispozitie pentru a tine valoarea immediate. \Rightarrow numarul nostru nu incape in imm.

Deci procesorul va face 2 operatii:

lui \$t0, 0x4996
 ori \$t0, \$t0, 0x2d2



**THE MIPS ARCHITECTURE OR
SOMETHING, I DON'T KNOW,
PLEASE WISHLIST AND FOLLOW
PALMRIDE ON STEAM.**

References

- [1] Dumitru Daniel Drăgulici. *Curs Arhitectura Sistemelor de Calcul*.
- [2] Larisa Dumitrache. *Tutoriat 2019*
- [3] Bogdan Macovei. *Laboratoare ASC 2019/ 2020*
- [4] Ben Eater. *video-uri YouTube*

Tutoriat 6

Stan Bianca-Mihaela, Stăncioiu Silviu

November 2020

**PROFI DE MATE DUPA CE SCRII PE
FOAIA DE EXAMEN CUM COVRIGII SUNT
PRODUSE DE PANIFICATIE, IAR APOI
DESCRII METAFIZIC NATURA
PARADOXALA A UNIVERSULUI PENTRU
CA AI UITAT CA DAI EXAMEN LA MATE,
NU LA ASC.**





TUTORIATE SCRISE FOARTE
FORMAL, CU BIBLIOGRAFIE
ADEVARATA SI VERIFICATE
DE MULTE ORI



TUTORIATE NEVERIFICATE,
PLINE DE TYPO-URI,
DEZACORDURI SI
MEME-URI

Contents

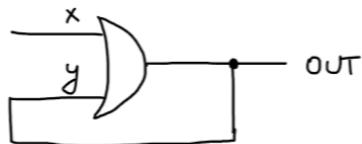
1	1-DS (Memorii)	3
1.1	Zavoare	3
1.1.1	Zavorul elementar	3
1.1.2	Zavorul elementar eterogen (SR latch)	4
1.2	Delay Flip-Flop (DFF)	6
2	Arhitectura MIPS	11
2.1	Reprezentarea internă a instrucțiunilor procesorului MIPS	12
2.2	Procesorul MIPS cu un ciclu	19
2.2.1	PC	20
2.2.2	Citirea instructiunii din memorie	21
2.2.3	Identificarea tipului de instructiune	22
2.2.4	Citirea registrilor	23
2.2.5	Scrierea in memorie / citirea din memorie	24
2.2.6	Executarea instructiunii	25
2.2.7	Unitatea de control	26

1 1-DS (Memorii)

Sistemele 1-DS sunt sisteme 0-DS inchise printr-un ciclu.

1.1 Zavoare

1.1.1 Zavorul elementar



x	y	$x+y$
0	0	0
0	1	1
1	0	1
1	1	1

Mai sus avem reprezentarea unui zavor elementar, impreuna cu tabelul de adevar pentru OR.

Sa spunem ca avem un circuit prin care, la apasarea unui switch, eu pot sa ii schimb valoarea de adevar a lui x.

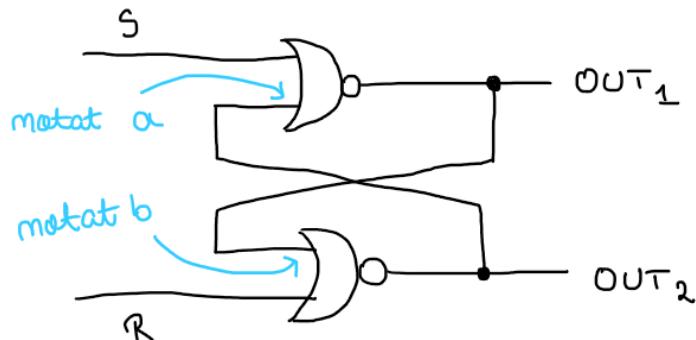
- Conectez circuitul la curent. Ambele valori, x si y, sunt 0. \Rightarrow OUT are valoarea 0.
- Ce observam? OUT isi transmite valoarea inapoi catre y. Deci avem o bucla infinita: atata timp cat x ramane 0, y il face pe OUT sa fie 0, iar OUT il face la randul lui pe y sa fie 0.
- Apas pe switch. Ce am zis ca face switch-ul? Schimba valoarea de adevar a lui x. \Rightarrow Acum valoarea de adevar a lui x este 1. Ce se intampla cu valoarea lui OUT? Valorile lui x si y trec prin poarta OR. Stim ca $1 \text{ OR } 0 = 1$. \Rightarrow valoarea lui OUT va fi 1.
- Stim ca OUT isi transmite valoarea sa inapoi catre y. Deci ce se intampla? y devine 1. \Rightarrow poarta OR primeste 1 si 1 \Rightarrow OUT e in continuare 1. Deci avem din nou o bucla infinita.
- Apas din nou pe switch. \Rightarrow valoarea lui x devine 0. \Rightarrow Poarta OR primeste un 0 si un 1 deci OUT=1. OUT isi transmite valoarea catre y si rezultatul ramane acelasi. Din nou am o bucla infinita. Mai mult decat atat, observ ca ori de cate ori as apasa eu switch-ul de acum incolo, OUT va fi tot 1.
- Singura solutie sa il fac pe OUT 0 este sa deconectez circuitul de la curent.

