5.4 JEDNOTAKTNA IMPLEMENTACIJA

Povezivanjem djelova datapath-a, namijenjenih kreiranju pojedinačnih tipova instrukcija u jedinstvenu cjelinu, dobija se jedinstveni datapath koji je u slučaju jednotaktne (jednostavne, odnosno eng. *single clock-cycle*) implementacije prikazan na slici 6. Jednotaktna implementacija pretpostavlja izvršavanja svih razmatranih instrukcija u toku trajanja jednog taktnog intervala. Podsjetimo, u taktovanim sistemima, taktni interval odredjuje period vremena u kome se pojedinačna funkcionalna jedinica može upotrijebiti, i to tačno jedan put, sa ulaznim podacima iz tog intervala. Ako raspolaženo sa jednim taktnim intervalom za izvršavanje razmatranih instrukcija, onda sve funkcionalne jedinice, neophodne za implementaciju ovih instrukcija, mogu biti upotrijebljene tačno jedan put, te ukoliko postoji potreba za višestrukom upotrebom bilo koje funkcionalne jedinice, ona se mora multiplicirati. To je slučaj sa memorijama i ALU i sabiračima za izračunavanje PC+4 i ciljne adrese grananja Target.

Memorija računara se, tokom izvršavanja pojedinačne instrukcije, može upotrijebiti dva puta:

1. Prilikom uzimanja/čitanja instrukcije i njenog dovodjenja u proces obrade (I korak izvršavanja svih instrukcija, tebela 2),
2. Prilikom čitanja podatka iz memorije (kod instrukcije *lw*) ili upisivanja podatka u memoriju (kod instrukcije *sw*).

Ukoliko bi se upotrebljavala jedna memorija, prilikom čitanja podatka bila bi izbrisana instrukcija (*lw*), koja još nije u potpunosti izvršena, a prilikom upisivanja podatka bila bi prepisana instrukcija (*sw*), koja još nije u potpunosti izvršena. Uz to, ne raspolaže se sa dodatnim taktom, da bi instrukcija *lw*/*sw* mogla sačuvati od prebrisavanja u medjuvremenu. Stoga je, u jednotaktnoj implementaciji, neophodno upotrijebiti 2 memorijske jedinice, jednu za čuvanje instrukcija (**Instruction memory**), a drugu za čuvanje podataka (**Data memory**), kao što je prikazano na slici 6.

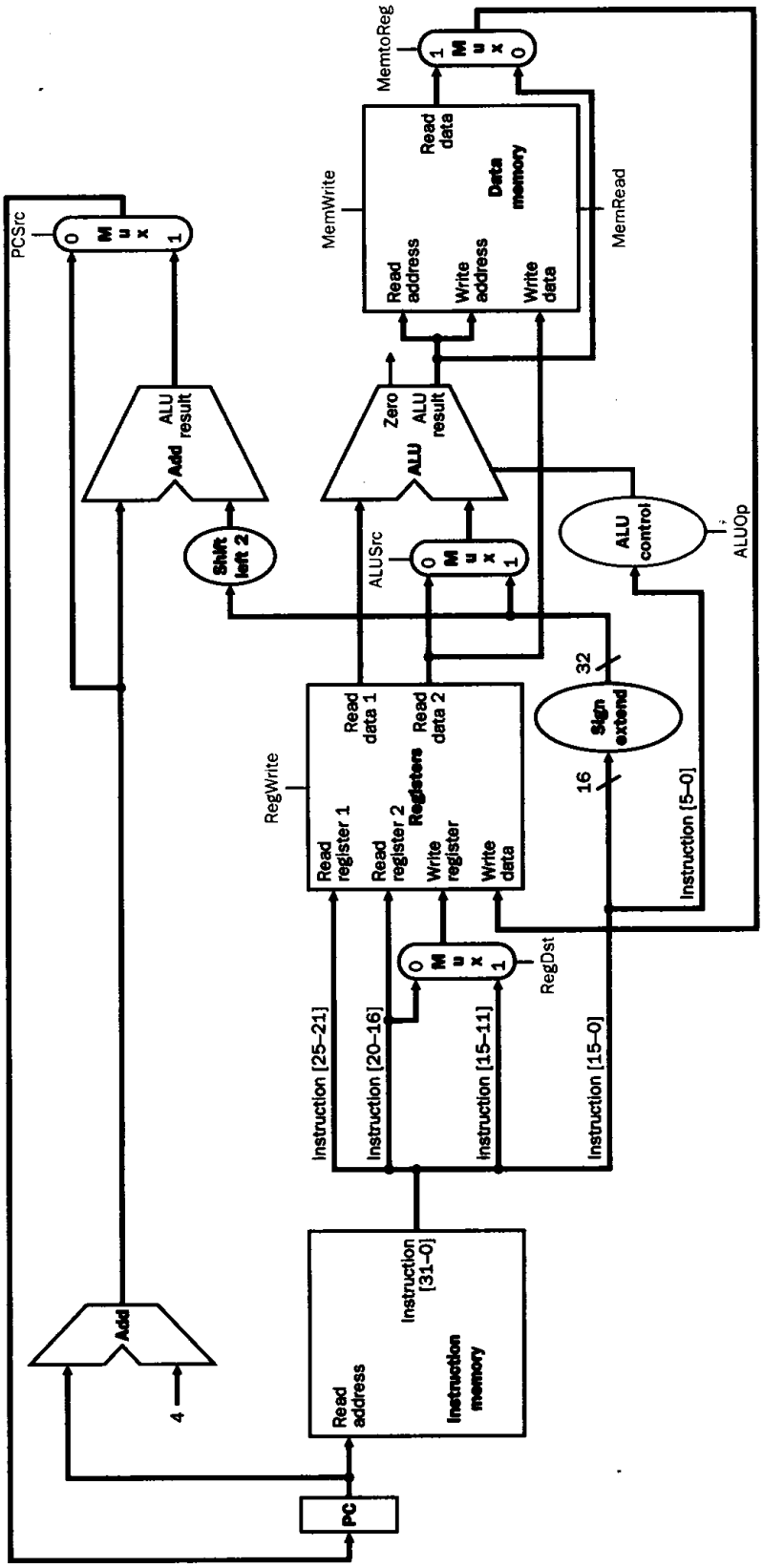
NAPOMENA 1: Tokom izvršavanja programa, instrukcije neće biti upisivane u Instruction memory, već samo čitane iz nje (instrukcije se upisuju u ovu jedinicu loaderom, i to prije početka izvršavanja programa). U Data memory, podaci mogu biti i čitani i upisivani za vrijeme izvršavanja instrukcija.

