5 PROCESOR

Centralna procesorska jedinica (eng. *Central Processing Unit* – CPU) ili skraćeno procesor je najznačajnija cjelina računarskih arhitektura/sistema. Uopšteno, procesor se sastoji iz Datapath-a (ALU, 32 procesorska registra, dodatnog kontrolnog interfejsa i medjusobnih veza ovih elemenata) i kontrolne jedinice. Kao što je elaborirano u poglavlju 4, ALU obavlja operacije koje se zahtijevaju kodom izvršavanih instrukcija. Operacije se izvršavaju nad operandima koji se nalaze u procesorskim registrima ($0–$31), a elementi kontrolnog interfejsa (multipleksori, dodatna logička kola, dekoderi) obezbjedjuju pravilan odabir i protok operanada, kao i podataka uzetih/pročitanih iz memorije računara i podataka koje je potrebno upisati u memoriju računara. Kontrolna jedinica postavlja kontrolne signale svih elemenata kontrolnog interfejsa, operativne memorije računara i ostalih upotrijebljenih memorijskih elemenata. Na ovaj način, kontrolna jedinica suštinski upravlja ispravnim funkcionisanjem računara kao sistema/cjeline.

Kada se govori o performansama računara obično se misli na:

- brzinu rada procesora, odnosno frekvenciju njegovog taktnog signala.

Medjutim, frekvencija rada procesora je samo jedna od tri faktora koja dominnantno utiču na performansama računara. Preostala dva su:

- broj instrukcija koje je potrebno izvršiti u assembler-skoj formi programa za izvršavanje odredjene akcije,
- broj taktnih intervala koje je neophodno izvršiti po instrukciji.

Broj instrukcija koje se izvršavaju u assembler-skoj formi odredjen je compiler-om, odnosno skupom instrukcija kojima compiler raspolaže. Naime, bez obzira da li se radi o programu koji se izvršava u nekom od viših programskih jezika, ili je riječ o akciji koja se izvršava pod komandom operativnog sistema, ovaj program/akcija mora biti zapisana/zadata u višem programskom jeziku. Potom se program/akcija prevode (kompajliraju) u assembler-sku formu. Assembler-ska forma će zauzeti manje linija koda (podsjetimo se, u assembler-skoj formi, u svakoj liniji koda može biti zapisana tačno jedna instrukcija) ukoliko compiler raspolaže širim skupom i moćnijim instrukcijama. Uz to, kada se kaže skup instrukcija u assembler-skoj formi misli se na arhitekturu kreiranu za podrazumijevani skup instrukcija. Podsjetimo, u poglavlju 3, razmatrali smo instrukcije u assemblerskoj formi. Od arhitekture neophodne za implementaciju intrukcija uvedenih u assembler-skoj formi u poglavlju 3, do sada je dizajnirana ALU (poglavlje 4).

Frekvencija rada procesora i broj taktnih intervala po instrukciji odredjeni su implementacijom procesora. Oni si tema izučavanja u ovom poglavlju. U tom cilju, biće razmatrane jednotaktna (ili prosta) i višetaktna (odnosno multitaktna) implementacija procesora za odredjeni skup instrukcija.

Implementirane će biti instrukcije koje predstavljaju srž MIPS seta instrukcija, i to:

- Data-transfer, odnosno memory-reference instrukcije *lw* i *sw*,
- Aritmetičko-logičke instrukcije (instrukcije R-ipa) *add*, *sub*, *and*, *or* i *slt*,
- Instrukciju uslovnog skoka/grananja *beq*, i
- Instrukciju bezuslovnog skoka *j*.

NAPOMENA: Ovim setom, odnosno podskupom instrukcija nijesu obuhvaćene sve MIPS instrukcije, čak ni sve instrukcije razmatrane u poglavlju 3. Medjutim, razmatrani podskup je reprezentativan i predstavlja osnovu za ilustraciju ključnih principa koji se upotrebljavaju u kreiranju datapath-a i kontrolne jedinice.

POSLJEDICA: Instrukcije, koje nijesu obuhvaćene prethodnim setom instrukcija, mogu se implementirati na način veoma sličan onom koji će biti prezentiran prilikom dizajniranja instrukcija iz prethodno navedenog seta. To će na kraju poglavlja biti ilustrovano na primjerima immediate instrukcija, kada će se postojeći procesor (za prethodno navedeni set instrukcija) dopuniti i korigovati tako da omogući implementaciju immediate instrukcija.

VAŽNO: Najveći broj koncepata koji će biti upotrebljeni u dizajniranju procesora za nevadeni set instrukcija predstavljaju koncepte koji se upotrebljavaju u dizajniranju širokog spektra računara, od računara sa viskim performansama, do mikroprocesora opšte i specijalizovane namjene.

5.1 PREGLED MIPS INSTRUKCIJA U ASSEMBLER-SKOM I MAŠINSKOM KODU

Prije nego započnemo dizajniranje, napravimo kratak pregled MIPS instrukcija i formata njihovog zapisivanja u assembler-skoj i mašinskoj formi, tabela 1.

Tabela 1. Posmatrane MIPS instrikcije i formati njihovog zapisivanja.

Aritmetičko-logičke instrukcije – *add*, *sub*, *mult*, *slt*, *sll*, *srl*…

Sintaksa: *add* $x, $y, $z
Format zapisivanja: R-tip
Šematski prikaz:

Polje:	op	rs	rt	rd	shamt	funct
Br. bitova:	6	5	5	5	5	6
Sadržaj:	0	y	z	x	n<31	32,34, …

Data-transfer instrukcije – *lw*, *sw*

Sintaksa: *lw* (*sw*) $x, Astart($y)
Format zapisivanja: I-tip
Šematski prikaz:

Polje:	op	rs	rt	Address
Br. bitova:	6	5	5	16
Sadržaj:	35, 43	y	x	Astart

Immediate instrukcije *addi*, *muli*, *slti* ...

Sintaksa: *addi* $x, $y, C
Format zapisivanja: I-tip
Šematski prikaz:

Polje:	op	rs	rt	Immediate
Br. bitova:	6	5	5	16
Sadržaj:	8	y	x	C

Branch instrukcije (instrukcije uslovnog skoka/grananja) – *beq*, *bne*

Sintaksa: *beq* (*bne*) $x, $y, offset
Format zapisivanja: I-tip
Šematski prikaz:

Polje:	op	rs	rt	Immediate
Br. bitova:	6	5	5	16
Sadržaj:	4, 5	x	y	Offset

Instrukcije bezuslovnog skoka – *j*, *jal*

Sintaksa: *j* address
Format zapisivanja: J-tip
Šematski prikaz:

Polje:	op	Address
Br. bitova:	6	26
Sadržaj:	2, 3	Address

ZABILJEŠKA: Polja mašinskog koda zapisivanja instrukcija zauzimaju tačno odredjena mjesta u kodu, tabela 1, i mogu se predstaviti shodno bitovima koja zauzimaju. Na primjer, op polju odgovara najviših 6 bitova instrukcije, ili op=Instruction [31–26], kao i rs=Instruction [25–21], rt=Instruction [20–16], rd=Instruction [15–11], shamt=Instruction [10–6], address/immediate/offset=Instruction [15–0], funct=Instruction [5–0], i address polje instrukcija bezuslovnog skoka address=Instruction [25–0]. Na šematskim prikazima, polja instrukcija će, po pravilu, biti označavana na ovaj način.

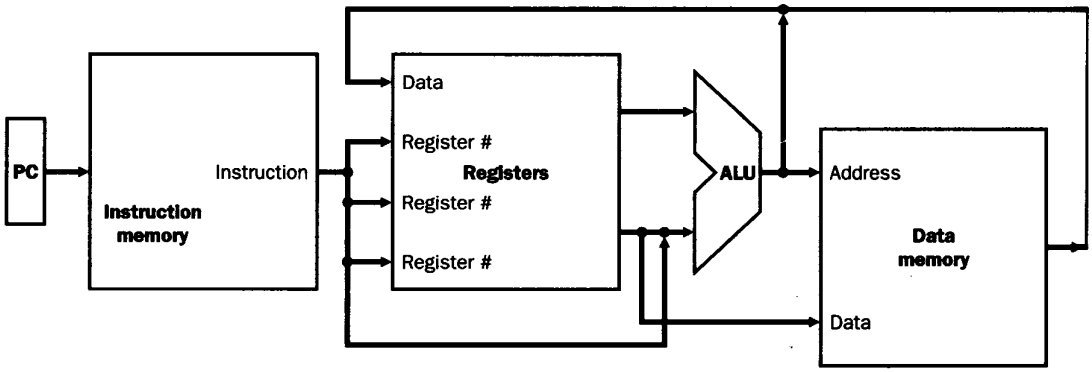
5.2 OPŠTI POGLED NA IMPLEMENTACIJU

Na osnovu znanja stečenih u oblasti programiranja u assembler-u, te prezentiranog kratkog pregleda MIPS inskrukcija i formata njihovog zapisivanja u assembler-skom i u mašinskom obliku, tabela 1, mogu se zapaziti koraci koje svaka pojedninačna instrukcija mora obaviti tokom izvršavanja, kao što je prikazano u tabeli 2. U tabeli 2 dati su detalji izvršavanja MIPS instrukcija po koracima. Medjutim, za sada ćemo pažnju posvetiti samo koracima i osnovnim razlozima za njihovo obavljenje, bez dubljeg ulaska u detalje. Detalji izvršavanja po koracima detaljno će biti razmatrani kasnije.

Tabela 2. Neophodni koraci koje je po trebno izvršiti u cilju implementacije MIPS instrukcija.

<table>
<tr><th>Korak</th><th>Data-transfer instrukcije (lw, sw) – I tip</th><th>Aritmetičko-Log. instrukcije (R-tip)</th><th>Branch instrukcije (beq) – I tip</th><th>Bezuslovni skok (j) – J tip</th></tr>
<tr><td>I</td><td colspan="4">Akcija: Uzimanje instrukcije iz memorije i njeno donošenje u proces izvršavanja (IR←Mem[PC])
Paralelna akcija: Inkrementiranje sadržaja PC-a (PC←PC+4)</td></tr>
<tr><td>II</td><td colspan="4">Akcija: Dekodiranje instrukcije – čitanje sadržaja registara označenim poljima rs i rt, kao i sadržaja polja op, shamt, funct, address, offset, … (nepotreno postavljati kontrolne signale, pošto se čitanje memorijskih elemenata, sa izuzetkom memorije računara, ne kontroliše, već se kontroliše samo njihovo upisivanje)
Paralelna akcija: Izračunavanje ciljne adrese grananja (Target←(PC+4)+sign_extend(offset)<<2)</td></tr>
<tr><td>III</td><td colspan="4">Upotreba ALU za izračunavanje operacije zadate instrukcijom i generisanje izlaza ALU (ALUOut, Zero, Overflow)</td></tr>
<tr><td></td><td>ALUOut← ←Reg(rs)+sign_extend(address)</td><td>ALUOut← Reg(rs) <u>op</u> Reg(rt) (<u>op</u> – operacija zahtijevana instrukcijom)</td><td>PC←Target samo ako je Reg(rs)=Reg(rt) (Zero=1)</td><td>PC←(PC[28–31]+address)<<2</td></tr>
<tr><td>IV</td><td colspan="4">Upotreba rezultata ALU u cilju kompletiranja naredbe</td></tr>
<tr><td></td><td>lw
Čitanje Mem[ALUOut]</td><td>sw
Mem[ALUOut] ←Reg(rt)</td><td colspan="2">Reg(rd)←ALUOut</td></tr>
<tr><td>V</td><td>Reg(rt)← Mem[ALUOut]</td><td colspan="3"></td></tr>
</table>

Da bi instrukcija mogla biti izvršena, ona najprije mora biti pročitana (iz memorije računara) i dekodirana. Čitanjem se instrukcija dovodi u proces izvršavanja, dok se njenim dekodiranjem čitaju polja instrukcije zapisane u mašinskom kodu. Očitavanje polja neophodno je za izvršavanje svake pojedinačne instrukcije. Na primjer, tek nakon očitavanja vrijednosti zapisane u polju op, dolazi se do saznanja o kojoj instrukciji je riječ i što je sve neophodno preduzeti u cilju njenog daljeg izvršavanja. Slično je i sa ostalim poljima instrukcije: očitavanjem obilježja registara utvrdjuju se operandi nad kojima se obavlja operacija zahtijevana instrukcijom, očitavanjem adresnog/immediate/offset polja utvrdjuje se početna adresa niza (kod implementacije instrukcija *lw* i *sw*), konstanta koja služi kao operand (kod immediate instrukcija), odnosno PC relativne adrese (kod instrukcija *beq* i *bne*), ... Stoga su prva dva koraka izvršavanja (čitanje instrukcije i njeno dekodiranje) obavezna. Istovremeno,



Slika 1. Sažeti pogled na MIPS implementaciju različitih tipova instrukcija.

oni su zajednički koraci koje mora obaviti svaka instrukcija na početku svog izvršavanja.

Paralelno sa čitanjem instrukcije i njenim dekodiranjem, u zajeničkim koracima (prvom i drugom) izvršavaju se i akcije koje mogu biti od koristi za kompletiranje pojedinih instrukcija (i skratiti njihovo izvršavanje za jedan korak), a čije obavljanje ne nanosi štetu ni izvršavanju drugih instrukcija (instrukcija kojima ove akcije nijesu namijenjene). Akcije koje se izvršavaju u paraleli sa čitanjem i dekodiranjem instrukcije odnose se na kreiranje adrese instrukcije koja sljedeća treba da se izvrši. U prvom koraku, paralelno čitanju instrukcije, inkrementira se sadržaj PC registra (uvećava se za 4 byte-a, PC←PC+4, da bi se sadržaj PC registra postavio na adresu instrukcije koja je u memoriji računara zapisana odmah nakon tekuće instrukcije – o ovoj akciji je već bilo riječi prilikom razmatranja PC-relativnog adresiranja). U drugom koraku, paralelno dekodiranju pročitane instrukcije, pronalazi se ciljna adresa grananja/skoka (tzv. Target adresa), koja može biti od koristi ukoliko se zaključi, nakon dekodiranja, da je riječ o instrukcijama uslovnog skoka/grananja, te ukoliko je uslov grananja zadovoljen.

Nakon dekodiranja instrukcije, ima se uvid o kojoj instrukciji je riječ i može se znati što je potrebno napraviti u cilju kompletiranja svake pojedinačne instrukcije. Uopšteno rečeno, treći korak izvršavanja odnosi se na upotrebu ALU za izračunavanje operacije zadate instrukcijom i generisanje izlaza ALU (ALUOut, Zero, Overflow), dok je četvrti (i eventualni peti) korak namijenjen upotrebi rezultata ALU dobijenih u trećem koraku.

Instrukcije *lw* i *sw* upotrebljavaju ALU u cilju izračunavanja adrese memorijske lokacije koja treba da učestvuje u transferu podataka sa registarom, čije obilježje je upisano u rt polju instrukcije. Nakon toga, u četvrtom (i eventualnom petom) koraku vrši se transfer podataka iz memorije računara u registar označen poljem rt (kod instrukcije *lw*), odnosno transfer podataka u inverznom/obrnutom smjeru (kod instrukcije *sw*).

Aritmetičko-logičke instrukcije (instrukcije R-tipa) upotrebljavaju ALU u cilju obavljanja operacije zahtijevane izvršavanom instrukcijom (dizajnirana je, u poglavlju 4, ALU za obavljanje operacija AND, OR, sabiranje, oduzimanje, Set-on-less-than). Nakon toga, u četvrtom koraku, rezultat zahtijevane operacije, izračunat u trećem koraku, upisuje se u registar označem rd poljem instrukcije.

Instrukcije uslovnog skoka/grananja (branch instrukcije) upotrebljavaju ALU za 2 namjene:

1. Za izračunavanje ciljne adrese grananja (tzv. Target adrese),
2. Za ispitivanje ispunjenosti uslova jednakosti operanada, kada je potrebno da je Zero=1 (prilikom implementiranja instrukcije *beq*), odnosno za ispitivanje ispunjenosti uslova nejednakosti operanada, kada je potrebno da je Zero=0 (prilikom implementiranja instrukcije *bne*).

Primijetimo da se izračunavanje ciljne adrese grananja (Target adrese) vrši u drugom koraku, paralelno sa dekodiranjem instrukcije, dok se ispitivanje/provjera ispunjenosti uslova grananja i smještanje (u PC registar) adrese instrukcije koja treba da se izvrši nakon instrukcije *beq*/*bne* izvršava u trećem koraku.

Instrukcija bezuslovnog skoka *j* ne upotrebljava ALU. Za njeno izvršavanje, potrebno je kreirati (u PC registru) adresu na kuju treba izvršiti skok, o čemu će više riječi biti kasnije (nakon dizajniranja datapath-a za ostale razmatrane instrukcije).

Sažeti pogled na MIPS implementaciju različitih tipova instrukcija prikazan je na slici 1. Ona pokušava da obuhvati sve prethodna razmatranja. U PC registru nalazi se adresa instrukcije koja treba da se izvrši. PC registar svojim sadržajem adresira lokaciju Instruction memory sa koje će instrukcija biti pročitana i donesena u proces izvršavanja. Nakon toga, instrukcija se dekodira. Sadržajima polja rs i rt instrukcije označavaju se operandi u registrarskoj jedinici/fajlu (na slici 1 nazvanoj Registers, koja obuhvata procesorske registre $0–$31). U polju address/immediate/offset nalazi se početna adresa (kod instrukcija *lw* i *sw*), konstanta (kod immediate instrukcija), odnosno PC-relativna adresa (kod instrukcija *beq* i *bne*). Nakon dekodiranja, svaka instrukcija ima svoj tok izvršavanja:

- Operandi pročitani/uzeti sa izlaza Registers jedinice, iz registara označenih poljima rs i rt instrukcija R-tipa (aritmetičko-logičke instrukcije), dolaze na ulaz ALU, koja nad njima obavlja operaciju zahtijevanu izvršavanom instrukcijom. Rezultat dobijen na izlazu ALU upisuje se nazad u Registers jedinicu, u registar označen poljem rd instrukcije,
- Operand pročitan/uzet sa izlaza Registers jedinice, iz registra označenog poljem rs instrukcije, i adresa iz polja address instrukcija *lw* i *sw* predstavljaju operande nad kojima ALU izračunava adresu lokacije u Data memory koja će vršiti razmjenu podataka sa registrom označenim poljem rt instrukcije:
  - Ukoliko je riječ o instrukciji *lw*, adresa izračunata od strane ALU odgovara adresi lokacije sa koje će se pročitati podatak (Read address). Podatak pročitan iz Data memory upisuje se nazad u Registers jedinicu, u registar označen poljem rt instrukcije,
  - Ukoliko je riječ o instrukciji *sw*, adresa izračunata od strane ALU služi kao adresa za upisivanje podataka (Write address) u lokaciju kojoj izračunata adresa odgovara. Podatak, koji se upisuje u ovu lokaciju, uzima se sa izlaza Registers jedinice, iz registra označenog poljem rt instrukcije,
- Operandi pročitani/uzeti sa izlaza Registers jedinice, iz registara označenih poljima rs i rt instrukcija uslovnog skoka/grananja *beq*/*bne*, dovode se na ulaze ALU, koja provjerava jednakost/nejednakost sadržaja ovih registara. Zero izlaz ALU predstavlja rezultat koji će upravljati postavljanjem budućeg sadržaja PC registra. Ovaj dio nije prikazan na slici 1, jer je suviše detaljan za sažeti pogled prikazan na ovoj slici.