Am vazut deci cum functioneaza un zavor elementar.

Pentru demonstratia fizica vezi: <https://www.youtube.com/watch?v=KM0DdEaY5sY>

Am vrea acum sa avem o modalitate sa il facem pe OUT 0 oricand vrem noi. Cu acest scop in minte, introducem zavorul elementar eterogen.

1.1.2 Zavorul elementar eterogen (SR latch)



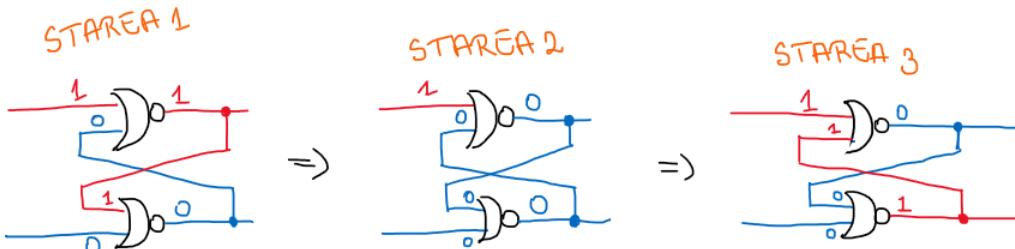
x	y	x NOR y
0	0	1
0	1	0
1	0	0
1	1	0

Avem mai sus schema unui zavor elementar eterogen, impreuna cu tabelul de adevar pentru NOR.

Acum avem 2 switch-uri: unul pentru S si unul pentru R.

Sa o luam din nou pe pasi:

- Conectam circuitul la curent. Pentru prima poarta NOR, ambele input-uri vor fi 0 \Rightarrow OUT₁ va fi 1.
- Pentru ca OUT₁ este 1, intrarea notata de mine cu b va fi tot 1 (OUT₂ isi propaga valoarea in b). R va fi 0 (inca nu am apasat pe switch). \Rightarrow OUT₂ = 0. Acest OUT₂ este propagat catre a, care era tot 0 inainte deci nu isi schimba valoarea. Avem din nou o bucla infinita.
- Apas pe switch-ul lui R. \Rightarrow valoarea lui R devine 1. Din tabel vedem ca 1 NOR 1 = 0 deci nu se schimba nimic pentru OUT₁ si OUT₂. Asa ca pot sa apas switch-ul lui R de oricate ori fara a schimba rezultatele. Mai apas o data ca sa il fac pe R din nou 0.
- Apas pe switch-ul lui S. Ce se intampla?



STAREA 1: S devine 1.

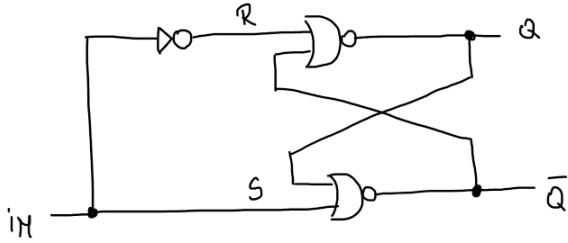
STAREA 2: 1 NOR 0 = 0. \Rightarrow OUT₁ = 0, OUT₁ isi transmite valoarea catre b, deci si b=0.