Izračunavanje adrese instrukcije koja sljedeća treba da se izvrši može obaviti ALU, bilo da je riječ o instrukciji koja je zapisana u prvoj sljedećoj memorijskoj lokaciji (kada je potrebno izračunati PC+4) ili da je riječ o instrukciji na koju se uslovno prelazi/grana (kada je potrebno izračunati ciljnu adresu grananja Target). Medjutim, tokom izvršavanja instrukcije, ALU je neophodno upotrijebiti za obavljanje operacije zahtijevane instrukcijom i to nad operandima koji su različiti od ulaza neophodnih za izračunavanje PC+4 ili ciljne adrese grananja Target. Pošto kod jednotaktne implementacije raspolažemo samo sa jednim taktnim intervalom, izračunavanje adrese sljedeće instrukcije obavljaju posebni sabirači označeni za Add (dvije posebne ALU su ALU_Operation=010 (poglavlje 4)).

NAPOMENA 2: Registers jedinica može se upotrebljavati za vrijeme izvršavanja iste instrukcije (R-tipa ili *lw*) i za čitanje operan(a)da i za upisivanje rezultata ALU/podatka, a ne multiplicira se. Naime, čitanje iz ove jedinice nije upravljano kontrolnim signalima i može se obaviti bilo kad tokom (pa i na početku) taktnog intervala, dok je upisivanje upravljano i obavlja se na kraju taktnog intervala, tako da raspolažemo čitavim taktnim intervalom izmedju trenutaka čitanja i upisivanja u Registars jedinicu.

Analizirajmo sada jednotaktnu implementaciju sa slike 6. Primijetimo da ova implementacija upotrebljava jednu Registers jedinicu i jedna ALU, iako se ove funkcionalne jedinice upotrebljavaju od strane različitih tipova instrukcija i to na različite načine (pogledaj napomene 1 i 2 u sekciji 5.3.3). U tabelama 3 i 4 navedeni su načini upotrebe Registers jedinice i ALU zavisno od instrukcija koje ih upotrebljavaju. Primijetimo da se Write register i Write data ulazi Registers jedinice upotrebljavaju na različite načine od strane instrukcija R-tipa i instrukcije *lw* (pogledaj napomenu 1 u sekciji 5.3.3), kao i da se na drugi ulaz ALU dovode različiti ulazni signali (operandi) tokom izvršavanja instrukcija R-tipa i instrukcije *beq*, sa jedne, odnosno instrukcija *lw* i *sw*, sa druge strane. Da bi se omogućilo dovodjenje različitih ulaznih signala na navedene ulaze Registers jedinice i ALU, ove ulaze je neophodno share-ovati dodavanjem multiplekora sa potrebnim brojem ulaza, kao što je prikazano na slici 6.

Ranije je uočeno (napomene 1 i 2 u sekciji 5.3.3) da multipleksori na Write registrer i Write data ulazima Registers jedinice, kao i multipleksor na drugom ulazu ALU treba da imaju po 2 ulaza, tj.



Slika 6. Jednotaktni datapath, sastavljen od djelova neophodnih za kreiranje različitih oblika instrukcija, sa predstavljenim kontrolnim signalima upotrijebljenih elemenata.

Tabela 3. Upotreba Registers jedinice za namjene koje su zahtijevane od strane različitih instrukcija. Mem[ALUOut] označava memorijsku lokaciju adresiranu rezultatom sa izlaza ALU, ALUOut označava rezultat sa izlaza ALU, dok simbol × označava "bilo što – ne upotrebljava se".

Ulazi	Instrukcije			
	lw	*sw*	R-tip	*beq*
Read register 1	rs	rs	rs	rs
Read register 2	×	rt	rt	rt
Write register	rt	×	rd	×
Write data	Mem[ALUOut]	×	ALUOut	×

Tabela 4. Upotreba ALU za namjene koje su zahtijevane od strane različitih instrukcija. Reg(rs) i Reg(rt) označavaju registare adresirane sadržajima polja rs i rt instrukcije, dok sign_extend(address) označava kopiranje znaka 16-bitne adrese, zapisane u address polju instrukcije, do 32-bitne dužine.

Ulazi	Instrukcije			
	lw	*sw*	R-tip	*beq*
I ulaz ALU	Reg(rs)	Reg(rs)	Reg(rs)	Reg(rs)
II ulaz ALU	sign_extend(address)	sign_extend(address)	Reg(rt)	Reg(rt)

po jedan selekcioni ulaz kojim će se upravljati propuštanjem odgovarajućih signala na navedene ulaze. Selekcioni ulazi ovih multipleksora nazvani su saglasno funkciji koju obavljaju u sistemu,

- selekcioni ulaz multipleksora na Write register ulazu Registers jedinice nazvan je ***RegDst*** (od eng. riječi Reg(ister) D(e)st(ination)), čime se ukazuje da se njime selektuje polje instrukcije (rt ili rd) kojim će biti odredjena destinacija (odredišni registar) u koji treba da bude upisan rezultat,
- selekcioni ulaz multipleksora na Write data ulazu Registers jedinice nazvan je ***MemtoReg***, da ukaže da se njime selektuje da li u registar označen rt ili rd poljem instrukcije treba da se upiše podatak pročitan iz Data memory ili rezultat sa izlaza ALU,
- selekcioni ulaz multipleksora na drugom ulazu ALU nazvan je ***ALUSrc*** (eng. ALU S(ou)rc(e)), da ukaže da se njime selektuje izvor operanda koji se dovodi na drugi ulaz ALU.

Osim navedenih multipleksora, jednotaktna arhitektura sa slike 6 uključuje i multipleksor na ulazu PC registra. Naime, u PC registar može biti upisana adresa instrukcije koja se u memoriji računara nalazi odmah nakon tekuće instrukcije (PC+4), ciljna adresa grananja Target (kod *beq* instrukcije), ili adresa beuslovnog skoka (kod *j* instrukcije), s tim što šema sa slike 6 ne uključuje implementaciju *j* instrukcije. Stoga je na slici 6 ovaj multipleksor prikazan sa dva ulaza i jednim selekcionim ulazom *PCSrc* (eng. *PC S*(*ou*)*rc*(*e*)), da ukaže da se njime selektuje izvor adrese za budući sadržaj PC registra. Implementiranjem *j* instrukcije, ovaj multipleksor će biti realizovan sa 3 ulaza i dva selekciona bita (poglavlje 5.5), ili kombinacijom 2 multipleksora 2/1 (sekcija 5.4.2).

Pored multipleksora i selekcionih ulaza koji upravljaju njihovim funkcionisanjem, memorijski elementi takodje moraju posjedovati kontrolu upisivanja podataka. Kontrolni signali memorijskih elemenata nazvani su po memorijskom elementu čije funkcionisanje kontrolišu i po funkciji koju obavljaju. Shodno tome, signal *RegWrite* kontroliše upisivanje podataka u Registers jedinicu, dok signal *MemWrite* kontroliše upisivanje podataka u Data memory.

