

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

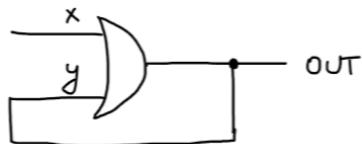
| | | |
|----------|---|-----------|
| 1 | 1-DS (Memorii) | 3 |
| 1.1 | Zavoare | 3 |
| 1.1.1 | Zavorul elementar | 3 |
| 1.1.2 | Zavorul elementar eterogen (SR latch) | 4 |
| 1.2 | Delay Flip-Flop (DFF) | 6 |
| 2 | Arhitectura MIPS | 11 |
| 2.1 | Reprezentarea internă a instrucțiunilor procesorului 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.

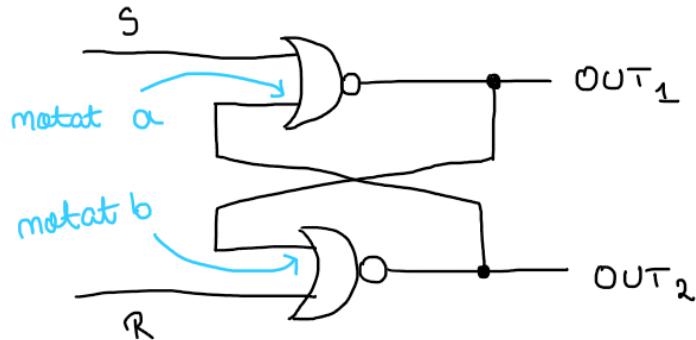
- Conectez circuitul la curent. Ambele valori, x si y, sunt 0. \Rightarrow OUT are valoarea 0.
- Ce observam? OUT isi transmite valoarea inapoi catre y. Deci avem o bucla infinita: atata timp cat x ramane 0, y il face pe OUT sa fie 0, iar OUT il face la randul lui pe y sa fie 0.
- Apas pe switch. Ce 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)



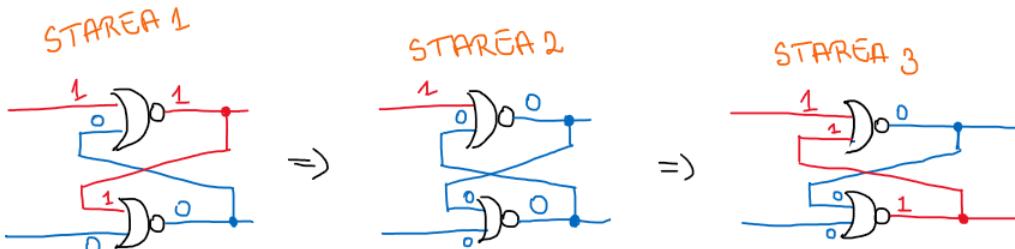
| x | y | x NOR y |
|---|---|---------|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

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.

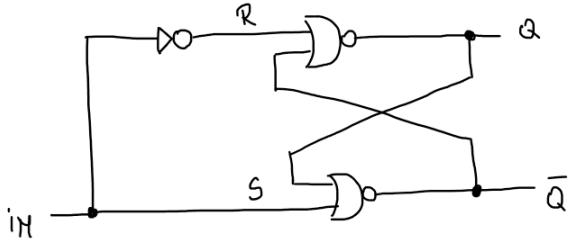
STAREA 2: 1 NOR 0 = 0. \Rightarrow OUT₁ = 0, OUT₁ isi transmite valoarea catre b, deci si b=0.

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.

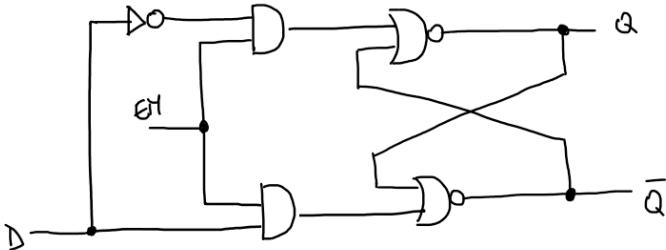
Observatie ! Mereu OUT₁ si OUT₂ au valori complementare. De aceea, ele se noteaza cu OUT₁ = Q si OUT₂ = \bar{Q} .

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



?

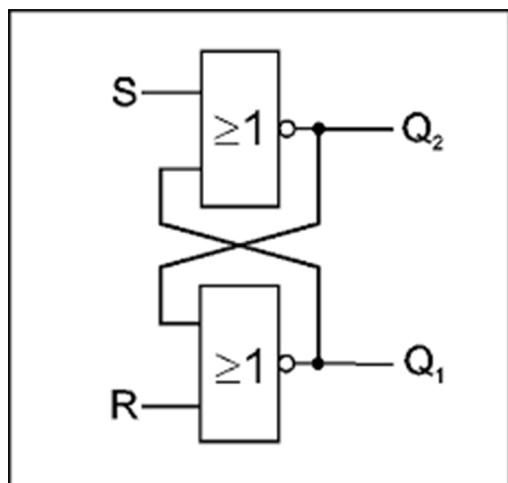
Mom: No we have



at home

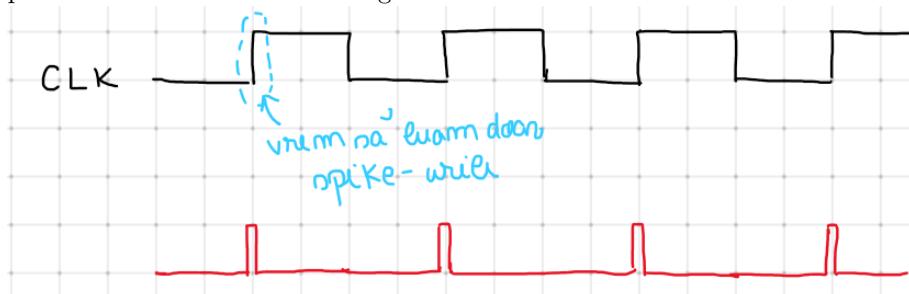


at home:

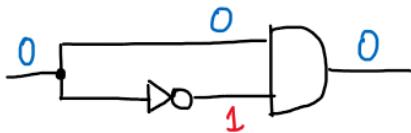


1.2 Delay Flip-Flop (DFF)

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.