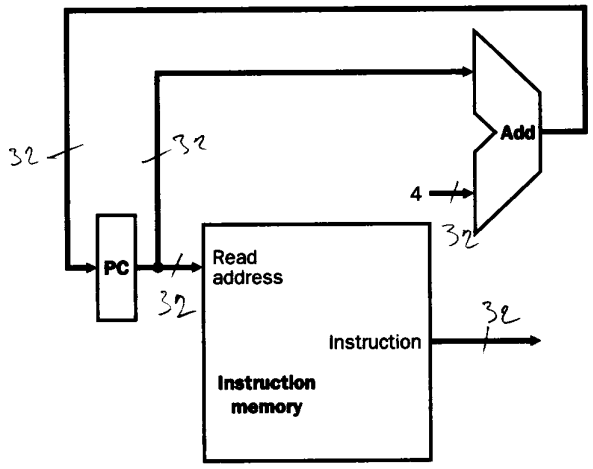
Primijetimo, odmah na strartu, da su drugi ulaz ALU i ulaz Data Registers jedinice podijeljeni (eng. *share*-ovani) za upotrebu izmedju različitih instrukcija:

- Drugi ulaz ALU može biti sadržaj registra/operand označen poljem rt instrukcija R-tipa, ali može biti i sadržaj address polja instrukcija *lw*/*sw*.
- Ulaz Data Registers jedinice može biti rezultat ALU (kod instrukcija R-tipa), ali može biti i podatak pročitan iz memorije računara (kod instrukcije *lw*).

Neophodnost podjele prethodno navedenih ulaza ALU i Registers jedinice može se primijetiti na slici 1, iako ovo nije i formalno prezentirano na slici. Share-ovanje/dijeljenje se obavlja postavljanjem multipleksora na odgovarajućim ulazima funkcionalnih jedinica, i to multipleksora čija veličina (broj ulaza) odgovara namjenama izmedju koji se funkcionalne jedinice dijele/share-uju.

5.3 SASTAVNI DJELOVI DATAPATH-A

Na osnovu navoda o potrebi share-ovanja/podjele pojedninih funkcionalnih jedinica, te njenog neprikazivanja na slici 1 može se primijetiti da je prikaz dat na slici 1 suviše sažet i da prikazuje nedovoljno detalja. U prethodnom tekstu pokušano je da se opiše svaki detalj prikazan na ovoj slici. Medjutim, za više detalja moramo proći kroz djelove datapath-a namijenjene izvršavanju pojedinačnih instrukcija, kroz njihovu integraciju u jedinstveni datapath, dodavanje kontrolnog interfejsa na svim



Slika 2. Dio datapath-a namijenjen donošenju instrukcije i inkrementiranju sadržaja PC registra.

funkcionalnim jedinicama koje je potrebno share-ovati/dijeliti i, na koncu, kroz dizajniranje kontrolne jedinice, što će biti razmatrano do detalja u nastavku.

5.3.1 Dio datapath-a namijenjen čitanju instrukcije i inkrementiranju sadržaja PC registra

Prvim (zajedničkim) korakom za implementaciju instrukcija predvidjeno je čitanje instrukcije iz memorije računara (odnosno, njeno donošenje u proces izvršavanja) i, u paraleli, inkrementiranje sadržaja PC registra. Dio datapath-a namijenjem izvršavanju ovih akcija prikazan je na slici 2.

Podsjetiimo, PC registar sadrži adresu instrukcije koja treba da se izvrši. Shodno tome, PC registar svojim sadržajem adresira, na "Read address" ulazu Instruction memory (memorijska jedinica namijenjena čuvanju instrukcija), lokaciju u kojoj je instrukcija zapisana. Na izlazu Instruction memory dobija se pročitana instrukcija koju treba izvršiti u narednim koracima. U paraleli, sadržaj PC registra dovodi se na prvi ulaz 32-bitnog sabirača, na čiji drugi ulaz se dovodi konstanta 4 da bi sabirač obavio operaciju PC+4. Rezultat ove operacije upisuje se, sa prvom sljedećom aktivnom ivicom takta, nazad u PC registar. Kontrola upisivanja podataka u memorijske elemente (time i u PC registar) nije predmet našeg razmatranja u ovom trenutku, te stoga nije ni prikazana na slici 2. Kontrola upisivanja će biti razmatrana kasnije prilikom dizajniranja kontrolne jedinice.

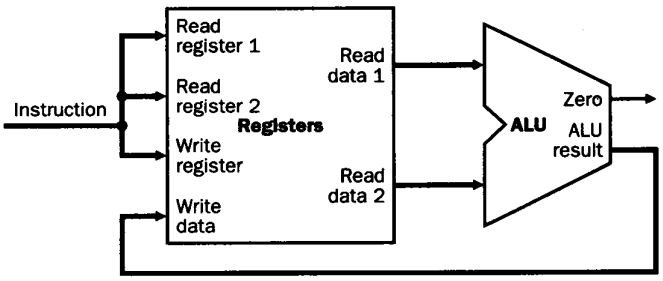
Primijetimo da je sadržaj PC registra 32-bitni, kao i kod pročitane instrukcije, u skladu sa razmatranjima iz poglavlja 3 (razmatramo, na koncu, 32-bitnu MIPS arhitekturu). Shodno 32-bitnom dizajnu sabirača, konstanta 4 koja se dovodi na njegov drugi ulaz takodje mora biti 32-bitna ($00...0100_{(2)}$). Primijetimo da je šemetski simbol sabirača veoma sličan simbolu ALU, uz navedenu oznaku "Add" na sabiraču. Na izlazu Instruction memory nalazi se instrukcije pročitana i donesena u proces izvršavanja. Ona (njena polja) će predstavljati ulaz ostalih djelova datapath-a.

5.3.2 Dio datapath-a namijenjen kreiranju aritmetičko-log. instrukcija (instrukcija R-tipa)

Podsjetimo na format zapisivanja aritmetičko-logičkih instrukcija (instrukcija R-tipa), tabela 1. Poljima rs i rt ovih instrukcija označeni su procesorski registri čiji sadržaji predstavljaju operande nad kojima ALU izvršava operaciju zahtijevanu instrukcijom (ALU je dizajnirana za obavljenje operacija AND, OR, sabiranje, oduzimanje ili Set-on-less-than). Rezultat zahtijevane operacije, sa izlaza ALU, upisuje se nazad u procesorski registar označen poljem rd instrukcije.

Procesorski registri ($0–$31) čiji sadržaji predstavljaju operande ALU nalaze su u Registers jedinici, kao i registar u koji se upisuje rezultat zahtijevane operacije. Shodno tome, Registers jedinica treba da posjeduje ulaze za adresiranje registara čiji sadržaji će predstavljati operande, kao i ulaz za adresiranje registra u koji treba da se upiše rezultat operacije:

– Ulazi za adresiranje registara čiji sadržaji predstavljaju operande nazivaju se "Read register 1" i "Read register 2" adresnim ulazima (uzimanjem operanada iz Registers jedinice vrši se



Slika 3. Dio datapath-a namijenjen kreiranju instrukcija R-tipa.

čitanje (eng. *Read*) odgovarajućih registara). Shodno navodima na početku ove sekcije, na ulaze Read register 1 i Read register 2 dovode se redom sadržaji polja rs i rt instrukcije R-tipa. Ovim adresnim ulazima odgovaraju izlazi "Read data 1" i "Read data 2" Registers jedinice, sa kojih se uzimaju (čitaju – eng. read) sadržaji registara označenih na "Read register 1" i "Read register 2" adresnim ulazima. Drigim riječima, svakom od operanada (podataka) koje treba pročitati iz Registers jedinice odgovara po jedan ulaz i izlaz – jedan adresni ulaz i njemu odgovarajući izlaz za pročitani podatak.

– Ulaz za adresiranje registra u koji treba da se **upiše** (eng. *Write*) rezultat operacije naziva se "Write register" adresnim ulazom. Shodno navodima na početku ove sekcije, na Write register adresni ulaz dovodi se sadržaj polja rd instrukcija R-tipa. Ovom adresnom ulazu odgovara ulaz "Write data" koji se upotrebljava za donošenje rezultata (sa izlaza ALU) u cilju njegovog upisivanja (eng. write) u registar označen na "Write register" adresnom ulazu. Drugim riječima, podatku (rezultatu sa izlaza ALU) koji treba upisati u Registers jedinicu odgovaraju 2 ulaza – adresni i njemu odgovarajući ulaz za podatak koji je potrebo upisati.

Primijetimo da su adresni ulazi Regisetrs jedinice (Read register 1, Read register 2 i Wrire register) 5-bitni, pošto služe za adresiranje 32 različita registra ($0–$31), a 32 različita obilježja ovih registara mogu biti zapisana sa 5 bitova (2^5=32). Sa druge strane, izlazi za čitanje podataka (Read data 1 i Read data 2), kao i Write data ulaz (za upisivanje podatka/rezultata ALU) su 32-bitni, pošto su sadržaju registara i rezultat sa izlazu ALU 32-bitni.

5.3.3 Dio datapath-a namijenjen kreiranju memory-reference instrukcija *lw* i *sw*

Da bi prišli implementaciji dijela datapath-a namijenjenog kreiranju memory-reference instrukcija *lw* i *sw*, sublimirajmo simbolički kod zapisivanja, tabela 1, i rezultate koji se postižu izvršavanjem ovih instrukcija, tabela 2. Simbolička forma zapisivanja instrukcija *lw* i *sw*, tabela 1,

lw/*sw* $x, Astart($y)

gdje je obilježje indeks registra y upisano u polju rs mašinskog koda instrukcija, obilježje registra x u polju rt mašinskog koda instrukcija, a Astart u address-nom polju mašinskog koda instrukcija, rezultira sljedećim ishodima, tabela 2:

Reg(rt)←Mem[Reg(rs)+sign_extend(address)] (kod instrukcije *lw*)

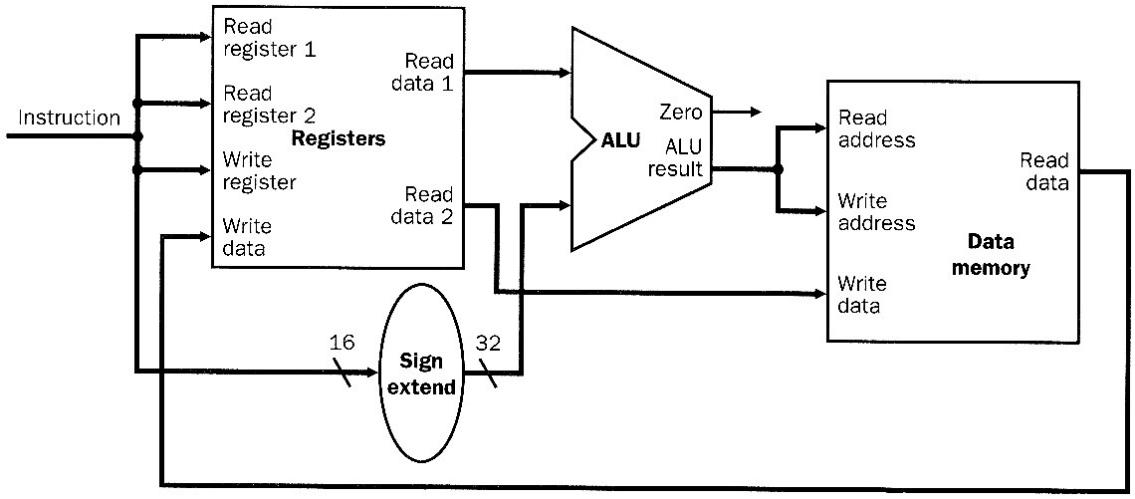
Mem[Reg(rs)+sign_extend(address)]←Reg(rt) (kod instrukcije *sw*)

gdje je sa sign_extend označeno kopiranje znaka 16-bitne adrese, zapisane u address-nom polju instrukcija, do 32-bitne dužine (pogledaj uvodne napomene poglavlja 4), sa Mem[Add] memorijska lokacija označena adresom Add, a sa Reg(rs) i Reg(rt) registari označeni poljima rs i rt instrukcija *lw* i *sw*.

Pojednostavljeno, operandi nad kojima ALU funkcioniše u slučaju instrukcija *lw*/*sw* nalaze se

– u registru označnom rs poljem instrukcija, i

– u address-nom polju instrukcije, pri čemu sadržaj address-nog mora biti produžen, kopiranjem znaka broja (sign_extend) do 32-bitne dužine prije pristupa na drugi ulaz ALU.



Slika 4. Dio datapath-a namijenjen kreiranju memory-reference instrukcija *lw* i *sw*.

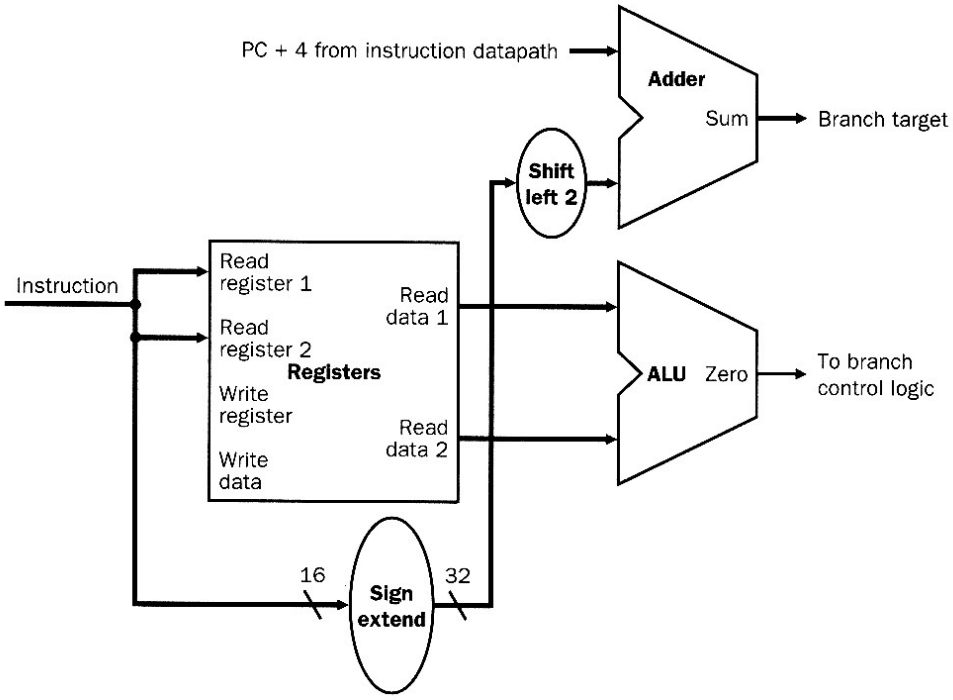
Shodno tome, u cilju implementiranja *lw*/*sw* instrukcija, na adresni ulaz Read register 1 Registers jedinice treba dovesti rs polje instrukcije, da bi se na Read register 1 izlazu iste jedinice dobio prvi operand nad kojim funkcioniše ALU. Na drugi ulaz ALU dovodi se 16-bitni sadržaj address-nog polja instrukcije, produžen, kopiranjem njegovog znaka, do 32-bitne dužine. ALU treba da izvrši sabiranje operanada i da svojim rezultatom adresira lokaciju Data memory

- sa koje će biti pročitan podatak koji će kasnije biti upisan u registar označen poljem rt u slučaju instrukcije *lw*, ili
- u koju će biti upisan podatak pročitan iz registra označenog poljem rt u slučaju instrukcije *sw*.

Razmotrimo najprije implementiranje instrukcije *lw*. Rezultat ALU adresira, na "Read address" ulazu, lokaciju Data memory koja na Read data izlazu daje podatak (sadržaj) sa adresirane lokacije koji treba upisati u registar označem poljem rt instrukcije. Shodno tome, sadržaj rt polja instrukcije treba dovesti na adresni Write register ulaz Registers jedinice, čime se adresira registar u koji će biti upisan podatak pročitan iz Data memory jedinice.

Razmotrimo sada implementaciju instrukcije *sw*. Rezultat ALU adresira, na "Write address" ulazu, lokaciju Data memory u koju treba upisati podatak pročitan iz registra označenog rt poljem instrukcije. Shodno tome, sadržaj rt polja instrukcije treba dovesti na adresni Read register 2 ulaz Registers jedinice, da bi se na Read data 2 izlazu iste jedinice dobio sadržaj adresiranog registra. Pročitani sadržaj registra dovodi se na Write data ulaz Data memory u cilju upisivanja u lokaciju adresiranu rezultatom ALU.

NAPOMENA 1: Primijetimo da se ista Registers jedinica (procesorski registri $0–$31) upotrebljava prilikom implementacija instrukcija R-tipa (sekcija 5.3.2) i prilikom implementacije instrukcija *lw*/*sw*. Medjutim, prilikom implementacije instrukcija R-tipa, registar u koji se upisuje rezultat ALU adresira se dovodjenjem **sadržaja rd polja instrukcije na Write register adresni ulaz Registers jedinice**, dok se **na Write data ulaz iste jedinice dovodi rezultat ALU (ALU result)**. Suprotno tome, prilikom implementacije instrukcioje *lw*, registar u koji se upisuje podatak, pročitan iz Data memory, adresira se dovodjenjem **sadržaja rt polja instrukcije na Write register adresni ulaz Registers jedinice**, dok se **na Write data ulaz iste jedinice dovodi podatak pročitan sa Read data izlaza Data memory**. Drugim riječima, ista Registers jedinica upotrebljava se za različite namjene prilikom implementacija instrukcija R-tipa i instrukcije *lw*. Da bi to bilo moguće, Write register adresni ulaz i Write data ulaz Registers jedinice treba share-ovati za različite namjene (ulaze) postavljanjem multipleksora na ovim ulazima, i to multipleksora čija veličina (broj ulaza) odgovara namjenama (ovdje instrukcijama R-tipa i instrukcije *lw* – dakle multipleksora sa po 2 ulaza). Medjutim, ovo je zadatak koji ćemo realizovati kasnije – prilikom kreiranja jedinstvenog datapath-a.



Slika 5. Dio datapath-a namijenjen kreiranju *beq* instrukcije.

NAPOMENA 2: Primijetimo da se ista ALU upotrebljava prilikom implementacija instrukcija R-tipa (sekcija 5.3.2) i prilikom implementacija instrukcija *lw*/*sw*. Prilikom implementacije instrukcija R-tipa, **na drugi ulaz ALU dovodi se sadržaj registra označenog poljem rt instrukcije**. Sa druge strane, prilikom izvršavanja instrukcija *lw* i *sw*, **na drugi ulaz ALU dovodi se 16-bitni sadržaj address-nog polja instrukcija, produžen, kopiranjem njegovog znaka, do 32-bitne dužine**. Drugim riječima, ista ALU upotrebljava se za različite namjene prilikom implementacija instrukcija R-tipa i instrukcija *lw*/*sw*. Da bi se to omogućilo, drugi ulaz ALU treba share-ovati za različite namjene (ulaze) postavljanjem multipleksora na ovom ulazu, i to multipleksora čija veličina (broj ulaza) odgovara namjenama (ovdje instrukcijama R-tipa i instrukcijama *lw*/*sw* – dakle multipleksora sa 2 ulaza). Ipak, ovo je zadatak koji će biti realizovan prilikom kreiranja jedinstvenog datapath-a.