NAPOMENA 3: Pored signala *MemWrite*, Data memory sadrži i kontrolni signal *MemRead*, kojim se upravlja čitanjem podataka iz ove jedinice. Primijetimo da, medju memorijskim elementima, samo Data memory sadrži kontrolne signale kojima se upravlja upisivanjem, ali i čitanjem podataka. Ostali memorijski elementi sadrže samo kontrolni signal koji upravlja upisivanjem podatka. Postavlja se pitanje što je razlog ovome? Prije nego odgovorimo na postavljeno pitanje, primijetimo da se, u slučaju Data memory, jedna te ista adresa, izračunata od strane ALU, upotrebljava i kao adresa memorijske lokacije sa koje se podatak čita (Read address na slici 6) i kao adresa memorijske lokacije u koju je podatak potrebno upisati (Write address na slici 6). Drugim riječima, ukoliko bi se omogućilo

čitanje podatka sa memorijske lokacije u istom trenutku kada bi se mogao upisivati neki drugi podatak u istu lokaciju, to bi moglo uzrokovati nepouzdan rad računara (uzrokovalo bi neizvjesnost da li pročitani podatak predstavlja "aktuelni" ili "bajati" podatak – podatak prije ili poslije izvršenog upisivanja). Sa druge strane, pouzdanost funkcionisanja je primarna karakteristika koju računar mora da zadovolji. Stoga, da bi se obezbijedila visoka pouzdanost funkcionisanja računara, realizuje se, za razliku od ostalih memorijskih elemenata, kontrola i upisivanja i čitanja podataka u Data memory.

NAPOMENA 4: Primijetimo da ne postoji kontrola upisivanja podataka u dva memorijska elementa: u PC registar i u Instruction memory, slika 6. Naime, arhitektura sa slike 6 je jednotaktna, tj. svaka od instrukcija izvršava se u toku trajanja jednog taktnog intervala, tako da se upisivanje nove adrese u PC registar vrši na kraju svakog od njih. Drugim riječima, osnovni taktni signal predstavlja kontrolni signal upisivanja adrese u PC registar, a taktni signal se ne predstavlja na slikama, već se podrazumijeva da ovaj signal kontroliše funkcionisanje svih memorijskih elemenata (na slikama su predstavljani samo kontrolni signali koji zajedno sa taktnim signalom upravljaju funkcionisanjem upotrijebljenih elemenata). Sa druge strane, kako je već navedeno u napomeni 1 na početku ove sekcije, tokom izvršavanja programa, instrukcije neće biti upisivane u Instruction memory (instrukcije se upisuju u ovu jedinicu loaderom, i to prije početka izvršavanja programa), već samo čitane iz nje. Shodno tome, Instruction memory ne uključuje kontrolni signal upisivanja instrukcija u nju.

Na koncu, primijetimo da ALU upotrebljavaju sve instrukcije, ali na različite načine. Uz to, više instrukcija upotrebljavaju ALU u cilju izvršavanja iste operacije, ali sa različitim operandima (ulazima). Na primjer,

- instrukcije *add*, sa jedne, i *lw*/*sw*, sa druge strane, zahtijevaju od ALU izvršavanje operacije sabiranja (pogledaj tabelu 2),
- instrukcije *sub*, sa jedne, i *beq*, sa druge strane, zahtijevaju od ALU izvršavanje operacije oduzimanja (pogledaj tabelu 2).

Podsjetimo da se operacije ALU zadaju postavljanjem ALU_Operation signala, poglavlje 4. Postavlja se pitanje: Kako postaviti iste ALU_Operation signale za različite tipove instrukcija (prethodno navedenim primjerima, *add* i *sub* pripadaju instrukcijama R-tipa, *lw*/*sw* – data transfer instrukcijama, a *beq* – instrukcija uslovnog skoka/grananja). Postavljanje ALU_Operation signala kod instrukcija R-tipa može obavljati funct polje mašinskog koda ovih instrukcija (funct poljem se, na koncu, definiše operacija koju obavlja instrukcija R-ipa), ali funct polje ne uključuju mašinski kodovi ostalih tipova instrukcija. Stoga, postavljanje ALU_Operation bitova mora biti kontrolisano ne samo funct poljem mašinskog koda zapisivanja instrukcija R-tipa, već i tipovima instrukcija koje mogu biti izvršavane. U tom cilju, jedinstveni datapath sa slike 6 uključuje **ALU control** jedinicu (kombinaciono kolo) koja će, na osnovu funct polja mašinskog koda zapisivanja instrukcija R-tipa i obilježja (nazvanih ALUOp signalima) svih mogućih tipova instrukcija koje upotrebljavaju ALU, postavljati ALU_Opetation kontrolne signale ALU.

5.4.1 Dizajniranje ALU control jedinice – Kontrole funkcionisanja ALU

Kao što je detaljno elaborirano u poglavlju 4, ALU_Operation signali ALU sastoje se od 3 kontrolna signala. Zavisno od kombinacije ovih signala, ALU izvršava jednu od operacija, sublimiranih u tabeli 5. Uz operacije i kontrolne signale ALU_Operation, u tabeli 5 predstavljene su i instrukcije koje upotrebljavaju pojedine operacije ALU tokom svog izvršavanja.

Tabela 5. ALU, kontrolni signali koju upravljaju njenim funkcionusanjem i tipovi instrukcija koji upotrebljavaju odredjeni način funkcionisanja ALU.

ALU_Operation			Funkcija ALU	Tip instrukcije koji upotrebljava ALU za odredjeni način funkcionisanja
2	1	0		
0	0	0	AND	R-tip (*and*)
0	0	1	OR	R-tip (*or*)
0	1	0	Add	R-tip (*add*), mem-reference (*lw*/*sw*)
1	1	0	Substract	R-tip (*sub*), uslovni skok (*beq*)
1	1	1	Set-on-less-than	R-tip (*slt*)

ZABILJEŠKA 1: Primijetimo da se kombinacije 011, 100, 101 ALU_Operation kontrolnih signala ne upotrebljavaju za definisanje posebne operacije ALU. Ovdje ćemo navedenu činjenicu upotrijebiti u cilju minimizacije dizajna ALU control jedinice. Ipak, primijetimo da se ista činjenica može upotrebiti i za proširivanje seta mogućih operacija koje je ALU sposobna da obavi.

ZABILJEŠKA 2: Primijetimo da tri različita tipa instrukcija (instrukcije R-tipa, memory-reference instrukcije i instrukcije uslovnog skoka/grananja) upotrebljavaju ALU za svoje odredjene namjene, tebela 5. Drugim riječima, potrebna su 2 bita za označavanje 3 različita tipa instrukcija. Ovi signali se nazivaju ALUOp signalima ($ALUOp_1$, $ALUOp_0$) i predstavljaju ulazne signale ALU control jedinice, koja se dizajnira u cilju generisanja ALU_Operation kontrolnih signala ALU. Kombinacije ALUOp signala koje odgovaraju pojedinim tipovima instrukcija predstavljene su u Tabeli 6.

Tabela 6. Kombinacije ALUOp signala koje definišu tipove instrukcija koje upotrebljavaju ALU za odredjene namjene.