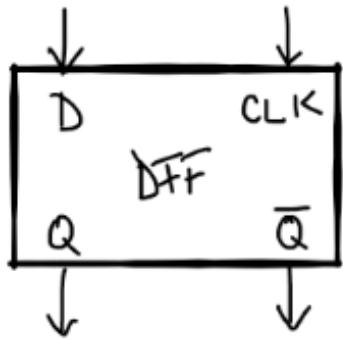


Ne intereseaza sa obtinem din acest clock un circuit care sa ne izoleze acele spike-uri de la high la low. Acest circuit pe care il cautam noi se numeste edge detector. Un exemplu de un astfel de circuit este:



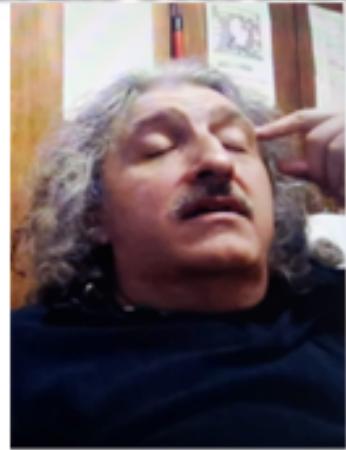
Observam starea in care este acum edge detector-ul, cu valorile scrise pe fiecare ramura. Daca input-ul isi schimba valoarea, pentru un timp foarte scurt, ramura de sus o sa fie 1, iar ramura de jos isi face nagatia putin mai greu si va ramane tot 1. Deci, pentru un timp foarte foarte scurt, ambele vor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

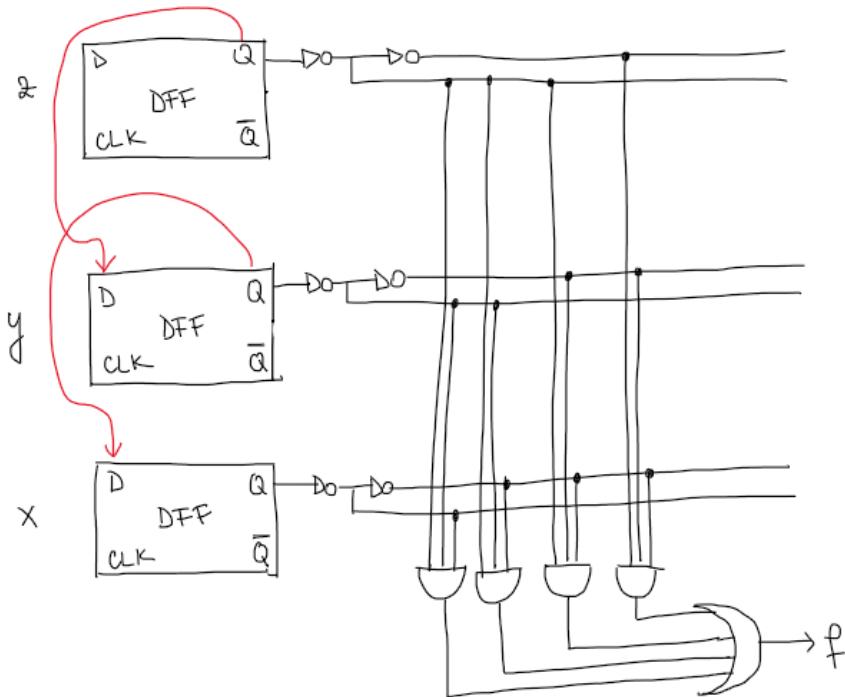


CAND ESTI IN MINECRAFT SI
CONSTRUIESTI UN D FLIP
FLOP IN LOC SA PUI UN
SIMPLU REPEATER



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

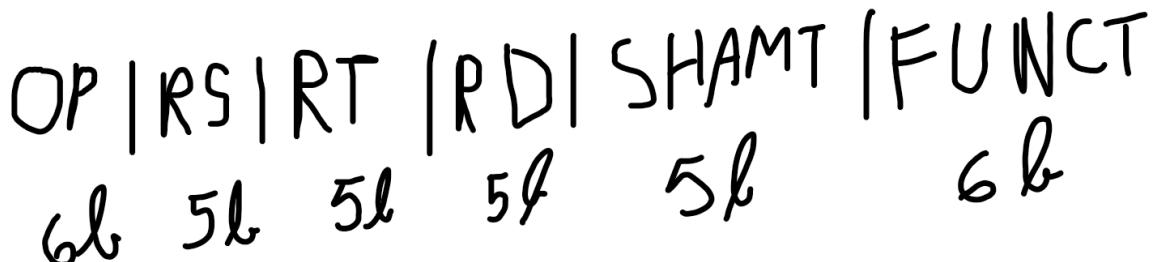
În MIPS avem 3 clase de instrucțiuni, și anume:

- 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:

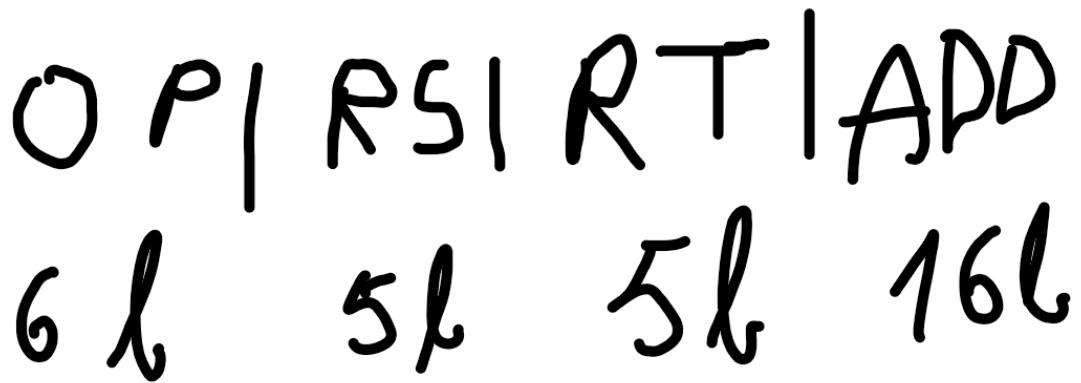
0000 0000 0000 0010 1000 1000 1000 0000

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)

$\text{op} \rightarrow 100011$
 $\text{rs} \rightarrow \$t2 = \$10 = 01010$
 $\text{rt} \rightarrow \$t0 = \$8 = 01000$
 $\text{imm} \rightarrow = 0000000000000000100$

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 = 1111111111111101, adică FFFD în hexa

