MIPS procesor - Kontrola i Datapath

Konstrukcija ALU-a, koji obavlja operacije "+" i "-" (aritmetičke) i "**and**" i "**or**" (logičke), počiva na korišćenju sljedećih blokova: I-kolo, Ili-kolo, invertor i multiplekser

1. AND gate (c = a · b)
   I-kolo

a	b	c = a · b
0	0	0
0	1	0
1	0	0
1	1	1

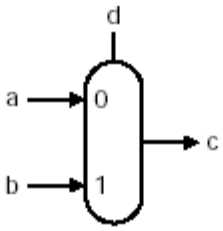
2. OR gate (c = a + b)
   Ili-kolo

a	b	c = a + b
0	0	0
0	1	1
1	0	1
1	1	1

3. Inverter (c = ā)

a	c = ā
0	1
1	0

4. Multiplexor
   (if d = = 0, c = a;
   else c = b)

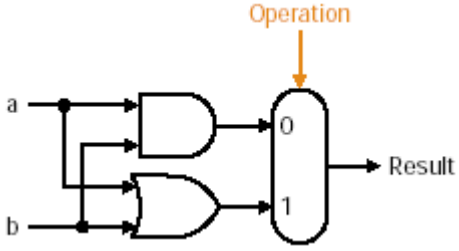


d	c
0	a
1	b

Slika 6.1

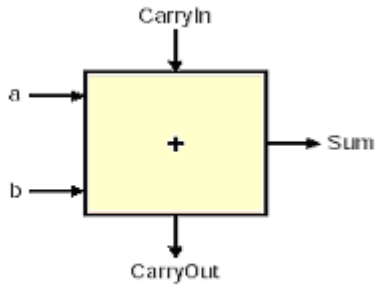
Pošto je "*riječ*"=32 bita, treba nam 32-bitna ALU jedinica, pa probajmo napraviti 1-bitnu ALU i onda spojiti takve 32-je.

Prvi korak je da riješimo "**and**" i "**or**" (onosno "i" i "ili" operacije),



Slika 6.2

Očigledno, signal "*operation*" bira koja se operacija od ove dvije radi.
Sad dodajmo još sabiranje, za koje moramo imati 2 izlaza zbog prenosa (1+1=11),



sabirač

Slika 6.3

što možemo predstaviti tabelom

inputi			outputi		
a	**b**	**carryin**	**carryout**	**sum**	**komentar**
0	0	0	0	0	0+0+0=00
0	0	1	0	1	0+0+1=01
0	1	0	0	1	0+1+0=01
0	1	1	1	0	0+1+1=10
1	0	0	0	1	1+0+0=01
1	0	1	1	0	1+0+1=10
1	1	0	1	0	1+1+0=10
1	1	1	1	1	1+1+1=11

Slika 6.4

Sa gornje tabele je jasno

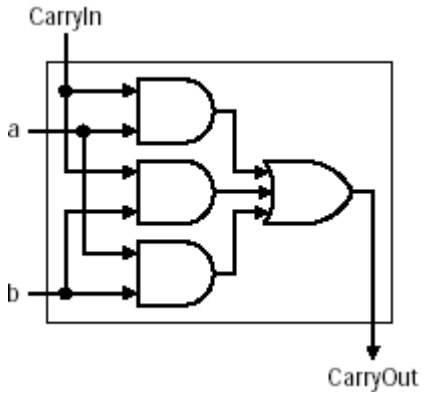
carryout=(b*carryin)+(a*carryin)+(a*b)+(a*b*carryin)

gdje je a+b=a **OR** b i a*b=a **AND** b

Očigledno, kad je ***a*b*carryin=true*** onda je sigurno još jedan od gornjih sabiraka = **true** pa možemo pojednostaviti gornju tabelu sa

Carryout=b*carryin+a*carryin+(a*b)

što se može predstaviti kao

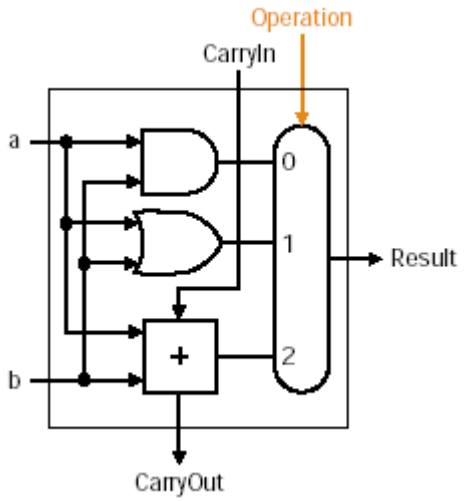


Slika 6.5

Za “**sum**” iz gornje tabele možemo pisati:

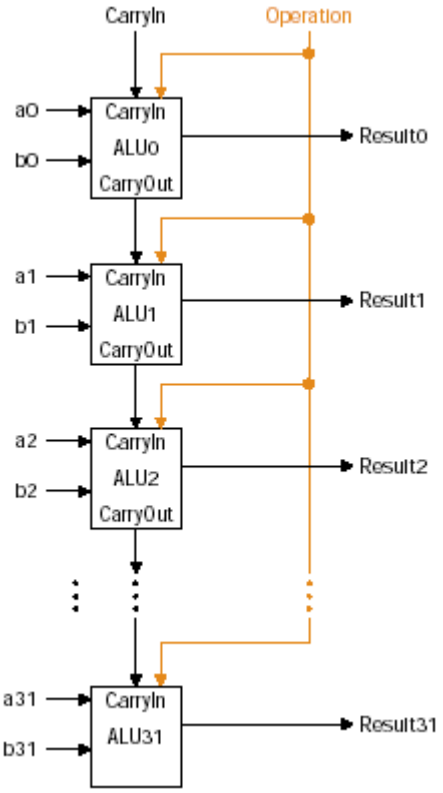
$$\boldsymbol{sum=(a*\bar{b}*\overline{carryin})+(\bar{a}*b*\overline{carryin})+(\bar{a}*\bar{b}*carryin)+(a*b*carryin)}$$

tako da sada 1-bitni ALU koji obavlja operacije “i”,”ili” i sabiranje izgleda



Slika 6.6

a shodno gore rečenom, 32-bitna ALU jedinica izgleda



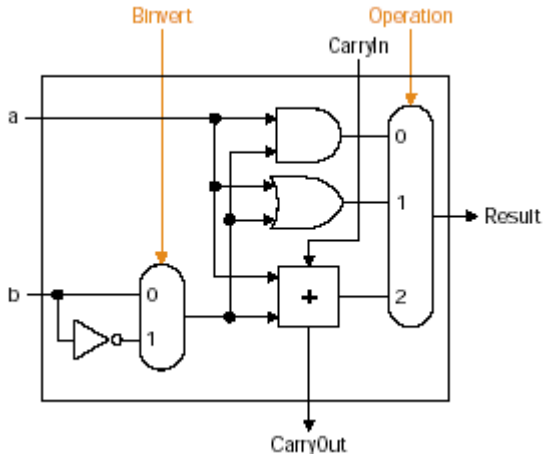
Slika 6.7

Vidi se da je *carryout* najnižeg bita povezan sa *carryin*-om bita najveće težine.

Sad je potrebno omogućiti i oduzimanje, što se radi preko drugog komplementa, a mi koristimo multiplekser 2:1 da bi izaberali b ili $\bar{b}$,

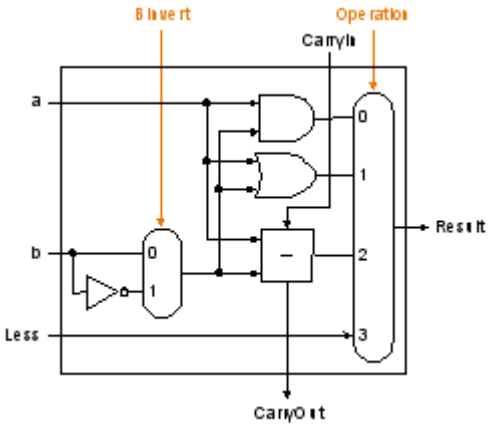
$$a+\bar{b}+1=a+(\bar{b}+1)=a+(-b)=a-b ,$$

i stavimo *carryin* bit na "1" (radi onoga +1) pa 1-bitni ALU izgleda

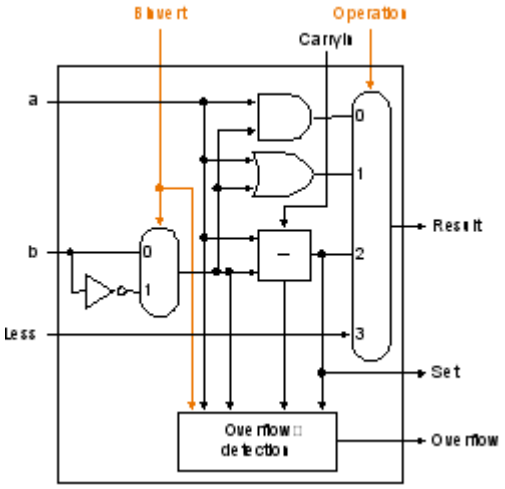


Slika 6.8

Ali sada treba dodati multiplekseru i varijantu za "*set in less than*", to jest poređenje koje setuje bit najmanje težine u "1" kad je R_S-R_T<0 (R_S<R_T) (razlika negativna), ili u "0" kad je razlika pozitivna. Za ovo bi bilo dovoljno "*sign bit*" iz izlaza sabirača staviti na bit najmanje težine, ali "*result*" izlaz iz ALU-a najveće težine nije izlaz iz sabirača, zato se za ALU *output* za "*less*" operaciju uzima *input* vrijednost od ***LESS***.



Slika 6.9



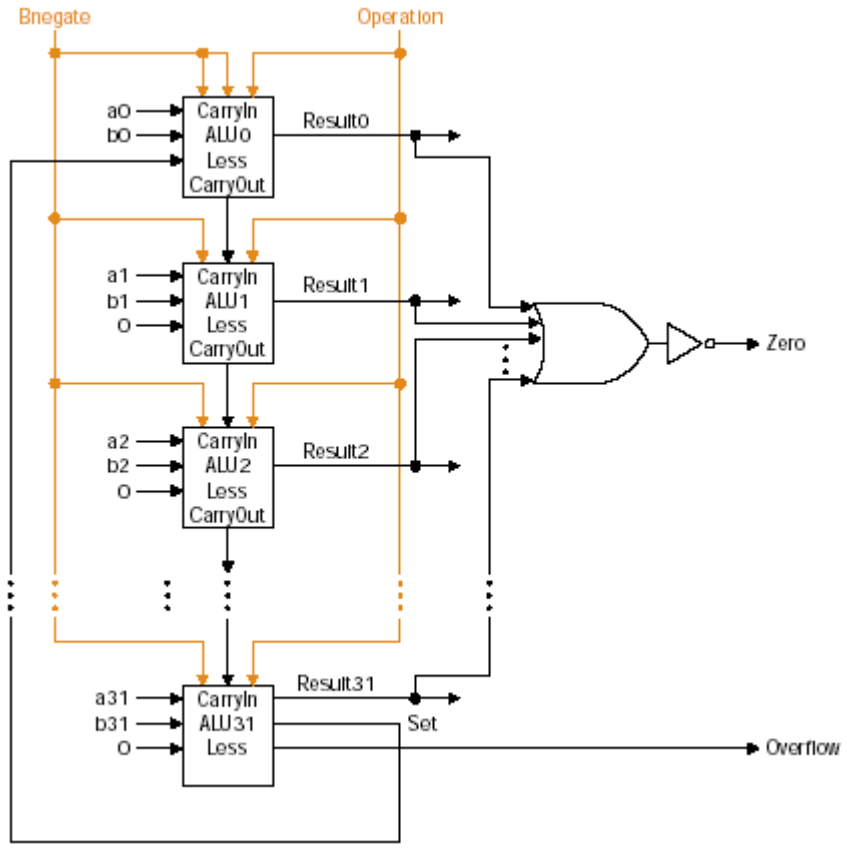
Slika 6.10

Znači ovo je ALU (ALU bita najveće težine je dat na slici 6.10) koji ima direktan *input* za «***SLT***» operaciju. Za ALU-31 (slika 6.10) je dodat još jedan izlaz iz sabirača nazvan «***SET***». Primijetimo da kad hoćemo oduzimanje, stavljamo i *Carryin* i *Binvert* na «1», a za drugo na «0» , pa možemo još uprostiti ALU tako što ćemo umjesto ta dva signala koristiti jedan ulaz «***Bnegate***». Još treba omogućiti podršku za uslovne *BRANCH* instrukcije, koje ispituju da li su 2 registra jednaka ili ne. To je najlakše utvrditi oduzimanjem, a onda testiranjem je li rezultat=0 (a-b=0) $\Rightarrow$a=b. Znači, ako imamo *hardware* koji testira je li rezultat=0, možemo testirati jednakosti. Najjednostavnije je sve *outpute* dovesti na «*ili*» kolo, a onda kroz inventor:

$$\text{zero}=(\overline{result31 + result30 + ... + result0}),$$

i sada imamo ALU koji obavlja:

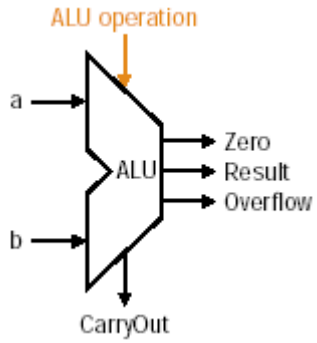
sabiranje, *oduzimanje*, *i* , *ili*, ili »*set on less than*«.



Finalni ALU (na koji je dodat «*Zero*» detektor) formiran od 1-bitnih ALU-a.

Slika 6.11

Obično se za ovakav ALU koristi slijedeći simbol



Slika 6.12

i obično se, znači, dodaju kola koja ispituju da li je rezultat=0, detektuju *overflow* i obavljaju operaciju «*set on less than*», pa se često umjesto «***i***» i «***ili***» kola koristi «***ekskluzivno ili***» kolo $x \neq y \Rightarrow 1$ i $x=y \Rightarrow 0$ jer je efikasnije: **sum=a+b+carryin**, a pomjerači se obično stavljaju van ALU-a.

Značenje tro-bitnih kontolnih signala za ALU može se prikazati kao:

```
000 = and  (i)
001 = or   (ili)
010 = add  (sabiranje)
110 = subtract (oduzimanje)
111 = slt (dodijeli vrijednost 1 ako je manji od)
```

7. PROCESOR: DATAPATH I KONTROLA

Performanse računara zavise od 3 ključna faktora: broja instrukcija, dužine takta, broja taktova po instrukciji. Kompajler određuje broj instrukcija za određeni program, a dužina takta i broj taktova su određeni procesorom. *Datapath* i *kontrolna jedinica* dizajnirane su posebno za sljedeće tipove instrukcija:

- instrukcije vezane za memoriju: *load word* (**lw**) i *store word* (**sw**)
- aritmetičko-logičke instrukcije **add**, **sub**, **and**, **or** i **slt**
- instrukcije uslovnog skoka (*BRANCH*) **beq**, i bezuslovnog skoka (**j**) na kraju.

Ostale instrukcije se izvode slično (**mult, div** i dr). Dva osnovna principa pri konstrukciji procesora su:

što jednostavnije i da se česte situacije učine što bržima.