ALUOp		Tip instrukcije koji upotrebljava ALU za odredjeni način funkcionisanja
1	0	
0	0	Memory-reference (*lw*, *sw*)
0	1	Instr. uslovnog skoka/grananja (*beq*)
1	0	R-tip (*add*, *sub*, *and*, *or*, *slt*)

ZABILJEŠKA 3: Primijetimo da se kombinacija bitova $ALUOp_{(1,0)}$=11 ne upotrebljava za definisanje odredjenog tipa instrukcije koja upotrebljava ALU. Ovdje ćemo navedenu činjenicu upotrijebiti u cilju minimizacije dizajna ALU control jedinice. Ipak, u cilju prilagodjavanja ALU za upotrebu od strane dodatnih tipova instrukcija, ova kombinacija ALUOp bitova može biti upotrijebljena, kao što će kasnije biti pokazano, na primjeru immediate instrukcija.

ZABILJEŠKA 4: Samo mašinski kod zapisivanja instrukcija R-tipa uključuje funct polje, kojim se jednoznačno definiše operacija zahtijevana instrukcijom (funct polje sadrži vrijednosti $32_{(10)}$ u slučaju instrukcije *add*, $34_{(10)}$ u slučaju instrukcije *sub*, $36_{(10)}$ u slučaju instrukcije *and*, $37_{(10)}$ u slučaju instrukcije *or* i $42_{(10)}$ u slučaju instrukcije *slt*). Drugim riječima, kada bi samo instrukcije R-tipa upotrebljavale ALU za izvršavanje odredjenih operacija, ALU_Operation kontrolni signali bili bi definisani samo funct poljem ovih instrukcija. Medjutim, više tipova instrukcija upotrebljava istu ALU, tako da se ALU_Operation kontrolni signali postavljaju na osnovu funct polja polja instrukcija R-tipa, te 2-bitnih ALUOp kontrolnih signala, kojim se definiše tip instrukcije koja upotrebljava ALU.

ZABILJEŠKA 5: Mašinski kodovi ostalih implementiranih tipova instrukcija (memory-reference i instrukcije uslovnog skoka/grananja) ne uključuju funct polje. Ova činjenica će se upotrijebiti u cilju minimizacije dizajna ALU control jedinice.

Sublimirajmo, u funkcionalnoj tabeli ALU control jedinice, činjenice navedene u zabilješkama 1–5. Napomenimo da je nepostojanje funct polja u mašinskom kodu zapisivanja memory reference instrukcija (*lw* i *sw*) i Branch instrukcije *beq* – instrukcije uslovnog skoka/grananja – označeno sa ×, odnosno vrijednošći "bilo što".

Tabela 7. Funkcionalna tabela ALU control jedinice.

Instrukcija	ALUOp		funct (Instruction [5–0])						ALU funkcija	ALU_Operation		
	1	0	F5	F4	F3	F2	F1	F0		2	1	0
lw (mem-ref)	0	0	×	×	×	×	×	×	ADD	0	1	0
sw (mem-ref)	0	0	×	×	×	×	×	×	ADD	0	1	0
beq (Branch)	0	1	×	×	×	×	×	×	Substract	1	1	0
add (R-tip)	1	0	1	0	0	0	0	0	ADD	0	1	0
sub (R-tip)	1	0	1	0	0	0	1	0	Substract	1	1	0
and (R-tip)	1	0	1	0	0	1	0	0	AND	0	0	0
or (R-tip)	1	0	1	0	0	1	0	1	OR	0	0	1
slt (R-tip)	1	0	1	0	1	0	1	0	Set-on-less-than	1	1	1

Sada je potrebno odrediti izlaze $ALU\_Operation_{(2,1,0)}$ u funkciji 8 ulaznih signala: 2 ALUOp bita i 6-birnog funct polja (Instruction [5–0]). Primijetimo da bi se $ALU\_Operation_{(2,1,0)}$ izlazi mogli odrediti direktno iz tabele 7, dizajniranjem 2-stepene I-ILI logičke strukture za svaki od ovih izlaza. Ipak, na taj način, ne bi se postigla minimalna forma dizajnirane ALU control jedinice (u cilju njene minimizacije, u tabeli 7 uključeno je samo nepostojanje funct polja kod memory-reference i branch instrukcija).

Ukoliko primijetimo (zabilješka 3) da se kombinacija bitova $ALUOp_{(1,0)}=11$ ne upotrebljava za definisanje odredjenog tipa instrukcije, kombinaciju $ALUOp_{(1,0)}=01$, upotrebljavanu kod *beq* instrukcije, možemo, u svrhu minimiziranja ALU control jedinice, sažeti kombinacijom $ALUOp_{(1,0)}=\times 1$, a kombinaciju $ALUOp_{(1,0)}=10$, upotrebljavanu kod instrukcija R-tipa, sažeti kombinacijom $ALUOp_{(1,0)}=1\times$.

Primijetimo takodje da najviša 2 bita funct polja uzimaju vrijednosti F5=1 i F4=0 kod svih razmatranih instrukcija R-tipa. Drugim riječima, ovi bitovi ne utiču na postavljanje odredjenog ALU_Operation kontrolnog signala, već svih ALU_Operation kontrolnih signala, i to na isti način, tako da se funkcionalna tabela 7 može sažeti tako što će F5 i F4 uzeti "bilo što" vrijednosti u slučaju svih razmatranih instrukcija, kao što je prikazano u tabeli 8.

Tabela 8. Sažeta funkcionalna tabela ALU control jedinice.

ALUOp		Funct (Instruction [5–0])						ALU_Operation		
1	0	F5	F4	F3	F2	F1	F0	2	1	0
0	0	×	×	×	×	×	×	0	1	0
×	1	×	×	×	×	×	×	1	1	0
1	×	×	×	0	0	0	0	0	1	0
1	×	×	×	0	0	1	0	1	1	0
1	×	×	×	0	1	0	0	0	0	0
1	×	×	×	0	1	0	1	0	0	1
1	×	×	×	1	0	1	0	1	1	1

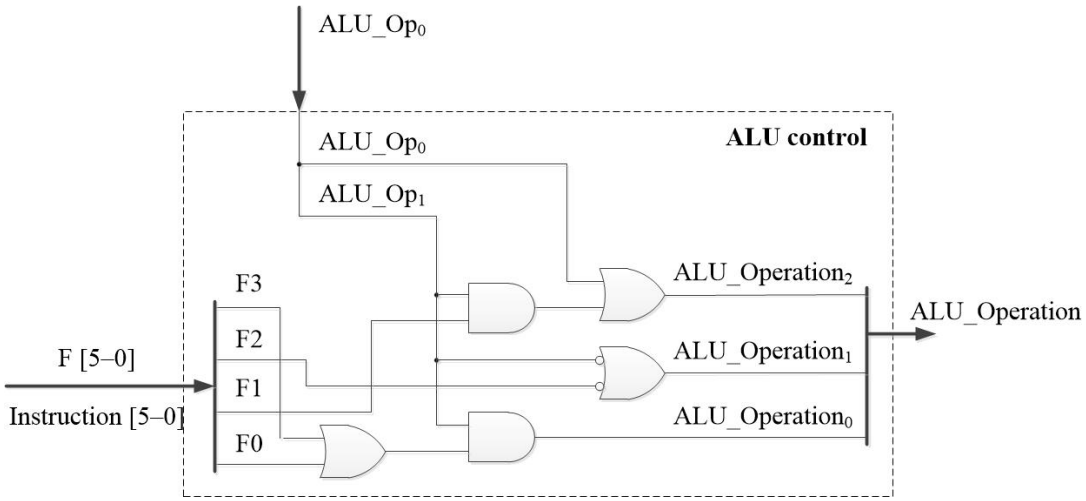
Na ovaj način, stvoreni su uslovi za pronalaženje minimalnih izraza za $ALU\_Operation_{(2,1,0)}$.