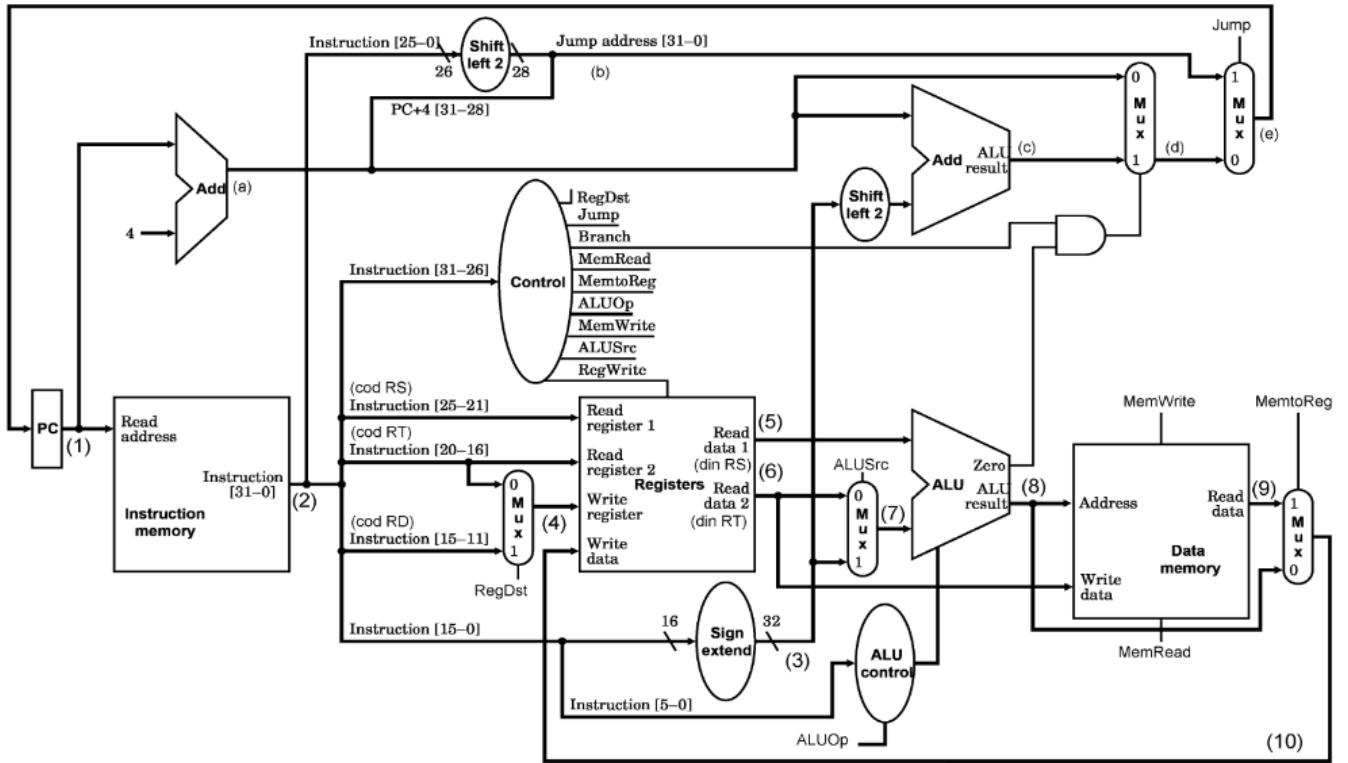
Pentru beq problema ne cere să scriem și reprezentarea binară și reprezentarea în hexa, deci vom avea:

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 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:

*no ignorează complet
jucările și
cărora*

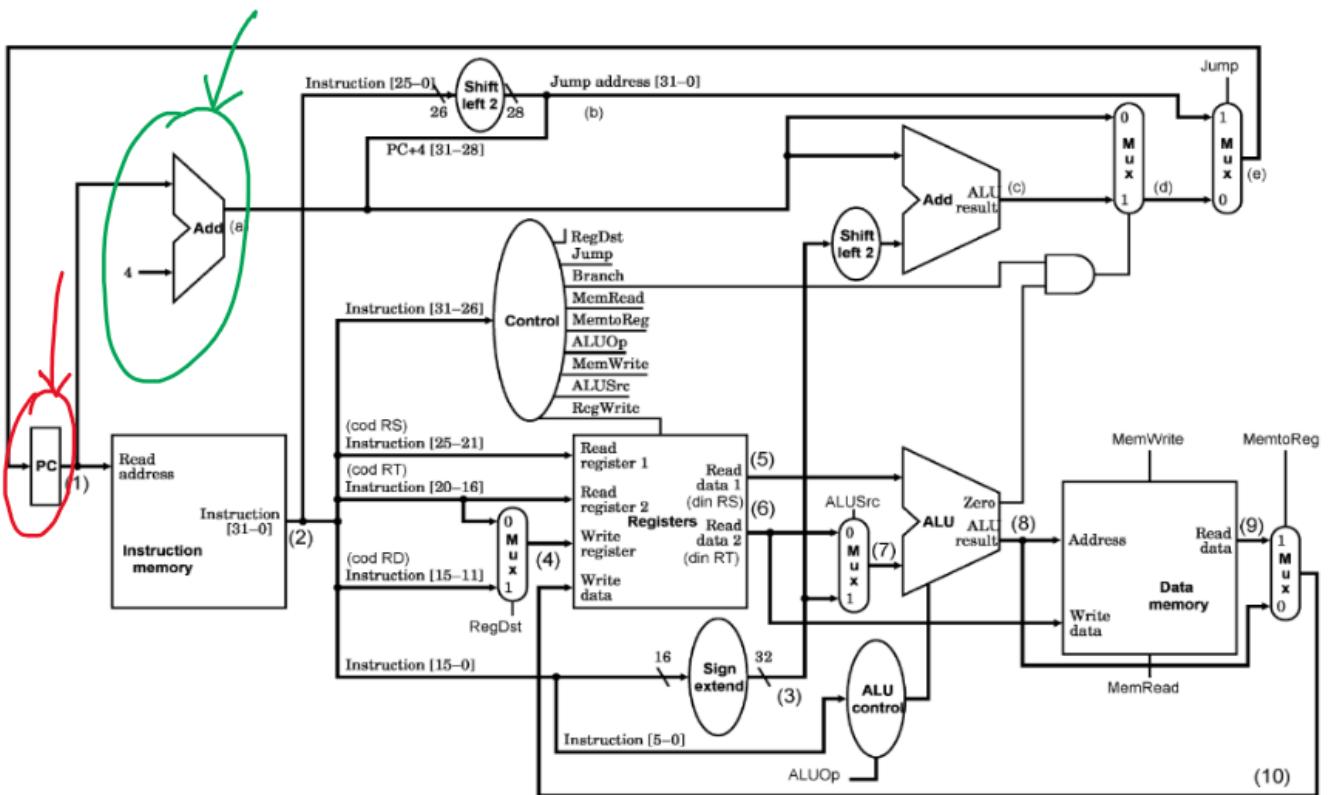
| Instruction | RegDst | ALUSrc | Memto-Reg | Reg Write | Mem Read | Mem Write | Branch |
|-------------|--------|--------|-----------|-----------|----------|-----------|--------|
| R-format | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| lw | 0 | 1 | 1 | 1 | 0 | 0 | |
| sw | X | 1 | X | 0 | 0 | 1 | 0 |
| beq | X | 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 | 111 (slt) |