STAREA 3: In a doua poarta NOR: 0 NOR 0 = 1 \Rightarrow OUT₂ = 1, OUT₂ isi transmite valoarea catre a, deci a=1. \Rightarrow In prima poarta NOR: 1 NOR 1 = 0. \Rightarrow OUT₁ nu isi schimba valoarea si ne-m intors la o bucla infinita.

- In final, vedem ca am schimbat valorile de output. Initial aveam OUT₁ = 1 si OUT₂ = 0, acum avem OUT₁ = 0 si OUT₂ = 1
- Analog, ca sa schimbam din nou outputurile, acum trebuie sa apasam pe switch-ul lui R.

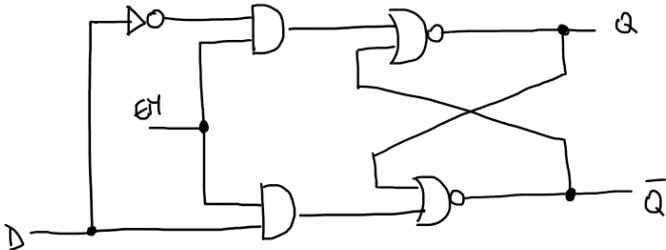
Observatie ! Mereu OUT₁ si OUT₂ au valori complementare. De aceea, ele se noteaza cu OUT₁ = Q si OUT₂ = \bar{Q} .

Putem acum sa facem o mica simplificare. Vrem sa avem un singut input, nu doua. Am vazut ca, desi aveam doua input-uri, la orice moment de timp unul dintre ele nu facea nimic, ori de cate ori ii schimbam valoarea. Asa ca putem transforma circuitul in:



Ce am reusit sa facem acum? Daca avem un switch prin care putem schimba valoarea lui IN, de fiecare data cand apasam switch-ul, Q si \bar{Q} isi schimba valorile. Daca initial $Q=0$ si $\bar{Q}=1$, dupa apasarea switch-ului $Q=1$ si $\bar{Q}=0$.

Acum, nu ne place ca noi putem schimba valoarea lui Q (si implicit si a lui \bar{Q}) oricand. Vrem sa avem un enabler care sa imi spuna cand am voie sa schimb valorile si cand nu. Schema care rezolva aceasta problema este:



Acum putem sa schimbam valoarea lui Q doar daca $EN=1$. Acest circuit obtinut poarta numele de D-latch. Pentru demonstratia cu circuite vezi: <https://www.youtube.com/watch?v=peCh859q7Qt> = 26s

Me: Mom, can I have



?

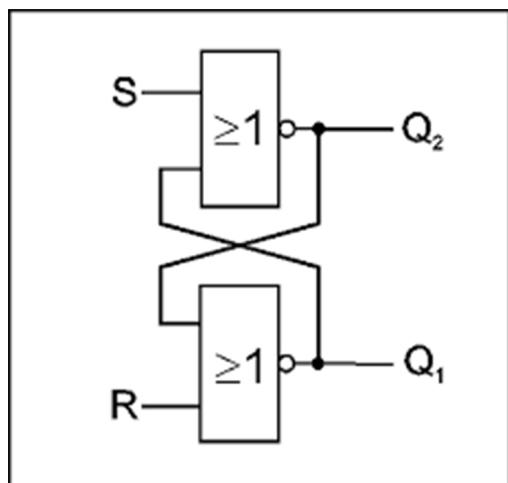
Mom: No we have



at home

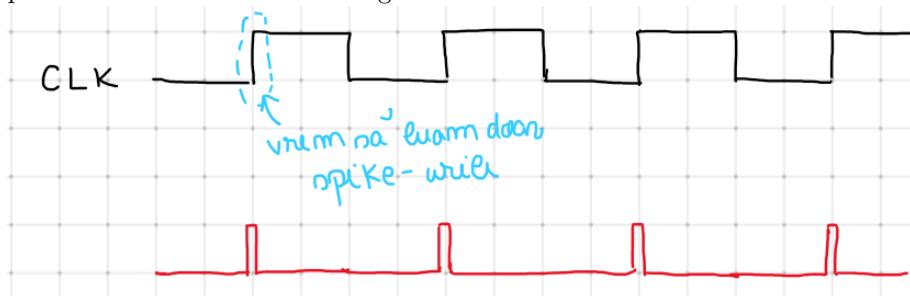


at home:

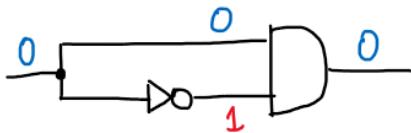


1.2 Delay Flip-Flop (DFF)

Vrem sa inlocuim enable-ul de mai devreme cu un clock CLK. Un clock este un circuit care isi schimba periodic valoarea de la low la high.