Primijetimo odmah da minimizacija upotrebom Karnough-ovih mapa, u ovom slučaju, ne predstavlja adekvatan izbor. Naime, 8 ulaznih signala zahtijeva minimizaciju upotrebom 16 Karnough-ovoh mapa za 4 promjenljive. Stoga ćemo minimizaciju obaviti upotrebom "bilo što" stanja i sažimanjem funkcionalne tabele samo na kombinacije ulaznih signala koji direktno utiču na postavljanje/set-ovanje odgovarajućeg izlaznog signala.

1. Posmatrajmo izlazni signal $ALU\_Operation_2$. Postavljanje/set-ovanje prve dvije vrijednosti ovog signala (prvi i drugi red tabele) obavlja isključivo originalna vrijednost $ALUOp_0$ signala ($ALUOp_0=1$ samo u drugom redu funkcionalne tabele), dok njegovu drugu po redu 1-cu (četvrti red tabele) set-uju jedinične vrijednosti signala $ALUOp_1$ i F1, a treću po redu 1-cu (sedmi red tabele) set-uju jedinične vrijednosti signala $ALUOp_1$, F3 i F1,

$$\begin{aligned} ALU\_Operation_2 &= ALUOp_0 + ALUOp_1 \cdot F1 + ALUOp_1 \cdot F3 \cdot F1 = \\ &= ALUOp_0 + ALUOp_1 \cdot F1 \cdot (1 + F3) = \\ &= ALUOp_0 + ALUOp_1 \cdot F1 \end{aligned}$$

2. Posmatrajmo izlazni signal $ALU\_Operation_1$. Njegove prve dvije vrijednosti (prvi i drugi red tabele) set-uje isključivo intertovana vrijednost $ALUOp_1$ signala ($ALUOp_1=0$ samo u ovim redovima tabele), dok preostale vrijednosti ovog signala (treći do sedmog reda tabele) set-uju jedinična vrijednost $ALUOp_1$ signala i invertovana vrijednost F2 signala (primijetimo da, za $ALUOp_1=1$, signal F2 ima invertovane vrijednosti u odnosu na signal $ALU\_Operation_1$),



Slika 7. ALU control jedinica.

$$\begin{aligned}\text{ALU\_Operation}_1 &= \overline{\text{ALUOp}_1} + \text{ALUOp}_1 \cdot \overline{\text{F2}} = \\ &= \overline{\text{ALUOp}_1} \cdot (1 + \overline{\text{F2}}) + \text{ALUOp}_1 \cdot \overline{\text{F2}} = \\ &= \overline{\text{ALUOp}_1} + \overline{\text{F2}} \cdot (\overline{\text{ALUOp}_1} + \text{ALUOp}_1) = \\ &= \overline{\text{ALUOp}_1} + \overline{\text{F2}}\end{aligned}$$

3. Posmatrajmo izlazni signal ALU_Operation_0. Ovaj signal set-uje se samo u šestom i sedmom redu tabele 7. Da bismo napisali njegov logički izraz, posmatrajmo svaku od njegovih set-ovanih vrijednosti kao da samo ona postoji. Dakle, ukoliko pretpostavimo da postoji samo 1-na vrijednost ALU_Operation_0 signala iz šestog reda, ALU_Operation_0 signal bio bi odredjen logičkim proizvodom originalnih vrijednosti signala ALUOp_1 i F0. Ukoliko, pak, pretpostavimo da postoji samo 1-na vrijednost ALU_Operation_0 signala iz sedmog reda, ALU_Operation_0 signal bio bi odredjen logičkim proizvodom originalnih vrijednosti signala ALUOp_1 i F3. Odnosno,

$$\begin{aligned}\text{ALU\_Operation}_0 &= \text{ALUOp}_1 \cdot \text{F0} + \text{ALUOp}_1 \cdot \text{F3} = \\ &= \text{ALUOp}_1 \cdot (\text{F0} + \text{F3})\end{aligned}$$

ALU control jedinica, implementirana na osnovu izvedenih izlaza, prikazana je na slici 7.

5.4.2 Dizajniranje glavne kontrolne jedinice jednotaktne arhitekture

Kao što je već rečeno u uvodu ove glave, ali i u toku dizajniranja jedinstvenog datapath-a za jednotaktnog arhitekturu, glavna kontrolna jedinica upravlja funkcionisanjem procesora i njegovog datapath-a. U tom cilju, glavna kontrolna jedinica postavlja kontrolne signale svih upotrijebljenih elemenata kontrolnog interfejsa sa slike 6 (upotrijebljenih multipleksora i ALU control jedinice), kao i kontrolne signale upotrijebljenih memorijskih elemenata. Potreba za njihovim uvodjenjem detaljno je analizirana prilikom dizajniranja jedinstvenog datapath-a jednotaktne implementacije. Ovdje će biti razmatrano generisanje ovih signala. Navedimo najprije kontrolne signale koje treba da generiše glavna kontrolna jedinica:

- Po 1 kontrolni signal na selekcionim ulazima svakog od 4 upotrijebljena multipleksora 2/1:
  RegDst signal multipleksora na Write register adresnom ulazu Registers jedinice,
  MemtoReg signal multipleksora na Write data ulazu Registers jedinice,
  ALUSrc signal multipleksora na drugom ulazu ALU,
  PCSrc signal multipleksora na ulazu PC registra.
- 3 kontrolna signala memorijskih elemenata:
  RegWrite kontrolni signal upisivanja podatka/rezultata ALU u Regitars jedinicu,

MemWrite kontrolni signal upisivanja podatka u Data memory,
MemRead kontrolni signal čitanja podatka iz Data memory.

– 2 kontrolna signala ALU control jedinice:
2-bitni ALUOp signal ($ALUOp_{(1,0)}$).