2.2.1 PC

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

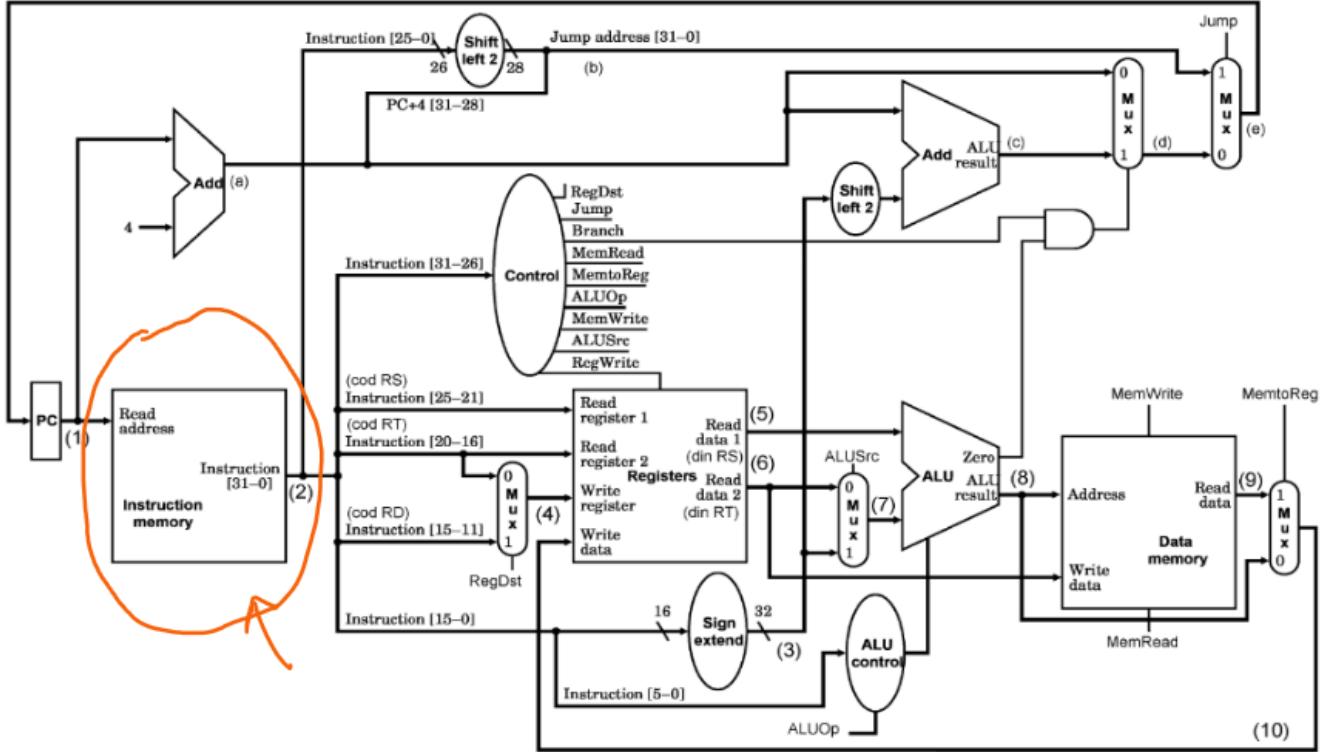


Ne putem gandi la PC ca la un pointer catre urmatoarea instructiune. De fiecare data cand se executa o instructiune, sa "incrementeaza" acest "pointer" pentru a arata catre urmatoarea instructiune. Incrementarea se face cu 4. De ce 4? Reprezentarea instructiunilor MIPS in calculator are 32 biti. Un byte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

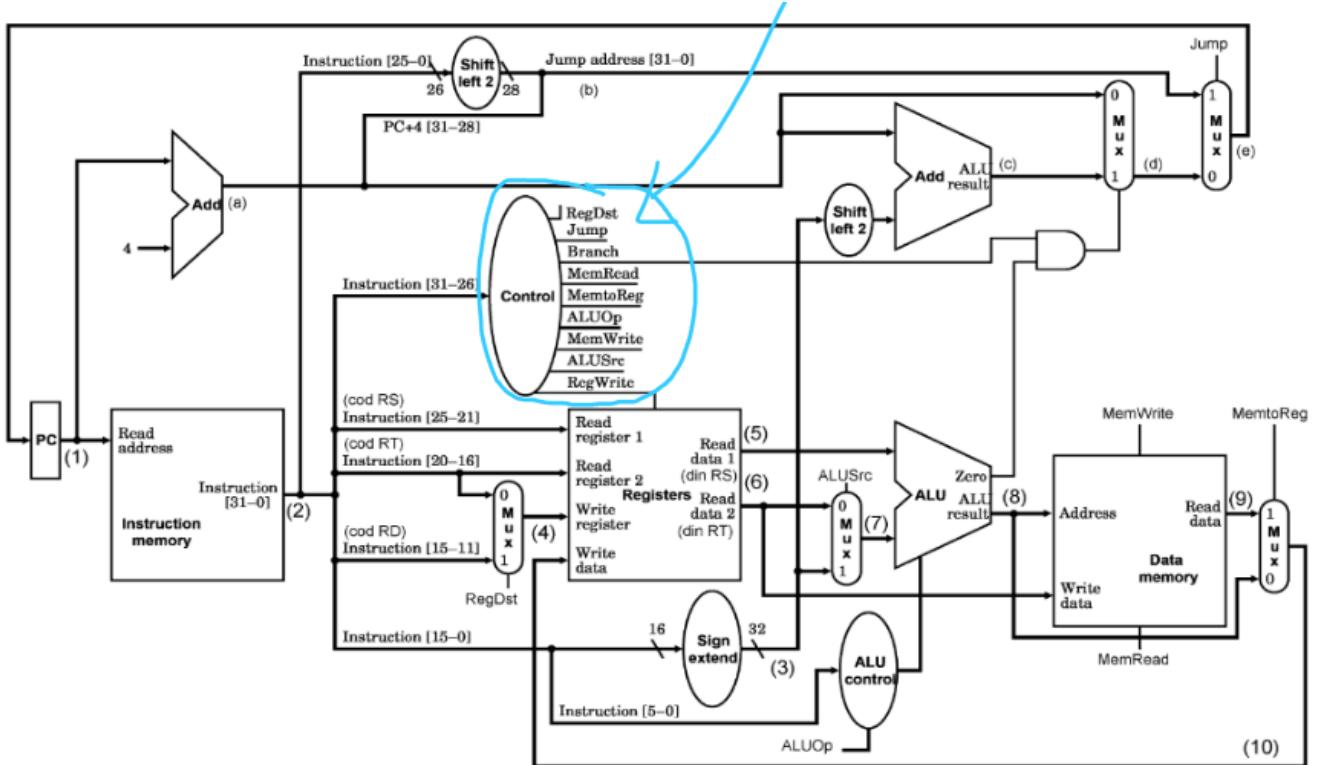
Observatie! In timpul executarii instructiunii, PC "arata" catre urmatoarea instructiune. De aceea, la 2.1 Reprezentarea interna a instructiunilor procesorului MIPS, trebuie sa adaugam acel -1.

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 incercuita 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. Acosta e comun tuturor tipurilor de instructiuni (R, I, J) si se afla mereu in primii 6 biti.