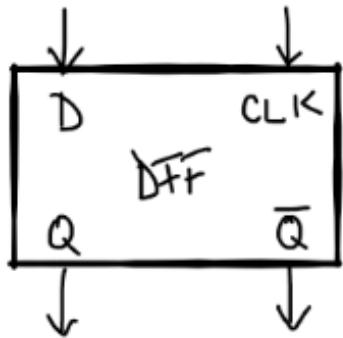


Ne intereseaza sa obtinem din acest clock un circuit care sa ne izoleze acele spike-uri de la high la low. Acest circuit pe care il cautam noi se numeste edge detector. Un exemplu de un astfel de circuit este:



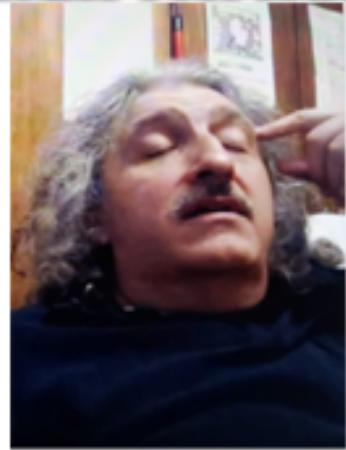
Observam starea in care este acum edge detector-ul, cu valorile scrise pe fiecare ramura. Daca input-ul isi schimba valoarea, pentru un timp foarte scurt, ramura de sus o sa fie 1, iar ramura de jos isi face nagatia putin mai greu si va ramane tot 1. Deci, pentru un timp foarte foarte scurt, ambele vor avea valoarea 1 deci output-ul va fi 1. Am obtinut astfel un edge detector.

D Flip-Flop este un D-latch ca mai sus, doar ca un loc de EN are un edge detector conectat la CL. Simbolul lui este:



Pentru mai multe detalii: <https://www.youtube.com/watch?v=YW-GkUguMM>

YOUR CRUSH **HER EX** **HER FATHER**



HER BROTHER



HER MOTHER



YOU

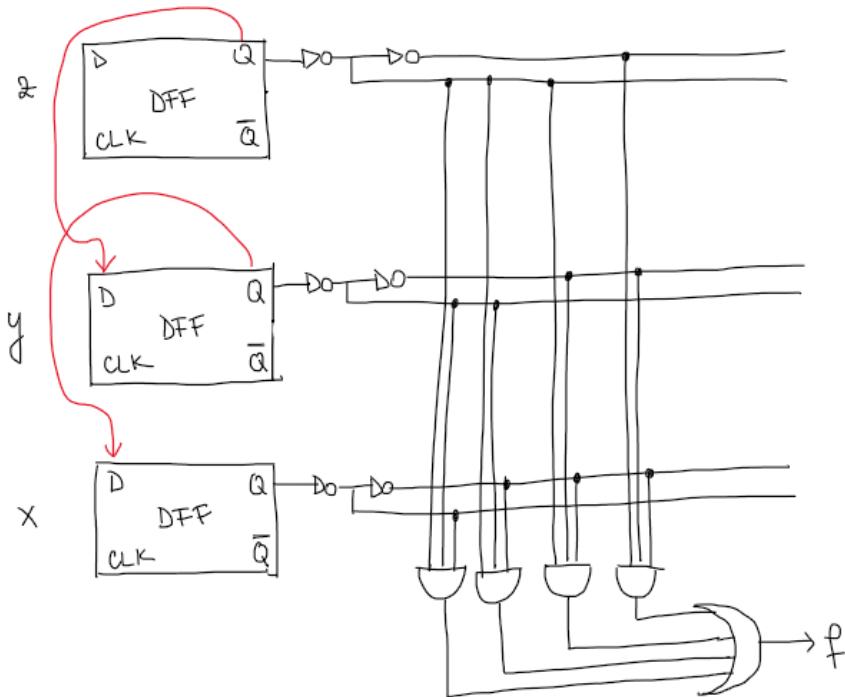


CAND ESTI IN MINECRAFT SI
CONSTRUIESTI UN D FLIP
FLOP IN LOC SA PUI UN
SIMPLU REPEATER



Exemplul 1 Construiti un circuit 1-DS care citeste unul cate unul o secventa de biti si de fiecare data scoate 1 daca $x \leq y \leq z$.