Implementacija za sve gornje tipove instrukcija je takva da su prva 2 koraka identična i glase:

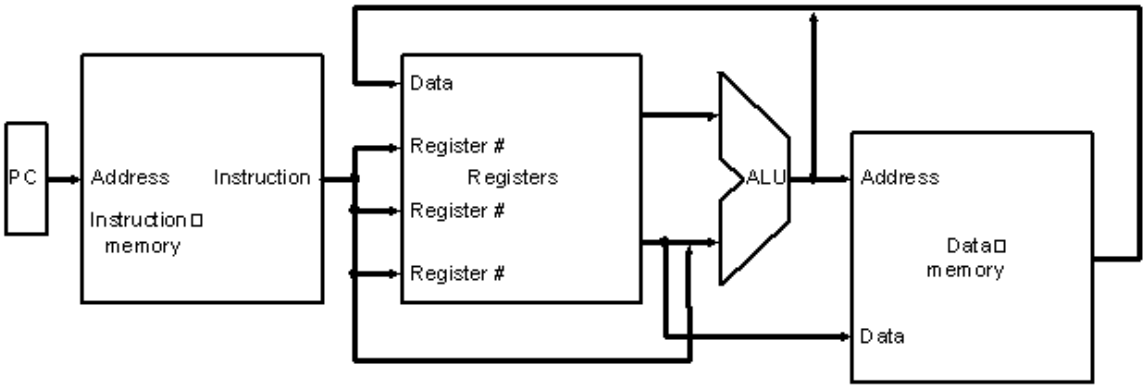
1.Pošalji brojač (PC) u memoriju gdje postoji kod za donošenje instrukcije;

2.Pročitaj 1 ili 2 registra koristeći instrukciju da izaberemo registar za čitanje. Za upis instrukcije treba pročitati samo 1 regitar, ali za sve druge instrukcije treba čitati 2 registra.

Nakon ova 2 koraka, treba završiti instrukciju zavisno od tipa, iako je za sva tri tipa ista dužina, nezavisno od *opcoda*. Za sva tri tipa postoje sličnosti, npr: svi tipovi koriste ALU nakon čitanja registara. *Informacije za memoriju* koriste ALU za računanje adrese, *aritmetičko-logičke* za izvršavanje *opcoda*, a *uslovne* za poređenje. Dakle, slično je za sve. Nakon korišćenja ALU-a, za različite tipove su različite finalizacije. *Instrukcije za memoriju* trebaju pristup memoriji koja sadrži podatke da završe smještanje ili uzmu "*riječ*" koja je učitana. *Aritmetičko-logičke instrukcije* moraju upisati podatke iz ALU-a nazad u registre. Na kraju, za *uslovne instrukcije* se treba promijeniti adresa sljedeće instrukcije zavisno od rezultata uslova. Ovo simbolično možemo vidjeti na slici 7.1.

Logička konvencija koja je usvojena kaže da signal može biti **true** ili **false**, pa ćemo za visok logički nivo koristiti izraz "**pobuđen** (*asserted*)" i "**pobuditi** (*assert*)" za signal koji treba biti doveden u visoki nivo. Funkcionalne jedinice MIPS-a imaju 2 tipa logičkih elemenata:

-*elementi koji sadrže stanje* i *elementi koji operišu vrijednostima podataka*. Ovi zadnji su svi "**kombinacioni**" što znači da im *outputi* zavise samo od trenutnih (aktuelnih) *inputa*. Ako im damo isti *input*, *kombinacioni elemenat* uvijek daje isti *output*. ALU na slici 7.1 je *kombinacioni elemenat* (jer nema unutrašnji sastav) . Drugi elementi na slici nijesu *kombinacioni*, ali **sadrže stanje** (jer imaju unutrašnji sastav). Njih ako i izvadimo iz računara i vratimo poslije restarta, imaće iste sadržaje, što znači da ti elementi stanja potpuno karakterišu mašinu. Na slici 7.1 *inktrucijska* i *data-memorija* kao i registri su primjeri *elemenata stanja*.



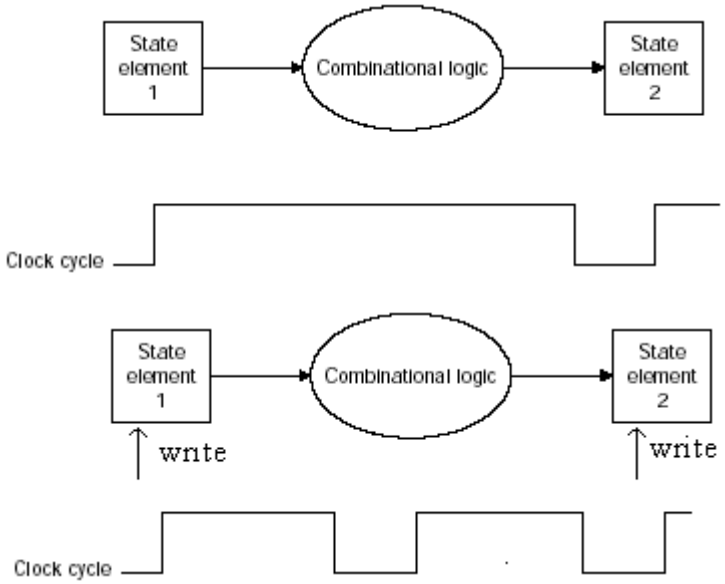
Slika 7.1

Elementi stanja imaju najmanje 2 ulaza i 1 izlaz. Zahtijevani *inputi* su vrijednost podataka koji treba biti upisan, i *takt* koji određuje kad će podatak na ulazu biti upisan. Izlaz daje vrijednost upisanu u prethodnom taktu. (*D flip-flop* je najjednostavniji primjer). Takt samo služi da odredi kad će podatak u *elemenat stanja* biti upisan, jer iz *elementa stanja* se može čitati bilo kada.

Logičke komponente koje imaju stanje se zovu i "**sekvencijalne**", jer im izlazi zavise od oba, i od ulaza i od unutrašnjeg sadržaja (stanja).

Taktovanje definiše kad podatak može biti pročitan i kad može biti upisan. Vrlo je bitno razdvojiti vrijeme upisa i čitanja, jer ako je neki podatak upisan kad je i čitanje, pročitana vrijednost može odgovarati staroj vrijednosti, ili novoj, ili čak nekom miksu ove dvije vrijednosti, a to bi bilo nedopustivo.

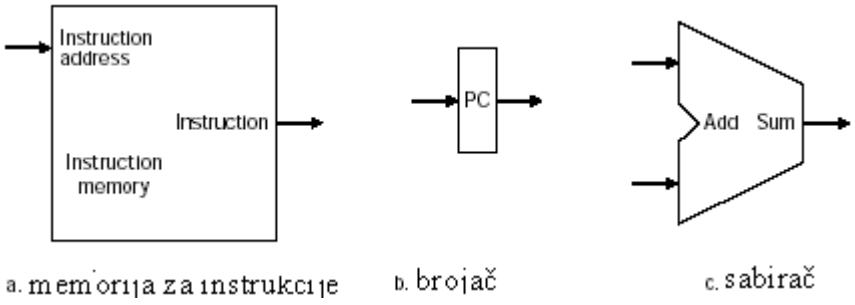
Zato se uvodi metod *ivice – okidanja*, dakle upisuje se samo na ivici takta (slika 7.2). Razlikuju se jedno-taktni i više-taktni sistemi. Na slici 7.2 je dat prikaz izvršavanja bloka operacija u 1-nom taktu, i niže na slici isti primjer u više taktova. Svi *elementi stanja* imaju takt kao *input* (*write* signal). U suštini koristi se više-taktni sistem, iako je kontrola kod njega mnogo kompleksnija.



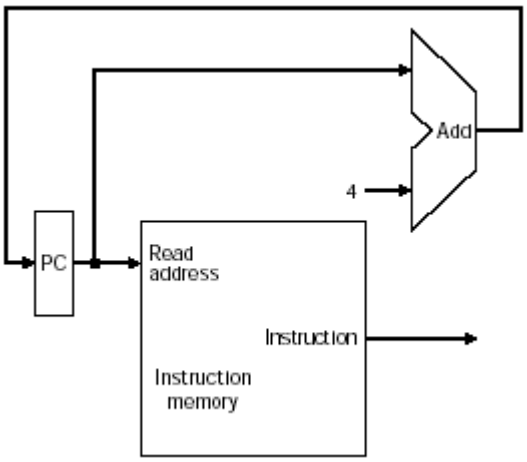
Slika 7.2

7.1 GRAĐENJE DATAPATH-a

Najprije pogledajmo glavne komponente potrebne za izvršavanje neke MIPS instrukcije. Počnimo od mjesta za smještanje instrukcije. **Memorijska jedinica**, koja je *element stanja*, čuva instrukcije i snabdijeva ostale elemente instrukcijama datim adresom (slika 7.3). Adresa instrukcije se isto mora čuvati u *elementu stanja* koji zovemo **brojač (PC)**, i treba nam **sabirač** za inkrementiranje brojača na adresu slijedeće instrukcije (sabirač samo radi tu operaciju). Izvršavanje instrukcije mora početi donošenjem instrukcije iz memorije. Tada i brojač inkrementiramo za sljedeću instrukciju (4 bita kasniju). Ovo je pokazano na slici 7.4.



Slika 7.3



Slika 7.4

Sada razmotrimo svaki tip instrukcije posebno.

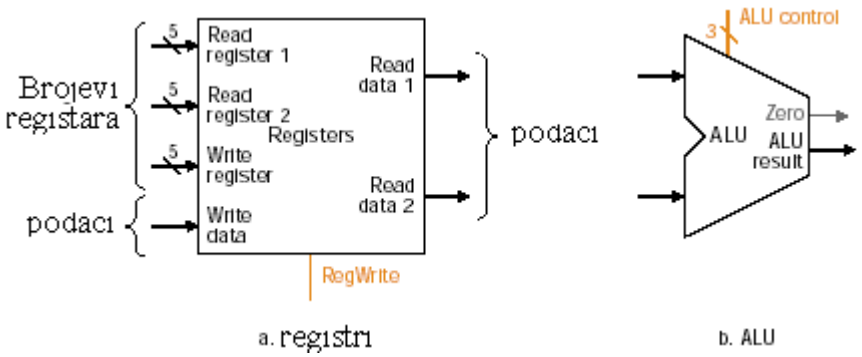
Pogledajmo *R-tip*.

Tu imamo 2 registra iz kojih čitamo, ide ALU operacija nad sadržajem iz njih, i upis rezultata. (To su instrukcije *R-tipa* ili aritmetičko-logičke instrukcije (**add**, **sub**, **and**, **or** i **slt**). Procesorova 32 registra su smještena u *Registar fajl*-u. Naprimjer:

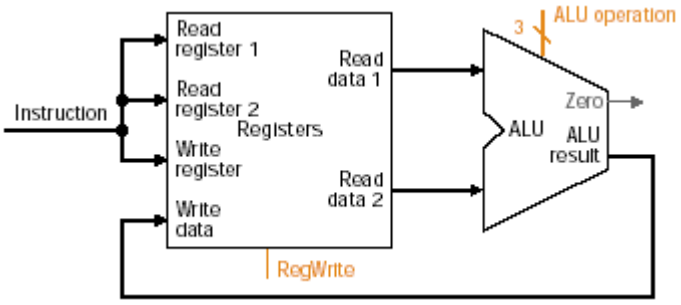
add $1, $2, $3 (saberi sadržaje registara 2 i 3 i stavi razultat u registar 1).

Znači, trebaju nam 3 operanda (registra), trebaju nam 2 registra (dvije *riječi* podataka) za čitanje i 1 za upis za svaku instrukciju. Za svaku *riječ* podatka koju treba pročitati iz registra treba nam 1 ulaz u registar koji sadrži broj registra iz kog čitamo i 1 izlaz koji nosi pročitanu vrijednost. Da upišemo *riječ* podatka trebamo 2 ulaza: 1 da pokaže broj registra gdje upisujemo i drugi koji nosi podatak za upis u registar. Znači ukupno 4 ulaza (3 za registre i 1 za podatke) i 2 izlaza (oba za podatke). To pokazuje slika 7.5.

Dakle *Ragister fajl* daje bilo koji izlaz bilo kojeg registra iz kojeg se čita, a upis je određen *taktom* i *write* signalom koji mora biti "**pobuđen**" (*asserted*) kad dođe ivica takta. Sa 5 bita se upisuje broj registra (ima ih 32, $32=2^5$), pa su i *inputi* podataka i 2 *outputa* po 32 bita. *Datapath* za *R-tip* je dat na slici 7.6.



Slika 7.5



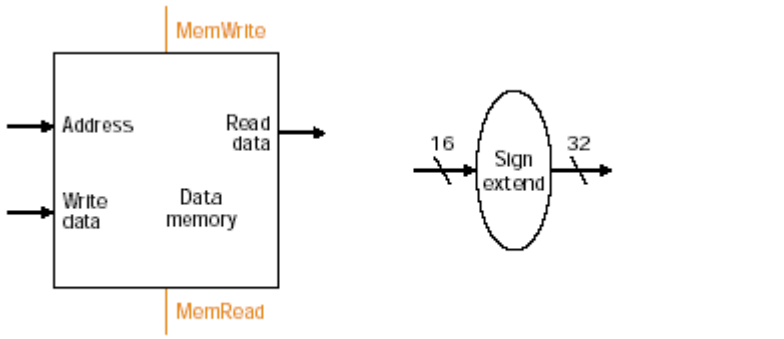
Slika 7.6

Pogledajmo *I-tip*:

Tu spadaju instrukcije *load* i *store* koje imaju izgled

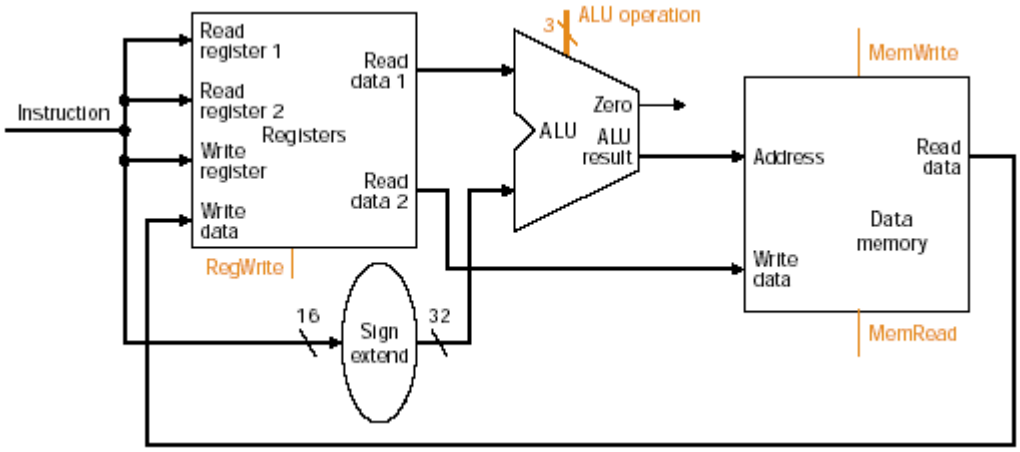
lw$1, offset_value($2) ili **sw$1,offset_value($2).**

Ove instrukcije računaju adresu memorije dodajući osnovnom registru (*$2*) 16-bitni znak iz *offseta* koji sadrži instrukcija. Ako je instrukcija *store*, vrijednost za smještanje mora biti pročitana iz *$1*, a za *load* mora se vrijednost pročitana iz memorije upisati u *$1*. Znači treba nam i *Registar fajl* i ALU, kao kod *R-tipa*. Slika 7.8 pokazuje kombinovanje elemenata za izgradnju *Datapath-a*, pretpostavljajući da je instrukcija već donešena. *Inputi* broja registara iz *Registar fajla* dolaze iz polja instrukcija kao i vrijednost *offset*-a, koja poslije *proširivanja-znakom* (*sign-extention*) postaje drugi ALU *input*. Na slici 7.7 su prikazani i *znakom-prošireni* 16-bitni *offset* i *data-memorija* za upis ili čitanje sa kontrolnim signalima.



a. memorija za podatke b. jedinica za znakom-proširenje

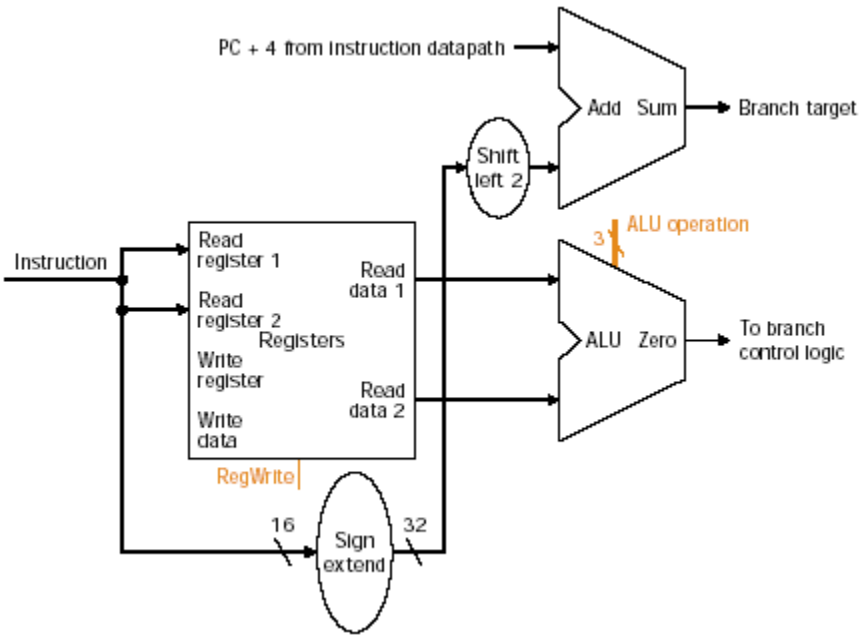
Slika 7.7



Slika 7.8

Pogledajmo uslovni tip (*BRANCH*)

Instrukcija **beq** ima tri operanda, 2 registra koji se porede, i 16-bitni *offset* za računanje adrese nove instrukcije koja se dodaje brojaču. Inače, brojač se uvećava kao PC+4 za adresu sljedeće instrukcije, a *offset* polje je pomjereno za 2 bita ulijevo radi lakšeg rada za neke instrukcije. Ako uslov nije zadovoljen, vrijednost PC-a se uvećava za 4, a ako jeste onda ga mijenja izračunata vrijednost sa *offset*-om (slika 7.9). Znači trebaju nam 2 registra i ALU-i koji će oduzeti njihove sadržaje, pa ako je *Zero* signal na izlazu iz ALU-a *pobuđen*, znači da su vrijednosti ta dva registra iste.