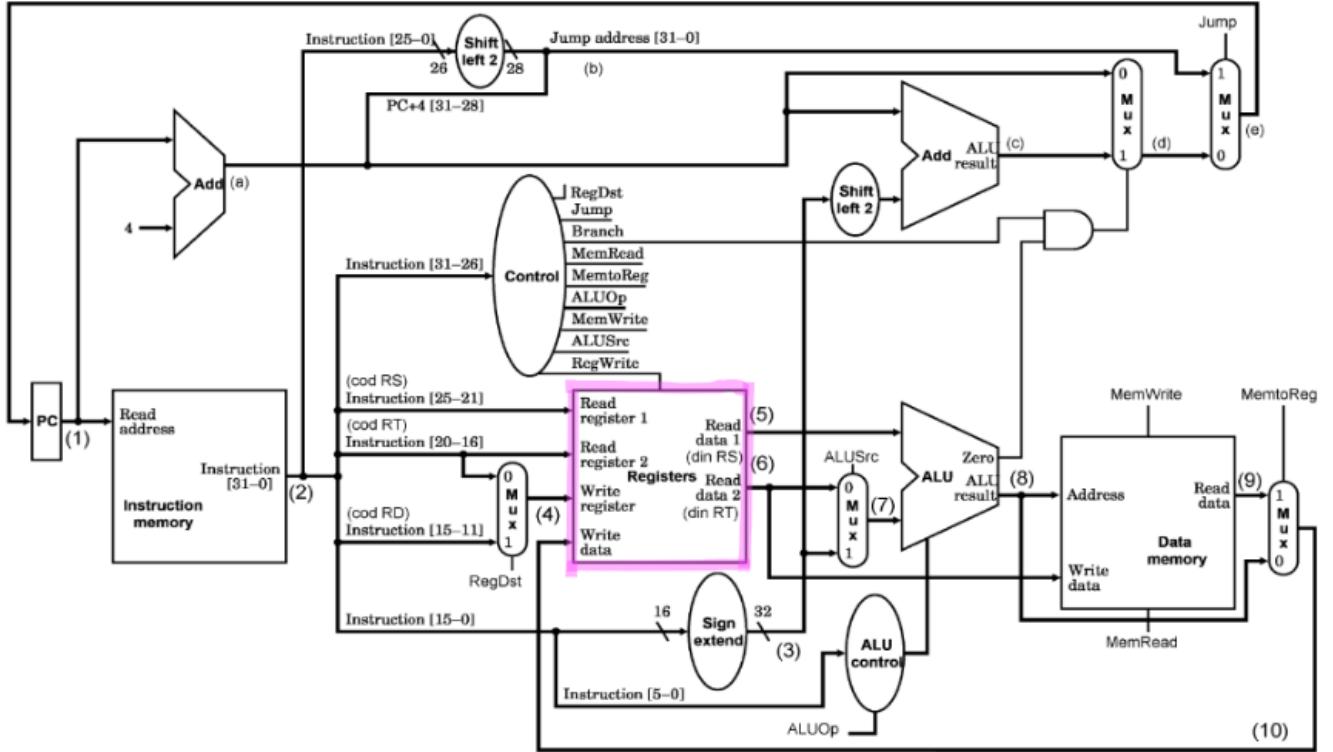
Observam ca in Control Unit intra bitii de la 31 la 26. Dar noi am spus ca opcode este mereu in primii 6 biti. Ne dam seama deci, ca asta este doar o chetiere de notatie. In memorie, fiecare bit este indexat de la 31 la 0.(putem sa ne gandim ca ultimul bit este cel mai nesemnificativ bit si de aceea are index 0).

The diagram illustrates the MIPS instruction format with the following fields:

- opcode**: bits 31...26
- rs**: bits 25...21
- rt**: bits 20...16
- rd**: bits 15...11
- shamt**: bits 10....6
- funct**: bits 5...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.

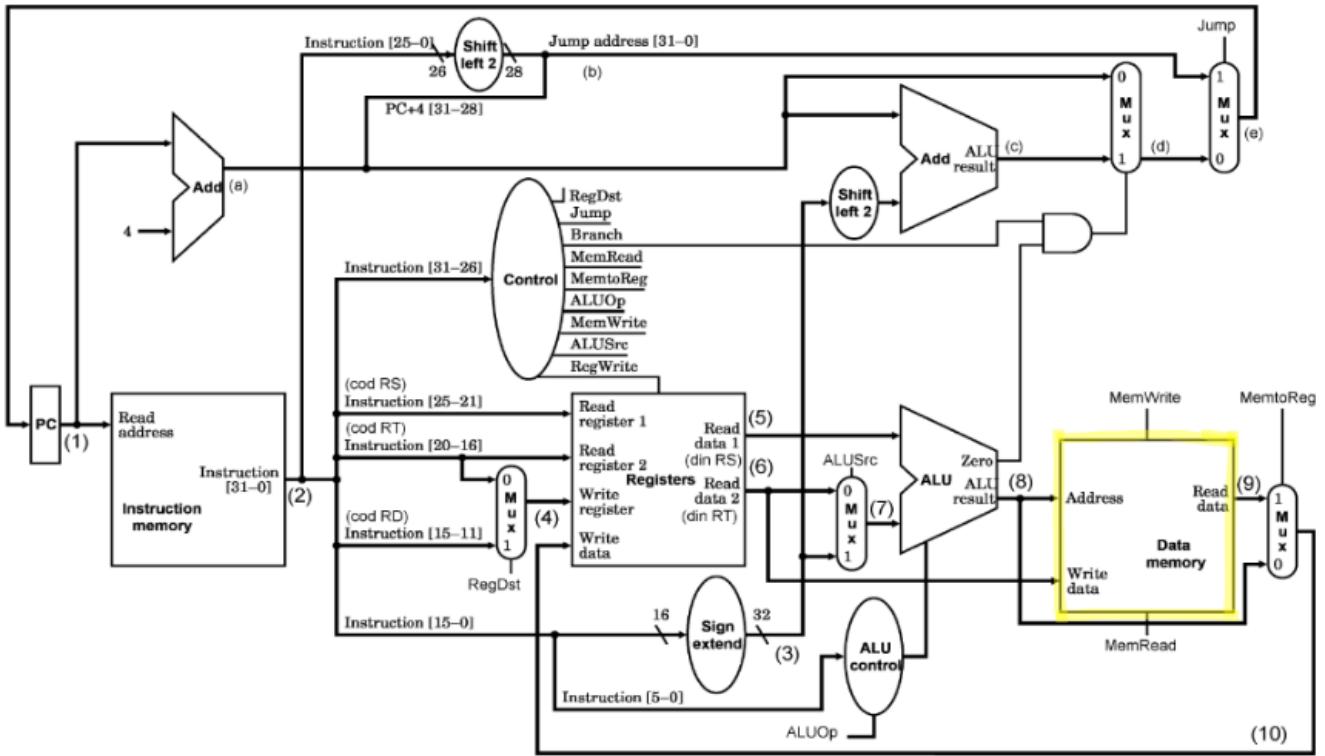
2.2.4 Citirea registrilor



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 scrie în registrul $\$t$)
- Read register 1: citește care este registrul $\$s$ (poate să fie $\$t_0$, $\$v_2$, etc.).
- Read register 2: citește care este registrul $\$t$ (poate să fie $\$t_0$, $\$v_2$, 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 ca 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: citește **valoarea** din registrul $\$s$.
- Read data 2 ca output: citește **valoarea** din registrul $\$t$.
- Write data: valoarea care trebuie scrisă în Write Register.