Ne amintim ca am facut deja tabelul acestei functii in tutoriatul 4 si stim ca in FND-ul sau apar liniile (0), (1), (3) si (7).



2 Arhitectura MIPS



2.1 Reprezentarea internă a instrucțiunilor procesorului MIPS

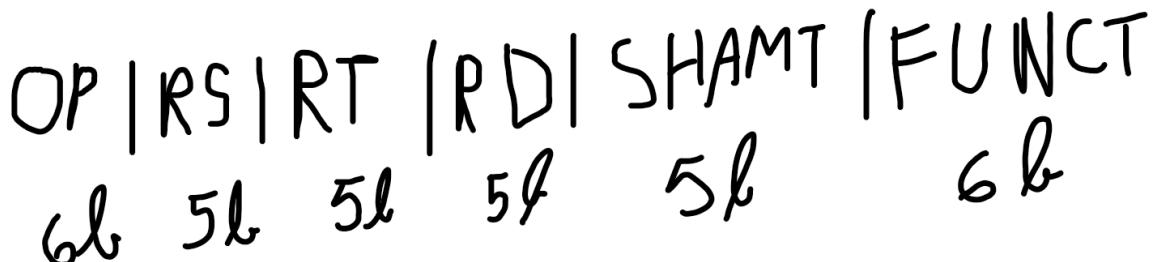
În MIPS avem 3 clase de instrucțiuni, și anume:

- R
 - Operații aritmetice, care nu utilizează valori imediate (de exemplu: add \$t0, \$t1, \$t2. Contraexemple: addi \$t0, 1 nu este de tip R deoarece se folosesc valori imediate)
 - Set: slt, seq, sre, ...
 - Shiftări logice: sll, slr
- I
 - Operații cu valori constante indicate: load, store, branch conditions (ble, blt, ...)
- J
 - jumps (j, jal)

Fiecare instrucțiune poate fi reprezentată în binar pe 32 de biți. Biții din reprezentarea binară a instrucțiunii se deduc din clasa de instrucțiuni din care face parte, regiștrii implicați și valorile constante implicate. Astfel, avem următoarele reprezentări pentru:

Clasa de instrucțiuni R:

Reprezentarea binară a unei instrucțiuni de tip R este compusă din mai multe secțiuni, și anume: op, rs, rt, rd, shamt și func. op și func ocupă 6 biți, iar restul ocupă 5 biți.



Când vrem să reprezentăm o instrucțiune în binar, putem deduce fiecare secțiune din ea în felul următor:

- op - În clasa R este mereu 000000.
- rs - Registrul sursă 1 (adică codul registrului reprezentat în binar. Fiecare registru are un număr. De exemplu registrul: \$t0 este echivalent cu registrul \$8. Găsiți în tabel codurile fiecărui registru).
- rt - Registrul sursă 2.
- rd - Registrul destinație.
- shamt - Shift amount. Se completează ≠ 0 când se face shiftare.
- func - În pereche cu op se decide ce instrucțiune MIPS se aplică. (găsim în tabel valorile pentru func care ne trebuie)

Exemplul 2

add \$t0, \$t1, \$t2

op → 000000 (pentru că instrucțiunea ∈ R)

rs → \$t1 = \$9 = 01001 (registru \$t1 are valoarea 9, again, găsim în tabel valorile)

rt → \$t2 = \$10 = 01010

rd → \$t0 = \$t8 = 01000

func → 100000 (este dat)

Acum vom pune la un loc toate valorile obținute, deci vom avea (le-am grupat direct în bucăți de câte 4 pentru a ne fi ușor după să le transformăm în baza 16):

0000 0001 0010 1010 0100 0000 0010 0000

Acum vom transofma rezultatul obținut în baza 16 (exact cum făceam și în primele 2 tutoriate), deci vom avea:

0x012A4020

Exemplul 3

sll \$s1, \$v0, 2 (este instrucțiune de tip R, iar $func = 000000$)

op → 000000 (instrucțiunea ∈ R)

rs → 00000 (nu se completează aici deoarece avem un singur regisztru sursă)

rt → 00010 (codul pentru regisztrul \$v0)

rd → \$s1 = \$17 = 10001

shamt → 2 = 00010

func → 000000

Reprezentarea în baza 2 este:

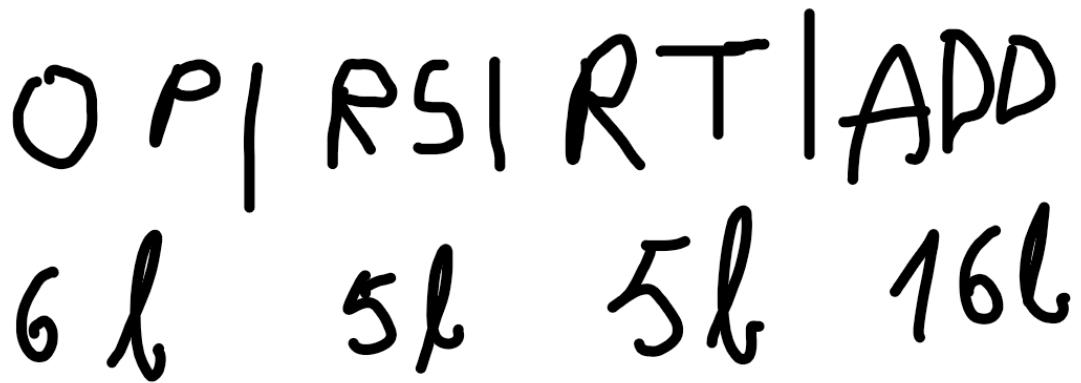
0000 0000 0000 0010 1000 1000 1000 0000

Iar în baza 16 avem:

0x00028880

Clasa de instrucțiuni I:

Secțiunile pentru aceste instrucțiuni sunt: op, rs, rt și add/ imm. Op ocupă 6 biți, rs și rt ocupă 5 biți, iar add/ imm restul de 16 biți.



Avem:

- op - Operația, avem în tabel codificarea binară.
- rs - Registru sursă.
- rt - Registru sursă/ destinație după caz.
- add/ imm - Câmp de adresă/ valoare imm, după caz.

Exemplul 4

lw \$t0, 4(\$t2)

$\text{op} \rightarrow 100011$
 $\text{rs} \rightarrow \$t2 = \$10 = 01010$
 $\text{rt} \rightarrow \$t0 = \$8 = 01000$
 $\text{imm} \rightarrow = 0000000000000100$

Exemplul 5

beq \$t1, \$s0, 28

$\text{op} \rightarrow 000100$
 $\text{rs} \rightarrow \$t1 = \$9 = 01001$
 $\text{rt} \rightarrow \$s0 = \$16 = 10000$
 $\text{add} \rightarrow 28 \sim \frac{28}{4} = 7 = 0000000000000111$

Clasa de instrucțiuni J:

Aici secțiunile sunt doar op (6 biți) și add (26 de biți).

OP | ADA
6l 26l

Exemplul 6

j 28

Tabelul cu valorile regiștrilor:

\$zero:0	\$t7:15	\$gp:28
\$at:1	\$s0:16	\$sp:29
\$v0:2	\$s1:17	\$fp:30
\$v1:3	\$s2:18	\$ra:31
\$a0:4	\$s3:19	
\$a1:5	\$s4:20	
\$a2:6	\$s5:21	
\$a3:7	\$s6:22	
\$t0:8	\$s7:23	
\$t1:9	\$t8:24	
\$t2:10	\$t9:25	
\$t3:11	\$k0:26	
\$t4:12	\$k1:27	
\$t5:13		
\$t6:14		

Imaginea de mai sus este extrasă din tutoriatul de ASC tinut anul trecut de Larisa Dumitracă.

La examen, pentru a găsi *op* și *func* pentru anumite instrucțiuni, folosiți acest link (se va descărca un pdf - am observat că au sters pagina catre care ducea link-ul pus de Silviu):
Instrucțiuni MIPS.

Folosiți-vă de secțiunea encoding a instrucțiunilor pentru a deduce cum să completați reprezentarea binară a instrucțiunilor.