5.3.4 Dio datapath-a namijenjen kreiranju instrukcije uslovnog skoka/grananja *beq*

Da bi prišli implementaciji dijela datapath-a namijenjenog kreiranju instrukcije uslovnog skoka/grananja *beq*, sublimirajmo simbolički kod zapisivanja, tabela 1, i rezultate koji se postižu izvršavanjem ove instrukcija, tabela 2. Simbolička forma zapisivanja instrukcije *beq*, tabela 1,

beq $x, $y, Offset

gdje su obilježja registara x i y redom upisana u polja rs i rt mašinskog koda instrukcije, a Offset u immediate polju mašinskog koda instrukcije, rezultira sljedećim ishodima, tabela 2:

– Izračunavanje ciljne adrese uslovnog skoka/grananja,

$$Target \leftarrow (PC+4)+sign\_extend(offset)<<2$$

– Ispitivanjem ispunjenosti uslova jednakosti operanada (sadržaja registara označenih poljima rs i rt instrukcije) i, ukoliko je uslov jednakosti zadovoljen, skokom na izračunatu ciljnu adresu grananja Target,

$$PC \leftarrow Target, \text{ ali samo ako je } Reg(rs)=Reg(rt)\ (Zero=1)$$

gdje je sa sign_extend označeno kopiranje znaka 16-bitne PC-relativne adrese (Offset-a), zapisane u immediate polju instrukcije, do 32-bitne dužine (pogledaj uvodne napomene poglavlja 4), sa <<2 shift-ovanje u lijevu stranu za 2 mjesta (množenje PC-relativne adrese sa

4), a sa Reg(rs) i Reg(rt) registri označeni poljima rs i rt instrukcije *beq*. Ukoliko uslov jednakosti nije zadovoljen, u PC registra se upisuje njegova inkrementirana vrijednost, PC←PC+4.

NAPOMENA: Prilikom izvršavanja instrukcija uslovnog i bezuslovnog skoka, buduća destinacija programa (adresa instrukcije koja sljedeća treba da se izvrši) mora biti adresa lokacije, a ne bilo kod drugog byte-a. Drugim riječima, u ovim slučajevima, adresa sljedeće instrukcije mora se završavati sa $00_{(2)}$. Pošto je činjenica da $00_{(2)}$ mora postojati u adresi sljedeće instrukcije, ove dvije nule se ne zapisuju u address-nim poljima mašinskog koda instrukcija uslovnog i bezuslovnog skoka (pogledaj napomenu 6 u poglavlju 3.7). Medjutim, prilikom implementacije instrukcija uslovnog i bezuslovnog skoka mora se zapisati $00_{(2)}$ na kraju adrese instrukcije koja sljedeća treba da se izvrši. To se obavlja shift-ovanjem/pomjeranjem, za 2 mjesta u lijevu stranu, sadržaja address-nih polja mašinskog koda instrukcija uslovnog i bezuslovnog skoka. Uz to, PC-relativna adresa zapisana u address-nom polju mašinskog koda instrukcije *beq*, uskladjena je sa adresom instrukcije koja u memoriji računara slijedi neposredno nakon instrukcije uslovnog skoka/grananja, odnosno uskladjena je sa PC+4 (pogledaj napomenu u poglavlju 3.7, i na strani 30).

Razmotrimo hardware-sku mplementaciju prethodnih navoda prikazanu na slici 5. 32-bitni sabirač Add izračunava ciljnu adresu grananja Target. Shodno tome, kao i navodima iz prethodne napomene, na njegove ulaze dovode se

- sadržaj PC registra inkrementiran za 4 (PC+4), koji je izračunat u dijelu datapath-a prezentiranom na slici 2 (nazvan "instruction datapath" na slici 5),
- 16-bitni sadržaj address-nog polja *beq* instrukcije produžen, kopiranjem znaka, do 32-bitne dužine i shift-ovan u lijevu stranu za 2 mjesta.

ALU ispituje ispunjenost uslova jednakosti operanada uzetih iz registara označenih poljima rs i rt instrukcije. Shodno tome, na Read registar 1 i Read reister 2 adresne ulaze Registers jedinice dovode se sadržaji polja rs i rt (kao kod instrukcija R-tipa). ALU na svom izlazu generuše Zero signal koji svojom jediničnom vrijednošću, Zero=1, signalizira ispunjenost uslova jednakosti operanada (za Zero=0, uslov jednakosti operanada nije zadovoljen, odnosno operandi su različiti).

Primijetimo sljedeće 2 činjenice:

1. Prilikom hardware-ske implementacije *beq* instrukcije, upotrebljava se ista Registers jedinica kao prilikom implementacije instrukcija R-tipa i instrukcija *lw*/*sw*, iako se u slučaju *beq* instrukcije ne vrši povratno upisivanje u Registers jedinicu (kao prilikom izvršavanja instrukcija R-tipa i *lw* instrukcije). Medjutim, o ovoj činjenici vodiće računa kontrolni signali Registers jedinice i kontrolnog interfejsa koji će biti postavljen na ulazima Registers jedinice (neophodnost upotrebe kontrolnog interfejsa kod Registers jedinice razmatrana je u napomeni iz sekcije 5.3.3).
2. Upisivanje ciljne adrese grananja Target u PC registar ukoliko je ispunjen uslov grananja (Zero=1) nije implementiran na slici 5, kao ni upisivanje inkrementiranog sadržaja PC registra (PC+4) ukoliko uslov grananja nije zadovoljen (Zero=0). Ovaj dio datapath-a biće implementiran prilikom kreiranja jednistvenog datapath-a i kontrolne jedinice.

5.3.5 Dio datapath-a namijenjen kreiranju instrukcije bezuslovnog skoka *j*

Instrukcija bezuslovnog skoka *j* implementira se zamjenom nižih 28 bitova tekućeg sadržaja PC registra (pogledaj napomenu 7 i konvenciju navedene u poglavlju 3.7) sa 26-bitnom adresom iz address-nog polja mašinskog koda instrukcije, tabela 1, shift-ovanom za 2 mjesta u lijevu stranu (pogledaj napomenu navedenu u sekciji 5.3.4). Najviša 4 bita PC registra, PC [31–28], zadržavaju svoj tekući sadržaj (pogledaj napomenu 7 i konvenciju iz poglavlja 3.7). Ipak, implementacija instrukcije *j* biće predstavljena tek prilikom dizajniranja potpunog datapath-a sa kontrolnom jedinicom.

Razmatrani djelovi datapath-a namijenjeni kreiranju pojedinačnih oblika instrukcija biće upotrijebljeni za krairanje jedinstvenog datapath-a u slučaju jednotaktne, ali i u slučaju višetaktne implementacije procesora.

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.

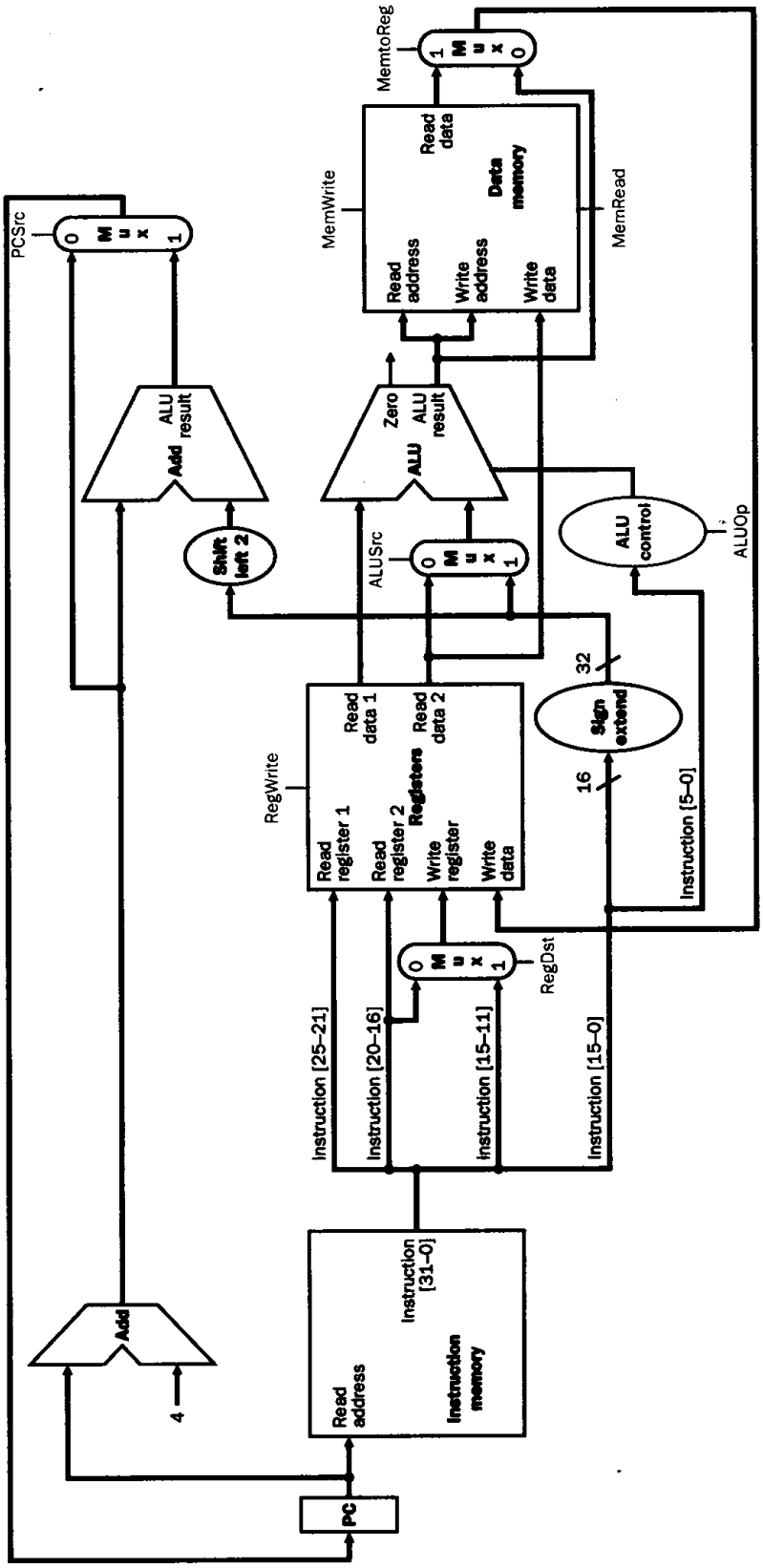
NAPOMENA 1: Tokom izvršavanja programa, instrukcije neće biti upisivane u Instruction memory, već samo čitane iz nje (instrukcije se upisuju u ovu jedinicu loaderom, i to prije početka izvršavanja programa). U Data memory, podaci mogu biti i čitani i upisivani za vrijeme izvršavanja instrukcija.

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

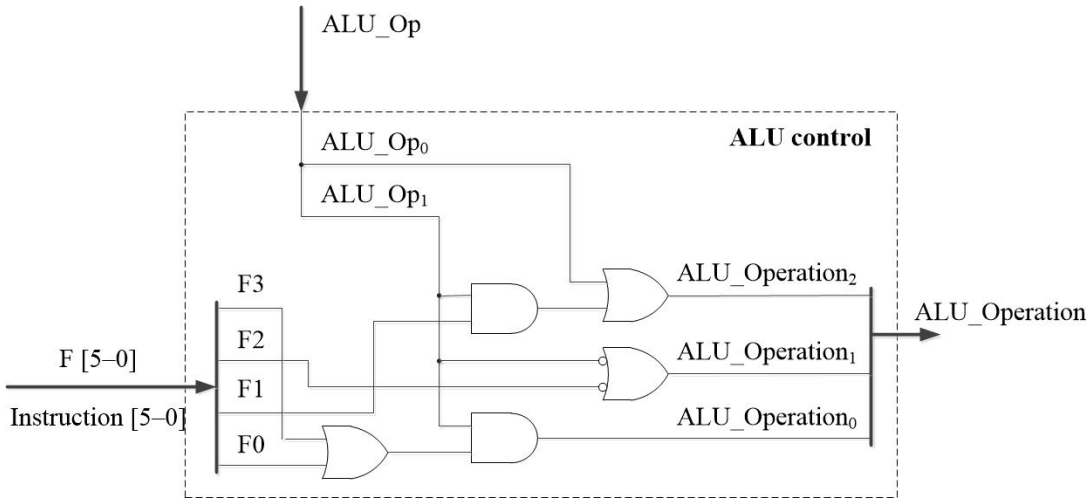
Na ovaj način, stvoreni su uslovi za pronalaženje minimalnih izraza za $ALU\_Operation_{(2,1,0)}$.

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,

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

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, dizajniranja u cilju kontrole izvršavanja operacija zahtijevanih instrukcijama iz tabele 7.

$$\begin{aligned} \text{ALU\_Operation}_1 &= \overline{\text{ALUOp}_1} + \text{ALUOp}_1 \cdot \overline{\text{F2}} = \\ &= \overline{\text{ALUOp}_1} \cdot (1 + \overline{\text{F2}}) + \text{ALUOp}_1 \cdot \overline{\text{F2}} = \\ &= \overline{\text{ALUOp}_1} + \overline{\text{F2}} \cdot (\overline{\text{ALUOp}_1} + \text{ALUOp}_1) = \overline{\text{ALUOp}_1} + \overline{\text{F2}} \end{aligned} \tag{2}$$

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,

$$\begin{aligned} \text{ALU\_Operation}_0 &= \text{ALUOp}_1 \cdot \text{F0} + \text{ALUOp}_1 \cdot \text{F3} = \\ &= \text{ALUOp}_1 \cdot (\text{F0} + \text{F3}) \end{aligned} \tag{3}$$

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,

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

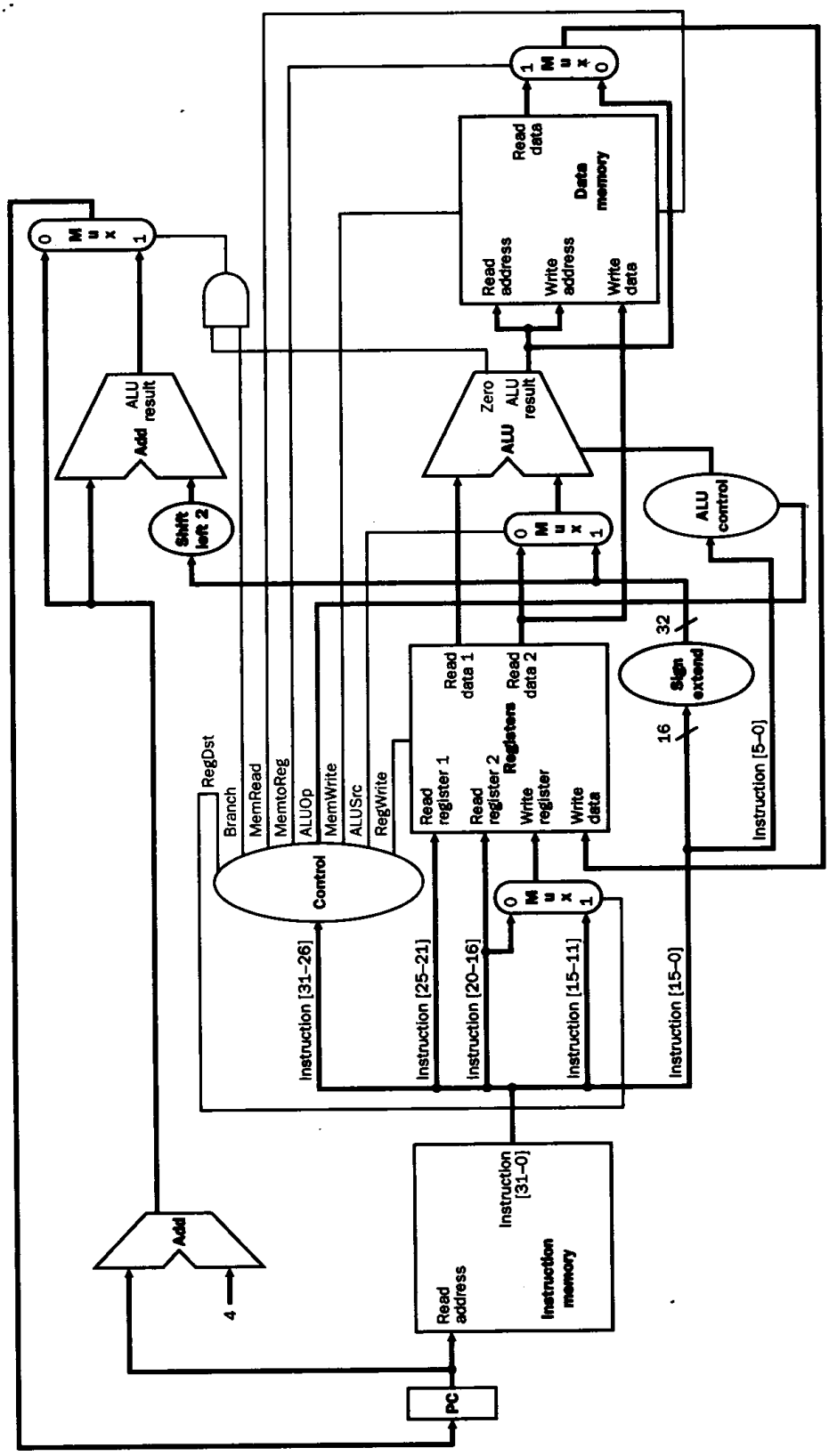
(*PCSrc* signal odlučuje da li će se u PC registar upisati ciljna adresa grananja Target ili PC+4).

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.

5.5 MULTITAKTNA IMPLEMENTACIJA

Jednotaktna arhitektura predvidja izvršavanje instrukcija obavljenjem niza koraka, navedenih u tabeli 2 u poglavlju 5.2, koji odgovaraju funkcionisanju upotrijebljenih elemenata. Prilikom implementacije svake od instrukcija, ovi koraći se obavljaju jedan za drugim i svi oni zajedno se izvršavaju u toku trajanja jednog taktnog intervala. Posljednja osobina implicira multipliciranje svih funkcionalnih elemenata koje je potrebno upotrijebiti više nego jedan put za vrijeme izvršavanja instrukcije, što je detaljno elaborirano u sekciji 5.4.

Ipak, sve implementirane instrukcije ne zahtijevaju obavljanje istih koraka, niti, što je još važnije, istog broja koraka tokom svog izvršavanja. Time se kod jednotaktne implementacije implicira i potencijalno rasipanje značajnog vremena prilikom izvršavanja većine istrukcija. Naime, dužina taktnog intervala mora biti odredjena tako da obezbijedi pouzdano izvršavanje svih instrukcija, pa i vremenski najzahtjevnije instrukcije (*lw*, koja zahtijeva obavljanje 5 koraka tokom svog izvršavanja – pogledaj tabelu 2). Medjutim, pošto je taktni interval fiksne dužine, kod jednotaktne implementacije je i izvršavanje ostalih instrukcija takodje odredjeno istim taktnim intervalom. Drugim riječima, prilikom implementacije instrukcija uslovnog i bezuslovnog skoka, koje zahtijevaju 3 koraka za svoje izvršavanje, rasipa se približno 40% vremena (odnosno, rasipaju se 2 nepotrebna od ukupno 5 koraka sa kojima je odredjena fiksna dužina taktnog intervala). Sa druge strane, prilikom implementacije instrukcija koje zahtijevaju 4 koraka za svoje izvršavanje, rasipa se približno 20% vremena (odnosno, rasipa se 1 nepotrebni od ukupno 5 koraka sa kojima je odredjena fiksna dužina taktnog intervala).