2.2.5 Scrierea in memorie / citirea din memorie

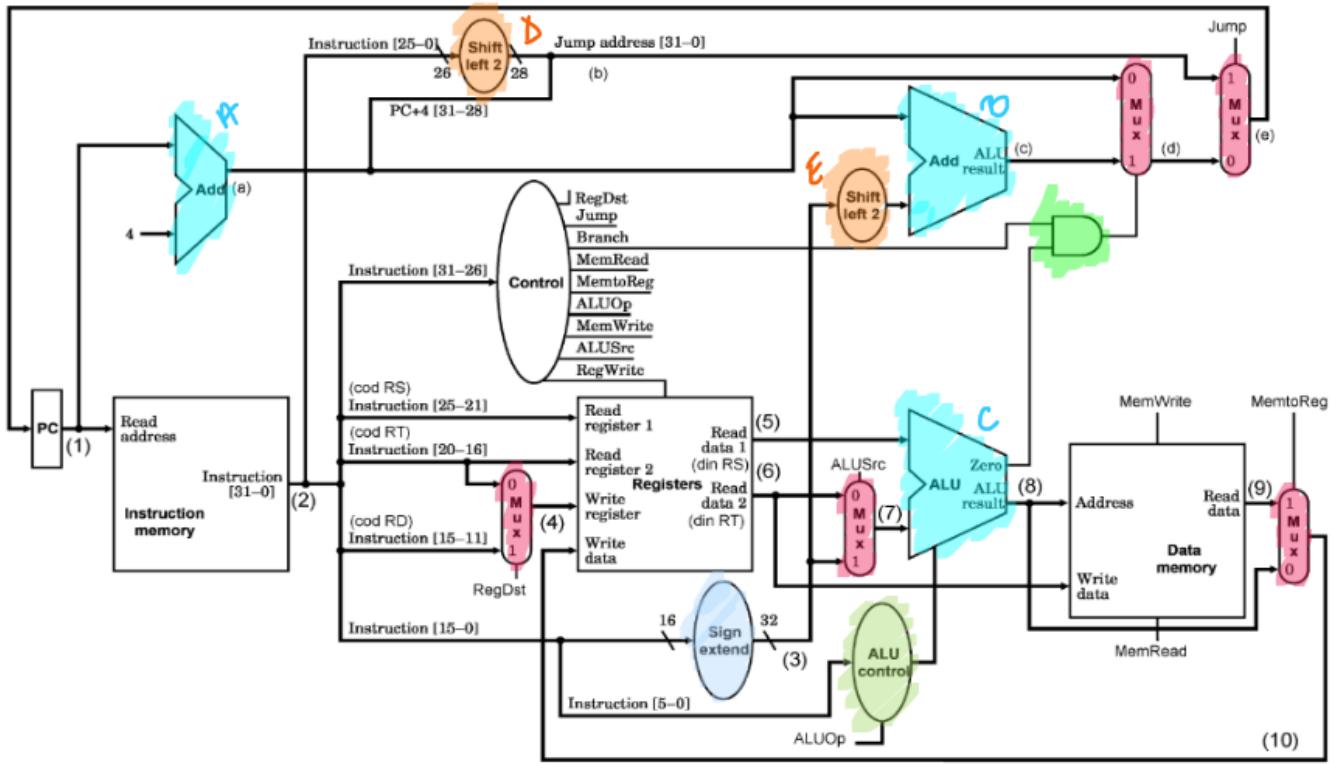


Pentru citirea/scrierea in memorie (aka lw si sw).

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 [RESTANTA SEPTEMBRIE 2020]