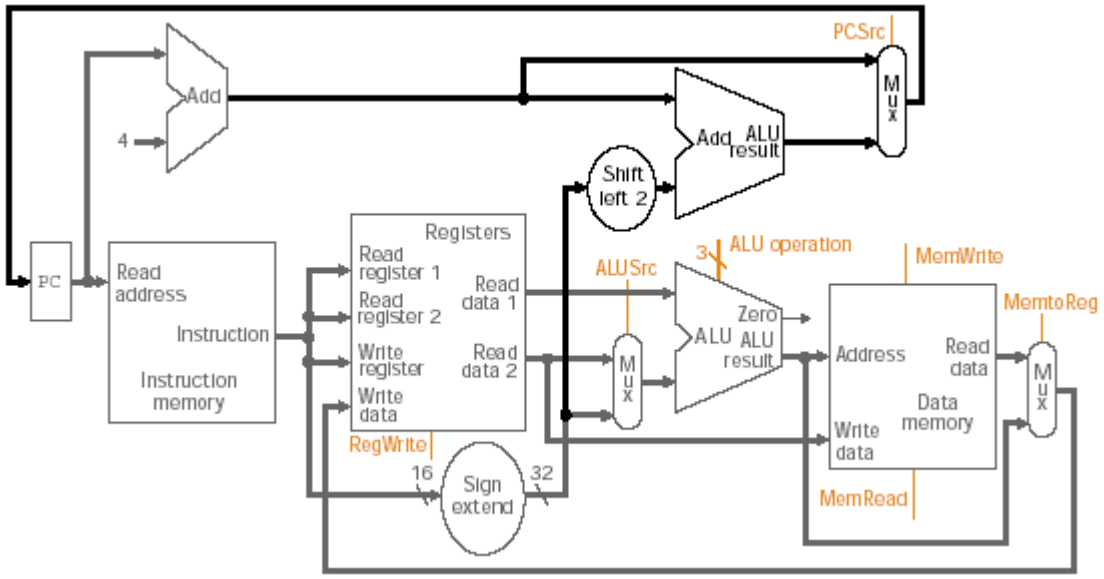


Slika 7.9

Pogledajmo "*JUMP*"-tip:

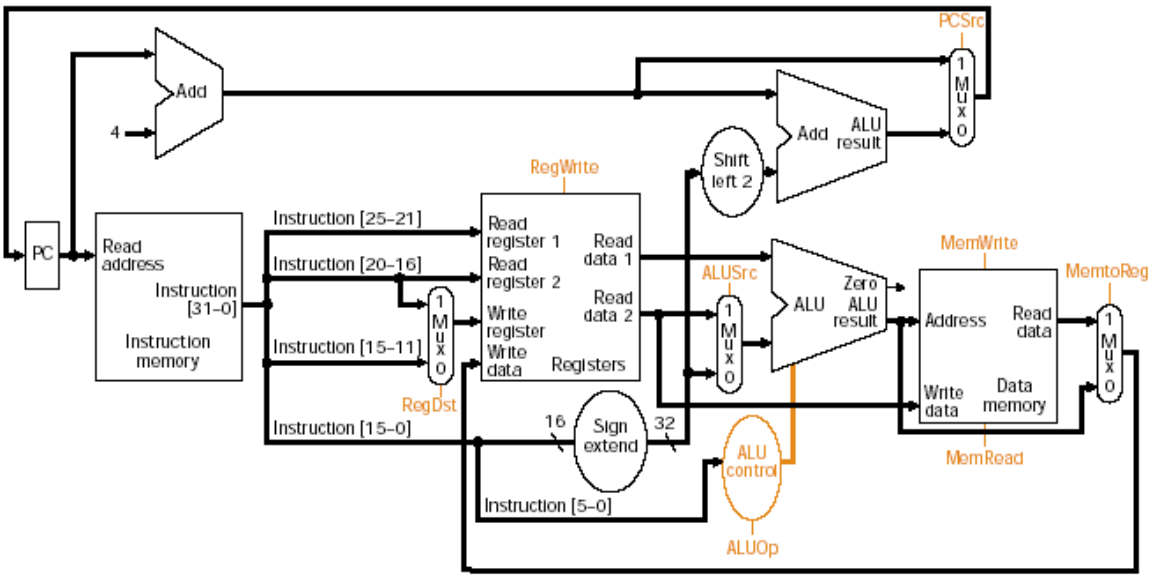
Kod ovog tipa se vrijednost PC-a mijenja sa 26 bita najmanje težine instrukcije, pomjerenim ulijevo 2 bita (samo se doda 00 na *JUMP offset*).

Sada možemo napraviti prostu *Datapath* šemu, pri čemu, pošto je jedno-taktni sistem, ako želimo neki elemenat koristiti 2 puta, moramo ga duplirati. Da bi razlučili kakav je *Datapath* za različite tipove instrukcija, koristimo multiplekser kao na slici 7.10.



Slika 7.10

Na slici 7.11 je prikazan *Datapath* sa multiplekserom i kontrolnim linijama.

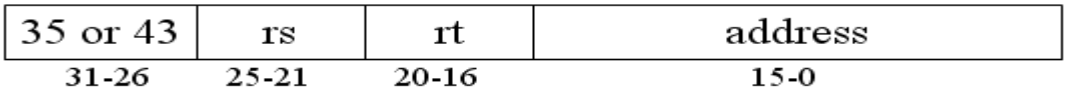


Slika 7.11

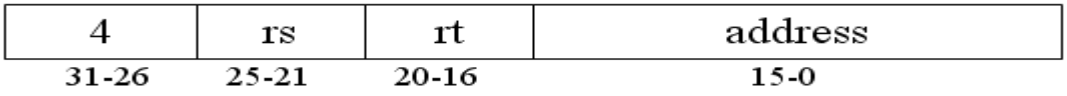
Korisno je ovdje vidjeti pojedinačno i izgled svih tipova instrukcija radi lakšeg praćenja *Datapath*-a i *kontrole* (slika 7.12) kao i tabelu značenja kontrolnih signala (slika 7.13).

R-Type, add, subtract, and, or, slt

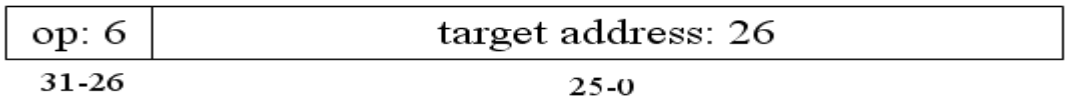
I-Type, lw, sw



I-type, beq



J-Type

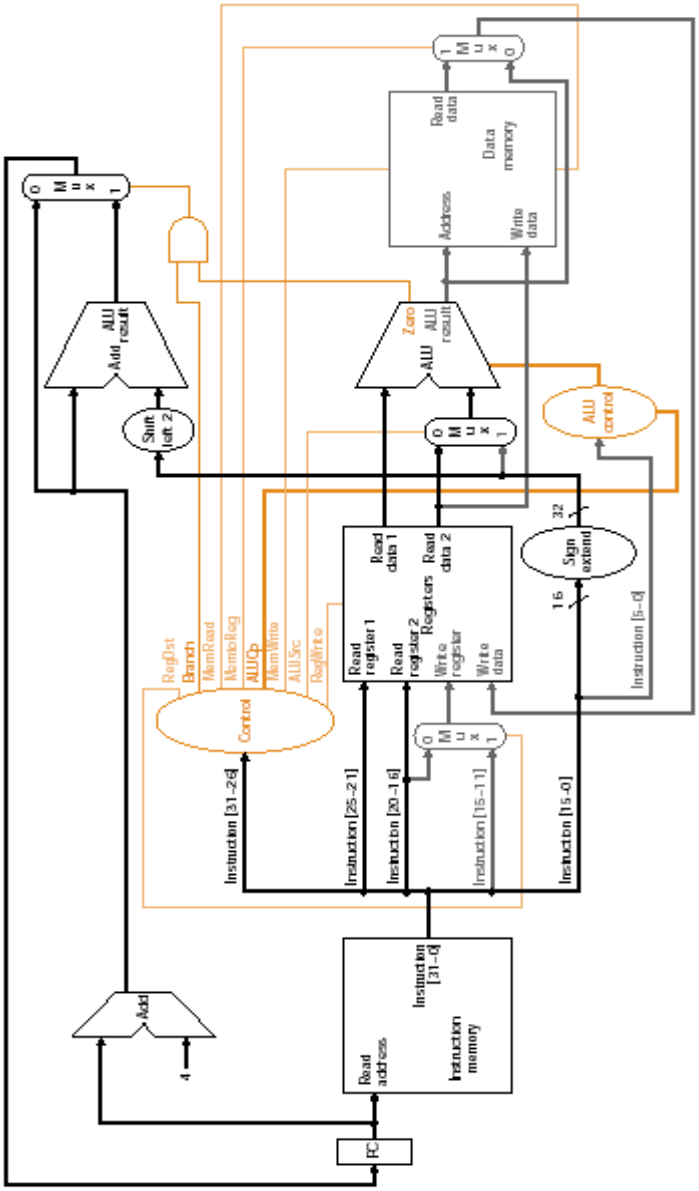


Slika 7.12

Ime signala	Efekat kad nije pobuđen	Efekat kad je pobuđen
ALUSrcA	drugi ALU operand je iz drugog izlaza Registar fajla	drugi ALU operand su znakom-prošireni 16 bita manje težine instrukcije
RegWrite	nema	write data podatak se upiše u registar dat write registar brojem
MemtoReg	regisarski write data input = ALU izlazu	registarski write data input = memory
RegDst	destinacija za upis u reg. = rt	Reg. dest. no. = rd
MemRead	nema	Read data1=Read address
MemWrite	nema	sadržaj Write address zamijenjen vrijednošću Write data

Slika 7.13

Kompletna šema *Datapath*-a sa kontrolnim signalima data je slikom 7.14.



Slika 7.14

Pojedinačne *Datapath*-ove za različite tipove instrukcija nećemo ovdje raditi, već, pošto o kojoj se instrukcji radi znamo iz 6-bitnog «*opcode*» polja (napišimo ga kao $Op_5\ Op_4\ Op_3\ Op_2 Op_1\ Op_0$), samo dajemo tabelu istinitosti za razne instrukcije i vrijednosti *opcode*-a sa kontrolnim signalima.

ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	010
X	1	X	X	X	X	X	X	110
1	X	X	X	0	0	0	0	010
1	X	X	X	0	0	1	0	110
1	X	X	X	0	1	0	0	000
1	X	X	X	0	1	0	1	001
1	X	X	X	1	0	1	0	111

Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
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

Signal	R-format	lw	sw	beq
Op5	***0***	***1***	***1***	***0***
Op4	***0***	***0***	***0***	***0***
Op3	***0***	***0***	***1***	***0***
Op2	***0***	***0***	***0***	***1***
Op1	***0***	***1***	***1***	***0***
Op0	***0***	***1***	***1***	***0***
RegDst	1	0	X	X
ALUSrc	0	1	1	0
MemtoReg	0	1	X	X
RegWrite	1	1	0	0
MemRead	0	1	0	0
MemWrite	0	0	1	0
Branch	0	0	0	1
ALUOp1	1	0	0	0
ALUOp2	0	0	0	1

Slika 7.15

7.2 NEDOSTACI JEDNO-TAKTNOG RADA

Takt mora biti iste dužine za sve instrukcije, odnosno, pošto je *Load* najduža instrukcija (5 koraka), takt mora toliko i trajati, a mnoge instrukcije zapravo traju puno kraće, tako da u suštini princip da česte stvari učinimo brzima, ovdje nije moguće sprovesti i zato se koristi više-taktni sistem rada. Sljedeća tabela reprezentuje razlike u trajanju za različite tipove instrukcija (slika 7.16).

Tip Instrukcije	memorijske instrukcije	čitanja iz registara	ALU operacije	memorijski podaci	upis u registre	Ukupno trajanje
Operacije R-tipa	10	5	10	0	5	30ns
Load	10	5	10	10	5	40ns
Store	10	5	10	10		35ns
BRANCH	10	5	10	0		25ns

JUMP	10					10ns

Slika 7.16

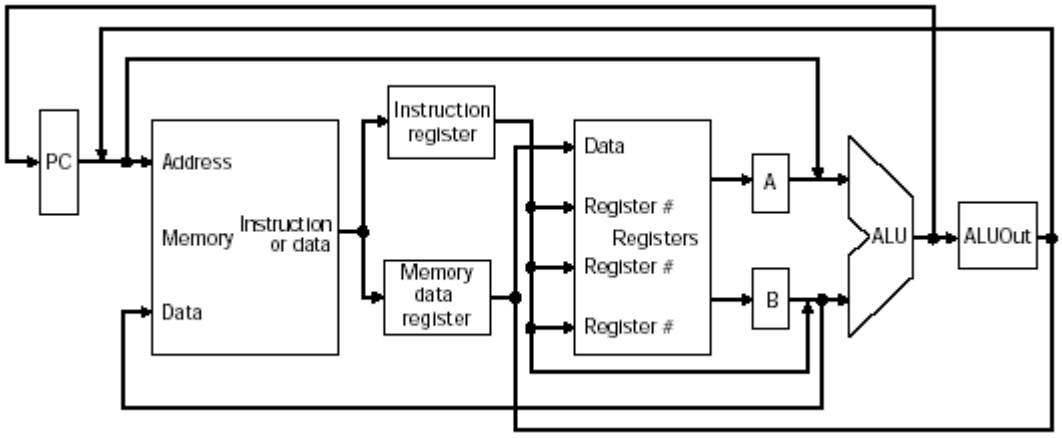
7.3 VIŠE-TAKTNA IMPLEMENTACIJA

Kod više-taktne implementacije svaki korak u izvršavanju traje jedan takt, tako da postoji mogućnost da se neka funkcijska jedinica koristi više puta za instrukciju samo u različitim taktovima. To smanjuje hardverske zahtjeve, pa je to uz mogućnost izvršavanja instrukcije kroz različiti broj taktova glavna prednost ovog sistema. Osnova više-taktnog sistema je prikazana na slici 7.17, a razlike u odnosu na jedno-taktnu verziju su:

-jedna memorijska jedinica se koristi i za instrukcije i za podatke;

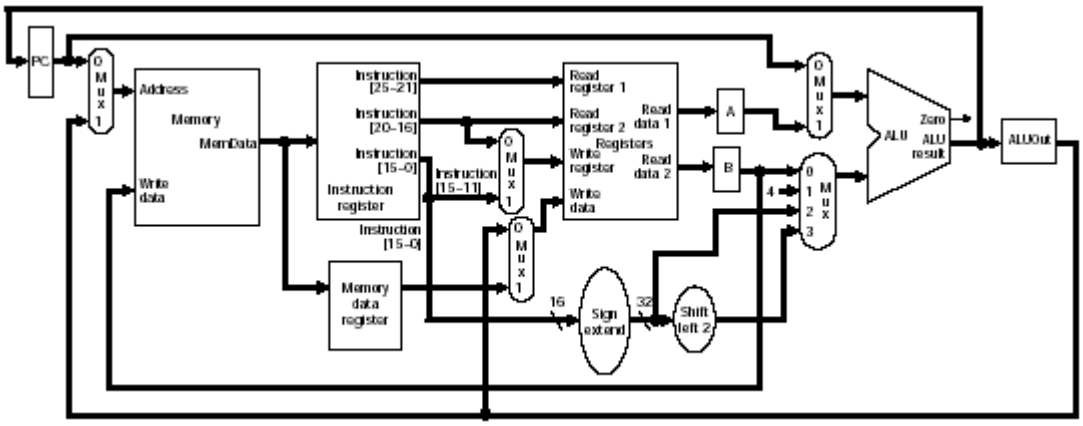
-registar se koristi za čuvanje instrukcije nakon što je pročitana.