U cilju prevazilaženja navedenih nedostataka jednotaktne arhitekture, implementira se multitaktna (višetaktna, odnosno eng. *multiple clock-cycle*) arhitektura. Multitaktna arhitektura predvidja izvršavanje instrukcija obavljenjem istog niza koraka, kao u slučaju jednotaktnog dizajna, navedenih u tabeli 2, ali za razliku od jednotaktne arhitekture, svaki od koraka obavlja se u toku posebnog taktnog intervala. Na ovaj način, postižu se sljedeće karakteristike:

- Svaka instrukcija uzima odredjeni broj taktnih intervala za svoje izvršavanje, i to onaj broj taktova koji odgovara koracima koje instrukcija zahtijeva. Preciznije, instrukcije uslovnog i bezuslovnog skoka, koje zahtijevaju 3 koraka, biće izvršavane u toku 3 taktna intervala, instrukcija *lw*, koja zahtijeva 5 koraka, biće izvršavana u toku 5 taktnih intervala, a instrukcije koje zahtijevaju 4 koraka, biće izvršavane u toku 4 taktna intervala,
- Ne multipliciraju se funkcionalni elementi koji se upotrebljavaju u implementaciji, čime se redukuje hardware-ska složenost multitaktne implementacije u poredjenju sa jednotaktnom. Naime, u jednotaktnoj implementaciji, slika 8, ALU se multiplicira 3 pita (ALU i 2 sabirača), a memorija 2 puta (Instruction i Data memory). Kod multitaktne implementacije, za izvršavanje instrukcija upotrebljavaju se najmanje 3 taktna intervala, tako da se bilo koji elemenat, pa i ALU, može najmanje 3 puta upotrijebiti tokom izvršavanja bilo koje instrukcije (ograničenje je da se elementi mogu upotrijebiti tačno jedan put po taktnom intervalu),
- Svaka instrukcija upotrebljava samo neophodan broj taktnih intervala za svoje izvršavanje i nema rasipanja vremena u smislu da neka instrukcija, ili više njih uzimaju više taktnih intervala nego što je neophodno. U tom smislu, na nivou više izvršavanih instrukcija (djelova programa ili čitavih programa), multitaktna implementacija može poboljšati vrijeme izvršavanja zahtijevano jednotaktnom implementacijom.

  NAPOMENA 1: Dužina taktnog intervala multitaktne implementacije treba da obezbijedi pouzdano izvršavanje svakog od koraka koji se obavlja tokom izvršavanja instrukcija (pogledaj tabelu 2). Shodno tome, dužina ovog taktnog intervala odredjena je vremenski najzahtjevnijim korakom. Primijetimo da pristup memoriji računara (u cilju čitanja ili upisivanja podataka) zahtijeva najduže vrijeme za svoje izvršavanje, tako da je izvršavanjem ove akcije odredjena dužina taktnog intervala multitaktne implementacije. Ipak, pošto je taktni interval fiksne dužine, sve ostale akcije/koraci, koje zahtijevaju kraće vrijeme za svoje izvršavanje od pristupa memoriji, izvršavaju se u toku istog taktnog intervala, što takodje dovodi do izvjesnog rasipanja vremena. Sada je neophodno procijeniti: u kom slučaju

(jednotaktne ili multitaktne implementacije) je rasipanje vremena značajnije i, shodno tome, koja implementacija obezbjedjuje bolje vremenske karakteristike?

Odgovor na ovo pitanje može se dati sa 2 nivoa:

- Izvršavanja jedne intrukcije – na primjer vremenski najzahtjevnije *lw*, ili
- Izvršavanja grupe instrukcija.

NAPOMENA 2: Odgovor posmatran sa nivoa jedne instrukcije prilično je jednostavan. Jednotaktnom implementacijom ne rasipa se vrijeme prilikom implementacije instrukcije *lw*. Naime, taktni interval jednotaktne implementacije odredjen je vremenom neophodnim za izvršavanje ove instrukcije, a koraci neophodni za njeno izvršavanje uzimaju tačno onoliko vremena koliko je potrebno za njihovo obavljanje. Sa druge strane, kod multitaktne implementacije instrukcije *lw*, svi koraci koji zahtijevaju kraće vrijeme od vremena pristupa memoriji (memoriji se pristupa u I i u IV koraku izvršavanja *lw* instrukcije) uzrokuju rasipanje vremena. Drugim riječima, jednotaktnom implementacijom se optimizira vrijeme izvršavanja pojedinačne insttrukcije *lw*.

Odgovor na postavljeno pitanje s aspekta grupe izvršavanih instrukcija (dijela ili čitavog programa) značajno je složeniji i zahtijeva dublju analizu. U cilju njegovog kreiranja, posmatrajmo sljedeći primjer izvršavanja grupe instrukcija.

Primjer: Pretpostavimo da se izvršava grupa od 5 instrukcija, od kojih je jedna *lw* instrukcija, jedna instrukcija uslovnog ili bezuslovnog skoka, koja zahtijeva 3 koraka za svoje izvršavanje, te 3 instrukcije koje zahtijevaju 4 koraka za svoja izvršavanja. Pretpostavimo da je za pristup memoriji potrebno vrijeme od 12 jedinica vremena (j.v.), za izvršavanje operacije ALU – 9 j.v., a za upisivanje podataka u Registers jedinicu 8 j.v. Napomenimo da pretpostavljene proporcije u vremenima izvršavanja prethodno navedenih akcija odgovaraju realnoj situaciji.

Izračunajmo najprije zahtijevanu dužinu taktnog intervala kod jednotaktne implementacije (T_{SCI}, gdje skraćenica SCI odgovara Single Clock-cycle Implementation – jednotaktnoj implementaciji), a potom i kod multitaktne implementacije (T_{MCI}, gdje skraćenica MCI odgovara Multiple Clock-cycle Implementation – multitaktnoj implementaciji):

- Kod jednotaktne implementacije, dužina taktnog intervala odredjena je vremenom neophodnim za izvršavanje vremenski najzahtjevnije instrukcije *lw*. Razmatranjem tabele 2 iz sekcije 5.2, možemo primijetiti da *lw* instrukcije zahtijeva
  - dvostruki pristup memoriji (u I koraku za čitanje instrukcije iz memorije i u IV koraku za čitanje podatka iz memorije),
  - izračunavanje adrese lokacije (od strane ALU) sa koje će podatak biti pročitan (u III koraku),
  - dekodiranje instrukcije (u II koraku) koje se izvršava u paraleli sa izračunavanjem (od strane sabirača/ALU) ciljne adrese grananja Target, tako da će vrijeme izvršavanja ovog koraka biti odredjeno vremenom potrebnim sabiraču/ALU za izračunavanje ciljne adrese grananja,
  - upisivanje podatka u odredjeni registar Registers jedinice (V korak).

  Shodno prethodno navedenom, te pretpostavljenim vremenskih zahtjevima pojedinih operacija, zaključujemo da dužina taktnog intervala kod jednotaktne implementacije iznosi:

  $$T_{SCI}=(2\times12+2\times9+8)\text{ j.v.}=50\text{ j.v.},$$

  te da je T_{SCI} ujedno vrijeme izvršavanja svih implementiranih instrukcija.
- Kod multitaktne implementacije, dužina taktnog intervala odredjena je vremenom neophodnim za izvršavanje vremenski najzahtjevnije operacije/koraka koji se obavlja tokom izvršavanja instrukcija. Već je navedeno da je pristup memoriji vremenski najzahtjevija operacija, te da se ona obavlja u I koraku svih razmatranih instrukcija (pogledaj tabelu 2). Uz

to, pretpostavljeno je da da vrijeme pristupa memoriji iznosi 12 j.v., tako da dužina taktnog intervala kod multitaktne implementacije iznosi:

$$T_{MCI}=12 \text{ j.v.}$$

Vrijeme izvršavanja svake pojedinačne instrukcije kod multitaktne implementacija odgovara umnošku T_{MCI} sa brojem koraka koje ta instrukcija zahtijeva za svoje izvršavanje.

Vremena izvršavanja pojedinih instrukcija i ukupno vrijeme izvršavanja zadate grupe instrukcija u jednotaktnoj (SCI) i multitaktnoj (MCI) imaplementaciji predstavljeni su u tabeli 10.

Tabela 10. Grupa instrukcija zadatih u primjeru, vrijeme izvršavanja svake od njih u slučaju jednotaktne (SCI) i multitaktne implementacije (MCI), kao i ukupno vrijeme izvršavanja zadate grupe instrukcija.

Razmatrane instrukcije	SCI	MCI
(Bez)uslovni skok (3 CLK)	50 j.v.	36 j.v.
Instrukcija (4 CLK)	50 j.v.	48 j.v.
Instrukcija (4 CLK)	50 j.v.	48 j.v.
Instrukcija (4 CLK)	50 j.v.	48 j.v.
lw (5 CLK)	50 j.v.	60 j.v.
UKUPNO:	250 j.v.	240 j.v.

U skladu sa razmatranjima iz napomene 2 iz ovog poglavlja, primijetimo da jednotaktna imlementacija (SCI) poboljšava vrijeme izvršavanja instrukcije *lw* i to 20% u odnosu na multitaktnu implementaciju (MCI). Medjutim, multitaktna implementacija obezbjedjuje brže izvršavanje pretpostavljene grupe od 5 instrukcija. Preciznije, zavisno od instrukcija koje sačinjavaju grupu, multitaktna implementacija može obezbjedjivati poboljšane vremenske karakteristike u odnosu na odgovarajuću jednotaktnu implementaciju, i to iz razloga uzimanja više od jednog taktnog intervala po instrukciji i, što je još važnije, različitog (samo neophodnog) broja taktnih intervala za izvršavanje svake od različitih instrukcija pojedinačno. Drugim riječima, vrijeme izvršavanja ne mora biti nedostatak multitaktne implementacije u poredjenju sa odgovarajućom jednotaktnom.

ZAKLJUČAK: Saglasno navedenom, u odnosu na odgovarajuću jednotaktnu implementaciju, multitaktna implementacija

1. Obezbjedjuje značajne redukcije u upotrijebljenom hardware-u,
2. Zavisno od grupe instrukcija čije izvršavanje se zahtijeva, ne mora imati inferiorne vremenske karakteristike.

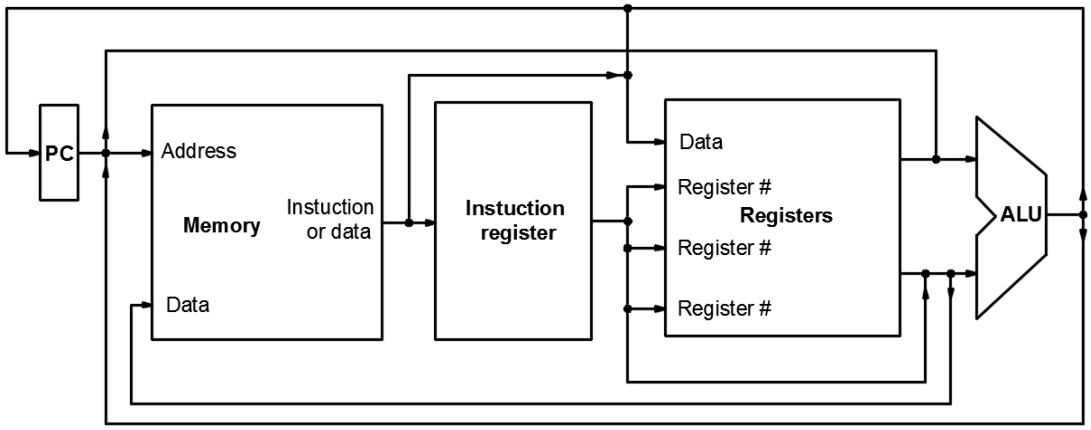
Shodno tome, multitaktna implementacija predstavlja praktično aplikabilan dizajn koji se upotrebljava u svim modernim računarskim sistemima. Stoga će joj u nastavku biti posvećena potpuna pažnja.

5.5.1 Datapath za multitaktnu implementaciju

U prethodnom poglavlju uočeno je da multitaktna arhitektura treba da obezbjedi implementaciju instrukcija iz posmatranog seta (instrukcije R-tipa (*add*, *sub*, *and*, *or*, *slt*), *lw*, *sw*, *beq* i *j*) u 3 do 5 taktnih intervala, odnosno upotrebu svake od zahtijevanih funkcionalnih jedinica 3 do 5 puta tokom izvršavanja svake instrukcije. Stoga je, u poredjenju sa jednotaktnom arhitekturom sa slike 8, kod multitaktne arhitekture nepotrebna upotrijeba ALU i 2 dodatna sabirača (2 "zaglupljene" ALU sa kontrolnim signalima 010), kao i dvije memorije (Instruction memory i Data memory), već samo

- jedne ALU koja će obavljati funkciju i ALU, sa slike 8, i 2 dodatna sabirača (u prva 3 koraka izvršavanja),
- jedne Memory koja će obavljati funkcije Instruction i Data memory sa slike 8.

Shodno navedenom, opšti pogled na multitaktni datapath za implementaciju razmatranog seta instrukcija (instrukcije R-tipa (*add*, *sub*, *and*, *or*, *slt*), *lw*, *sw*, *beq* i *j*) prikazan je na slici 9. Primijetimo da datapath sa slike 9, za razliku od jednotaktnog datapath-a sa slike 1, odnosno jednotaktnog dizajna



Slika 9. Sažeti prikaz datapath-a koji je namijenjen multitaktnom izvršavanju instrukcija.

sa slike 8, uključuje jednu memoriju i jednu ALU, ali i da su, shodno tome, dodatno share-ovani pojedini ulazi ovih jedinica, kao i da je dodat registar označen sa Instruction register (IR).

ZAPAŽANJE 1: ALU upotrijebljana kod multitaktnog datapath-a sa slike 9, obavlja funkcije ALU i dva dodatna sabirača upotrijebljenih kod jednotaktne arhitekture sa slike 8. Shodno tome, broj potencijalnih operanada ALU mora biti povećan, odnosno oba ulaza ALU sa slike 9 moraju biti share-ovana (svaki od njih za odgovarajuće ulaze ALU i ulaze 2 sabirača sa slike 8) u skladu sa razmatranjima prezentiranim u tabeli 11.

Tabela 11. Funkcionalni elementi/uredjaji (i njihovi ulazi) upotrijebljeni u jednotaktnoj arhitekturi sa slike 8 čije funkcije u miltitaktnoj implementaciji izvršava ALU sa slike 9.

Uredjaj	I ulaz	II ulaz
ALU	Reg(rs)	Reg(rt), sign_extend(address)
Add (za izračunavanja PC+4)	PC	Const. 4
Add (za izračunavanje Target)	PC+4	sign_extend(address)<<2

Kao i ranije, share-ovanje se vrši dodavanjem multipleksora sa odgovarajućim brojem ulaza na prvom i na drugom ulazu ALU sa slike 9. Primijetimo da će multipleksor na prvom ulazu biti novouveden u poredjenju sa jednotaktnom arhitekturom sa slike 8, a multipleksor na drugom ulazu ALU proširen u odnosu na odgovarajući multipleksor sa slike 8.

- Multipleksor na prvom ulazu ALU treba da ima 2 ulaza (za operande Reg(rs) i sadržaj PC registra, tabela 11), te jedan selekcioni ulaz, koji će biti nazvan shodno funkciji koju obavlja, odnosno *ALUSelA* (selektuje prvi, odnosno operand A ALU).

  NAPOMENA 1: Na prvi pogled, na osnovu razmatranja iz tabele 11, multipleksor na prvom ulazu ALU trebao bi da ima 3 ulaza (za operande Reg(rs), PC i PC+4). Medjutim, kao što je može primijetiti iz tabele 2 (poglavlje 5.2) i kao što će biti implementirano u nastavku, nakon izračunavanja (u prvom koraku/taktnom intervalu), PC+4 upisuje se nazad u PC registar, tako da u trenutku upotrebe PC+4 (u drugom koraku u cilju izračunavanja ciljne adrese Target), sadržaj PC registra suštinski je PC+4 s aspekta instrukcije koja se izvršava. Iz ovog razloga, multipleksor na prvom ulazu ALU treba da posjeduje 2 ulaza (za operande Reg(rs) i PC), gdje PC u prvom koraku sadrži adresu izvršavane instrukcije, dok u drugom koraku sadrži adresu instrukcije koja je u memoriji zapisana odmah nakon izvršavane instrukcije.

- Multipleksor na drugom ulazu ALU treba da ima 4 ulaza:
  - 2 ranije prepoznata ulaza (kod jednotaktne arhitekture sa slike 8) za operande Reg(rt) i sign_extend(address),
  - 1 ulaz za operand Const. 4, koja se dovodi na drugi ulaz prvog sabirača upotrijebljavanog kod jednotaktne arhitekture (slika 8) za formiranje adrese prve sljedeće lokacije (PC+4),

- 1 ulaz za operand sign_extend(address)<<2, koji se dovodi na drugi ulaz drugog sabirača upotrebljavanog kod jednotaktne arhitekture (slika 8) za formiranje ciljne adrese grananja Target.

Multipleksor na drugom ulazu ALU pored 4 ulaza za operande treba da sadrži i 2 selekciona ulaza, koji će biti nazvani shodno funkciji koju obavljaju, odnosno *ALUSelB* (selektuje drugi, odnosno operand B ALU).

ZAPAŽANJE 2: Memory upotrijebljana kod multitaktnog datapath-a sa slike 9, obavlja funkcije Instruction memory i Data memory upotrijebljenih kod jednotaktne arhitekture sa slike 8, u skladu sa razmatranjima prezentiranim u tabeli 12.

Tabela 12. Memorijski elementi/uredjaji (i njihovi adresni ulazi) upotrijebljeni u jednotaktnoj arhitekturi sa slike 8 čije funkcije u multitaktnoj implementaciji izvršava Memory jedinica sa slike 9.

Uredjaj	Read address	Write address
Instruction memory	PC	×
Data memory	ALUOut	ALUOut

Shodno činjenici, prikazanoj i u tabeli 12, da se obje ove memorije u jednotaktnoj implementaciji upotrebljavaju za uzimanje/čitanje instrukcija/podataka (Data memory i za upisivanje podataka), na Read address ulaz jedinstvene Memory (na slici 9, zbog njene opštosti i ne ulaženja u detalje, označen sa Address) može biti dovedena adresa sadržana u PC registru (u cilju uzimanja/čitanja instrukcije sa odgovarajuće lokacije), a može biti dovedena adresa izračunata od strane ALU (u cilju čitanja podatka sa odgovarajuće lokacije prilikom implementacije instrukcije *lw*). U prilog navedenom,

– Read address ulaz Memory jedinice mora biti share-ovan dodavanjem multipleksora sa 2 ulaza

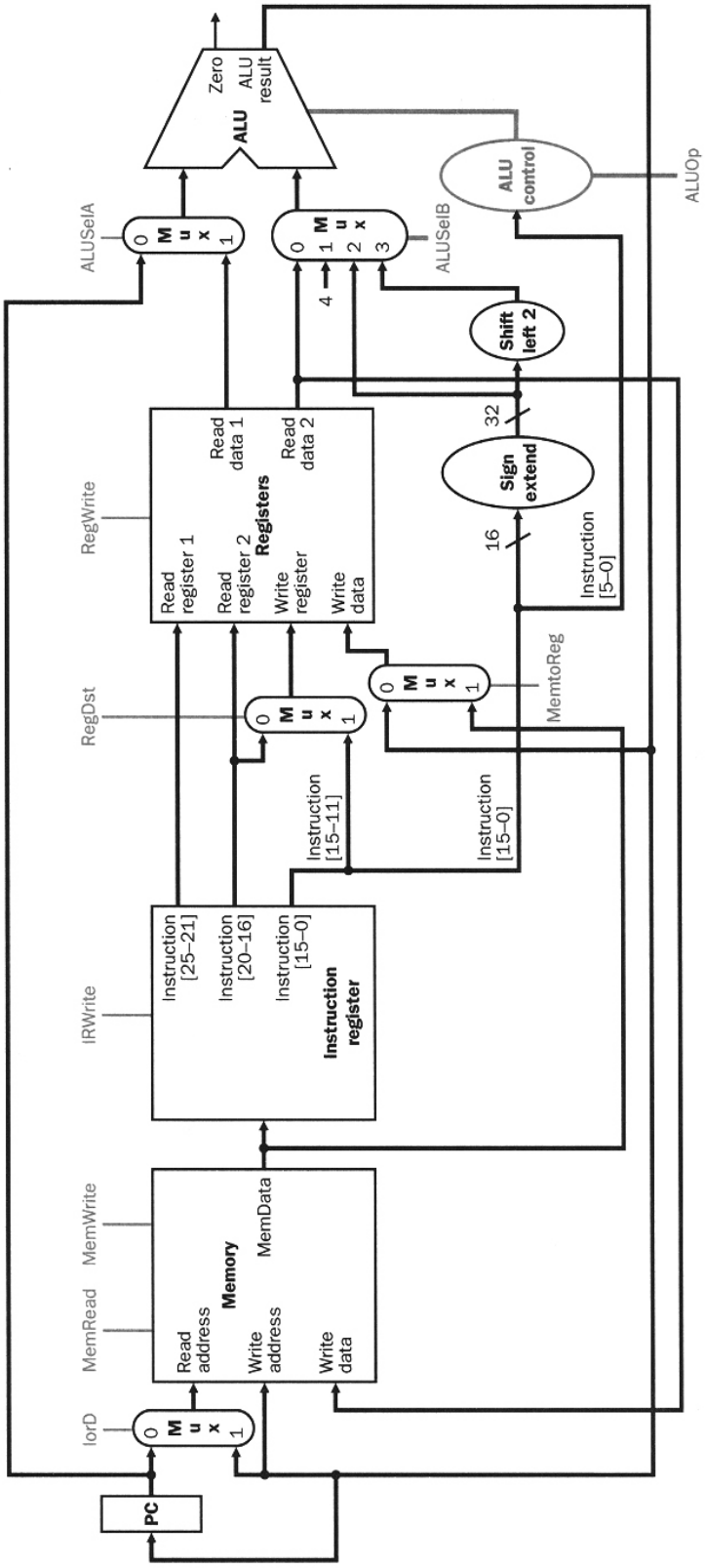
  - Na prvi ulaz ovog multipleksora dovodit će se sadržaj PC registra, u cilju obezbjedjivanja uslova za potencijalno uzimanje/čitanje instrukcije sa odgovarajuće lokacije,
  - Na drugi ulaz ovog multipleksora dovodit će se adresa izračunata od strane ALU (ALUOut), u cilju obezbjedjivanja uslova za potencijalno čitanje podatka sa odgovarajuće lokacije prilikom implementacije instrukcije *lw*.