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
              li $t2, 5      beq $t2, $0, et
```

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) si noi avem instructiunea lw \$t4, 0(\$t3)

- \$s=\$t3=11

- $\$t = \$t4 = 12$

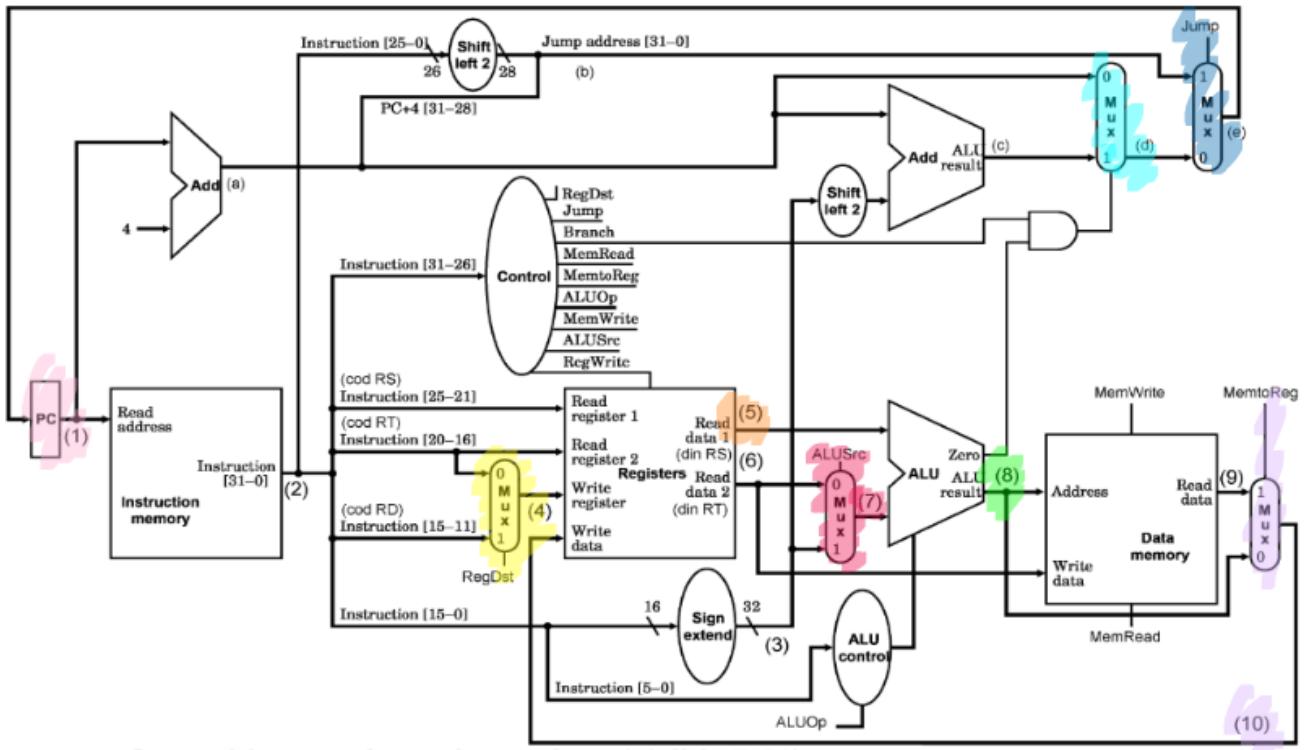
- offset=0

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

Aveți un x care inițial 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 $\$s$ (adică din $\$t3$) este adresa de memorie a lui x pe care o notăm cu β
- valoarea din $\$t$ (adică din $\$t4$) este 5

| | 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 | 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 către instrucțiunea curentă.
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 este o "variabila" care se foloseste in cazul instructiunilor de tip branch. Atunci cand am o instructiune de tip branch, valoarea lui ALUzero imi spune daca conditia din branch este indeplinita sau nu. Cum Instructiunea noastra momentan este lw, valoarea lui ALUzero este ? (nu am nicio conditie pusa, deci nu am cum sa stiu valoarea sa de adevar).
- (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 treaba cu \$t2. Deci valoarea lui \$t2 ramane aceeasi.

Incepem sa completam si randul pentru beq, incepand cu valorile pe care le luam direct 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 | St2 |
|---------|----------|---|---|---|---|----------|----|-----|-----|--------|------------|-------------|---------------|-----------|----|------------|
| Initial | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | α 5 |
| lw | α | | | | | | | | | 0 1 | 00 010 | 1 | | | | |
| beq | | | | | | | | | | 1 X | X 1 110 | 0 | | | | |

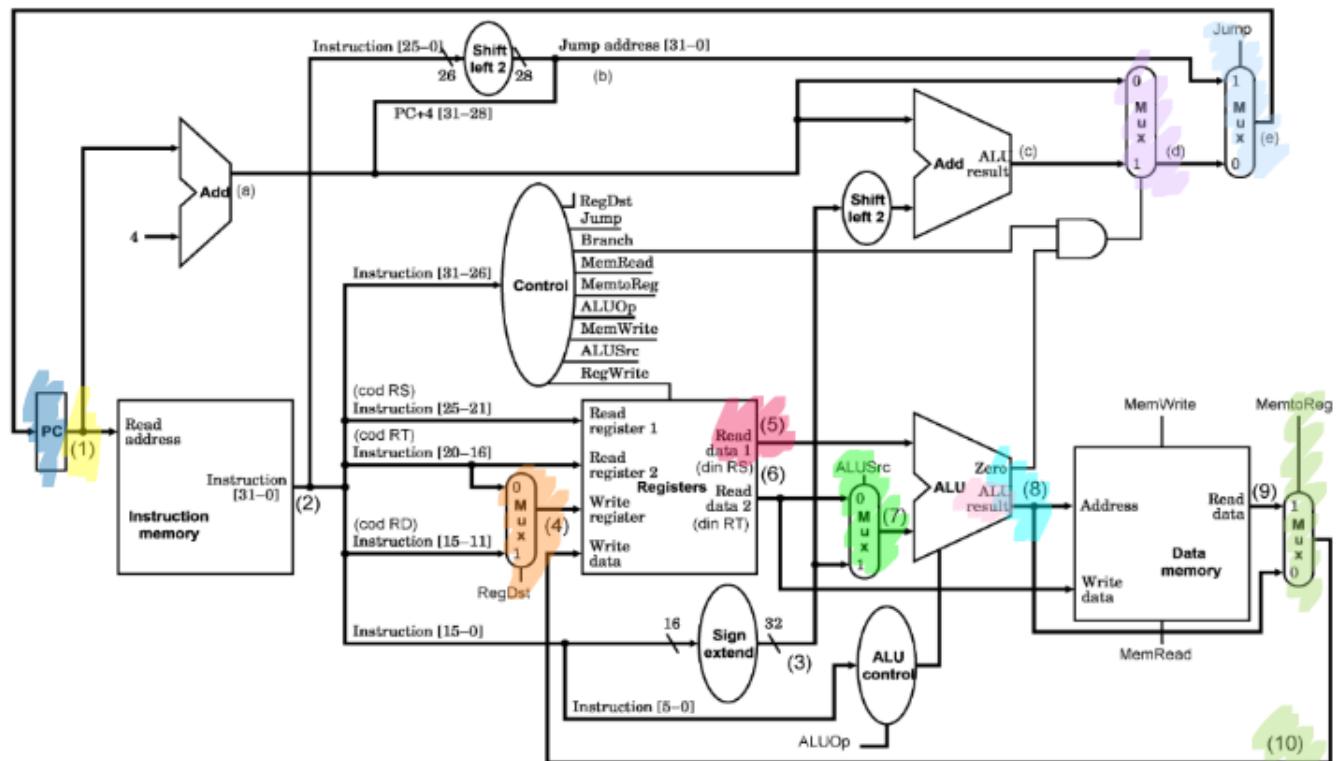
| Instruction | RegDst | ALUSrc | Memto-Reg | Reg Write | Mem Read | Mem Write | Branch | ALUOp | ALUOp2 |
|-------------|--------|--------|-----------|-----------|----------|-----------|--------|-------|--------|
| R-format | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 010 | 000 |
| lw | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 010 | 001 |
| sw | X | 1 | X | 0 | 0 | 1 | 0 | 010 | 010 |
| beq | X | 0 | X | 0 | 0 | 0 | 1 | 010 | 011 |

ALU Control (slide 7.36)

| | ALUOp | | Camp functie | | | | | | Operatie |
|-----------------|--------------------|--------------------|--------------|----|----|----|----|----|--------------|
| | ALUOp ₁ | ALUOp ₀ | F5 | F4 | F3 | F2 | F1 | F0 | |
| lw/sw | 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) |
| R- or format | 1 | X | X | X | 0 | 1 | 0 | 1 | 001 (or) |
| slt | 1 | X | X | X | 1 | 0 | 1 | 0 | 111 (slt) |

Continuam cu restul valorilor:

| | 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 | 12 | 0 | 12 | ? | 5 | alpha+4 | 0 | 1 | 00 | 010 | 1 | $\alpha+4$ | 5 | |
| beq | $\alpha+8$ | ? | 0 | 0 | 0 | 1 | ? | α | α | 1 | X | X1110 | 0 | α | 0 | |



Să parcurgem ce am completat și de ce:

- (1): Înainte `lw` PC avea valoarea α . Înainte de sub a avut valoarea $\alpha+4$. Înainte de `beq` va avea valoarea $\alpha+8$.
- (4): RegDst are valoarea X. Deci nu stim ce iese pe la 4. Si nici nu avem nevoie sa stim. Pentru `beq`, nu avem registru destinatie, deci nu ne intereseaza ce intra pe la 4.
- (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). Care e valoarea din \$s?

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