Imamo *Instrukcijski registar* (IR) koji je neophodan jer se memoriji može pristupiti zbog podataka i kasnije u toku izvršavanja instrukcije. Takođe, koristi se jedna ALU jedinica, što je bolje nego ALU i dva sabirača što je slučaj kod jedno-taktnih sistema.



Slika 7.17

Zbog različitih funkcijskih jedinica za razne svrhe, neophodni su i multiplekseri radi mogućnosti izbora. Pošto se memorija koristi i za instrukcije i za podatke, multiplekserom biramo izmedju dva izvora za adresu u memoriji (prvi je brojač (PC) instrukcija, a drugi ALU rezultat - kad želimo pristupiti podatku). Takođe nam trebaju i multiplekser za prvi ulaz u ALU koji bira da li je to brojač ili sadržaj prvog registra, i multiplekser na drugom ulazu ALU-a koji je proširen sa dva na četiri ulaza, pri čemu su dodati kao drugi ulaz u MUX ″broj 4″(za inkrementaciju brojača na slijedeću instrukciju) i kao četvrti ulaz u MUX vrijednost *offseta* koji je znakom-proširen i pomjeren ulijevo za 2 mjesta a koristi se za računanje adrese kod *BRANCH* instrukcija (slika 7.18).



Slika 7.18

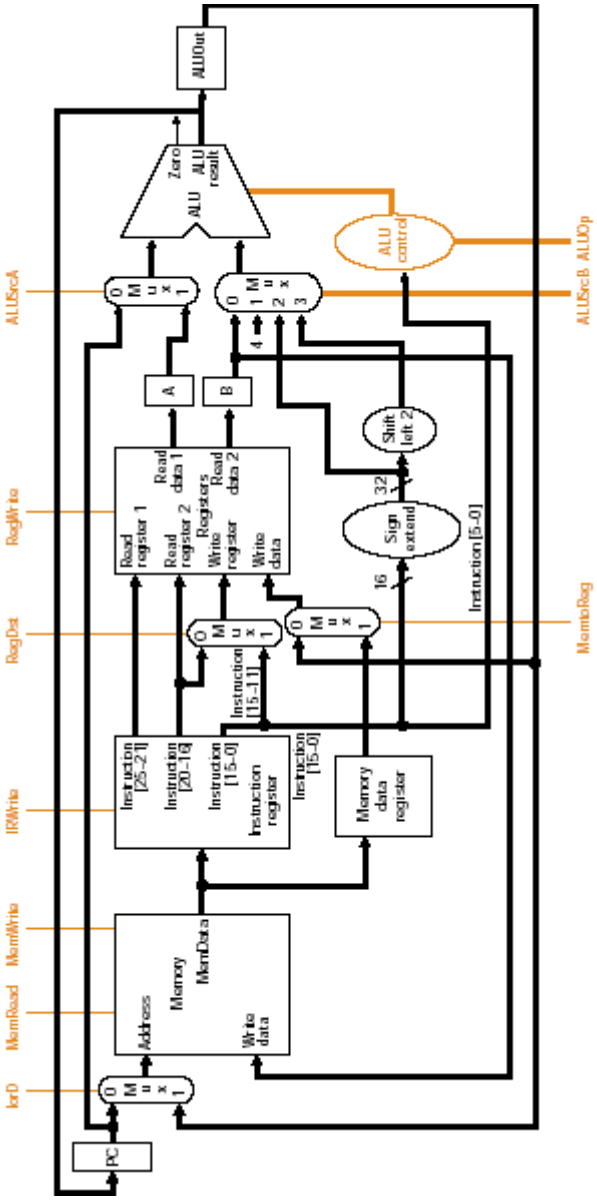
Znači, na ovaj način smo smanjili broj memorijskih jedinica sa dvije na jednu i izbjegli 2 sabirača koja imamo kod jedno-taktnih sistema, što je značajna ušteda sa aspekta cijene samog *hardware*-a. Kako se radi o više-taktnom sistemu za izvršavanje instrukcije, trebaju nam i razni kontrolni signali:

-*write* signali za svaki *elemenat stanja*: memoriju, PC, registar za opštu upotrebu i *Instrukcijski registar*;

-*read* signal za memoriju;

-*inputi* za ALU-e zahtijevaju 1-no bitne kontrolne signale, a za ALU sa četiri ulaza trebaju nam 2-bitni kontrolni signali.

Na slici 7.19 prikazan je *Datapath* sa kontrolnim signalima.



Slika 7.19

No, trebaće nam i signali koji će kontrolisati kad će nova vrijednost PC-a biti upisana i koja mu je vrijednost dodijeljena. Stoga su nam od velikog značaja tabele koje pokazuju značenje kontrolnih signala, zavisno da li su pobuđeni ili ne (slika 7.20) (*AluSelB* i *AluOp* su posebno prikazani jer su oni 2-bitni signali).

Ime signala	Efekat kad nije pobuđen	Efekat kad je pobuđen
ALUSelA	prvi ALU operand = PC	prvi ALU operand = sadržaju Reg[rs]
RegWrite	nema	write data podatak se upiše u registar dat write registar brojem
MemtoReg	regisarski write data input = ALU izlazu	registarski write data input = memory

RegDst	destinacija za upis u reg. = Reg[rt]	destinacija za upis u reg. = Reg[rd]
MemRead	nema	Read data1=Read address
MemWrite	nema	sadržaj Write address zamijenjen vrijednošću Write data
IorD	Memory address = PC	Memory address = ALU izlazu
IRWrite	nema	IR = Memory

1-bitni kontrolni signali

Ime signala	Vrijednost	Efekat
ALUOp	00	ALU sabira
	01	ALU oduzima
	10	operacija ALU-a određena funkcijskim poljem
ALUSelB	000	drugi ALU input = Reg[rt]
	001	drugi ALU input = 4
	010	drugi ALU input = znakom-proširenom IR[15-0]
	100	drugi ALU input = nulama-proširenom IR[15-0]

2-bitni kontrolni signali

Slika 7.20

Da bi smanjili broj signalnih linija, dizajneri računara koriste iste BUS-ove (magistrale podataka) za različite upise i čitanja (npr. 5 ulaza ide ka ALU-u, ali samo nam 2 trebaju istovremeno u određenom trenutku, pa je stoga ovo puno bolja opcija nego da koristimo veliki multiplekser ispred ALU-a, jer samo jedan elemenat koristi magistralu u tom trenutku).

7.4 RAZBIJANJE IZVRŠAVANJA INSTRUKCIJE NA KORAKE PO TAKTOVIMA

Pošto imamo *Datapath*, pogledajmo što se zbiva u svakom taktu posebno, pošto će nam to pomoći da utvrdimo koji nam možda još elementi u *Datapath-u* fale (naprimjer privremeni registri) i koje eventualno kontrolne signale treba dodati. Potreban je dakle registar koji će čuvati vrijednost signala uvjek kad postoji jedan od slijedeća dva uslova:

1.Signal je izračunat u jednom, a koristi se u drugom taktu;

2.*Input* funkcijskog bloka, čiji je ovaj signal *output*, se može promjeniti prije nego što se signal upiše u *elemenat stanja.*

Naprimjer, treba smjestiti instrukciju u *Instrukcijski registar* (IR) jer funkcijska jedinica (memorija) koja daje tu vrijednost, mijenja *output* prije nego što je završeno korišćenje instrukcije, ili kad se ALU koristi za neku instrukciju *R-tipa* ne trebamo smještati taj *output* iako se on neće koristiti prije slijedećeg takta. To je zbog toga jer se izlaz iz ALU-a ne mijenja (stabilan je) tokom trajanja takta u kojemu je upisan u *Registar fajl.* Izlaz ALU-a je stabilan jer su *inputi* za ALU izlazi iz *Registar fajla* koji su određeni sa *Rs* i *Rt* poljima u *Instrukcijskom registru,* koji je stabilan jer je *elemenat stanja* u koji se upisuje jednom u toku izvršavanja jedne istrukcije. Dakle, funkcijske jedinice od *Registar fajla* do ALU-a prave jedan kombinaciono-logički blok, čiji *inputi* dolaze iz *Instrukcijskog registra* (*elemenat stanja*) i čiji se *outputi* upisuju u *Registar fajl* (koji je takođe *elemenat stanja*), naravno u raznim taktovima za razliku od jedno-taktnog sistema.

Suština je naći balans između količine odrađenog posla u svakom krugu, i minimizacije samog trajanja takta kao dijela tog kruga. Zato možemo izvršavanje instrukcije podijeliti u pet koraka, koji svaki traje 1 takt izbalansiranog trajanja. Naprimjer, svedimo svaki korak tako da sadrži najviše jednu ALU operaciju, ili jedan pristup *Registar fajlu*, ili jedan pristup memoriji. Na ovaj način, takt će biti kratak onoliko kolika je dužina trajanja ovih operacija.

Kod jedno-taktnog *Datapath-a*, svaka instrukcija treba skup *Datapath* elemenata radi izvršavanja. Mnogi od tih elemenata rade u seriji tako što je *output* jednog *input* drugog. Neki elementi rade i u paraleli, naprimjer, PC se inkrementira i instrukcija se čita istovremeno. Slično je i u više-taktnom radu, pa su sve operacije u jednom koraku paralelne tokom takta, a ostali koraci idu u seriji po slijedećim taktovima.

Pogledajmo ovih pet koraka detaljno:

1.Donošenje instrukcije (*FETCH*)

Donesimo instrukciju iz memorije i inkrementiramo brojač

IR = Memory [PC];
PC = PC + 4;

Opis:

Pošaljimo brojač u memoriju kao adresu, pročitajmo i donesimo instrukciju u *Instrukcijski registar* (IR), gdje je smiještamo. Da bi se ovo desilo, moramo pobuditi kontrolne signale *MemRead* i *IRWrite*, i setovati *IorD* na 0 da bi brojač PC bio uzet kao izvor adrese. Takođe, sad inkrementiramo PC za 4 zbog čega je potrebno postaviti *ALUSelB* na 01 i *ALUSelA* na 0 a *ALUOp* na 00 (da bi ALU sabirao). Na kraju želimo smjestiti inkrementiranu adresu instrukcije nazad u PC. Inkrementacija PC-a i pristup instrukciji u memoriji dešavaju se paralelno.

2.Dekodiranje instrukcije i pristup registru

Kao u prošlom, ni u ovom koraku ne znamo koja je instrukcija, znači ova dva koraka su zajednička za sve instrukcije. Ovdje čitamo sadržaje dva registra *Rs* i *Rt*, jer to ne škodi za kasniji rad. Možda će nam njihovi sadržaji kasnije zatrebati, pa ih zovimo A i B, respektivno. *Outpute* registara nije potrebno čuvati u privremenom registru, jer im se *inputi* ne mijenjaju tokom izvršavanja instrukcije (a time ni *outputi*).

Takođe, računamo *Target* adresu za *BRANCH* (uslovne) instrukcije preko ALU-a, što takođe ne smeta jer tu vrijednost možemo ignorisati ako instrukcija nije *BRANCH*. Pošto ne znamo je li instrukcija *BRANCH* (a kamoli gdje treba skočiti), zato jer nam ALU treba za druge operacije u kasnijim taktovima, izračunatu adresu smiještamo u *Target.* Ovo je dobro zbog toga jer smanjujemo broj taktova potrebnih za izvršavanje instrukcija, a moguće je zbog samog formata instrukcija. Recimo, ako instrukcija ima dva registra kao *inpute*, to su uvjek *Rs* i *Rt* polja, i ako je *BRANCH* instrukcija, *offset* je uvjek zadnjih 16 bita u registru:

A = *Register* [IR [25-21]];
B = *Register* [IR [20-16]];
Target = PC + (*znakom-proširen* (IR [15-0])‹‹2);

Opis:

Pristupimo *Registar fajlu* i pročitajmo registar koristeći *Rs* i *Rt* polja (za ovo ne treba nikakav kontrolni signal). Izračuna se *BRANCH Target* adresa i smjesti se u *Target*. Potrebno je znači *ALUSelB* staviti na 11 (jer je tada *offset* i *znakom-proširen* i pomjeren ulijevo 2 mjesta), *ALUSelA* na 0 i *ALUOp* na 00. Da bi dodali *Target* registar treba nam kontrolni signal za taj registar koji mora biti pobuđen. Pristup registrima i računanje *BRANCH Targeta* idu paralelno. Nakon ovoga koraka (takta), za različite tipove instrukcija slijede različiti koraci.

3.Izvršavanje, računanje memorijske adrese, ili finalizacija BRANCH instrukcije

Ovo je prvi korak (takt) kod kojega je *Datapath* zavisan od tipa instrukcije. U svim slučajevima ALU koristi operande iz prethodnog koraka, i izvodi jednu od 3 operacije zavisno od tipa instrukcije. Rezultat operacije zovemo ALU *output* i koristimo ga u slijedećim koracima. Pošto su *inputi* za ALU stabilni, ovu vrijednost ne treba čuvati u registru.

Znači, svaki signal setovan u ovom taktu, a koji utiče na ALU rezultat, mora ostati isti dok se ALU rezultat ne upiše u registar ili nam više ne bude potreban.

Zavisno od tipa instrukcije imamo slijedeće varijante:

-Memorijski tip instrukcija

ALUoutput = A + *znakom-proširen* (IR [15-0]);

Opis:

ALU sabira operande da bi dobio novu memorijsku adresu. Potrebno je znači da vrijednost *ALUSelA* bude 1, što znači uzimanje *outputa* prvog registra za prvi input ALU-a, i da vrijednost *ALUSelB* bude 10, čime drugi ALU *input* biva *znakom-proširena* vrijednost *offseta*. *ALUOp* je na 00, znači sabiramo.

-Aritmetičko-logičke instrukcije (R-tip)

ALUoutput = A op B;

Opis:

ALU obavlja operaciju specificiranu u polju *opcode* nad sadržajem dva registra pročitana u predhodnom taktu. Potrebno je da je *ALUSelA* = 1 i *ALUSelB* = 00, što znači
da će *outputi Registar fajla* biti uzeti kao *inputi* ALU-a. *ALUOp* treba biti 10, pa funkcijski kod određuje setovanje *ALUOp* kontrolnog signala.

-BRANCH (uslovne) instrukcije

if (A == B) PC = *Target;*

Opis:

ALU radi poređenje da li su jednaki sadržaji dva registra pročitana u predhodnom taktu. *Zero* signal ALU-a se koristi da odredi je li uslov zadovoljen ili ne. Treba znači da je *ALUSelA* = 1 i *ALUSelB* = 00, kao i kod *R-tipa* instrukcija. *ALUOp* treba biti 01 za oduzimanje zbog poređenja. *Write* signal mora biti *okinut (trigered)* za novu vrijednost PC-a ako je *Zero output* pobuđen.