  Multipleksor na Read address ulazu Memory jedinice pored 2 ulaza za odgovarajuće adrese treba da sadrži i 1 selekcioni ulaz, koji će biti nazvani shodno funkciji koju obavljaja, odnosno *IorD* (selektuje adresu lokacije u kojoj se nalazi instrukcija (I) ili podatak (D)).

– Na Write address ulaz Memory jedinice dovodit će se samo jedan signal – adresa izračunata od strane ALU (ALUOut), u cilju adresiranja lokacije u koju će potencijalno (prilikom implementacije instrukcije *sw*) biti upisan podatak doveden na Write data ulaz ove jedinice (na slici 9, zbog njene opštosti i ne ulaženja u detalje, Write data ulaz je označen sa Data).

Primijetimo da će multipleksor dodat na Read address ulazu Memory jedinice takodje biti novouveden u poredjenju sa jednotaktnom arhitekturom sa slike 8.

ZAPAŽANJE 3: Prilikom izvršavanje *lw* instrukcije, Memory jedinica upotrebljava se dva puta i to oba puta u funkciji čitanja (instrukcije, a potom i podatka). Naime, tom prilikom se, u toku prvog koraka, instrukcija čita iz Memory jedinice (u cilju njenog donošenja u proces izvršavanja), dok se tokom njenog izvršavanja (u IV koraku), iz Memory jedinice čita podatak. Primijetimo da prilikom čitanja podatka, instrukcija *lw* još uvijek nije izvršena, te da je potrebno izvršavanje još jednog koraka u cilju njenog kompletiranja (pogledaj tabelu 2 u sekciji 5.2). Drugim riječima, ukoliko se ne obezbijedi čuvanje instrukcije nakon njenog uzimanja/čitanja, pa do kraja njenog izvršavanja, podatak pročitan u IV koraku će prepisati izvršavanu instrukciju *lw* ili njene još uvijek neupotrijebljene djelove, odnosno ova instrukcija neće se moći izvršiti do kraja, te će dalji rad računara biti nepouzdan. Ova neželjena situacija prevazilazi se uvodjenjem dodatnog registra Instruction register (IR) koji je namijenjen čuvanju mašinskog koda instrukcije, od trenutka njenog uzimanja/čitanja iz Memory jedinice do njenog kompletiranja. Nakon toga, bezbjedno je, prilikom izvršavanja instrukcije *lw*, pročitati podatak iz Memory jedinice.



Slika 10. Detaljniji prikaz datapath-a za multitaktnu implementaciju sa naznačenim kontrolnim signalima memorijskih jedinica i prikazanih multipleksora. Primijetimo da na slici nije prikazan dio namijenjen upisivanju u PC registar, kao ni kontrolni signal upisivanja u PC registar.

Datapath za multitaktnu implementaciju koji sadrži značajno više detalja u odnosu na datapath sa slike 9, i koji je kreiran na osnovu prethodnih zapažanja 1–3, prezentiran je na slici 10.

ANALIZA: Pažljivom analizom datapath-a sa slike 10, pored prethodno detaljno razmatrane upotrebe samo jedne Memory jedinice (pogledaj zapažanje 2) i samo jedne ALU (pogledaj zapažanje 1), kao i uključivanja (u implementaciju) IR registra (pogledaj zapažanje 3), mogu se primijetiti i sljedeće činjenice koje dodatno razlikuju multitaktnu implementaciju sa slike 10 od jednotaktne implementacije sa slike 8:

1. Datapath sa slike 10 uključuje 2 dodatna multipleksora. Kao što je naznačeno u zapažanjima 1 i 2, to su multipleksori MUX 2/1 sa po jednim selekcionim ulazom. Jedan multipleksor MUX 2/1 dodat je na prvom ulazu ALU, a drugi na Read address ulazu Memory jedinice,
2. Postojećem multipleksoru MUX 2/1 na drugom ulazu ALU sa slike 8 dodata su 2 ulaza, tako da je na drugom ulazu ALU multipleksor MUX 4/1,
3. Ostali upotrijebljeni multipleksori na slici 10 (na Write register ulazu i Write data ulazu Registers jedinice) i kontrolni signali (*RegDst* i *MemtoReg*) dovedeni na njihove selekcione ulaze odgovaraju multipleksorima i pripadajućim kontrolnim signalima upotrijebljenim na istim ulazima Registers jedinice kod jednotaktne implementacije sa slike 8 (za funkcije ovih multipleksora pogledaj napomene 1 i 2 u sekciji 5.3.3 i razmatranja iz poglavlja 5.4),
4. Upotrijebljeni memorijski elementi (Memory jedinica, Registers jedinica i IR registar) posjeduju kontrolu upisivanja podataka. Nazivi njihovih kontrolnih signala (*MemWrite*, *RegWrite* i *IRWrite*) odgovaraju memorijskim elementima čije funkcionisanje kontrolišu i funkciji koju obavljaju,
5. Kao i u slučaju jednotaktne implementacije, pored kontrole upisivanja, Memory jedinica sadrži i kontrolni signal čitanja *MemRead* (potreba za uvodjenjem *MemRead* kontrolnog signala detaljno je razmatrana u napomeni 3 u poglavlju 5.4).

NAPOMENA 2: Na slici 10 nije uključen dio datapath-a koji obezbjedjuje upisivanje u PC registar, a shodno tome, ni kontrola upisivanja u PC registar. Ovaj dio datapath-a biće uključen kada i kontrolna jedinica i svi kontrolni signali koji utiču na kontrolu upisivanja u PC registar (treba predvidjeti kontrolu bezuslovnog i uslovnog upisivanja u ovaj registar u cilju implementacija instrukcije uslovnog skoka/grananja *beq*). Ipak, jednostavno je uočiti da na ulazu PC registra treba dodati multipleksor sa 3 ulaza (MUX 3/1), čiji ulazi treba da odgovaraju budućim potencijalnim sadržajima PC registra:

1. PC+4, kod implementacije instrukcija koje se sukcesivno izvršavaju (jedna-za-drugom),
2. Adresa Target, kod implementacije instrukcije uslovnog skoka/grananja *beq*,
3. Adresa bezuslovnog skoka, kod implementacije instrukcije *j*,

što je realizovano na slici 11. Primijetimo da ovaj multipleksor (MUX 3/1) treba da sadrži 2 selekciona ulaza na koja se dovodi 2-bitni *PCSource* kontrolni signal. Ipak, kontrolni signali i njihovo set-ovanje biće tema našeg interesovanja tokom kreiranja glavne kontrolne jedinice multitaktne implementacije.

5.5.2 Uključivanje registara za privremeno smještanje podataka u datapath za multitaktnu implementaciju

Kod jednotaktne implementacije (poglavlje 5.4) ne postoji mogućnost, ni potreba za uključivanjem regist(a)ra za privremeno smještanje podataka. Nema mogućnosti, jer jednotaktna implementacija raspolaže samo sa jednim taktnim intervalom tokom izvršavanja pojedinačne instrukcije i ne postoji dodatni taktni interval na čijoj bi se jednoj od ivica mogao u registru sačuvati podatak od prebrisavanja, da bi se isti kasnije (tokom izvršavanja iste instrukcije) mogao upotrijebiti. Nema potrebe, jer svaka akcija koja se obavlja mora biti izvršena u toku taktnog intervala kojim se raspolaže.

Sa druge strane, kod multitaktnie implementacije, svaka od instrukcija izvtršava se u toku trajanja nekoliko taktnih interval (3–5 taktnih intervala, zavisno od instrukcije koja se izvršava). Drugim riječima, kod višetaktne implementacija postoji (i te kako postoji) mogućnost za upotrebu regist(a)ra za privremeno smještanje podataka. Samo se postavlja pitanje da li postoji potreba za tim?

Odgovor na ovo pitanje već je dat u zapažanju 3 u sekciji 5.5.1. Naime IR registar, upotrijebljen prilikom dizajniranja datapath-a za multitaktnu implementaciju (slike 9 i 10), predstavlja registar za privremeno smještanje instrukcije (uzete/pročitane iz Memory jedinice u prvom taktnom intervalu i smještene u IR registar do kraja njenog izvršavanja – do III, IV ili V taktnog intervala, zavismo od trajanja instrukcije koja se izvršava). Potreba za uključivanjem IR registra u multitaktni implementaciju više je nego očigledna i detaljno je elaborirana u zapažanju 3 u sekciji 5.5.1.

Osim instrukcije, može postojati potreba za privremenim čuvanjem podat(a)ka ili rezultata ALU. Uočimo 2 kriterijuma neminovne upotreba registara za privremeno čuvanje podataka:

1. Instrukcija/podatak se uzima/čita iz memorijskog elementa u jednom taktnom intervalu, upotrebljava se u sljedećem ili u nekoliko sljedećih taktnih intervala, tokom njene/njegove upotrebe iz istog memorijskog elementa se uzima/čita novi podatak, a prvopročitana/i instrukcija/podatak nije u medjuvremenu sačuvana od prebrisavanja,
2. Rezultat se izračunava (od strane kombinacione logike – funkcionalnog elementa) u jednom taktnom intervalu, upotrebljava se u sljedećem ili nekoliko sljedećih taktnih intervala, tokom njegove upotrebe ista kombinaciona logika vrši nova izračunavanja, a prvodobijeni rezultat nije u medjuvremenu sačuvan od prebrisavanja.

Primijetimo da je uključivanje IR registra u multitaktnu implementaciju posljedica kriterijuma 1. od dva prethodno navedena kriterijuma neminovne upotrebe registra za privremeno čuvanje podataka. Razmotrimo što je sa rezultatima koje kreira ALU na svom izlazu i da li postoji potreba za uključivanjem dodatnih registara za privremeno smještanje njenih rezultata. ALU vrši izračunavanja PC+4, ciljne adrese grananja Target i operacije zahtijevane instrukcijom:

- PC+4 izračunava se u I taktnom intervalu izvršavanja instrukcija i na kraju istog taktnog intervala upisuje se nazad u PC registar (pogledaj tabelu 2). Drugim riječima, izračunati rezultat sačuva se u memorijskom elementu (PC registru) u toku istog taktnog intervala, tako da nema potrebe za njegovim čuvanjem i u nekom drugom memorijskom elementu.

  NAPOMENA 1: PC+4, nakon izračunavanja, upisuje se nazad u PC registar, jer će, u najvećem broju slučajeva, adresa instrukcije koja sljedeća treba da se izvrši biti upravo PC+4 (od ovoga pravila odstupaju samo instrukcije bezuslovnog skoka i instrukcije uslovnog skoka/grananja, ali ove potonje samo kada je uslov grananja zadovoljen).

- Ciljna adresa grananja Target izračunava se u II taktnom intervalu izvršavanja instrukcija, upotrebljava se u III taktnom intervalu ukoliko se izvršava *beq* instrukcija i ukoliko je zadovoljen uslov grananja (pogledaj tabelu 2) i nema smisla da se nakon izračunavanja upiše nazad u PC registar (time bi se izgibilo PC+4, izračunato u I taktnom intervalu i upisano u PC registar na kraju istog taktnog intervala). Medjutim, u III taktnom intervalu, ALU će biti upotrijebljena za izračunavanje operacije zahtijevane instrukcijom, pa čak i ukoliko je riječ o instrukciji *beq* (za poredjenje operanada Reg(rs) i Reg(rt)). Drugim riječima, ukoliko Target adresa, izračunata u II taktnom intervalu, ne bude sačuvana, biće prebrisana već u III taktnom intervalu i to tokom ispitivanja uslova grananja (prije konačne upotrebe adrese Target). Ovim je zadovoljen kriterijum 2. za uključivanje registra za privremeno smještanje ciljne adrese grananja. Registar je uključen u implementaciju, kao što je prikazano na slici 11, i nazvan je Target, shodno podatku (adresi) koji čuva od prebrisavanja (pogledaj sliku 11).

  NAPOMENA 2: Izračunata ciljna adresa grananja ne upisuje se nazad u PC registar nakon njenog izračunavanja (na kraju II taktnog intervala). Naime, time bi se prebrisala vrijednost PC+4 upisana na kraju I taktnog intervala, a sa druge strane, ne postoji garancija da će izračunata ciljna adresa grananja uopšte biti upotrijebljena do kraja izvršavanja instrukcije (upotrebljava se samo ukoliko se izvršava *beq* instrukcija, a to još uvijek ne znamo za vrijeme izračunavanja ciljne adrese grananja, i samo ukoliko je uslov grananja zadovoljen).

- Operacija zahtijevana instrukcijom izvršava se u III taktnom intervalu (pogledaj tabelu 2), a dobijeni rezultat upotrebljava se u IV taktnom intervalu (prilikom implementacije instrukcija R-tipa i *sw* instrukcije – pogledaj tabelu 2) ili u IV i u V taktnom intervalu (prilikom

implementacije *lw* instrukcije – pogledaj tabelu 2). Medjutim, na ulaze ALU nalaze se isti potencijalni operandi tokom svakog od navedenih taktnih intervala. Naime, instrukcija, odnosno sva njena polja (pa i ona polja čijim sadržajima se označavaju registri iz Registers jedinice koji sliže kao potencijalni operandi ALU i address/immediate/offset polje koje takodje može obavljati ulogu operanda ALU) sačuvani su u IR registru i ne mogu biti izmijenjeni do kraja izvršavanja instrukcije (na koncu, to je namjena IR registra). Sa druge strane, selekciju operanada ALU zahtijevanih instrukcijom obavljaju kontrolni signali *ALUSelA* i *ALUSelB* multipleksora na ulazima ALU, dok selekciju operacije ALU zahtijevane instrukcijom obavljaju *ALUOp* kontrolni signali, a vrijednosti kontrolnih signala postavlja/set-uje glavna kontrolna jedinica koju ćemo mi dizajnirati. Dakle, selekcija operanada i funkcionisanje ALU je pod potpunom kontrolom dizajnera glavne kontrolne jedinice, tako da se do kraja izvršavanja instrukcije može obezbijediti zadržavanje istog rezultata na izlazu ALU, odnosno neprebrisavanje rezultata ALU, dobijenog u III taktnom intervalu. Drugim riječima, ne postoji potreba za uključivanjem dodatnog registra za privremeno smještanje rezultata ALU.

SUMARUM: Na slici 11, prikazan je kompletni dizajn za multitaktnu implementaciju, koji uključuje 2 registra za privremeno smještanje podataka/rezultata ALU: Instruction registar (IR – za detalje pogledati zapažanje 3 u sekciji 5.5.1) i Target registar za privremeno smještanje ciljne adrese grananja.

NAPOMENA 3: Arhitektura za multitaktnu implementaciju, prikazana na slici 11, uključuje glavnu kontrolnu jedinicu sa njenim ulazima, op poljem instrukcije (Op [5–0]=Instruction [31–26]), i izlazima–kontrolnim signalima–povezanim sa selekcionim ulazima upotrijebljenih multipleksora i kontrolnim ulazima upotrijebljenih memorijskih elemanata. Dodata je i kontrola upisivanja u PC registar (pogledaj napomenu 2 u sekciji 5.5.1). Ipak, dizajniranje glavne kontrolne jedinice i kontrola funkcionisanja datapath-a biće detaljno razmatrane u sljedećoj sekciji.