SUMARUM: Dakle, ukupno 9 kontrolnih signala potrebno je generisati. Kao što je već rečeno, njih generiše glavna kontrolna jedinica na osnovu 6-bitnog op polja instrukcije (op=Op[0–5]=Instruction [31–26], vidji zabilješku u poglavlju 5.1). Stoga, šemi prikazanoj na slici 6 potrebno je dodati blok glavne kontrolne jedinice sa ulazima Op [0–5]=Instruction [31–26], te izlaznim kontrolnim linijama koje se povezuju sa selekcionim ulazima upotrijebljenih multipleksora, te kontrolnim signalima memorijskih elemenata i ALU control jedinice, kao što je prikazano na slici 8.

Primijetimo da glavna kontrolna jedinica samostalno generiše sve kontrolne signale u sistemu, osim *PCSrc* kontrolnog signala. Naime, *PCSrc* signal odlučuje da li će budući sadržaj PC registra biti PC+4 (adresa instrukcije koja je u Instruction memory zapisana odmah nakon instrukcije koja se trenutno izvršava) ili ciljna adresa grananja Target. Podsjetimo da se ciljna adresa grananja Target upisuje u PC registar samo u slučaju implementiranja instrukcije uslovnog skoka/grananja *beq* i to samo ako je uslov grananja zadovoljen (ako je Zero=1). Shodno tome, glavna kontrolna jedinica generiše kontrolni signal *Branch*, koji svojom jediničnom vrijednošću treba da ukaže da se izvršava *beq* instrukcija, ali tek u kombinaciji sa signalom Zero treba da set-uje/postavi *PCSrc* kontrolni signal,

$$PCSrc = Branch \cdot \text{Zero}$$

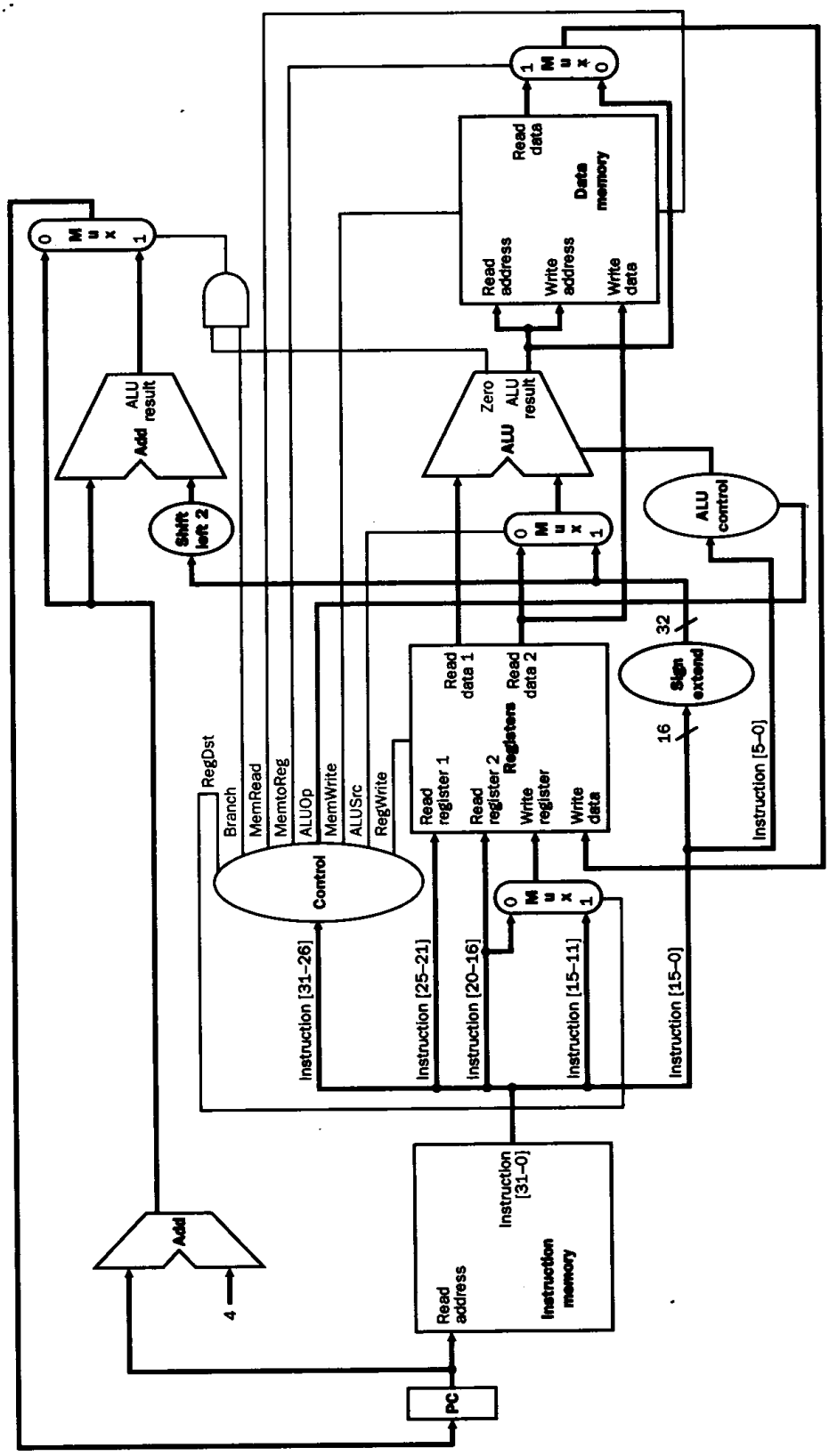
(*PCSrc* signal odlučuje da li će se u PC registar upisati ciljna adresa grananja Target ili PC+4).

NAPOMENA 1: Jednotaktna arhitektura sa slike 8 namijenjena je implementaciji instrukcija R-tipa, *lw*, *sw* i *beq*. Na ovoj slici još uvijek nije uključena implementacija instrukcije bezuslovnog skoka *j* (biće uključena prilikom dizajniranja višetaktne arhitekture). Ipak, primijetimo da, slično instrukciji *beq*, instrukcija *j* rezultira smještanjem odgovarajuće adrese (bezuslovnog skoka) u PC registar. Stoga bi implementacija instrukcije *j* zahtijevala dodatni multipleksor 2/1, koji bi se nalazio na ulazu PC registra, a čiji bi jedan od ulaza (ulaz 1) odgovarao adresi bezuslovnog skoka kreiranoj na način opisan u sekciji 5.3.5 (biće implementiran kod višetaktne arhitekture), a dugu ulaz (ulaz 0) odgovarao izlazu multipleksora sa selekcionim ulazom *PCSrc*. Shodno tome, dodatni muiltipleksor imao bi jedan selekcioni ulaz na koji bi se dovodio kontrolni signal, recimo *Jump* signal. Za 1-nu vrijednost *Jump* signala u PC registar upisivala bi se adresa beuslovnog skoka, a za njegovi 0-tu vrijednost u PC registar upisivalo bi se PC+4 ili adresa uslovnog skoka Target zavisno od vrijednosti signala *PCSrc*. Drugim riječima, implementacija instrukcije *j* zahtijevala bi dodatni multipleksor 2/1 i dodatni kontrolni signal, te bi glavna kontrolna jedinica trebala generisati ukupno 10 kontrolnih signala.