4.Pristup memoriji ili finalizacija instrukcije *R-tipa*

Tokom ovog koraka, učitavaju se ili smještaju podaci *u* ili *iz* memorijskih pristupa i upisuje se rezultat aritmetičko-logičke operacije. Izlaz iz memorije zovemo *memory-data*, jer ne korespondira sa registarskim izlazom, zato što je taj *output* stabilan tokom slijedećeg takta, sve dok se ne upiše u registar.
I ovdje razlikujemo dva tipa instrukcija:

-memorijski tip instrukcije

memory-data = Memory [*ALUoutput*];

ili

Memory [*ALUoutput*] = B;

Opis:
Ako je instrukcija *Load*, podatak se vraća iz memorije i mi tu vrijednost zovemo *memory-data*. Ako je instrukcija *Store*, tada se podatak upisuje u memoriju. U oba slučaja adresa je izračunata u prethodnom koraku *ALUoutput*. Kontrolni signali za ALU-e moraju ostati isti setovani u prethodnom taktu. Za *Store*, izvor operand zvani B je pročitan u koraku prije dva takta. Signal *MemRead* (za *Load*) ili *MemWrite* (za *Store*) moraju biti pobuđeni zavisno od instrukcije. Također, signal *IorD* mora imati vrijednost 1 da bi memorijska adresa došla iz ALU-a, a ne iz PC-a.

-aritmetičko-logička instrukcija (R-tip)

Reg [IR (15-11)] = *ALUoutput*;

Opis:
Smješta se rezultat ALU operacije u *Result* registar. Signal *RegDst* mora biti 1 da bi *Rd* polje (biti 15-11) bilo korišćeno kao mjesto za upis u registru. *RegWrite* mora biti pobuđen, i *MemtoReg* mora biti 0 pa je upisan *ALUoutput* (suprotno *memory-data outputu*). *ALUSelA*, *ALUSelB* i *ALUOp* moraju ostati nepromijenjeni jer se upis u *Rd* okida na ivici takta pa ne utiče na tekuće čitane podatke.

5.Korak upisa-nazad

Reg [IR [20-16]] = *memory-data*;

Opis:
Upišu se učitani podaci iz memorije u *Registar fajl*. Znači *MemtoReg* = 1, za upis rezultata iz memorije, i *RegWrite* = 1 da se omogući upis, i *RegDst* = 0 za izbor *Rt* polja (biti 20-16) za smještanje. *ALUSelA*, *ALUSelB* i *ALUOp* moraju ostati nepromijenjeni i do kraja ovog takta.
Ovih pet koraka je sumirano tabelom na slici 7.21.

Ime koraka	Akcija za R-tip instrukciju	Akcija za instrukciju memorijskog tipa	Akcija za Branch instrukciju	Akcija za Jump instrukciju
Instruction fetch (donošenje instrukcije)	IR = Memory[PC] PC = PC + 4			
Instruction decode/register fetch (dekodiranje i pristup registru)	A = Reg [IR[25-21]] B = Reg [IR[20-16]] ALUOut = PC + (sign-extend (IR[15-0]) << 2)			
Izvršavanje,traženje adrese , branch/ jump kompletiranje	ALUOut = A op B	ALUOut = A + sign-extend (IR[15-0])	if (A ==B) then PC = ALUOut	PC = PC [31-28] II (IR[25-0]<<2)
Pristup memoriji ili kompletiranje R-tipa	Reg [IR[15-11]] = ALUOut	Load: MDR = Memory[ALUOut] or Store: Memory [ALUOut] = B		
Citanje iz memorije		Load: Reg[IR[20-16]] = MDR		

Slika 7.21

Znači sada bi smo mogli odrediti kontrolne signale za svaki takt pojedinačno, ali prije nego što dizajniramo kontrolnu jedinicu moramo dodati *write* kontrolu za PC i multipleksere za odabir vrijednosti koja će se upisati u PC, kao i *Target* registar i njegov kontrolni signal. Takođe dodaćemo i kontrolni signal za *JUMP* instukciju, pa ćemo imati tri mogućnosti za vrijednost koja će se upisati u PC. To su:

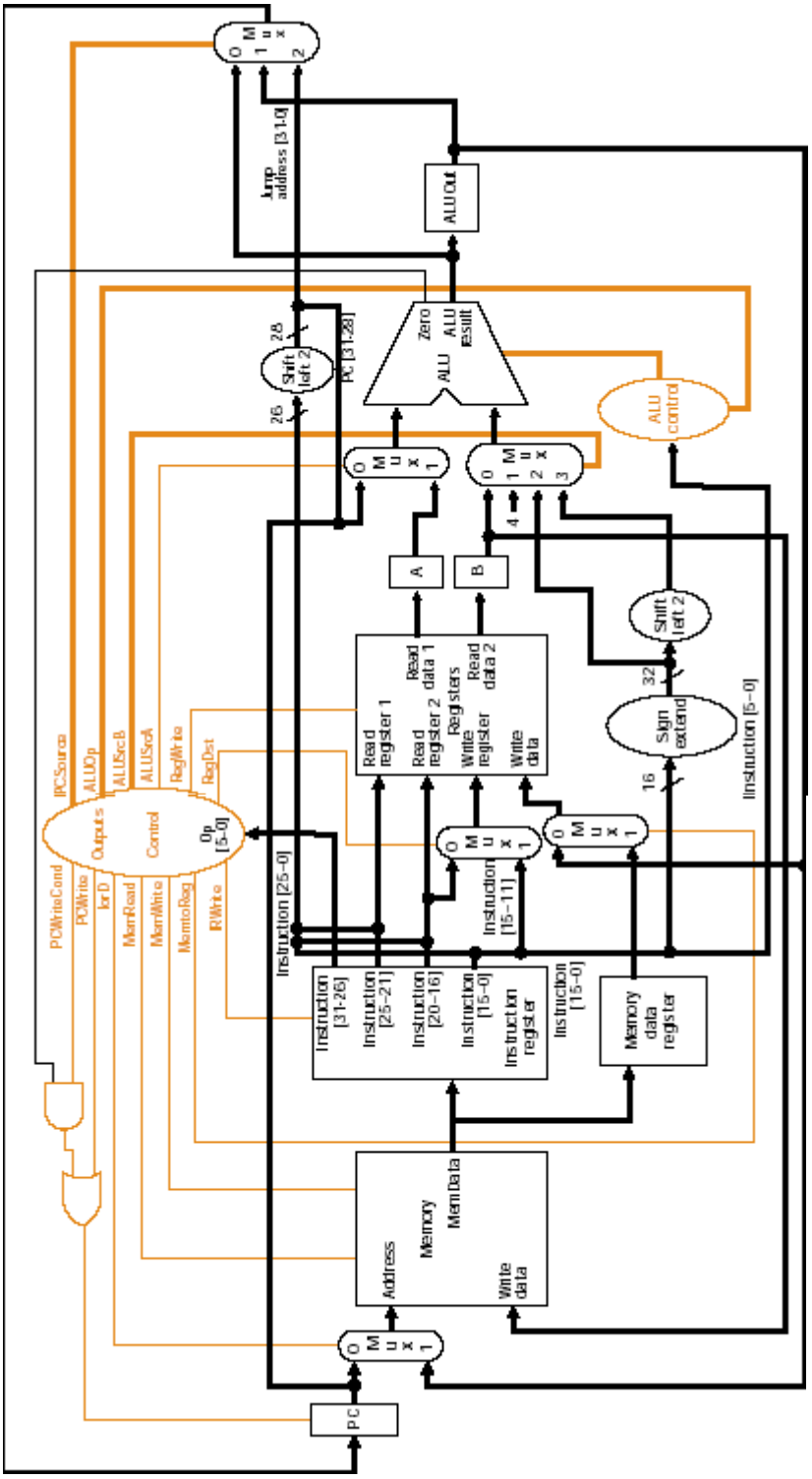
1. *ALUoutput*, kad se brojač inkrementira za slijedeću instrukciju sekvencijalno;

2. *Target registar*, kad je kod *BRANCH* instrukcije zadovoljen uslov, i tu nam treba i *TargetWrite* signal za upis u registar;

3. 26 bita manje težine iz *Instrikcijskog registra* (IR) pomjereni ulijevo za 2 mjesta i povezani sa 4 bita od brojača PC koji idu na mjesto bita veće težine (ukupno 32 bita), kad se radi o *JUMP* instrukciji.

Da bi smo odredili koju od gore navedene mogućnosti koristimo, uvodimo 2-bitni kontrolni signal *PCSource* sa slijedećim vrijednostima: za *ALUoutput* (00), za *Target* (01) i za IR (10), koji očigledno upravlja multiplekserom koji ima tri ulaza.

Kod jedno-taktnog sistema rada, PC se upisuje dvojako. Ako instrukcija nije uslovna PC se upiše bezuslovno, a kad je uslovna inkrementirani PC se zamijeni vrijednošću iz *Target*-a samo ako je *Zero* signal pobuđen. Znači, trebaju nam dva PC *write* signala, *PCWrite* i *PCWriteCond*. *PCWriteCond* i *Zero* signal iz ALU-a idu na *I-kolo*, pa se to kombinuje sa *PCWrite* preko *ILI-kola* da dobijemo *write* signal za PC.
Na slici 7.22 se vidi kompletan *Datapath* i *Kontrola*, sa svim dodatnim kontrolnim signalima, *Target* registrom i multiplekserom za PC, a tabela 7.23 pokazuje efekte kontrolnih signala.



Slika 7.22

Ime signala	Efekat kad nije pobuđen	Efekat kad je pobuđen
PCWrite	nema	PC=PCSource
PCWriteCond	nema	Ako je Zero signal iz ALU-a pobuđen onda je PC=PCSource
TargetWrite	nema	Target reg. = izlazu iz ALU -a

1-bitni kontrolni signali

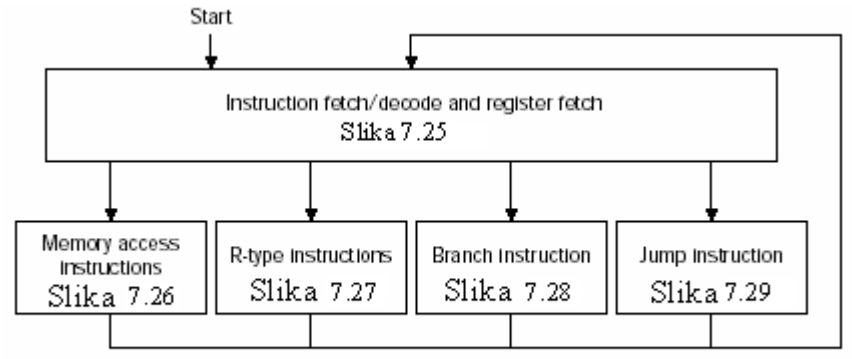
Ime signala	Vrijednost	Efekat
PCSource	00	PC = ALU outputu
	01	PC = sadržaju Target-a
	10	PC =PC+4[29-26] : IR[25-0]<<2

2-bitni kontrolni signal

Slika 7.23

7.5 DEFINISANJE KONTROLE

Kad imamo sve kontrolne signale i znamo kad trebaju biti pobuđeni ili ne, lako je implementirati *Kontrolnu jedinicu*. Tu su nam od velike važnosti tabele sa opisom stanja kontrolnih signala. Postoje dvije različite tehnike za specificiranje kontrole. Prva je zasnovana na *"određenim stanjima mašine"* i obično se grafički prikazuje, a druga je *"mikroprogramiranje"* i prikazuje se kroz program. Mi ćemo se ovdje osvrnuti na prvi način *"određena stanja mašine"* koji je sastavljen od niza karakterističnih stanja i pravaca koji pokazuju kako se ta stanja mijenjaju i ukazuju na slijedeće stanje, koje u suštini zavisi od toga da li su neki kontrolni signali pobuđeni ili ne. Svako stanje odgovara jednom taktu. Prva dva stanja su ista za sve tipove instrukcija, a od stanja 2 do stanja 4 sve zavisi od polja *opcode.* Nakon poslijednjeg stanja, vraćamo se opet na inicijalno stanje za novu instrukciju. Slika 7.24 pokazuje ugrubo *"određeno stanje mašine"*.

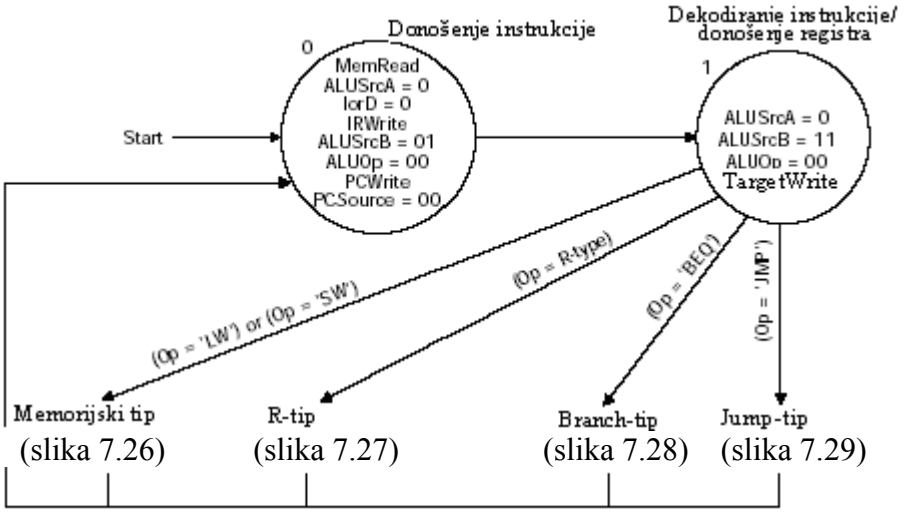


Slika 7.24

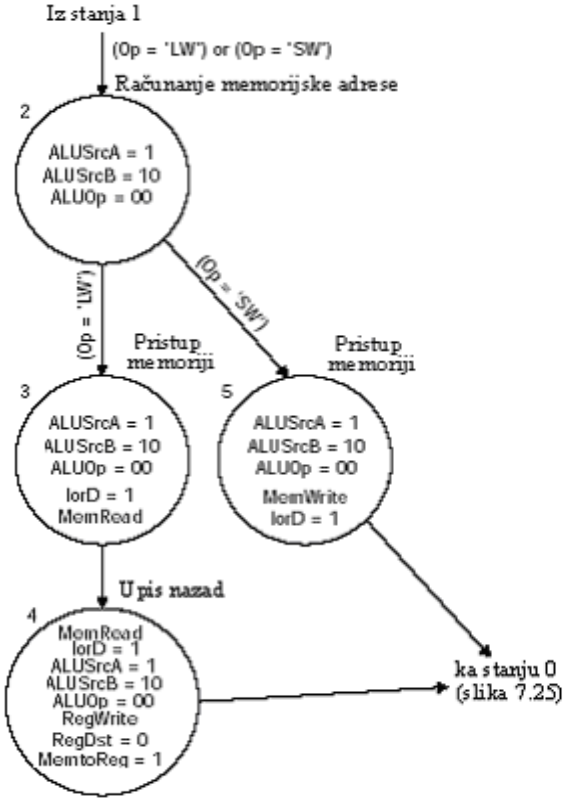
Slika 7.25 pokazuje korake 0 i 1 *Datapath*-a. Nakon stanja 1 imamo četiri mogućnosti zavisno od tipa instrukcije (memorijski tip, *R-tip*, uslovni (*BRANCH*) tip, *JUMP* tip). To je ustvari dekodiranje instrukcije. Za memorijski tip je ovaj korak prikazan slikom 7.26, zavisno od togaradi li se o *Load* ili *Store* instrukciji.

R-tip prikazan je na slici 7.27, dok su *BRANCH* i *JUMP* instrukcije prikazane slikama 7.28 i 7.29, respektivno.