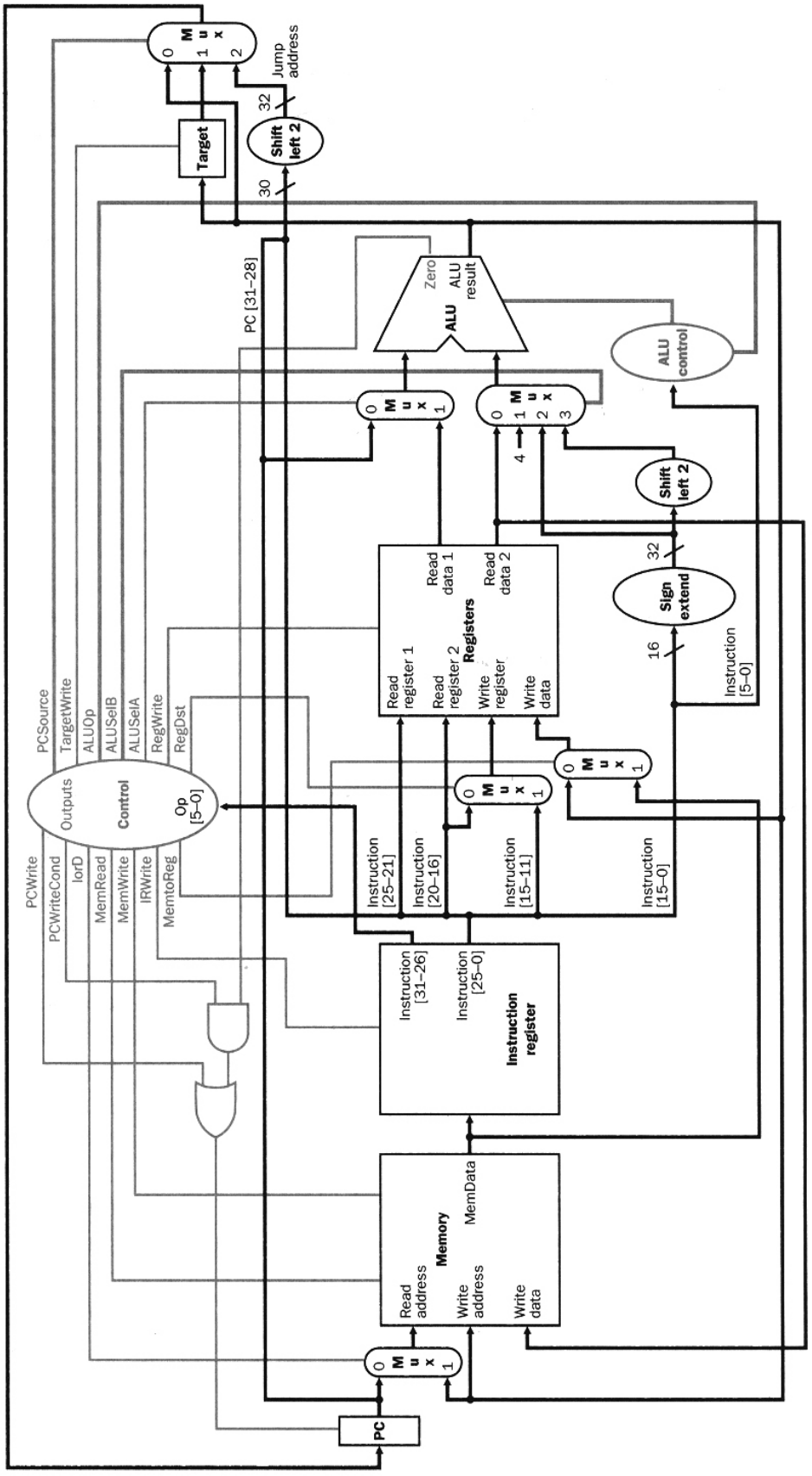
NAPOMENA 4: Na slici 11, prikazana je i realizacije instrukcije bezuslovnog skoka *j*. Realizuje se formiranjem 32-bitne adrese bezuslovnog skoka (Jump address na slici 11) i njenim dovodjenjem na ulaz br. 2 multipleksora koji se nalazi na ulazu PC registra (pogledaj napomenu 2 iz sekcije 5.5.1). Jump address se formira kombinacijom najviša 4 bita tekućeg sadržaja PC registra, PC [31–28], i 26-bitnog adresnog polja instrukcija J-tipa, Instruction [26–0], te shift-ovanjem tako dobijene 30-bitne adrese za 2 mjesta u lijevu stranu, u skladu sa razmatranjima iz sekcije 5.3.5, napomenom iz sekcije 5.3.4, te napomenom 7 i konvencijom iz poglavlja 3.7.

Detaljno poredjenje (s aspekta hardware-ske složenosti) jednotaktne implementacije, date na slici 8, i multitaktne implementacije, prikazane na slici 11, formirano na osnovu analize iz sekcije 5.5.1 i zapažanja iz napomene 2 iz iste sekcije, sumirano je u tabeli 13.

Tabela 13. Jednotaktna (SCI) vs multitaktna implementacija (MCI) s aspekta hardwere-ske složenosti.

Uredjaji	SCI	MCI
Funkcionalni jedinice	ALU + 2×Add	ALU $\begin{pmatrix} +\text{MUX 2/1 na I ulazu ALU} \\ +2 \text{ ulaza na MUX na II ulazu ALU} \end{pmatrix}$
Memorijske jedinice	Instruction + Data	Memory (+ MUX 2/1 na Read asddress ulazu)
Privremeni registri	0	2 (IR + Target)

ZAKLJUČNA RAZMATRANJA: Multitakna implementacija sa slike 11 uključuje po jednu veliku funcionalnu (ALU), odnosno memorijsku (Memory) jedinicu, za razliku od jednotaktne implementacije koja uključuje ALU i 2 dodatna sabirača, te 2 velike memorijske jedinice (Instruction i Data memory). Ovo se "plaća" (od strane multitaktne implementacije) sa 2 dodata multipleksora MUX 2/1, 2 dodata ulaza na još jednom multipleksoru MUX 2/1, te sa 2 registra za privremeno smještanje instrukcije/rezultata ALU (IR i Target registar). Medjutim, multipleksori i registri su mali i veoma jeftini elementi, koji, uz to, zahtijevaju malu količinu energije u poredjenju sa ALU i velikim memorijskim jedinicama. Stoga, možemo zaključiti da multitaktna implementacija optimizira ne samo hardware-sku složenost i gabarite sistema, već i njegovu cijenu i potrošnju energije, kao i vrijeme izvršavanja grupe instrukcija i izvršavanih programa (kao što je pokazano u sekciji 5.5). Shodno tome, multitaktna implementacija predstavlja praktično aplikabilno rješenje za dizajniranje procesora.



Slika 11. Kompletni dizajn za multitaktnu implementaciju.

5.6 DIZAJNIRANJE GLAVNE KONTROLNE JEDINICE MULTITAKTNE IMPLEMENTACIJE

Glavna kontrolna jedinica generiše selekcione ulaze svih multipleksora i kontrolne ulaze svih memorijskih elemenata upotrijebljenih u multitaknoj implementaciji prikazanoj na slicu 11. Svoje izlaze glavna kontrolna jedinica generiše na osnovu operacionog koda, op=Op [5–0]=Instruction [31–26], instrukcije koja se izvršava. Primijetimo da glavna kontrolna jednica multitaktne implementacije generiše značajno veći broj izlaznih signala u odnosu na glavnu kontrolnu jedinicu jednotaktne implementacije sa slike 8. Ova činjenica implicira zaključak o komplikovanijoj, odnosno značajno složenijoj kontroli multitaktne u odnosu na jednotaktnu implementaciju, u što ćemo se uvjeriti u ovom poglavlju. Pojednostavljeno, komplikovanija kontrola multitaktne implementacije može se posmatrati kao još jedan način "plaćanja" jednostavnije, manje gabaritne, enetgetski manje zahtjevne i jeftinije implementacije u odnosu na jednotaktnu implementaciju.

Prije nego predjemo na dizajniranje glavne kontrolne jedinice multitaktne implementacije sa slike 11, pobrojmo najprije ranije razmatrane (pogledaj zapažanja 1 i 2, analizu i napomenu 2 iz sekcije 5.5.1) kontrolne signale koje ona treba da generiše:

- Kontrolne signale na selekcionim ulazima svakog od 6 upotrijebljenih multipleksora:
  IorD signal multipleksora na Raed address ulazu Memory jedinice,
  RegDst signal multipleksora na Write register adresnom ulazu Registers jedinice,
  MemtoReg signal multipleksora na Write data ulazu Registers jedinice,
  ALUSelA signal multipleksora na prvom ulazu ALU,
  2-bitni *ALUSelB* signal multipleksora na drugom ulazu ALU,
  2-bitni *PCSource* signal multipleksora na ulazu PC registra.

- 7 kontrolnih signala memorijskih elemenata:
  IRWrite kontrolni signal upisa instrukcije u IR registar,
  MemWrite kontrolni signal upisa podatka u Memory jedinicu,
  MemRead kontrolni signal čitanja podatka iz Memory jedinice
  RegWrite kontrolni signal upisa podatka/rezultata ALU u Regitars jedinicu,
  TargetWrite kontrolni signal upisa rezultata ALU u Target registar,
  PCWrite kontrolni signal bezuslovnog upisa adrese sljedeće instrukcije u PC registar,
  PCWriteCond kontrolni signal uslovnog upisa adrese sljedeće instrukcije u PC registar.

- Kontrolne signale ALU control jedinice:
  2-bitni ALUOp signal ($ALUOp_{(1,0)}$).

NAPOMENA 1: Kao što je već razmatrano (pogledaj napomenu 2 iz sekcije 5.5.1), adresa instrukcije koja sljedeća treba da se izvrši može biti PC+4 (kod instrukcija koje se sukcesivno izvršavaju), ciljna adresa grananja (nakon izvršavanja instrukcije uslovnog skoka) ili adresa bezuslovnog skoka (nakon izvršavanja instrukcije bezuslovnog sloka – Jump address na slici 11). Adrese PC+4 i Jump address bezuslovno se upisuju u PC registar, dok se ciljna adresa grananja upisuje u PC registar samo ukoliko je zadovoljen uslov grananja (ukoliko je Zero=1). Drugim riječima, potrebno je predvidjeti kontrolu bezuslovnog upisa, ali i kontrolu uslovnog upisa adrese sljedeće instrukcije u PC registar. U tom cilju, *PCWrite* kontrolni signal upotrebljava se za kontrolu bezuslovnog skoka u PC registar, a *PCWriteCond* (zajedno sa signalom Zero) za kontrolu uslovnog upisa u PC registar.

NAPOMENA 2: Primijetimo suštinsku razliku izmedju kontrolnih signala multipleksora i kontrolnih signala memorijskih elemenata. Obje vrijednosti (1 i/ili 0) kontrolnih signala multipleksora proizvode odredjenu akciju, a samo jedinična vrijednost kontrolnog signala memorijskog elementa proizvodi akciju zahtijevanu od odgovarajućeg elementa. Što pod time podrazumijevamo?

- Što se tiče kontrolnih signala multipleksora, odgovor ćemo dati posmatrajući kontrolni signal *IorD* multipleksora na Read address ulazu Memory jedinice. Jasno je da svaka linija u sistemu, time i linija koja odgovara *IorD* kontrolnom signalu, mora da "nosi" vrijednost ili logičke 0 ili logičke 1. Kada je *IorD*=0, na Read address ulaz Memory jedinice dovodi se sadržaj PC registra, a kada je *IorD*=1, na Read address ulaz Memory jedinice dovodi se adresa izračunata

od strane ALU. Dakle, i vrijednost 0 i vrijednost 1 kontrolnog *IorD* signala proizvode odredjenu akciju (dovodjenje sadržaja PC registra ili adrese izračunate od strane ALU na Read address ulaz Memory jedinice) i strogo se mora voditi računa, prilikom upotrebe multipleksora, koja će se vrijednost kontrolnog signala postaviti na njegove selekcione ulaz(e),

– Što se tiče kontrolnih signala memorijskih elemenata, odgovor ćemo dati posmatrajući *MemWrite* kontrolni signal Memory jedinice. Samo 1-na vrijednost *MemWrite* kontrolnog signala proizvodi akciju (upis u Memory jedinicu), dok 0-a vrijednost ovog signala onemogućava upis u Memory jedinicu (ne proizvodi akciju, već onemogućava akciju upisa). Suštinski, kontrolni bitovi memorijskih elemenata proizvode akciju (upis u memorijski element ili čitanje iz elementa, ali čitanje samo u slučaju Memory jedinice) samo kada uzmu vrijednost 1. Za vrijednost 0 kontrolnog signala, odgovarajuća akcija se onemogućava. Stoga, zbog većeg broja kontrolnih signala koje generiše glavna kontrolna jedinica multitaktne implementacije, samo se navodi naziv kontrolnih signala memorijskih elemenata koji treba da uzmu vrijednost 1, bez označavanja =1 (samim njihovim navodjenjem pretpostavlja se njihova 1-na vrijednost). Ukoliko kontrolni signal memorijskog elemnta treba da uzme vrijednost 0, naziv ovog signala jednostavno se ne navodi.

5.6.1 Postavljanje kontrolnih signala neophodnih za relizaciju I koraka izvršavanja (uzimanje/čitanje instrukcije i inkrementiranje sadržaja PC registra)

U I koraku izvršavanja instrukcija (pogledaj tabelu 2 iz sekcije 5.2), uzima se instrukcija iz Memory jedinice i upisuje se u IR registar i, u paraleli, inkrementira se sadržaj PC registra i izračunato PC+4 upisuje se nazad u PC registar, odnosno:

$$IR \leftarrow Mem[PC], \tag{4}$$

$$PC = PC + 4. \tag{5}$$

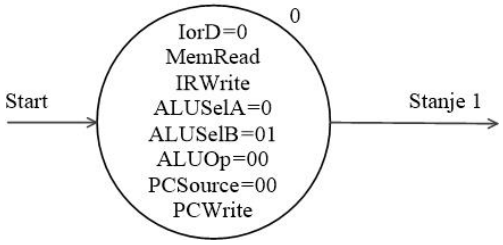
Postavimo najprije kontrolne signale neophodne za implementaciju akcije (4). U tom cilju, neophodno je obezbijediti da sadržaj PC registra adresira Memory jedinicu u svrhu čitanja instrukcije sačuvane u njoj, tj. potrebno je dovesti sadržaj PC registra na Read address ulaz Memory jedinice (*IorD*=0), omogućiti čitanje iz Memory jedinice (*MemRead*=1, uz napomenu da će u dijagramu stanja biti zapisivano samo *MemRead*, pošto je riječ o kontrolnom signalu memorijskog elementa – pogledaj napomenu 2 iz poglavlja 5.6), te obezbijediti upis pročitanje instrukcije u IR registar (*IRWrite*).

Postavimo sada kontrolne signale neophodne za implementaciju akcije (5). Pošto je riječ o izračunavanju, za implementaciju (5) upotrebljava se ALU. U tom cilju, na I ulaz ALU potrebno je dovesti sadržaj PC registra (u ovu svrhu, potrebno je postaviti *ALUSelA*=0), a na II ulaz ALU – konstantu 4 (*ALUSelB*=01). ALU treba da obavi operaciju sabiranja (ALUOp=00). Postavljenim kontrolnim signalima, obezbijedjeno je izračunavanje PC+4. Potrebno je obezbijediti još da se izračunato PC+4 upiše nazad u PC registar. Izračunato PC+4 najprije treba dovesti na ulaz PC registra (*PCSource*=00), te potom obezbijediti bezuslovni upis u PC registar (*PCWrite*, gdje se ne zapisuje =1, jer je riječ o kontrolnom signalu memorijskog elementa – pogledaj napomenu 2 iz sekcije 5.6).

Postavljeni kontrolni signali formiraju početno stanje (stanje 0), slika 12, kojim se implementira I korak izvršavanja instrukcija (pogledaj tabelu 2 iz poglavlja 5.2). Ovo stanje/korak obično se u literaturi naziva *donošenjem instrukcije* (eng. *Instruction fetch* (*IF*)).

NAPOMENA 1: Svi kontrolni signali koji nijesu postavljeni/zapisani u odgovarajućem stanju, nijesu potrebni za implementaciju tog stanja, odnosno pretpostavlja se da uzimaju vrijednost 0. Ovo je posebno važno za kontrolne signale memorijskih elemenata, jer se njihovom vrijednosti 0 spriječava upis u odgovarajuće memorijske elemente, odnosno onemogućiti promjena njegovog sadržaja. Vrijednost 0 kontrolnih signala multipleksora uzrokovat će odredjenu akciju (pogledaj napomenu 2 iz poglavlja 5.6), ali rezultat te akcije neće moći da izmijeni sadržaje memorijskih elemenata.

NAPOMENA 2: PC+4, izračunato od strane ALU prilikom implementacije akcije (5), dolazi na ulaze više elemenata: (i) na ulaz 0 multipleksora koji se nalazi na ulazu PC registra, (ii) na ulaz Target registra, (iii) na ulaz 0 multipleksora koji se nalazi na Write data ulazu Registers jedinice, (iv) na Write



Slika 12. Početno stanje (stanje 0) mašine (sekvencijalnog kola) sa konačnim brojem stanja za implementaciju glavne kontrolne jedinice multitaktne arhitekture sa slike 11.

address ulaz Memory jedinice i (v) na ulaz 1 multipleksora na Read address ulazu Memory jedinice. Medjutim, PC+4 treba da se upiše samo u PC registar, odnosno potreban je na ulazu (i), a ne smije se dopuštiti da uzrokuje štetu dovodjenjem na ulaze (ii)–(v). U cilju upisa PC+4 u PC registar, u stanju 0 postavljeni su kontrolni signali *PCSource*=00 i *PCWrite*. Istovremeno, nepostavljanjem kontrolnih signala *TargetWrite*, *RegWrite*, *MemWrite*, pretpostavlja se da ovi kontrolni signali uzimaju vrijednost 0 (vidi napomenu 1 iz ove sekcije), čime je spriječen upis rezultata PC+4 u Target registar i u odredjeni registar Registers jedinice, kao i upis podatata u Memory jedinicu u lokaciju označenu sa PC+4, odnosno onemogućeno je neodgovarajući uticaj rezultata PC+4. Uz to, u stanju 0 postavljen je signal *IorD*=0, čime je onemogućeno adresiranje Memory jedinice (na Read address ulazu) sa PC+4.

NAPOMENA 3: Postavljanjem kontrolnih signala ALUOp=00, obavlja se bezuslovno sabiranje operanada dovedenih na ulaze ALU (pogledaj tabele 6 i 7 iz sekcije 5.4.1). Ista operacija mogla bi se izvršiti postavljanjem signala ALUOp=10, ali uslovno – operaciju bi tada odredjivalo funct polje mašinskog koda instrukcije (funct=Instruction [5–0]=$32_{(10)}$ za sabiranje – pogledaj tabelu 7 iz sekcije 5.4.1). Ipak, nema razloga funkcionisanje ALU prepuštati funct polju, ako ono može biti odredjeno postavljanjem samo ALUOp bitova. Na koncu, u ovom trenutku (izvršavanja I koraka, kome odgovara stanje 0), ne raspolažemo funct poljem, a takodje sve instrukcije ne raspolažu funct poljem. Drugim riječima, u ovom stanju, operacija sabiranja ALU zadaje se kontrolnom signalima ALUOp=00.

NAPOMENA 4: Paralelno izvršavanje akcija (4) i (5) omogućeno je činjenicom da se za izvršavanje ovih akcija upotrebljavaju različiti funkcionalni elementi. Tokom izvršavanja akcije (4) pristupa se Memory jedinici u cilju čitanja instrukcije, a u cilju izvršavanja akcije (5) upotrebljava se ALU za izračunavanje PC+4. Primijetimo da se ove akcije ne bi mogle paralelno izvršavati ukoliko bi jedna od njih upotrebljavala rezultat one druge akcije, što ovdje nije slučaj.

5.6.2 Postavljanje kontrolnih signala neophodnih za relizaciju II koraka izvršavanja (dekodiranje instrukcije i izračunavanje ciljne adrese grananja)

U II koraku izvršavanja (pogledaj tabelu 2 iz sekcije 5.2), dekodira se instrukcija sačuvana u IR registru (u I koraku), pristupa se sadržajima registara koji su označeni pojedinim poljima instrukcije i, u paraleli, izračunava se ciljna adresa grananja i, nakon izračunavanja, upisuje se u registar Target,

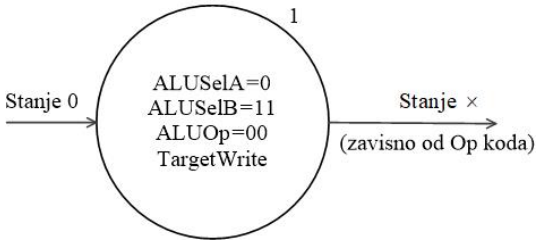
$$\text{pristup/čitanje Reg(IR [25–21])}, \tag{6}$$

$$\text{pristup/čitanje Reg(IR [20–16])}, \tag{7}$$

$$\text{Target=PC+sign\_extend(IR [15–0])<<2}, \tag{8}$$

gdje se, nakon I taktnog interala/koraka, u PC registru nalazi PC+4 (vidji napomenu 1 u sekciji 5.5.1).

Dekodiranje instrukcije podrazumijeva čitanje polja instrukcije sačuvane u IR registru, IR [31–0]≡Instruction [31–0], odnosno čitanje odgovarajućih bitova IR registra, gdje su op=Op[5–0]=IR [31–26], rs=IR [25–21], rt=IR [20–16], rd=IR [15–11], shamt=IR [10–6], address/immediate/offset =IR [15–0], funct=IR [5–0], i address polje instrukcija bezuslovnog skoka address=IR [25–0] (pogledaj zabilješku iz poglavlja 5.2). Nakon dekodiranja, poljima rs=IR [25–21] i rt=IR [20–16] označavaju se registri Registers jedinice u cilju čitanja njihovog sadržaja (akcije (6) i (7)). Pošto IR registar i Registars jedinica ne posjeduju kontrolne signale čitanja, čitanje sadržaja označenih registara može se obaviti u proizvoljnom trenutku i nije potrebno postavljati kontrolne signale u tom cilju.