Kreirajmo sada kontrolnu logiku glavne kontrolne jedinice jednotaktne arhitekture sa slike 8, čiji se izlazni signali generišu na osnovu ulaza koje predstavlja 6-bitno op polje, op=Instruction [31–26], odnosno bitovi Op [0–5]=(Op5, Op4, Op3, Op2, Op1, Op0). Stoga, u cilju dizajniranja glavne kontrolne jedinice, sami postavljamo stanja 0, 1 i × njenih pojedinih izlaza zavisno od tipa instrukcije koju je potrebno implementirati, kao što je prikazano u tabeli 9.

Tabela 9. Postavljanje vrijednosti kontrolnih signala neophodnih za izvršavanje instrukcija koje su implementirane jednotaktnom arhitekturom sa slike 8.

Instrukcija	Op [5–0]=Instruction [31–26]						*RegDst*	*ALUSrc*	*MemtoReg*	*RegWrite*	*MemRead*	*MemWrite*	*Branch*	ALUOp		*Jump*
	Op5	Op4	Op3	Op2	Op1	Op0								1	0	
R-tip	0	0	0	0	0	0	1	0	0	1	0	0	0	1	0	0
lw	1	0	0	0	1	1	0	1	1	1	1	0	0	0	0	0
sw	1	0	1	0	1	1	×	1	×	0	0	1	0	0	0	0
beq	0	0	0	1	0	0	×	0	×	0	0	0	1	0	1	0
j	0	0	0	0	1	0	×	×	×	0	0	0	×	×	×	1



Slika 8. Jednotaktna arhitektura procesora sa glavnom kontrolnom jedinicom i svim kontrolnim signalima u sistemu.

ZAPAŽANJE 1: Prilikom implementacije instrukcija R-tipa, sekcija 5.3.2, na ulaze ALU potrebno je dovesti sadržaje registara koji se označavaju poljima rs=Instruction [25–21] i rt=Instruction [20–16] instrukcije. U tom cilju, potrebno je postaviti *ALUSrc*=0. ALU obavlja operaciju zahtijevanu instrukcijom R-tipa ($ALUOp_{(1,0)}$=10), a dobijeni rezultat potrebno je upisati nazad u Registers jedinicu, u registar označen poljem rd=Instruction [15–11] instrukcije. Shodno tome, potrebno je postaviti sljedeće kontrolne signale *MemtoReg*=0, *RegDst*=1, te omogućiti upis rezultata u selektirani registar sa *RegWrite*=1. Data memory se ne upotrebljava tokom izvršavanja instrukcija R-tipa, ni za čitanje, niti za upis podatka, tako da je *MemRead*=0, *MemWrite*=0, a takodje nije riječ o instrukciji uslovnog skoka/grananja, *Branch*=0.

ZAPAŽANJE 2: Prilikom implementacije instrukcije *lw*, sekcija 5.3.3, na ulaze ALU potrebno je dovesti sadržaj registra koji se označava poljem rs=Instruction [25–21] instrukcije i sadržaj 16-bitnog address=Instruction [15–0] polja instrukcije produžen, kopiranjem znaka, do 32-bitne dužine. U tom cilju, potrebno je postaviti *ALUSrc*=1. ALU obavlja operaciju sabiranja zahtijevanu instrukcijom *lw* ($ALUOp_{(1,0)}$=00), a dobijeni rezultat adresira Data memory u cilju čitanja podatka (*MemRead*=1) sa adresirane lokacije. Podatak pročitan iz Data memory potrebno je upisati u Registers jedinicu, u registar označen poljem rt=Instruction [20–16] instrukcije. Shodno tome, potrebno je postaviti sljedeće kontrolne signale *MemtoReg*=1, *RegDst*=0, te omogućiti upis pročitanog podatka u selektirani registar sa *RegWrite*=1. Data memory se ne upotrebljava tokom izvršavanja instrukcije *lw* u cilju upisa podatka u nju, *MemWrite*=0, a takodje nije riječ o instrukciji uslovnog skoka/grananja, *Branch*=0.

ZAPAŽANJE 3: Prilikom implementacije instrukcije *sw*, sekcija 5.3.3, na ulaze ALU potrebno je dovesti sadržaj registra koji se označava poljem rs=Instruction [25–21] instrukcije i sadržaj 16-bitnog address=Instruction [15–0] polja instrukcije produžen, kopiranjem znaka, do 32-bitne dužine. U tom cilju, potrebno je postaviti *ALUSrc*=1. ALU obavlja operaciju sabiranja zahtijevanu instrukcijom *sw* ($ALUOp_{(1,0)}$=00), a dobijeni rezultat adresira Data memory u cilju upisa podatka na adresiranu lokaciju. Podatak koji je potrebno upisati u Data memory uzima se iz Registers jedinice, iz registra označenog poljem rt=Instruction [20–16] instrukcije. Upis podatka u Data memory omogućava se sa *MemWrite*=1. Tokom izvršavanja instrukcije *sw*, ne vrši se povratno upisivanja podatka (iz Data memory)/rezultata (sa izlazu ALU) u Registers jedinicu, tako da je *RegWrite*=0, te shodno tome (kada nema povratnog upisivanja u Registers jedinicu) potpuno je nevažno/irelevantno koju će vrijednost uzeti kontrolni signali *MemtoReg* i *RegDst*, *MemtoReg*=×, *RegDst*=× (ovim kontrolnim signalima, obavlja se selekcija registra i podatka (iz Data memory)/rezultata (sa izlazu ALU) koji se upisuje u selektirani registar). Data memory se ne upotrebljava tokom izvršavanja instrukcije *sw* u cilju čitanja podatka iz nje, *MemRead*=0, a takodje nije riječ o instrukciji uslovnog skoka/grananja, *Branch*=0.

ZAPAŽANJE 4: Prilikom implementacije instrukcije *beq*, sekcija 5.3.4, na ulaze ALU potrebno je dovesti sadržaje registara koji se označavaju poljima rs=Instruction [25–21] i rt=Instruction [20–16] instrukcije. U tom cilju, potrebno je postaviti *ALUSrc*=0. ALU obavlja operaciju oduzimanja zahtijevanu instrukcijom *beq* ($ALUOp_{(1,0)}$=01). Izvršavanjem ove operacije, na izlazu ALU postavlja se Zero signal, koji učestvuje, zajedno za signalom *Branch*=1 (*Branch*=1, pošto je riječ o instrukciji uslovnog skoka *beq*), u kreiranju *PCSrc* selekcionog ulaza multipleksora koji se nalazi na ulazu PC registra. Data memory se ne upotrebljava tokom izvršavanja instrukcije *beq*, ni za čitanje, niti za upis podatka, tako da je *MemRead*=0, *MemWrite*=0. Tokom izvršavanja instrukcije *beq*, ne vrši se povratno upisivanja podatka (iz Data memory)/rezultata (sa izlazu ALU) u Registers jedinicu, tako da je *RegWrite*=0, te shodno tome (kada nema povratnog upisivanja u Registers jedinicu) potpuno je nevažno/irelevantno koju će vrijednost uzeti kontrolni signali *MemtoReg* i *RegDst*, *MemtoReg*=×, *RegDst*=× (ovim kontrolnim signalima, obavlja se selekcija registra i podatka (iz Data memory)/rezultata (sa izlazu ALU) koji se upisuje u selektirani registar).