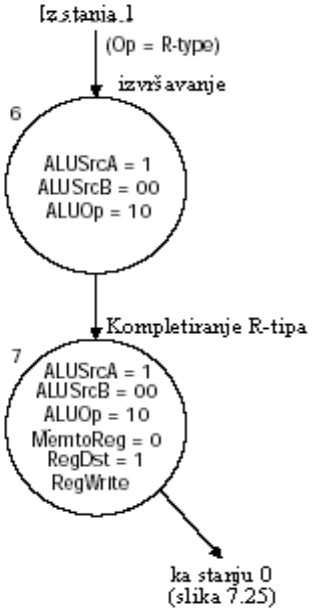
Kontrolni signali za kontolu rada multipleksera *ALUSelA* i *ALUSelB* često se u literaturi obilježavaju i kao *ALUSrcA* i *ALUSrcB*, pa su kao takvi prikazani i na dijagramima.



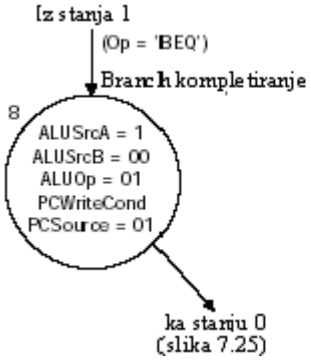
Slika 7.25



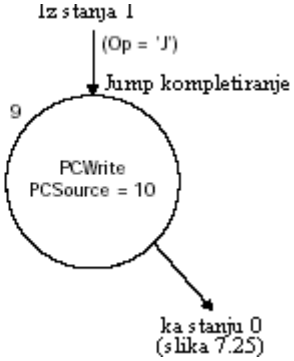
Slika 7.26



Slika 7.27

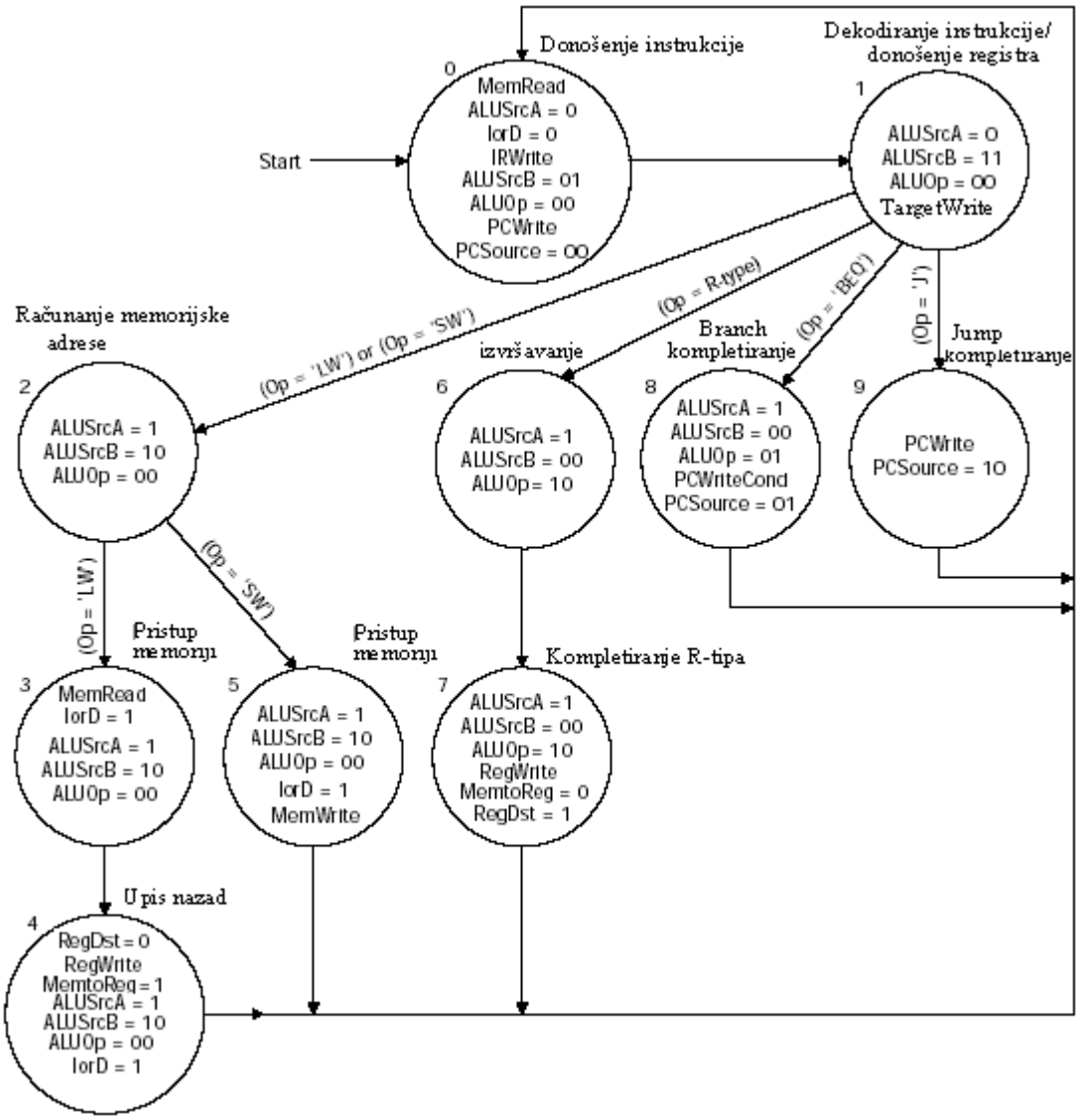


Slika 7.28



Slika 7.29

Na kraju, na slici 7.30 imamo kompletan prikaz *"konačnih stanja mašine"* sa detaljnim prikazom pobuđenih signala koji odgovaraju nekom određenom stanju. *"Konačno stanje mašine"* poznato je i po nazivu *Moore-ova mašina*, po Eduardu Moore-u i ima karakteristiku da *output* zavisi samo od trenutnog stanja, a kod drugog pomenutog tipa, poznatog i pod imenom *Mealy-jeva mašina,* i *input* i trenutno stanje određuju *output*. Pojašnjenja svih stanja i odgovarajućih im kontrolnih signala, data su poglavlju o razbijanju izvršavanja instrukcije na korake po taktovima.



Slika 7.30

8. ZAKLJUČAK

Kao što smo vidjeli, iako računar radi izuzetno komplikovane operacije, suština njegovog funkcionisanja leži u elementima koje smo predstavili i koji su u osnovi izuzetno jednostavni. Pored obješnjenja za rad sa osnovnim instrukcijama, u prilogu možemo vidjeti i neke atipične instrukcije koje se opet po logici mogu svrstati u jedan od tipova o kojima je bilo riječi, a na osnovu svega rečenog uz mali napor bi smo trebali lako riješavati i probleme konstrukcije *Datapath*-a i *Kontrole* i za neke nove instrukcije ili pseudoinstrukcije koje nemaju *hardware*-sku implementaciju u osnovnoj verziji *Datapath*-a prezentovanoj ovim radom. Savremeni procesori imaju sve veće zahtjeve i, praktično iz dana u dan, sve bolje performanse iako u suštini svi rade na principima o kojima smo govorili. Zahvaljujući razumjevanju rada računara i *hardware*-a koji ga čini, počev od memorija, registara, aritmetičko-logičke jedinice i ostalog, danas smo više nego ikad ranije spremni da napravimo *software* kojima bi uz pomoć računara riješavali najraznorodnije probleme i zahtjeve. Skoro da ne postoji ni jedna grana obrazovnog, društvenog i uopšte života, koju bi smo u XXI vijeku mogli zamisliti bez računara. To je od izuzetnog značaja zbog toga jer su se, zahvaljujući razvoju procesora, odnosno računara u cjelini, pojedine grane kao što su medicina, astronomija, ekonomija, telekomunikacije, filmska industrija, muzička industrija i druge razvile do neslućenih razmjera, i time u mnogome poboljšale i olakšale kvalitet življenja prije svega. Na kraju, iako nam se danas čini da je razvoj procesora dotakao tačku preko koje se teško može puno dalje naprijed, sa sigurnošću možemo tvrditi da nas čekaju nova iznenađenja u čijoj će osnovi ležati upravo univerzalni principi čije smo osnove upoznali tokom ovog rada.

TEORIJSKA PITANJA ZA PROVJERU ZNANJA

1. MIPS procesor izvršava instrukciju **beq $4, $5, 1021** i sadržaji registara $4 i $5 su različiti. Šta će tačno biti upisano u PC i koji signali omogućuju ovaj upis?

Odgovor:

Stanje 0: **PCWrite**=1, **PCSource**=00 → PC=PC+4.
Stanje 1: isto
Stanje 8: **PCWriteCond**=1, **Zero**=0, **PCWrite**=0 → isto

Dakle, u registar PC biće upisano PC+4.

2. Čemu služi **Shift left 2** jedinica na posljednjem ulazu multipleksora (drugi ulaz ALU) i u toku izvršavanja kojih instrukcija ona dolazi do izražaja?

Odgovor:

Shift left 2 (množenje sa 4) se koristi prilikom računanja adrese uslovnog skoka kod instrukcija uslovnog skoka.

3. U toku izvršavanja kojih instrukcija se vrši upis u **Target** registar? Šta se dešava sa njegovim sadržajem u toku izvršavanja instrukcije **jal 2000**?

Odgovor:

Upis u **Target** registar se vrši u toku izvršavanja svih instrukcija (stanje 1 je zajedničko). Sadržaj ovog registra u toku izvršavanja instrukcije **jal 2000** je PC+4+4*2000.

4. MIPS procesor redom izvršava instrukcije **slti $8, $0, 1** i **bne $8, $0, 111**. Šta će biti tačno upisano u PC registar nakon njihovog izvršenja? Pojasniti!

Odgovor:

slti $8,$0,1: PC=PC+4, u $8 je 1
bne $8,$0,111: PC=PC+4, uslov je ispunjen → PC=PC+4*111

Dakle, u registar PC biće upisano PC+8+4*111.

5. U toku je izvršavanje instrukcije **lw $20, 20($20)**. Objasniti proceduru formiranja adrese sa koje se preuzima podatak za upis u odgovarajući registar.

Odgovor:

Računanje adrese sa koje se preuzima podatak za upis u odgovarajući registar obavlja se u stanju 2: **ALU result**=IR[25-21]+signextended(IR[15-0]). Signal **ALU result** se vodi na ulaz 1 MUX-a na **Read address** ulazu **Memory** jedinice, kao adresa sa koje je potrebno pročitati podatak za upis u registar $20.

6. MIPS procesor izvršava instrukciju **bne $13, $0, 130**, a prije nje je izvršio instrukciju **addi $13, $0, 0**. Šta će biti upisano u PC u posljednjem taktu izvršavanja pomenute bne instrukcije? Objasniti!

Odgovor:

Nakon izvršenja instrukcije **addi $13, $0, 0**, u $13 je upisana nula i PC=PC+4. Obzirom da, u toku izvršenja instrukcije **bne $13, $0, 130**, neće doći do skoka (sadržaj $0 i $13 nije različit), sadržaj PC registra biće uvećan za 4.

Dakle, u registar PC biće upisano PC+8.

7. MIPS procesor izvršava instrukciju **beq $3, $7, 1021** i sadržaji registara $3 i $7 su isti. Šta će biti upisano u PC i koji signali omogućuju ovaj upis?

Odgovor:

Stanje 0: PC=PC+4 (**PCWrite**=1, **PCSource**=00)
Stanje 1: PC=PC+4*1021 (**PCWriteCond**=1, **Zero**=1, **PCSource**=01)
Dakle, u registar PC biće upisano PC+4+4*1021.

8. MIPS procesor redom izvršava instrukcije **slti $8, $0, 1021** i **bne $8, $0, 1021**. Šta će biti tačno upisano u PC registar nakon njihovog izvršenja? Pojasniti!

Odgovor:

slti $8, $0, 1021: PC=PC+4, u $8 je 1
bne $8, $0, 1021: PC=PC+4, uslov je ispunjen → PC=PC+4*1021

Dakle, u registar PC biće upisano PC+8+4*1021.

9. Čemu služi **Zero** signal na izlazu ALU? Pojasniti njegovu ulogu prilikom izvršavanja instrukcije **lw $3, 1021($10)**.

Odgovor:

Zero signal se dobija kao rezultat NOR operacije nad svim bitima rezultata ALU-e. Ovaj signal nema nikakvu ulogu prilikom izvršavanja instrukcije **lw $3, 1021($10)**.

10. Od MIPS instrukcija koje su implementirane datom šemom, koja se instrukcija izvršava najkraće, a koja najduže, i koliko traje njihovo izvršavanje izraženo u broju clock-ova (ciklusa)?

Odgovor:

Najduže traje **load** - pet taktova, a po tri takta traju **jump** i **branch**.

11. Zbog čega je potrebno uvesti **Target** registar u multicycle implementacionu šemu?

Odgovor:

Za realizaciju pojedinih instrukcija, potrebno je ALU-u koristiti više puta (u više taktova). U stanju 1 (koje je zajedničko) ALU se koristi za izračunavanje adrese na koju se usmjerava tok programa za slučaj da je u pitanju instrukcija uslovnog skoka. Izračunata adresa se čuva u **Target** registru. Nakon toga, ALU se može koristiti za potpunu implementaciju date instrukcije.

12. Čemu služi multiplekser na **Write data** ulazu **Registers** jedinice i u toku izvršavanja kojih instrukcija on dolazi do izražaja?

Odgovor:
Multiplekser na **Write data** ulazu **Registers** jedinice služi za odlučivanje da li će se u određeni registar **Registers** jedinice upisati rezultat neke aritmetičko-logičke operacije doveden sa **ALU result** izlaza ALU-e (instrukcije R-tipa), ili, podatak pročitan iz memorije (instrukcija **lw**).

13. U toku je izvršavanje instrukcije **sw $21, 120($19)**. U kom taktu izvršenja ove instrukcije se formira memorijska adresa u koju se upisuje sadržaj predmetnog registra? Objasniti proceduru formiranja ove adrese!

Odgovor:
Računanje adrese u koju se upisuje sadržaj registra obavlja se u stanju 2: **ALU result**= IR[25-21]+signextended(IR[15-0]). Signal **ALU result** se vodi na **Write address** ulaz **Memory** jedinice, kao adresa na koju je potrebno upisati sadržaj registra $21.

14. MIPS procesor izvršava instrukciju 1010 1110 1100 1100 0000 0100 0000 0010, koja se nalazi na adresi 100. Šta će tačno biti upisano u **Target** registar u toku izvršenja ove instrukcije i u kojem taktu? Objasniti! Koristi li se sadržaj ovog registra tokom izvršavanja ove instrukcije?

Odgovor:

OP=$101011_{(2)}=43_{(10)}$=0x2B→ instrukcija **sw**
address=$0000010000000010_{(2)}=1026_{(10)}$
Stanje 0: PC=PC+4=100+4=104
Stanje 1: **Target**=PC+4*1026=104+4*1026
Sadržaj registra **Target** se ne koristi tokom izvršavanja instrkcije.

15. Čemu služi **Shift left 2** jedinica na posljednjem ulazu multipleksora koji vodi u PC i u toku izvršavanja kojih instrukcija ona dolazi do izražaja?

Odgovor:
Shift left 2 služi za množenje sa 4, prilikom računanja adrese bezuslovnog skoka. Koristi se kod naredbi J-tipa.

16. U toku je izvršavanje instrukcije **beq $21, $19, 160**. Koliko iznosi adresa na koju se ″skače″ i od čega zavisi da li će doći do skoka?

Odgovor:
Target=PC+4+4*160
Da li će doći do skoka zavisi od sadržaja registara $21 i $19, odnosno od vrijednosti **Zero** signala. Naime, ako su sadržaji registara $21 i $19 jednaki, na **ALU result** izlazu ALU-e će biti nula (stanje 8). **Zero** signal u tom slučaju ima vrijednost jedan. Obzirom da je **PCWriteCond**=1 i **PCSource**=01, u PC registar se upisuje vrijednost iz **Target** registra. Kao posljedica, tok izvršavanja programa usmjerava se na instrukciju na adresi PC+4+4*160 (došlo je do ″skoka″). Ukoliko su sadržaji registara $21 i $19 različiti, na **ALU result** izlazu ALU-e će biti vrijednost različita od nule. **Zero** signal u tom slučaju ima vrijednost nula. Dakle, u PC registru bi ostala vrijednost PC+4 (nije došlo do ″skoka″).