Slika 13. Stanje 1 mašine (sekvencijalnog kola) sa konačnim brojem stanja za implementaciju glavne kontrolne jedinice multitaktne arhitekture sa slike 11.

Postavimo sada kontrolne signale neophodne za implementaciju akcije (8). Pošto je riječ o izračunavanju, za implementaciju (8) upotrebljava se ALU. U tom cilju, na I ulaz ALU potrebno je dovesti sadržaj PC registra (*ALUSelA*=0), a na II ulaz ALU – 16-bitno address/immediate/offset polje produženo, do 32-bitne dužine, kopiranje znaka broja i shift-ovano u lijevu stranu za 2 mjesta (*ALUSelB*=11). ALU treba da obavi operaciju sabiranja (ALUOp=00). Postavljenim kontrolnim signalima, obezbijedjeno je izračunavanje PC+sign_extend(IR [15–0])<<2. Primijetimo da je, na koncu, potrebno obezbijediti još upis izračunatog rezultata u Target registar (*TargetWrite*).

Postavljeni kontrolni signali formiraju stanje 1, slika 13, kojim se implementira II korak izvršavanja instrukcija (pogledaj tabelu 2 iz poglavlja 5.2). Ovo stanje/korak obično se u literaturi naziva *dekodiranjem instrukcije*/*donošenjem sadržaja registara* (eng. *Instruction decode*/*Register fetch* (*ID*)).

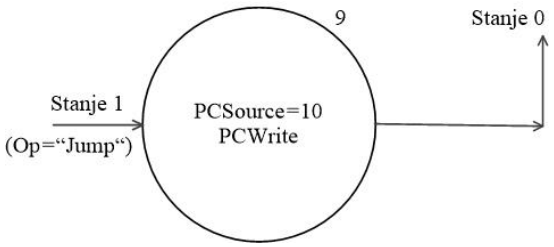
NAPOMENA 1: Postavljanjem kontrolnih signala ALUOp=00, obavlja se bezuslovno sabiranje operanada dovedenih na ulaze ALU (pogledaj tabelu 7 iz sekcije 5.4.1).

NAPOMENA 2: Paralelno dekodiranje instrukcije i izvršavanje akcije (8) omogućeno je činjenicom da se za izvršavanje ovih akcija upotrebljavaju različiti funkcionalni elementi. Tokom dekodiranja instrukcije pristupa se IR registru i Registers jedinici u cilju njihovog čitanja, a u cilju izvršavanja akcije (8) upotrebljava se ALU za zahtijevano izračunavanje. Primijetimo da nijedna od ove dvije akcije ne upotrebljavaju rezultat one druge akcije, tako da se mogu izvršavati u paraleli.

5.6.3 Kompletiranje instrukcije bezuslovnog skoka *j*

Instrukcija bezuslovnog skoka *j*, za svoje kompletiranje, zahtijeva izvršavanje još jednog koraka/taktnog intervala (pogledaj tabelu 2 iz poglavlja 5.2). U tom koraku/taktnom intervalu, adresu bezuslovnog skoka (Jump address na slici 11), koja se ne izračunava, već hardverski kreira (za pojašnjenje pogledaj napomenu 4 iz sekcije 5.5.2), potrebno je bezuslovno upisati u PC registar. U tom cilju, neophodno je dovesti (kroz multipleksor koji se nalazi na ulazu PC registra) Jump address na ulaz PC registra (*PCSource*=10), te potom obezbijediti njen bezuslovan upis u PC registar (*PCWrite*).

Postavljeni kontrolni signali formiraju stanje 9, slika 14, kojim se kompletira izvršavanja instrukcije *j* (pogledaj tabelu 2 iz poglavlja 5.2). Ovo stanje/korak obično se u literaturi naziva *kompletiranjem instrukcije bezuslovnog skoka* (eng. *Jump completion* (*JC*)).



Slika 14. Mašina sa konačnim brojem stanja za implementaciju instrukcije bezuslovnog skoka *j*.

Nakon kompletiranja instrukcije bezuslovnog skoka *j*, izvršavanje se vraća u stanje 0 u cilju uzimanja/čitanja sljedeće instrukcije iz Memory jedinice i njenog izvršavanja u sljedećih 3–5 koraka/taktnih intervala.

5.6.4 Kompletiranje instrukcije uslovnog skoka/grananja *beq*

Instrukcija uslovnog skoka/grananja *beq*, za svoje kompletiranje, takodje zahtijeva izvršavanje još jednog koraka/taktnog intervala (pogledaj tabelu 2 iz poglavlja 5.2). U tom koraku/taktnom intervalu, potrebno je:

(i) Najprije ispitati (upotrebom ALU) uslov grananja (jednakost sadržaja registara označenih poljima rs=Instruction [25–21] i rt=Instruction [20–16]), odnosno postaviti Zero izlaz ALU (ukoliko je Zero=1, uslov grananja je zadovoljen),

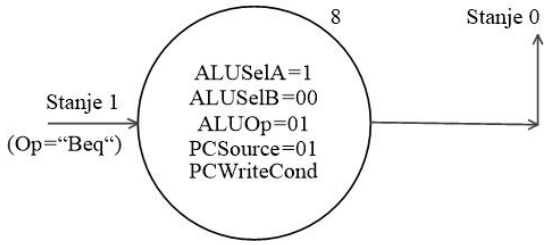
(ii) Upisati ciljnu adresu grananja, sačuvanu u Target registru, u PC registar ukoliko je uslov grananja zadovoljen (ukoliko je Zero=1),

$$PC \leftarrow Target, \text{ ako je } Zero=1. \quad (9)$$

U cilju ispitivanja uslova grananja (postavljanja Zero izlaza ALU), na I ulaz ALU potrebno je dovesti sadržaj registra koji se adresira rs=Instruction [25–21] poljem instrukcije (*ALUSelA*=1), a na II ulaz ALU – sadržaj registra koji se adresira rt=Instruction [20–16] poljem instrukcije (*ALUSelB*=00), te ALU natjerati da izvrši operaciju oduzimanja (ALUOp=01 – bitovi ALUOp uzimaju ove vrijednosti u slučaju implementacije instrukcije *beq* – pogledaj tabele 6 i 7 iz sekcije 5.4.1).

U cilju izvršavanja akcije (9), ciljnu adresu uslovnog skoka/grananja, izračunatu i sačuvanu u Target registru u II koraku/taktnom intervalu (stanje 1), potrebno je dovesti na ulaz PC registra, ali upisati u PC registar samo ukoliko je zadovoljen uslog grananja (Zero=1). Ciljna adresa grananja (sadržaj Target registra) dovodi se na ulaz PC registra kroz multipleksor koji se nalazi na ulazu PC registra (*PCSource*=01), a njen uslovni upis u PC registar obezbjedjuje se set-ovanjem/postavljanjem *PCWriteCond* kontrolnog signala.

Postavljeni kontrolni signali formiraju stanje 8, slika 15, kojim se kompletira izvršavanja instrukcije grananja *beq* (pogledaj tabelu 2 iz poglavlja 5.2). Ovo stanje/korak obično se u literaturi naziva *kompletiranjem instrukcije grananja* (eng. *Branch completion* (*BC*)).



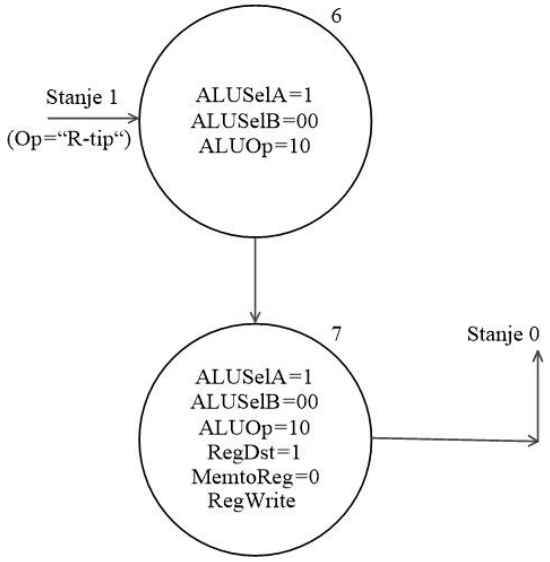
Slika 15. Mašina sa konačnim brojem stanja za implementaciju instrukcije *beq*.

NAPOMENA 1: Kontrolni signal *PCWriteCond* zajedno sa signalom Zero učestvuje u formiranju kontrolnog signala kojim se reguliše uslovni upis u PC registar. U tom cilju, ova 2 signala (*PCWriteCond* i Zero) dovode se na ulaze I kola, a na izlazu I kola generiše se signal koji kontroliše uslovni upis u PC registar. Na taj način, kotrolni signal *PCWriteCond*, generisan od strane glavne kontrolne jedinice, kontroliše upis u PC registar, ali uslovno – samo kada je Zero=1 (kada je zadovoljen uslov grananja zadat *beq* instrukcijom).

NAPOMENA 2: Kontrolni signal *PCWriteCond* upotrebljava se za kontrolu uslovnog upisa u PC registar i samo prilikom implementacije *beq* instrukcije (*beq* je jedina instrukcija uslovnog skoka/grananja koja će biti implementirana u ovom materijalu), dok se kontrolni signal *PCWrite* upotrebljava u cilju kontrole beuslovnog upisa u PC registar.

5.6.5 Kompletiranje instrukcija R-tipa

Za kompletiranje instrukcije R-tipa potrebno je izračunavanje operacije koja se zahtijeva instrukcijom i to nad operandima koji se nalaze u registrima označenim rs=IR [25–21] i rt=IR [20–16] poljima mašinskog koda instrukcije (pogledaj tabelu 2 iz poglavlja 5.2), te upisivanje izračunatog rezultata u registar označen rd=Instruction [15–11] poljem instrukcije,



Slika 16. Mašina sa konačnim brojem stanja za implementaciju instrukcija R-tipa.

$$\text{Reg(IR [15–11])} = \text{Reg(IR [25–21])} \ \underline{\text{op}} \ \text{Reg(IR [20–16])} \qquad (10)$$

gdje je op operacija zahtijevana izvršavanom instrukcijom R-tipa (*add*, *sub*, *and*, *or*, *slt*), a sadržaji registara Reg(IR [25–21]) i Reg(IR [20–16]) pročitani su u prethodnom (II) koraku/taktnom intervalu.

U cilju optimizacije vremena izvršavanja (umanjenja gubitaka u vremenu izvršavanja), akcija (10) uzima 2 koraka/taktna intervala (pogledaj tabelu 2 iz poglavlja 5.2). U prvom od ovih koraka/taktnih intervala izračunava se zahtijevana operacija, dok se u drugom koraku rezultat izračunat u prvom koraku/taktnom intervalu upisuje u odredišni registar,

1. U cilju izračunavanja zahtijevane operacije, na I ulaz ALU potrebno je dovesti sadržaj registra Registers jedinice označenog poljem rs=IR [25–21] (*ALUSelA*=1), a na II ulaz ALU – sadržaj registra Registers jedinice označenog poljem rt=IR [20–16] (*ALUSelB*=00), te, pošto je riječ o implementaciji instrukcija R-tipa, postaviti ALUOp=10 (pogledaj tabele 6 i 7 iz poglavlja 5.4.1). Postavljanje ovih kontrolnih signala odgovara stanju 6 (u literaturi nazivanim *izvršnim stanjem*, eng. *Execution* (*Ex*)), neophodnim za izvršavanje instrukcija R-tipa, slika 16.

   NAPOMENA 1: Postavljanjem kontrolnih bitova ALUOp=10, kontrola funkcionisanja ALU suštinski prepušta se funct=IR [5–0] polju instrukcija R-tipa. Drugim riječima, za ALUOp=10, funct polje instrukcija R-tipa definiše operaciju koja se instrukcijom zahtijeva (pogledaj tabelu 7 iz poglavlja 5.4.1).

2. U cilju upisivanja izračunatog rezultata u odredišni registar (registar označen rd=IR [15–11] poljem instrukcije) potrebno je obezbijediti sljedeće:

   2.1. Nepromjenljivost rezultata (iz prethodnog koraka/taktnog intervala) na izlazu ALU, zadržavanjem istih kontrolnih signala koji se odnose na funkcionisanje ALU, a postavljeni su u prethodnom koraku, odnosno *ALUSelA*=1, *ALUSelB*=00, ALUOp=10.

   NAPOMENA 2: Rezultat ALU, izračunat u III koraku/taktnom intervalu, nije sačuvan na kraju III koraka/taktnog intervala u nekom memorijskom elementu, a upotrebljava se u IV koraku. Da ovaj rezultat ne bi bio prebrisan (kreiranjem novog rezultata na izlazu ALU), neophodno je obezbjediti nepromjenljivost funkcionisanja ALU u III i u IV koraku. To se postiže zaržavanjem iste vrijednosti kontrolnih signala koji odredjuju funkcionisanje ALU (pogledaj paragraf koji se nalazi nakon napomene 2 u poglavlju 5.5.2).

   2.2. Dovesti rezultat sa izlaza ALU na Write data ulaz Registers jedinice (*MemtoReg*=0), poljem rd=IR [15–11] instrukcije označiti/adresirati registar u koji je potrebno upisati dovedeni rezultat (*RegDst*=1), te, na koncu, dozvoliti upis dovedenog rezultata u označeni registar (*RegWrite*).

Postavljeni kontrolni signali formiraju stanje 7, slika 16, kojim se kompletira izvršavanje instrukcija R-tipa (pogledaj tabelu 2 iz poglavlja 5.2). Ovo stanje/korak obično u literaturi se naziva *kompletiranjem instrukcija R-tipa* (eng. *R-type completion* (*RC*)).

NAPOMENA 3: Izvršavanjem akcije (10) u 2 koraka/taktna intervala, svaka od upotrijebljenih funkcionalnih jedinica upotrijebljava se tačno jedan put i to u odgovarajućem koraku,

- U prvom od ova 2 koraka, upotrebljava se ALU za izvršavanje zahtijevane operacije,
- U drugom – Registers jedinica za upis izračunatog rezultata u odgovarajući registar.

Uz to, taktni interval, ranije odredjen vremenom pristupa Memory jedinici (pogledaj napomenu 1 u poglavlju 5.5), dovoljno je dug za izvršavanje operacije ALU, ali i za upis razultata u odgovarajući registar Registers jedinice. Naime, konstatovano je tada da je vrijeme pristupa Memory jedinice zahtjevnije od vremena zahtijevanih od ostalih upotrijebljenih funkcionononalnih jedinica, pa i od vremena potrebnog ALU za izračunavanje i od vremena pristupa Registers jedinici. Medjutim, ukoliko bi se postupilo suprotno, odnosno ukoliko bi se akcija (10) izvršavala u jednom koraku/taktnom intervalu, upisivanje u Registers jedinicu moralo bi sačekati završetak izračunavanja u ALU (ove dvije operacije izvršavale bi se redno, a ne u paraleli kao što je bio slučaj u I i u II koraku – pogledaj sekcije 5.6.1 i 5.6.2 i stanja 0 i 1). Shodno tome, dužina taktnog intervala u ovom slučaju morala bi odgovarati sumi vremena neophodnog za izvršavanje operacije ALU i vremena neophodnog za pristup Registers jedninici. Dužina tako dizajniranog taktnog intervala bila bi veća od intervala odredjenog vremenom pristupa Memory jedinici (pogledaj primjer iz poglavlja 5.5), što bi impliciralo gubitke u vremenu izvršavanja koja se odnosi na ostale korake, a time bi impliciralo i gubitke u ukupnom vremenu izvršavanja pojedinačnih instrukcija. Stoga, taktni interval dizajnirane implementacije ostaje odredjen vremenom pristupa Memory jedinici, a akcija (10) izvršava se u 2 koraka/taktna intervala (III i IV).

NAPOMENA 4: Nakon pojašnjenja iz prethodne napomene 3, nameće se pitanje: kako se u I koraku izvršavanja instrukcija (stanje 0) može pristupiti Memory jedinici (u clju čitanja) i pročitana instrukcija potom upisati u IR registar u toku jednog te istog taktnog intervala, a za izračunavanje ALU i upisivanje dobijenog rezultata u odredjeni registar Registers jedinice potrebna su 2 taktna intervala? Ili, kako se u II koraku izvršavanja instrukcija (stanje 1) može izračunati ciljna adresa grananja u ALU i izračunata adresa potom upisati u Target registar u toku jednog te istog taktnog intervala, a za izračunavanje ALU i upisivanje dobijenog rezultata u odredjeni registar Registers jedinice potrebna su 2 taktna intervala? Odgovor na ovo pitanje je jednostavan. Registers jedinica predstavlja skup svih procesorskih registara ($0–$31), te se pristup odredjenom registru iz ovog skupa obezbjedjuje tek nakon njegovog adresiranja, odnosno nakon dekodiranja adresnih bitova dovedenih na Write register adresni ulaz Registers jedinice. Dekodiranje adrese će zahtijevati odredjeno vrijeme koje je, zajedno sa vremenom pristupa registru, značajno veće od vremena zahtijevanog za pristup pojedninačnom memorijskom elementu, kao što su IR registar ili Target registar.

5.6.6 Kompletiranje memory-reference instrukcija (*lw* i *sw*)

Kompletiranje memory-reference instrukcija (*lw* i *sw*) započinje izračunavanjem adrese memorijske lokacije čiji sadržaj će, do kraja izvršavanja instrukcija, biti razmijenjen sa sadržajem registra koji je označen rt=IR [20–16] poljem instrukcije. U cilju izračunavanja adrese memorijske lokacije, sabira se (od strane ALU) sadržaj registra, koji je označen rs=IR [25–21] poljem instrukcije, i sadržaj address-nog polja mašinskog koda instrukcija *lw*/*sw*, IR [15–0], produžen, kopiranjem znaka, do 32-bitne dužine (pogledaj tabelu 2 iz poglavlja 5.2),

$$\text{ALUOut}=\text{Reg(IR [25–21])} + \text{sign\_extend(IR [15–0])}. \quad (11)$$

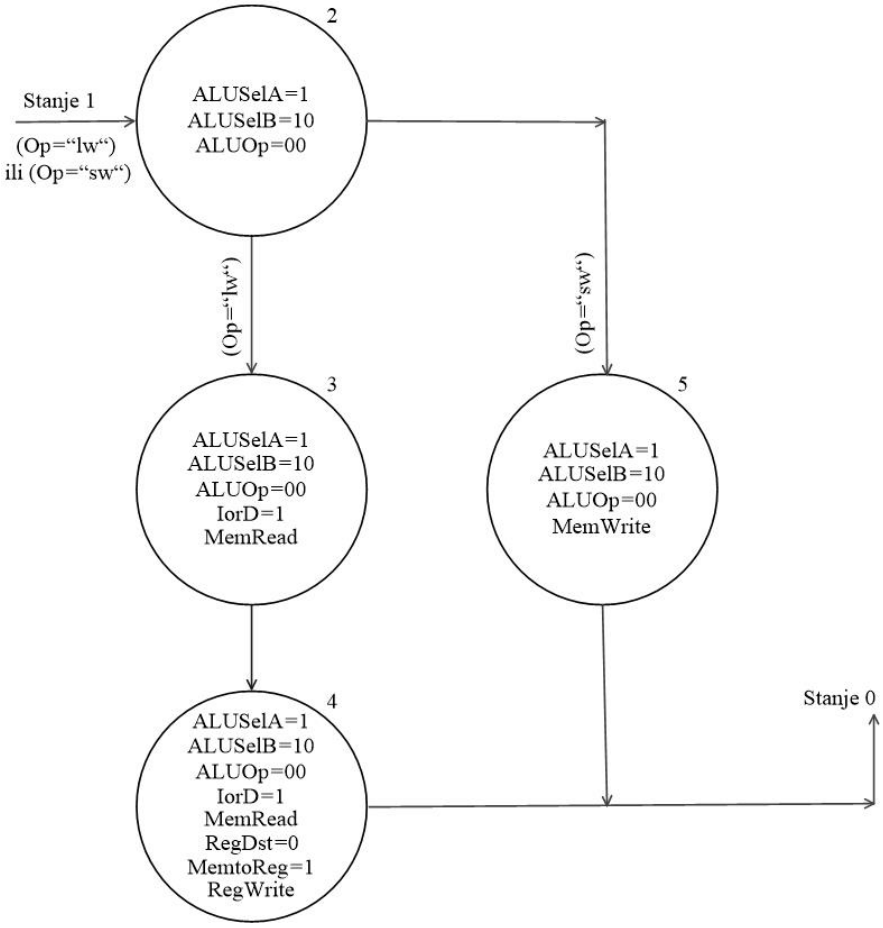
Prilikom implementacije instrukcije *lw*, sadržaj adresirane lokacije, Mem[ALUOut], potrebno je premjestiti u registar označen rt=IR [20–16] poljem instrukcije,

$$\text{Reg(IR [20–16])} \leftarrow \text{Mem[ALUOut]}, \quad (12)$$

dok je, u slučaju instrukcije *sw*, ovaj transfer podataka potrebno obaviti u suprotnom smjeru,

$$\text{Mem[ALUOut]} \leftarrow \text{Reg(IR [20–16])}, \quad (13)$$

gdje je, u slučaju instrukcije *sw*, Reg(IR [20–16]) pročitan u II koraku/taktnom intervalu.