Alternativ, folosiți-vă de acest link unde este scris direct codul pentru op/ func:

<http://alumni.cs.ucr.edu/~vladimir/cs161/mips.html>

Exemplu:

Exemplul 7 [Restanță MAI 2020]

Considerăm implementarea procesorului MIPS cu 1 ciclu / instrucțiune (vezi verso). Fie fragmentul de program:

```
1 li $t1 , 2
2 li $t2 , 3
3 li $t3 , 2
4 et :
5 add $t1 , $t2 , $t1
6 sub $t3 , $t1 , $t3
7 beq $t3 , $t2 , et
```

prog_mips.s

side note, la examen nu veți primi un program cu syntax-highlighting, duuh



Presupunem că în memorie instrucțiunea *sub* din program are adresa α .

a) Pentru instrucțiunile *sub* și *beq* din program scrieți câmpurile din reprezentarea lor internă (ex: op/rs/rt/imm, valorile se scriu hexa); pentru *beq* din program scrieți reprezentările ei binară (32 biți) și hexa (8 cifre hexa).

Rezolvare:

Observăm că *sub* face parte din clasa R de instrucțiuni, deci avem:

op \rightarrow 000000, adică 0 în hexa. (cum ne cere exercițiul)

rs \rightarrow \$t1 = \$9 = 01001, adică 9 în hexa.

rt \rightarrow \$t3 = \$11 = 01011, adică B în hexa.

rd \rightarrow \$t3 = \$11 = 01011, adică B în hexa.

func \rightarrow 100010, adică 22 în hexa.

Observăm că *beq* face parte din clasa I de instrucțiuni, deci avem:

op → 000100, adică 4 în hexa.

rs → \$t3 = \$11 = 01011, adică B în hexa.

rt → \$t2 = \$10 = 01010, adică A în hexa.

CUM FACEM SĂ AFLĂM VALOAREA LUI et? Noi mai devreme am văzut cum se codifică beq când avem o constantă multiplu de 4, dar acum cum facem...?

Trebuie să stim că beq nu aşteaptă o adresă de memorie absolută din program la care să facă jumpul, ci mai degrabă, un offset de la poziția lui către poziția unde vrem să facă jump-ul. Trebuie să ținem minte că atunci când se execută instrucțiunea beq, PC (un fel de registru care ține poziția curentă a instrucțiunii care se execută) are valoarea adresei instrucțiunii următoare lui beq. De aceea pentru a calcula poziția relativă trebuie să facem: $\frac{(\text{adresă_etichetă} - \text{adresă_instructiune_branch})}{4}$. Observăm că această formulă ne dă numărul de instrucțiuni dintre eticheta și beq (inclusiv) în cazul în care label-ul este înaintea lui et. Observăm că rezultatul este negativ în acest caz, deci îl vom codifica ca număr signed. În cazul în care eticheta se află după beq, formula ne va da offset-ul dintre instrucțiunea de după beq și instrucțiunea din dreptul label-ului. Iar în cazul în care label-ul este pus fix peste beq, atunci am avea -1.

Ne vom imagina că înainte de fiecare instrucțiune din următorul program avem un label, deci relativ la beq vom avea valorile din comentarii:

```
1 .data
2 .text
3 main:
4
5 li $t0 , 0          # -5
6 li $t1 , 0          # -4
7
8 li $a0 , 100        # -3
9 li $v0 , 1          # -2
10
11 beq $t0 , $t1 , et # -1
12 et:
13 syscall             # 0
14
15 li $v0 , 10         # 1
16 syscall             # 2
```

test_instr.s

Acestea fiind spuse, observăm că et este înaintea lui beq în problema noastră, deci vom lua numărul de instrucțiuni dintre et și beq (inclusiv), adică 3 instrucțiuni. Eticheta fiind înainte de beq, avem -3 pentru add, deci:

add → -3 = 111111111111101, adică FFFD în hexa

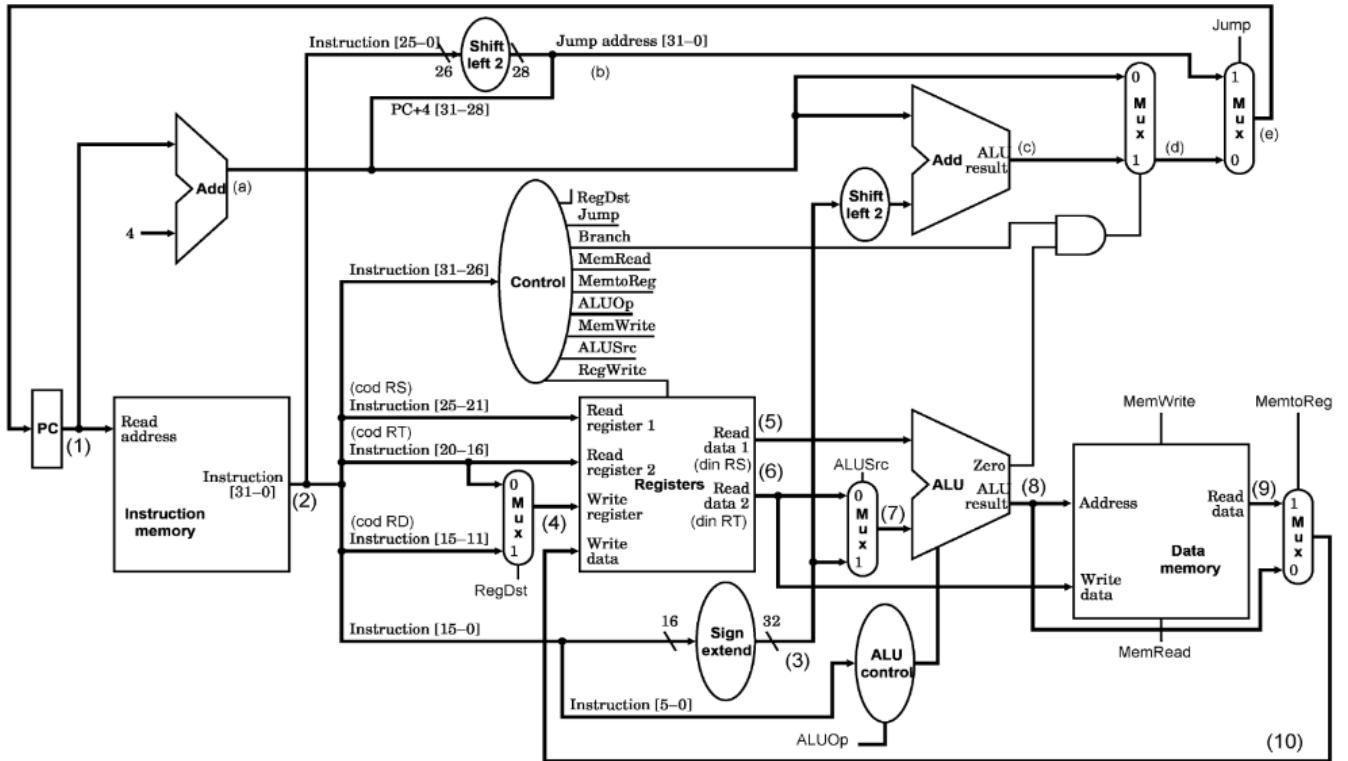
Pentru beq problema ne cere să scriem și reprezentarea binară și reprezentarea în hexa, deci vom avea:

0001 0001 0110 1010 1111 1111 1111 1101 (în binar)

0x116AFFFD (în hexa)

2.2 Procesorul MIPS cu un ciclu

Aşa arată procesorul MIPS cu un ciclu:



Inainte să aruncați laptopul pe fereastra, să ne gândim puțin ce face acest procesor:

- Face FETCH pentru o instructiune din memoria de instructiuni.
- Decodifica instructiunea: vream sa stim daca e ADD sau SUB, daca e de tip I sau R, etc.
- Citeste operanzii din registrii (\$rs, \$rd, op, etc.).
- Executa instructiunea.
- Scrie inapoi in memorie.

Voi mentiona acum si tabelele de ajutor, de care ne vom folosi pe tot parcursul subiectului 3:

Jignat complet
 Jignirea 2
 cu game

Instruction	RegDst	ALUSrc	MemtoReg	Reg Write	Mem Read	Mem Write	Branch
R-format	1	0	0	1	0	0	0
lw	0	1	1	1	1	0	0
sw	X	1	X	0	0	1	0
beq	X	0	X	0	0	0	1