17. U toku izvršavanja kojih instrukcija se vrši upis u **Target** registar? Šta se dešava sa njegovim sadržajem u toku izvršavanja instrukcije **beq $13, $17, 221**?

Odgovor:
Upis u **Target** registar se vrši u toku izvršavanja svih instrukcija (stanje 1 je zajedničko). **beq $13, $17, 221**: **Target**=PC+4+4*221

Nekoliko primjera različite konstrukcije Datapath-a i kontrole

1. Prikazati i objasniti sve promjene u *Datapath-u* i dijagramu stanja (uključujući kontrolne signale) potrebne za implementaciju ***sllv*** instrukcije.
Sintaksa nove instrukcije je:

Sllv $Rd, $Rt, $Rs

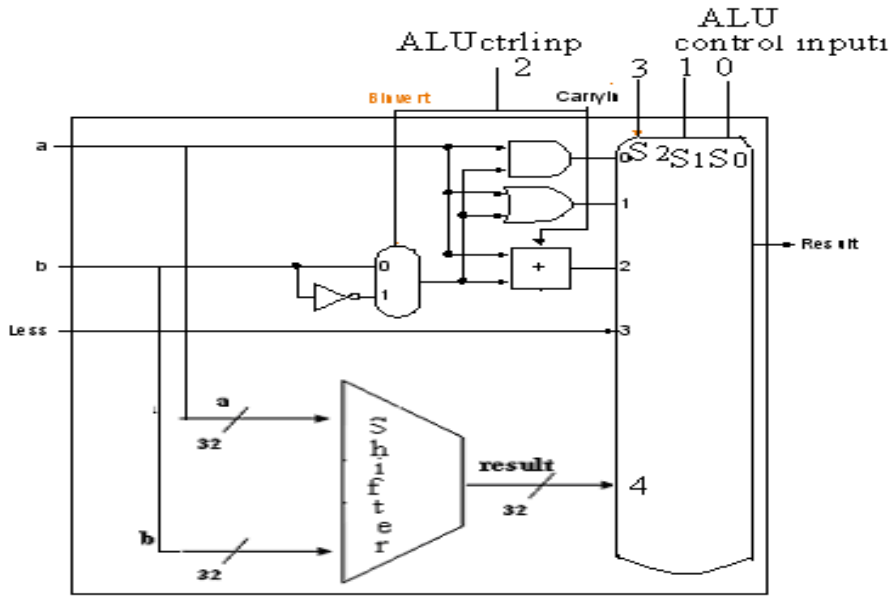
Format:

0	Rs	Rt	Rd	0	4

Op (6 bitova): operacioni kod instrukcije (0x0)
Rs (5 bitova): registar koji sadrži iznos za koji treba ″šiftovati″
Rt (5 bitova): registar u kome se nalazi operand koji treba ″šiftovati″
Rd (5 bitova): registar u kome se smješta rezultat *sllv* instrukcije

Opis: Pomjeranje sadržaja registra *Rt* ulijevo za iznos dat u registru *Rs* i smještanje rezultata u *Rd*.
Rješenje:

Pošto jedan MUX mora dati na izlazu sadržaj registra *Rs* (**a**), a drugi sadržaj registra *Rt* (**b**), da bi smo kao izlaz iz ALU-a dobili ″šiftovanu″ vrijednost iz registra *Rt* proširićemo ALU tako što ćemo dodati ″šifter″ koji radi logičko pomjeranje ulijevo, i njegov izlaz dovesti na ulaz "4″ MUX-a unutar ALU-a. Tu će podatak **a** (odnosno njegovih 5 bita najmanje težine, pošto je podatak 32-bitni) kao prvi ulaz u ″šifter″, određivati za koliko ćemo logički pomjeriti podatak **b** (drugi ulaz u ″šifter″). ″Šifter″ će raditi logičko pomjeranje ulijevo tako što će na onoliko mjesta koliko to pokazuju 5 bita najmanje težine od **a**, na bite najmanje težine **b**-a staviti nule i prethodno za isto toliko mjesta pomjeriti logički ulijevo sadržaj **b**. Da bi ALU ovu vrijednost, koja će uvjek postojati na ulazu "4″ u MUX, dao kao izlaz neophodno je da *ALUctrlinp*(biti 3 1 0) bude =100.

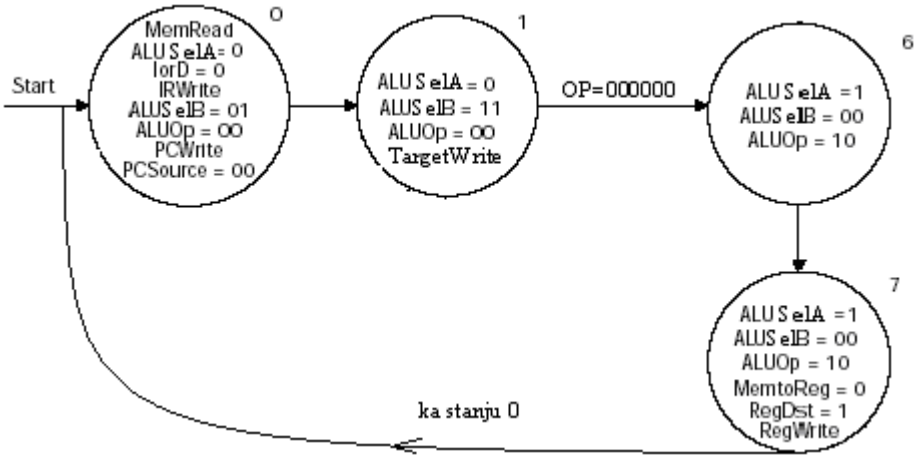


ALUOp mora biti =10 da bi funkcijsko polje odredilo ALU operaciju.

Dakle, tabela sa kodovima će biti proširena za ***sllv*** vrijednosti:

Instrukcija iz opcode-a	**ALUop** ALUop1 Aluop0	**Funkcijsko polje IR [5 - 0]**	**ALU control input 3 2 1 0**
LW	0 0	X X X X X X	0 1 0
SW	0 0	X X X X X X	0 1 0
Branch on equal	0 1	X X X X X X	1 1 0
R-tip (add)	1 0	1 0 0 0 0 0	0 1 0
R-tip (sub)	1 0	1 0 0 0 1 0	1 1 0
R-tip (and)	1 0	1 0 0 1 0 0	0 0 0
R-tip (or)	1 0	1 0 0 1 0 1	0 0 1
R-tip (slt)	1 0	1 0 1 0 1 0	1 1 1
sllv	**1 0**	**0 0 0 1 0 0**	**1 0 0 0**

U suštini, zbog vrijednosti *opcode*-a (000000) dijagram stanja će odgovarati *R-tipu* instrukcija:



U kontrolnoj jedinici nema nikakvih promjena, a za *Datapath* su i dalje dovoljna 4 flip-flopa.

2. Prikazati i objasniti sve promjene u *Datapath-u* i dijagramu stanja (uključujući kontrolne signale) potrebne za implementaciju ***xori*** instrukcije.
Sintaksa nove instrukcije je:

xori $Rt, $Rs, immediate

Format:

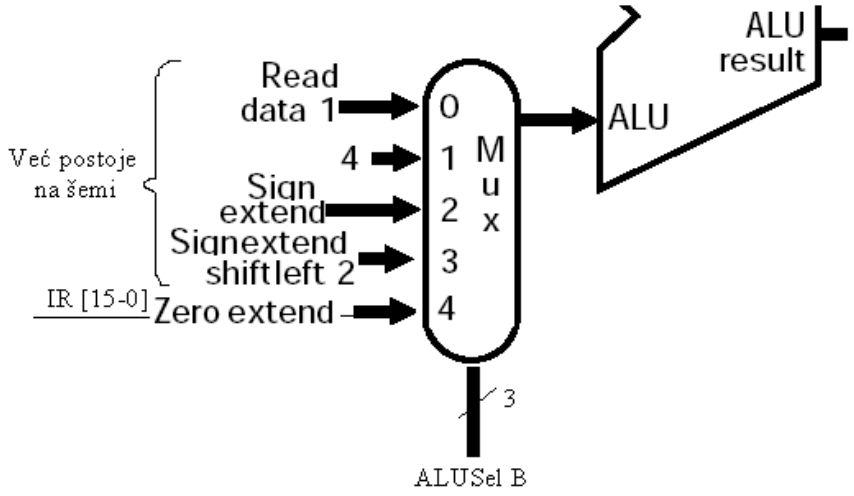
op	Rs	Rt	immediate

Op (6 bitova): operacioni kod instrukcije (001110)
Rs (5 bitova): registar koji sadrži jedan operand
Rt (5 bitova): registar u kome se smješta rezultat *xori* instrukcije
Immediate: 16-tobitna konstanta

Opis: 16-tobitna konstanta se proširuje nulama (proširuje se do 32 bita dodavanjem 16 nula na bitove više težine) i izvršava se "*ekskluzivno ILI*" operacija između bitova proširene konstante i sadržaja registra *Rs*.Rezultat se smješta u registar *Rt*.

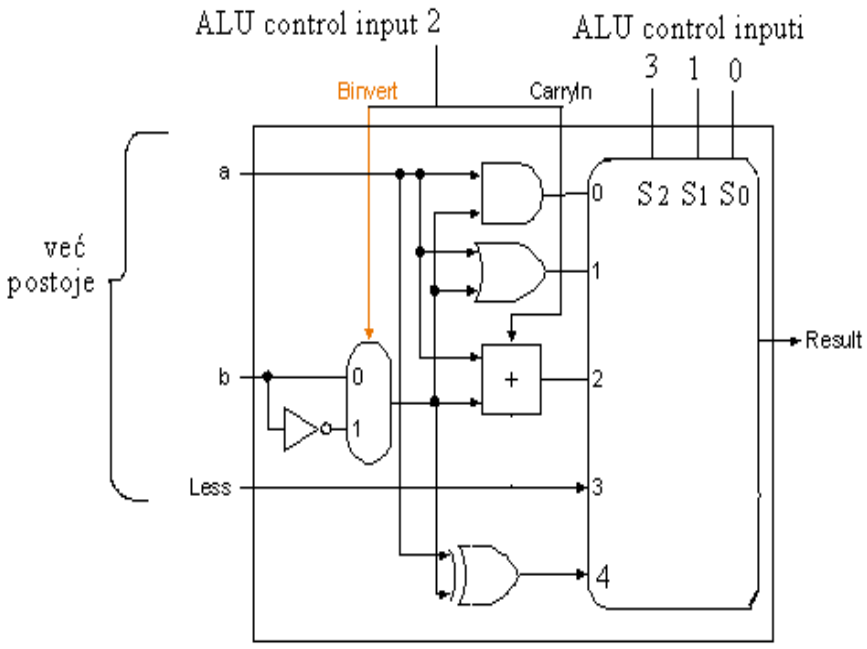
Rješenje:

Pošto se konstanta proširuje nulama, a u šemi (slika 7.22) nema *zero-extender*-a moramo ga dodati. Zato se proširuje MUX:



Sada znači *ALUSelB* ima 3 bita i kada je *ALUSelB* = 100 (binarno), konstanta proširena nulama dolazi na drugi ulaz ALU-a.

U ALU-u nije implementirana operacija XOR. Zato se ALU prepravlja:

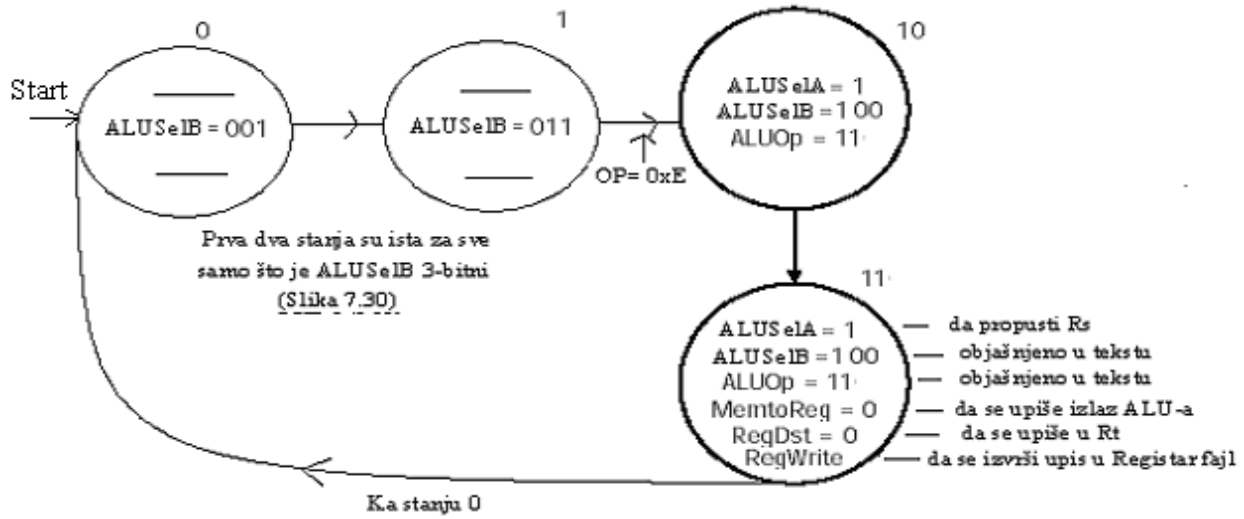


Dakle, tabela sa kodovima će biti proširena za ***xori*** vrijednosti:

Instrukcija iz opcode-a	**ALUop** ALUop1 Aluop0	**Funkcijsko polje IR [5 - 0]**	**ALU control input 3 2 1 0**
LW	0 0	X X X X X X	0 1 0
SW	0 0	X X X X X X	0 1 0
Branch on equal	0 1	X X X X X X	1 1 0
R-tip (add)	1 0	1 0 0 0 0 0	0 1 0
R-tip (sub)	1 0	1 0 0 0 1 0	1 1 0
R-tip (and)	1 0	1 0 0 1 0 0	0 0 0
R-tip (or)	1 0	1 0 0 1 0 1	0 0 1
R-tip (slt)	1 0	1 0 1 0 1 0	1 1 1
xori	**1 1**	**X X X X X X**	**1 0 0 0**

Pošto nema funkcijskog polja (tamo je operand-konstanta) koristimo *ALUop*=11.
Signal *ALUctrlinp2* (*bnegate*) ovdje nije bitan pa ga možemo staviti 0. Selekt ulazi MUX-a moraju biti =100 da propušte ulaz ″4″. Dakle *ALUctrlinp3* = 1, *ALUctrlinp 1*= 0 i *ALUctrlinp0* = 0. Iz tabele se vidi da je *ALUctr inp3 = Aluop1*Aluop0.*

Dijagram stanja ima izgled (prva dva stanja su kao u prethodnom primjeru osim što je *ALUSelB* ovdje 3-bitni, pa ćemo pokazati samo njega):



Dakle, dodali smo dva nova stanja (ukupno ih je 12: *0,1,…,11*), i dalje je dovoljno 4 flip-flopa za *Datapath*, dodali smo novu kontrolnu liniju za *ALUSelB* i promjenili funkcije *ALUctrlinp-uta* za MUX u ALU-u.