```

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

```

Să `beq` are forma: `beq $s, $t, imm`. La sub am scăzut din `$t2` 5, deci `$t2` are acum valoarea 0. => (5)=0

- (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 beq. ALUSrc=0 => Iese din multiplexor de a intrat pe la 0.
Ce a intrat pe la 0? Valoarea ui \$t, care era tot 0 => (7)=0
- (8): Avem ALU (Arithmetic and Logical Unit). Ce operatie efectueaza? Trebuie sa ne uitam la ceiese din ALUcontrol, adica ne uitam in tabela de ajutor la ALUCtrl pentru beq. Vedem ca operatia este scadere. Deci se face scaderea intre valorile din (5) si (7). Deci rezultatul este 0.
- Acum chiar avem o instructiune de tip branch, deci ALUzero are sens. Instructiunea noastra este beq, adica branch if equal. Deci ALUzero este 1 daca valorile din (5) si (7) sunt egale, si 0 daca nu sunt egale. In procesor, testam egalitatea prin scadere: doua numere sunt egale daca rezultatul scaderii lor este zero. Asadar, pentru beq:
ALUzero=1 daca (8)=0
ALUzero=0 daca (8) ≠ 0 In cazul nostru valoarea lui (8) chiar este 0 => ALUzero=1

Atentie! Functia de mai sus nu este valabila pentru orice instructiune de tip branch. De exemplu, pentru bne (branch if not equal, adica opusul lui beq) functia va deveni:

ALUzero=0 daca (8)=0

ALUzero=1 daca (8) ≠ 0

Un alt exemplu ar fi pentru blez(branch if less than or equal to 0). Aceasta instructiune are forma blez \$s, offset. Pentru aceasta instructiune, functia va deveni:

ALUzero=0 daca (5)>0

ALUzero=1 daca (5) ≤ 0. De ce (5)? Pentru ca in (5) intra valoarea din rs, iar instructiunea noastra blez testeaza daca valoarea din rs este mai mica sau egala cu 0.

- (10): Ceiese din multiplexorul a carui valoare este determinata de MemToReg. Pentru beq MemToReg are valoarea X => Nu stim ceiese pe la (10).
- (d): Rezultatul este determinat de o poarta AND intre Branch si ALUzero. Branch stim sigur ca are valoarea 1 pentru beq, si ALUzero=1. Deci in (d) avem valoarea care intra in multiplexor la 1. Ce intra in 1? Ceiese din acel ALU cu operatia de adunare. Pe sus intra valoarea lui PC+4, adica $(\alpha + 8) + 4 = \alpha + 12$. Pe jos intra imm shiftat cu 2 biti la stanga. imm=-3. Shiftarea la stanga cu 2 inseamna o inmultire cu 4. => Pe sus intra = -12. => Ceiese din ALU este $(\alpha + 12) - 12 = \alpha => (\alpha) = \alpha$
- (e): Un multiplexor determinat de valoarea lui Jump. Jump pentru beq e 0 => ieiese ce intra din (d) => α
- PC: Observam ca im PC intra fix ce a iesit din (e) => PC= α
- \$t2: Trebuie sa ne intoarcem putin la codul de MIPS si sa ne amintim ce se intampla acolo. \$t2 primește valoarea 5. Apoi avem operatia de lw pe \$t3 si \$t4, deci \$t4 este 5. Apoi avem un sub in care \$t2=0. In bew nu se schimba valoarea lui \$t2, doar o compara. Deci valoarea lui \$t2 ramane 0.

Mai departe, ne amintim ca cerinta era ca valorile sa fie scrise in tabel in hexa. Deci tabelul completat in hexa este:

| | 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 | C | p | 0 | p | ? | 5 | $\alpha\#$ | $\alpha\#$ | 0 | 1 | 00 | 010 | 1 | $\alpha\#$ | 5 |
| beq | $\alpha\#$ | ? | 0 | 0 | 0 | 1 | ? | α | α | 1 | X | x1 | 110 | 0 | α | 0 |

Observam ca pentru ALUOp si ALUCtrl nu transformam valorile in hexa pentru ca scrie ca trebuie sa fie pe 2 biti, respectiv pe 3 biti (am verificat si asa a completat si Dragulici la curs).

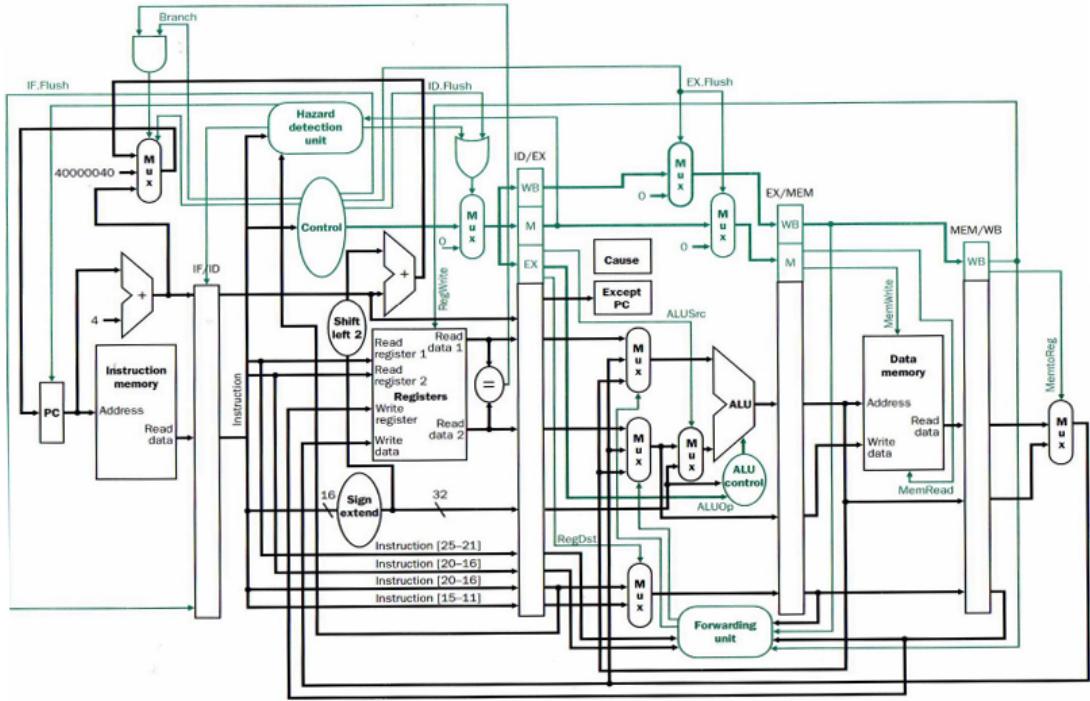
Exemplul 9 De ce spunem ca instructiunile de tip I pot sa ruleze 2 instructiuni in loc de una?
Luam ca exemplu:

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*