ZAPAŽANJE 5: Kao što je već naglašeno, na slici 8 nije implementirana instrukcija bezuslovnog skoka *j*. Ipak, shodno napomeni 1 iz ove sekcije, u tabeli 9 dodati su (i jasno odvojeni debelim isprekidanim linijama) instrukcija *j* i kontrolni signal *Jump* koji bi odgovarao implementaciji ove instrukcije (pogledaj napomenu 1 iz ove sekcije), a postavljene su i vrijednosti svih ostalih kontrolnih signala jednotaktne arhitekture sa slike 8 neophodne za implementiranje instrukcije *j*. Primijetimo da implementacija instrukcije *j* ne predvidja upotrebu bilo koje funkcionalne cjeline sa slike 8 (pogledaj

napomenu 1 iz ove sekcije), tako da kontrolni bitovi svih memorijskih elemenata treba da uzmu vrijednost 0, *RegWrite*=*MemRead*=*MemWrite*=0 (upotreba bilo kog memorijskog elementa nije predvidjena), dok kontrolni signali svih multipleksora (osim multipleksora sa selekcionim ulazom *Jump*, dodatog u cilju implementiranja instrukcije *j*) treba da uzmu vrijednosti "bilo-što", *RegDst*= *ALUSrc*=*MemtoReg*=*Branch*=ALUOp_1=ALUOp_0=×. Naravno, selekcioni ulaz *Jump* multipleksora dodatog (na šematski prikaz sa slike 8) na ulazu PC registra (pogledaj napomenu 1 iz ove sekcije) treba da selektira adresu bezuslovnog skoka (*Jump*=1), kreiranu na način opisan u sekciji 5.3.5.

U cilju konačne hardware-ske implementacije glavne kontrolne jedinice jednotaktne arhitekture sa slike 8, u tabeli 9 predstavljene su binarne vrijednosti za njene ulazne signale – bitove op polja, op=Op [5–0]=Instruction [31–26], implementiranih instrukcija (op=$0_{(10)}$=$000000_{(2)}$ u slučaju instrukcija R-tipa, op=$35_{(10)}$=$100011_{(2)}$ u slučaju *lw* instrukcije, op=$43_{(10)}$=$101011_{(2)}$ u slučaju *sw* instrukcije, op=$4_{(10)}$=$000100_{(2)}$ u slučaju *beq* instrukcije i op=$2_{(10)}$=$000010_{(2)}$ u slučaju *j* instrukcije).

Sada, ukoliko se tabela 9 posmatra kao funkcionalna tabela sa ulazima Op5, Op4, Op3, Op2, Op1, Op0, te izlazima koji odgovaraju kontrolnim signalima koji se kreiraju, jednostavno se može pristupiti dizajniranju glavne kontrolne jedinice jednotaktne arhitekture sa slike 8. Dizajn odgovara 2-stepenoj I–ILI strukturi, gdje je u prvom stepenu implementira 5 mogućih potpunih logičkih proizvoda i to sa 5 logičkih I kola sa po 6 ulaza (Op5, Op4, …, Op0 i/ili njihovih komplementiranih vrijednosti),

$$\text{LP}_1 = \overline{\text{Op5}} \cdot \overline{\text{Op4}} \cdot \overline{\text{Op3}} \cdot \overline{\text{Op2}} \cdot \overline{\text{Op1}} \cdot \overline{\text{Op0}}$$
$$\text{LP}_2 = \text{Op5} \cdot \overline{\text{Op4}} \cdot \overline{\text{Op3}} \cdot \overline{\text{Op2}} \cdot \text{Op1} \cdot \text{Op0}$$
$$\text{LP}_3 = \text{Op5} \cdot \overline{\text{Op4}} \cdot \text{Op3} \cdot \overline{\text{Op2}} \cdot \text{Op1} \cdot \text{Op0}$$
$$\text{LP}_4 = \overline{\text{Op5}} \cdot \overline{\text{Op4}} \cdot \overline{\text{Op3}} \cdot \text{Op2} \cdot \overline{\text{Op1}} \cdot \overline{\text{Op0}}$$
$$\text{LP}_5 = \overline{\text{Op5}} \cdot \overline{\text{Op4}} \cdot \overline{\text{Op3}} \cdot \overline{\text{Op2}} \cdot \text{Op1} \cdot \overline{\text{Op0}}.$$

U drugom stepenu, potpuni logički proizvodi LP_i, i=1,...,5 dovode se na ulaze ILI kola za kreiranje kontrolnih signala *ALUSrc*, *RegWrite*, ili se direktno vode sa izlaza odgovarajućih I kola na izlaze glavne kontrolne jedinice kao kontrolni signali *RegDst*, *MemtoReg*, *MemRead*, *MemWrite*, *Branch*, ALUOp_1, ALUOp_0, *Jump* koje je potrebno generisati,

$$\textit{RegDst}=\text{LP}_1$$
$$\textit{ALUSrc}=\text{LP}_2+\text{LP}_3$$
$$\textit{MemtoReg}=\text{LP}_2$$
$$\textit{RegWrite}=\text{LP}_1+\text{LP}_2$$
$$\textit{MemRead}=\text{LP}_2$$
$$\textit{MemWrite}=\text{LP}_3$$
$$\textit{Branch}=\text{LP}_4$$
$$\text{ALUOp}_1=\text{LP}_1$$
$$\text{ALUOp}_0=\text{LP}_4$$
$$\textit{Jump}=\text{LP}_5.$$

Navedimo, na koncu, da jednotaktna arhitektura ipak nije praktično aplikabilna. Jednostavno, zahtijeva multipliciranje funkcionalnih elemenata kadgod ih je potrebno upotrijebiti više puta tokom izvršavanja, čime se značajno povećava hardware-ska složenost sistema, gabarit, utrošak energije i cijena. Uz to, taktni interval mora biti dovoljno dug da obezbijedi izvršavanje svih razmatranih instrukcija – time i najduže od njih, *lw* instrukcije, koja za svoje izvršavanje zahtijeva 5 koraka, tabela 2. Medjutim, tako odabrani taktni interval biće suviše dug za izvršavanje ostalih instrukcija (napr., 2/5=40% vremena duži nego je neophodno za izvršavanje instrukcija uslovnog i bezuslovnog skoka). Drugim riječima, ni vrijeme izvršavanja nije optimizirano kod jednotaktih arhitektura. Stoga se dizajniraju odgovarajuće multitaktne arhitekture, kao što će biti pokazano u sljedećem poglavlju.