3. Prikazati i objasniti sve promjene u *Datapath-u* i dijagramu stanja (uključujući kontrolne signale) potrebne za implementaciju ***ori*** instrukcije.
Sintaksa nove instrukcije je:

ori $Rt, $Rs, immediate

Format:

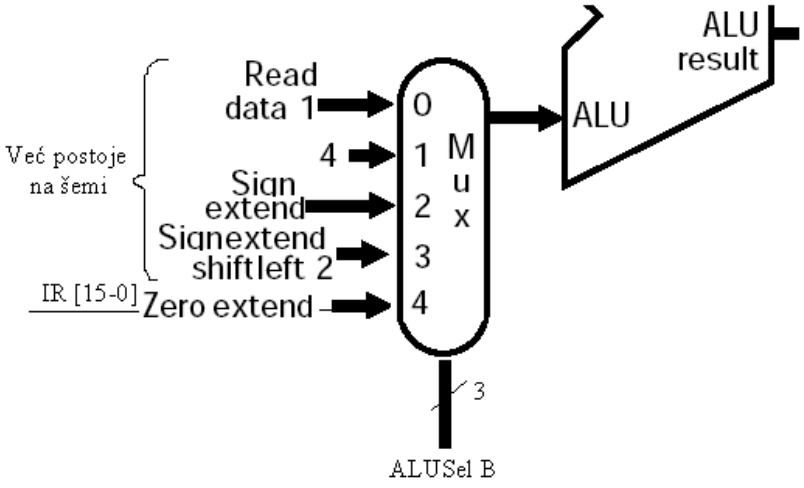
op	Rs	Rt	immediate

Op (6 bitova): operacioni kod instrukcije (0xD)
Rs (5 bitova): registar koji sadrži jedan operand
Rt (5 bitova): registar u kome se smješta rezultat *ori* instrukcije
Immediate: 16-tobitna konstanta

Opis: 16-tobitna konstanta se proširuje nulama (*zero-extended*) i vrši se logičko sabiranje između bitova proširene konstante i sadržaja registra *Rs*.Rezultat se smješta u registar *Rt*.

Rješenje:

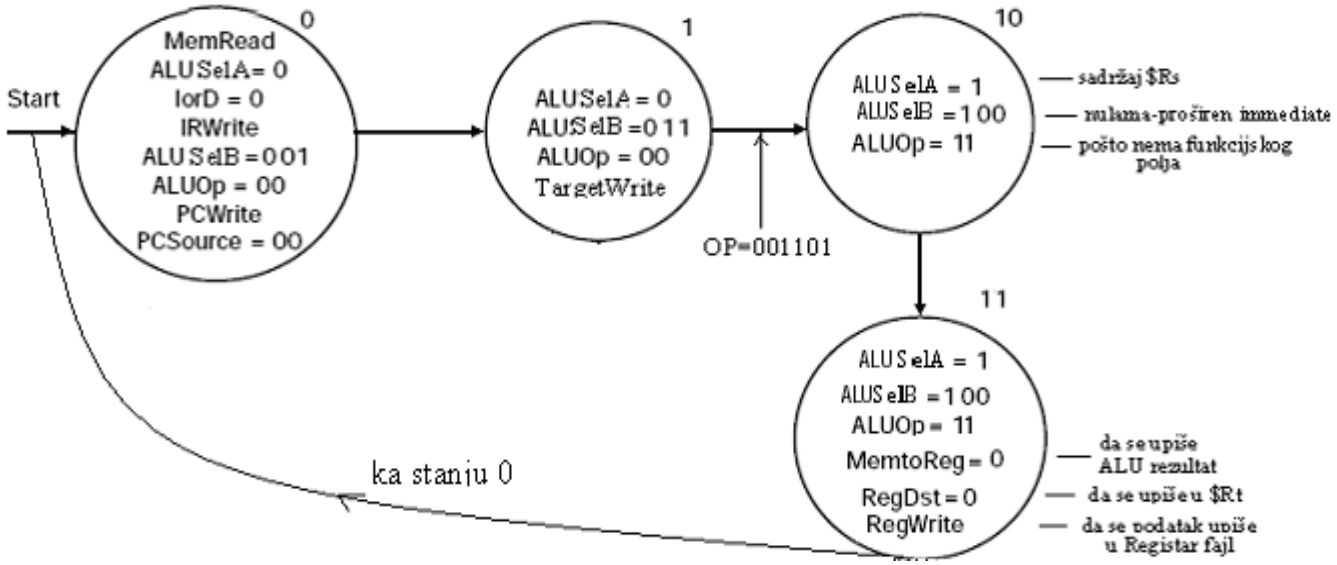
Pošto se vrijednost *immediate* proširuje nulama, a u šemi (slika 7.22) nema *zero-extender*-a moramo ga dodati kao peti ulaz (″4″) u MUX, kao na slici:



Sada znači *ALUSelB* ima 3 bita i kada je *ALUSelB* = 100 (binarno), konstanta proširena nulama dolazi kao drugi ulaz ALU-a, koji nije potrebno mijenjati jer je operacija ***or*** već implementirana tako da drugih promjena nema.
Dakle, tabela sa kodovima će biti proširena za ***ori*** vrijednosti:

Instrukcija iz opcode-a	**ALUop** ALUop1 Aluop0	**Funkcijsko polje IR [5 - 0]**	**ALU control input 2 1 0**
LW	0 0	X X X X X X	0 1 0
SW	0 0	X X X X X X	0 1 0
Branch on equal	0 1	X X X X X X	1 1 0
R-tip (add)	1 0	1 0 0 0 0 0	0 1 0
R-tip (sub)	1 0	1 0 0 0 1 0	1 1 0
R-tip (and)	1 0	1 0 0 1 0 0	0 0 0
R-tip (or)	1 0	1 0 0 1 0 1	0 0 1
R-tip (slt)	1 0	1 0 1 0 1 0	1 1 1
ori	**1 1**	**X X X X X X**	**0 0 1**

Pošto nema funkcijskog polja (tamo je operand-konstanta) koristimo *ALUop*=11, a
dijagram stanja će izgledati:



Dakle, dodali smo dva nova stanja (ukupno ih je 12), i dalje je dovoljno 4 flip-flopa, i dodali smo novu kontrolnu liniju za *ALUSelB*.

4. Prikazati i objasniti sve promjene u *Datapath-u* i dijagramu stanja (uključujući kontrolne signale) potrebne za implementaciju ***blez*** instrukcije.
Sintaksa nove instrukcije je:

blez $Rs, label

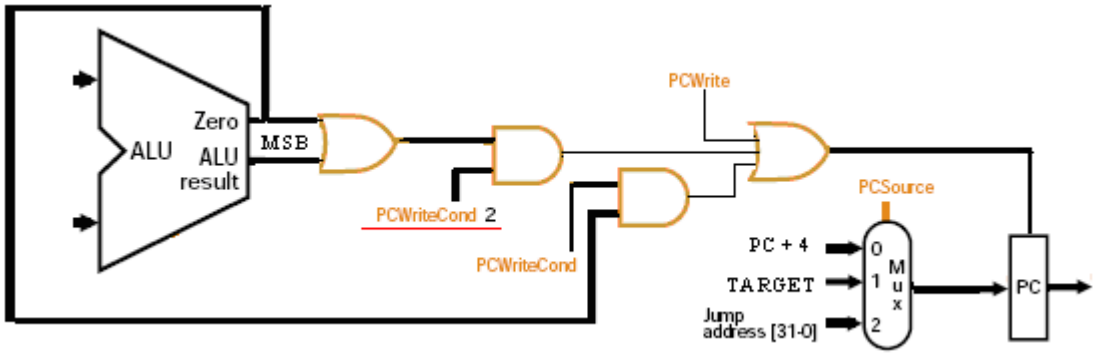
Format:

6	Rs	0	Offset

Opis: skočiti za onoliko lokacija koliko pokazuje *Offset*, ako je *Rs*≤0.

Rješenje:

Da bi sadržaj registra *Rs* bio ≤0, treba da mu je *sign-bit*=1 ili da je *zero-output* ALU-a jednak 1. To možemo predstaviti sa:

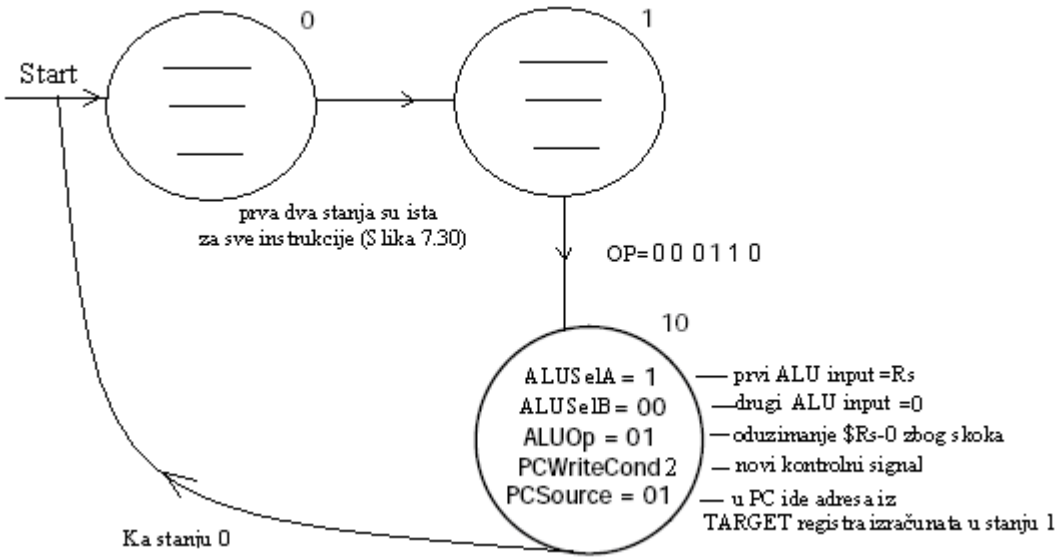


Znači, treba nam novi kontrolni signal PCWriteCond 2.

U tabeli sa kodovima, koristimo vrijednosti za ***beq*** instrukciju:

Instrukcija iz opcode-a	**ALUop** ALUop1 Aluop0	**Funkcijsko polje IR [5 - 0]**	**ALU control input 2 1 0**
LW	0 0	X X X X X X	0 1 0
SW	0 0	X X X X X X	0 1 0
Branch on equal	**0 1**	**X X X X X X**	**1 1 0**
R-tip (add)	1 0	1 0 0 0 0 0	0 1 0
R-tip (sub)	1 0	1 0 0 0 1 0	1 1 0
R-tip (and)	1 0	1 0 0 1 0 0	0 0 0
R-tip (or)	1 0	1 0 0 1 0 1	0 0 1
R-tip (slt)	1 0	1 0 1 0 1 0	1 1 1

Dakle, dijagram stanja će imati slijedeći izgled (prva dva stanja ostaju nepromijenjena):



Dakle, dovoljna su nam i dalje 4 flip-flopa, dodali smo jedan novi kontrolni signal i još jedno novo stanje na dijagramu tako da ih je ukupno 11, a treba shodno tome promjeniti i izgled kontrolne jedinice.

5. Prikazati i objasniti sve promjene u *Datapath-u* i dijagramu stanja (uključujući kontrolne signale) potrebne za implementaciju ***beg*** instrukcije.
Sintaksa nove instrukcije je:

beq $Rs, $Rt, label

Format:

4	Rs	Rt	Offset

Opis: skočiti za onoliko instrukcija koliko pokazuje *Offset*, ako je *Rs=Rt*.

Rješenje:

Zadatak je riješen tokom upoznavanja sa *Datapath*-om i prikazan slikama 7.25 i 7.28. *Datapath*-u i dijagram stanja u potpunosti odgovaraju osnovnim modelima prikazanim na slikama 7.22 i 7.30, a opis signala za ″treće stanje″ (pošto su prva dva ista za sve) dat je na strani 37.

6. Prikazati i objasniti sve promjene u *Datapath*-u i dijagramu stanja (uključujući kontrolne signale) potrebne za implementaciju ***slti*** instrukcije.
Sintaksa nove instrukcije je:

slti $Rt, $Rs, immediate

Format:

op	Rs	Rt	immediate

Op (6 bitova): operacioni kod instrukcije (001010)
Rs (5 bitova): registar koji sadrži jedan operand
Rt (5 bitova): registar u kome se smješta rezultat *slti* instrukcije
Immediate: 16-tobitna konstanta

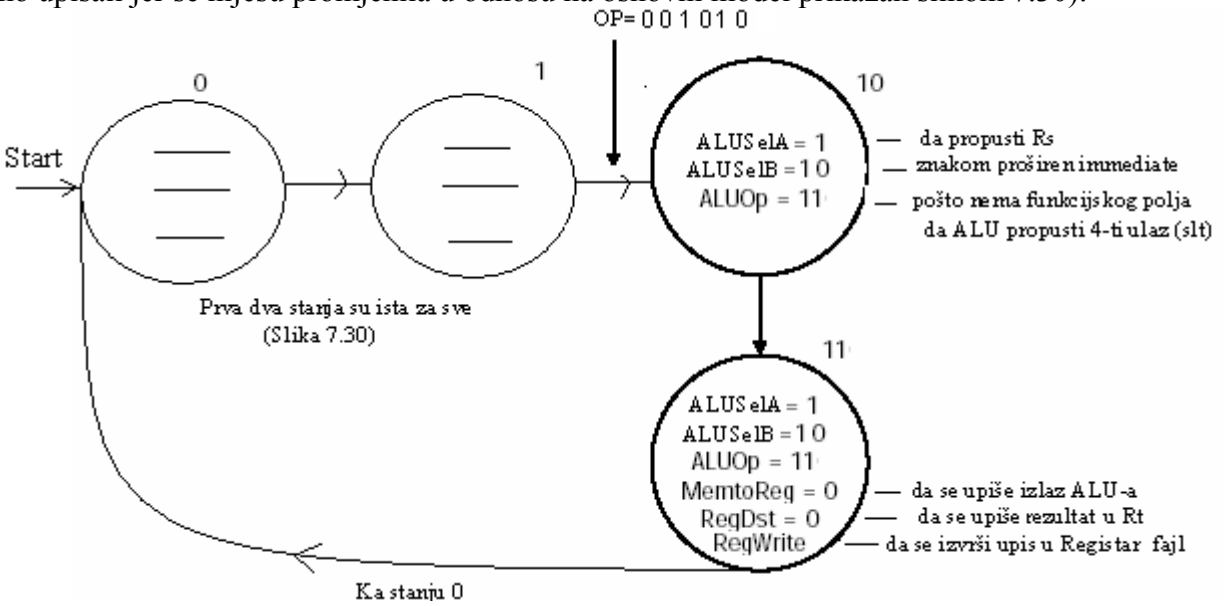
Opis: 16-tobitna konstanta se proširuje znakom (*sign-extended*) i oduzme se od sadržaja registra *Rs*. Posmatrajući obije vrijednosti kao cijele brojeve sa znakom, ako je *Rs* manje od proširene konstante, rezultat je jedinica, a u ostalim slučajevima rezultat je nula. Rezultat se smješta u registar *Rt*.
Rješenje:

Pošto je instrukcija ***slt*** već implementirana kroz *″less″* i *″set″* signale u ALU-u (ALU za bit najveće težine), nema promjena u *Datapath-u*, pa će se kodovi vidjeti iz tabele:

Instrukcija **iz opcode-a**	**ALUop** ALUop1 Aluop0	**Funkcijsko polje** **IR [5 - 0]**	**ALU control input** **2 1 0**
LW	0 0	X X X X X X	0 1 0
SW	0 0	X X X X X X	0 1 0
Branch on equal	0 1	X X X X X X	1 1 0
R-tip (add)	1 0	1 0 0 0 0 0	0 1 0
R-tip (sub)	1 0	1 0 0 0 1 0	1 1 0
R-tip (and)	1 0	1 0 0 1 0 0	0 0 0

R-tip (or)	1	0	1 0 0 1 0 1	0 0 1
R-tip (slt)	**1**	**0**	**1 0 1 0 1 0**	**1 1 1**

Na osnovu gornje tabele crtamo dijagram stanja (kontrolne signale vezane za prva dva stanja opet nijesmo upisali jer se nijesu promjenila u odnosu na osnovni model prikazan slikom 7.30):



Dakle, imamo nova dva stanja (10 i 11), i dalje su nam dovoljna 4 flip-flopa, nema novih kontrolnih signala, pa kako nema promjena u *Datapath-u* neće biti promjena ni u kontrolnom dijelu.

Literatura:

1. John L.Hennessy, David A.Patterson, *"Computer Organization and Design"*,Morgan Kaufmann Publishers,San Mateo,California, 1998

Webografija:

http://www.cs.princeton.edu/courses/archive/fall04/cos471/lectures
http://cs.uccs.edu/~cdash/cs216/overheads/
http://www-inst.eecs.berkeley.edu/~cs61c/su05/ cs
http://www-csag.ucsd.edu/teaching/cse141-w00/lectures/
http://www.ecst.csuchico.edu/~juliano/Architecture/Slides/PDF/ch05-2pp.pdf