Slika 17. Mašina sa konačnim brojem stanja za impletaciju memory-reference instrukcija (*lw* i *sw*).

Implementacije akcije (11) izvršava se u jednom (III) koraku. Ovaj korak je zajednički za *lw* i *sw* instrukcije (pogledaj tabelu 2 u poglavlju 5.2). U cilju njegove realizacije, na I ulaz ALU potrebno je dovesti sadržaj registra koji je označen poljem rs=IR [25–21] instrukcije (*ALUSelA*=1), na II ulaz ALU – 16-bitno address pilje instrukcije produženo, kopiranjem njegovog znaka, do 32-bitne dužine (*ALUSelB*=10), te omogućiti ALU da obavi operaciju sabiranja (ALUOp=00 – na koncu, implementacija instrukcija *lw*/*sw* zahtijeva postavljanje ALUOp=00 – pogledaj tabelu 6 iz sekcije 5.4.1). Ovim kontrolnim signalima formira se stanje 2 kojim se implementira III korak (u literaturi obično nazivan *izračunavanje memorijeske adrese* – eng. *Memory address computation* (*MC*)). Ovaj korak je neophodno obaviti prilikom izvršavanja instrukcija *lw* i *sw*, kao što je prikazano na slici 17.

Akcija (12), koja se obavlja tokom implementacije instrukcije *lw*, za svoje izvršavanje zahtijeva 2 koraka/taktna intervala (IV i V), dok akcija (13), koja se obavlja tokom implementacije instrukcije *sw*, za svoje izvršavanje zahtijeva 1 korak/taktni interval (IV korak/taktni interval).

NAPOMENA 1: Akcija (12) zahtijeva pristup Memory jedinici u cilju čitanja podatka iz nje, te pristup Registers jedinici u cilju upisa pročitanog podatka u odgovarajući registar. Shodno tome, akcija (12) zahtijeva vrijeme koje odgovara zbiru vremena pristupa Memory jedinici i vremena pristupa Registers jedinici. Naime, navedeni pojedinačni pristupi izvršavaju se redno, tako da ukupno vrijeme izvršavanja akcije (12) odgovara zbiru pojedinačnih vremena pristupa svake od jedinica (Memory i Registers). Medjutim, sumarno vrijeme pristupa Memory jedinici i pristupa Registers jedinici prevazilazi dužinu taktnog intervala odredjenu vremenom pristupa Memory jedinice i stoga, slijedeći principe navedene u razmatranjima u napomeni 3 iz sekcije 5.6.5, akcija (12) obavlja se u 2 taktna intervala (IV i V):

– U IV koraku/taktnom intervalu, vrši se pristup Memory jedinici u cilju čitanja podatka sa adresitrane memorijske lokacije (Mem[ALUOut]),

– U V koraku/taktnom intervalu, vrši se upis pročitanog podatka u registar označen poljem rt=IR [20–16] instrukcije *lw*.

NAPOMENA 2: Akcija (13) takodje zahtijeva serijski (redni) pristup, najprije Registers jedinici (u cilju čitanja podatka iz odgovarajućeg registra), a nakon toga Memory jedinici (u cilju upisa pročitanog podatka u adresiranu memorijsku lokaciju). Medjutim, potrebno je primijetiti da je podatak iz odgovarajućeg registra Registers jedinice pročitan prilikom dekodiranja instrukcije (u II koraku – stanje 1 – pogledaj sekciju 5.6.2), tako da je u toku IV koraka/taktnog intervala potrebno samo pročitani podatak upisati u adresiranu memorijsku lokaciju (Mem[ALUOut]).

NAPOMENA 3: Rezultat ALU (memorijska adresa), izračunat u III koraku/taktnom intervalu, upotrebljava se do kraja izvršavanja instrukcija *lw*/*sw*, a nije sačuvan (od prebrisavanja) u nekom memorijskom elementu na kraju III koraka/taktnog intervala, tako da se mora obezbijediti njegova nepromjenljivost do kraja izvršavanja obje instrukcije (*lw* i *sw*) – za detalje pogledaj napomenu 2 iz sekcije 5.6.5. U tom cilju, do kraja izvršavanja instrukcija *lw* (u IV i V koraku/taktnom intervalu) i *sw* (u IV koraku/taktnom intervalu), zadržavaju se iste vrijednost kontronih signala koji odredjuju funkcionisanje ALU u III koraku/taktnom intervalu (*ALUSelA*=1, *ALUSelB*=10, ALUOp=00).

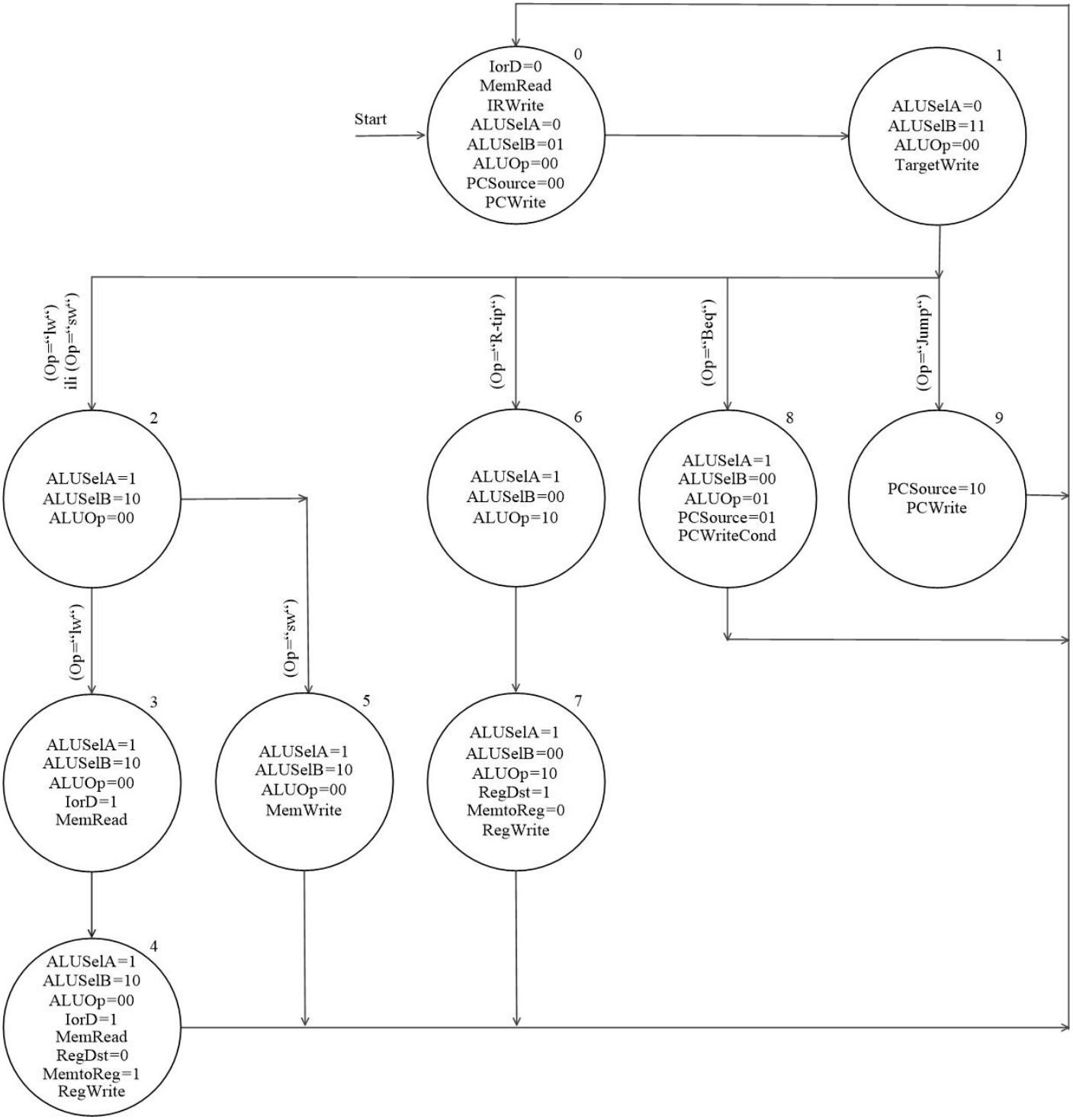
Razmotrimo najprije implementaciju instrukcije *lw*. Za svoje kompletiranje, ova instrukcija zahtijeva 2 dodatna koraka/taktna intervala (IV i V):

1. U IV koraku/taktnom intervalu, adresa izračunata (od strane ALU) u III koraku upotrebljava se za pristup Memory jedinici (*IorD*=1) u clju čitanja podatka (*MemRead*) sa adresirane memorijske lokacije. Ovim kontrolnim signalima, zajedno sa kontrolnim signalima koji obezbjedjuju nepromjenljivost funkcionisanja ALU u odnosu na III korak/taktni interval (*ALUSelA*=1, *ALUSelB*=10, ALUOp=00 – pogledaj napomenu 3 iz ove sekcije), definisano je stanje 3 kojim se implementira IV korak izvršavanja instrukcije *lw* (u literaturi obično nazivan *pristupanje memoriji* – eng. *Memory accress* (*MA*)), slika 17.
2. Podatak, pročitan iz Memory jedinice u IV koraku, nije upisan u neki memorijski elemenat za privremeno smještanje podataka na kraju ovog koraka, tako da se, iz istih razloga navedenih u napomeni 3 u ovoj sekciji i u napomeni 2 u sekciji 5.6.5, mora obezbijediti njegovo neprebrisavanje. Ono se obezbjedjuje zadržavanjem istih kontrolnih signala koji se odnose na čitanje podatka iz Memory jedinice, a koji su postavljeni u IV koraku/taktnom intervalu (*IorD*=1, *MemRead*). Nakon toga, podatak pročitan iz Memory jedinice, potrebno je dovesti na Write data ulaz Registers jedinice (*MemtoReg*=1), poljem rt=IR [20–16] označiti, na Write register ulazu Registers jedinice, registar u koji podatak treba upisati (*RegDst*=0), te, na koncu, obezbijediti upis u označeni registar Registers jedinicu (*RegWrite*). Ovim kontrolnim signalima, zajedno sa kontrolnim signalima koji obezbjedjuju nepromjenljivost funkcionisanja ALU u odnosu na III i IV korak/taktni interval (*ALUSelA*=1, *ALUSelB*=10, ALUOp=00 – pogledaj napomenu 3 iz ove sekcije), definisano je stanje 4 (u literaturi obično nazivano *upisivanje nazad* – eng. *Write back* (*WB*)), kojim se kompletira izvršavanje instrukcije *lw*, slika 17.

Razmotrimo, na koncu, implementaciju instrukcije *sw*. Za svoje kompletiranje, ova instrukcija zahtijeva 1 dodatni (IV) korak/taktni interval. U ovom koraku/taktnom intervalu, podatak koji je pročitan iz registra označenog rt=IR [20–16] poljem instrukcije (u II koraku/taktnom intervalu – tokom dekodiranja instrukcije) i koji je doveden na Write data ulaz Memory jedinice, potrebno je upisati (*MemWrite*) u memorijsku lokaciju adresiranu rezultatom ALU dovedenim na Write address ulaz Memory jedinice. Ovim kontrolnim signalom, zajedno sa kontrolnim signalima koji obezbjedjuju nepromjenljivost funkcionisanja ALU u odnosu na III korak/taktni interval (*ALUSelA*=1, *ALUSelB*=10, ALUOp=00 – pogledaj napomenu 3 iz ove sekcije), definisano je stanje 5 (u literaturi obično nazivano *pristupanje memoriji* – eng. *Memory accress* (*MA*)), kojim se kompletira izvršavanje instrukcije *sw*, slika 17.

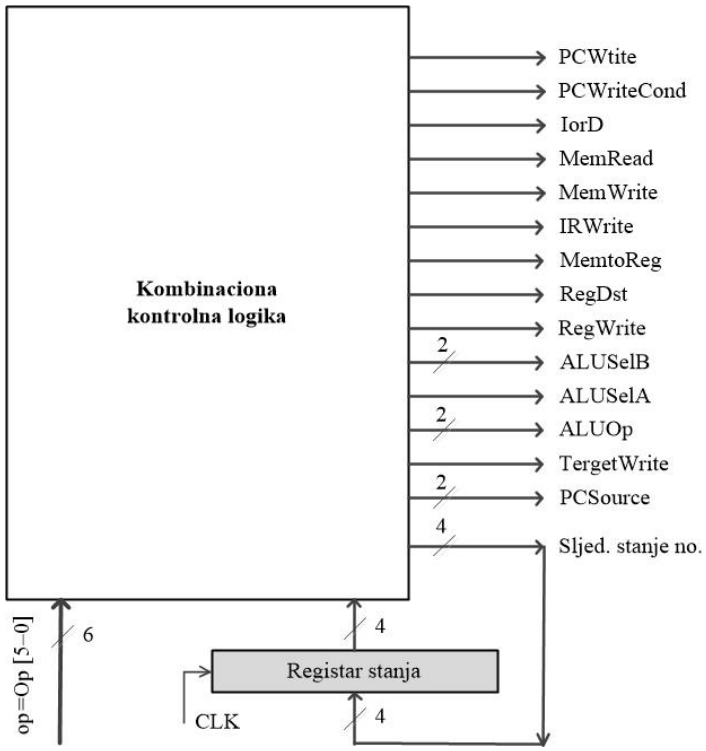
5.6.7 Definisanje glavne kontrolne jedinice multitaktne implementacije

Povezivanjem zajedničkih djelova za izvršavanje svih razmatranih instrukcija, prikazanih na slikama 12 i 13, i djelova za kompletiranje odredjenih tipova instrukcija, prikazanih na slikama 14-17,



Slika 18. Potpuni dijagram mašine sa konačnim brojem stanja kojom se implementira glavna kontrolna jedinica multitaktne arhitekture sa slike 11.

u jedinstvenu cjelinu dobija se potpuna mašina (sekvencijalno kolo) sa konačnim brojem stanja za multitaktnu implementaciju instrukcija R-tipa, *lw*, *sw*, *beq* i *j*. Potpuna mašina sa konačnim brojem stanja prikazana je na slici 18. Ona odgovara Moore-ovom tipu sekvencijalnog kola, pošto se njeni izlazi (kontrolni signali multipleksora i memorijskih elemenata upotrijebljenih u multitaktnoj implementaciji sa slike 11) postavljaju u funkciji stanja u kome se sekvencijalno kolo nalazi.



Slika 19. Logički dizajn glavne kontrolne jedinice multitaktne implementacije sa slike 11.

Strijelama je prikazan smjer prelaza izmedju stanja koji zavisi od implementirane instrukcije (njenog op koda, op=Op [5–0]=IR [31–26]), a oznakama iznad strijela prikazani su uslovi koji moraju biti zadovoljeni da bi se prešlo izmedju 2 stanja koja povezuje odgovarajuća strijela. Ukoliko se prelaz izmedju dva stanja beuslovno obavlja, nije zapisana oznaka iznad stijele koja odgovara tom prelazu.

Na grafičkoj prezentaciji sa slike 18, u svakom pojedinačnom krugu (stanju) navedeni su kontrolni signali koji se u tom stanju postavljaju. Kao što je i ranije navedeno, za kontrolne signale, koji nijesu navedenu u zapisu odredjenog stanja, pretpostavlja se da u tom stanju uzimaju vrijednosti 0 (ovo se posebno odnosi na kontrolne signale memorijskih elemanata, dok i suprotne vrijednosti kontrolnih bitova multipleksora ništa neće izmijeniti u funkcionisanju procesora – pogledaj napomenu 2 u sekciji 5.6).

Mašina sa konačnim brojem stanja, logički prikazana na slici 18, implementira se kombinacionom logikom čiji su ulazi op=IR [31–26] polje implementirane instrukcije i obilježje/broj prethodno izvršenog stanja, a izlazi – kontrolni signali multipleksora i memorijskih elemenata koje treba postaviti u zatečenom stanju i obilježje/broj stanja koje sljedeće treba izvršiti. Stoga, pored kombinacione logike, u implementaciju glavne kontrolne jedinice treba uključiti 4-bitni registar stanja (4-bitni registar, pošto mašina sa slike 18 sadrži ukupno 10 stanja za čije zapisivanje su potrebna 4 bita), kao što je prikazano na slici 19.

NAPOMENA: Mašina sa konačnim brojem stanja jedan je mogući način implementacije glavne kontrolne jedinice multitaktne arhitektire sa slike 11. Drugi mogući (moguće i jednostavniji) način za implementaciju glavne kontrolne jedinice je mikroprogramiranjem. Medjutim, ovo je tema daljeg izučavanja, koje izlazi iz okvira ovog materijala.

5.7 NADOGRADNJA/REDIZAJNIRANJE ARHITEKTURE ZA MULTITAKTNU IMPLEMENTACIJU U CILJU REALIZACIJE IMMEDIATE INSTRUKCIJA

Arhitektura za multitaktni implementaciju, prikazana na slici 11, i njena glavna kontrolna jedinica, čija je logika dizajniranja zasnovana na mašini sa konačnim brojem stanja prikazana na slikama 18 i 19, obezbjedjuju implementaciju instrukcija R-tipa (*add*, *sub*, *and*, *or*, *slt*), *lw*, *sw*, *beq* i *j*. U ovom poglavlju razmatraćemo načine za nadogradnju/redizajniranje arhitekture sa slika 11, 18 i 19 u cilju implementacije najčešće upotrebljavanih immediate instrukcija *addi*, *slti*, *ori* i *andi*. Nadogradnju ćemo razmatrati pojedinačnim dodavanjem svake od navedenih immediate instrukcija. Nakon toga ćemo razmotriti mogućnost jednovremenog uključivanja sve 4 immediate instrukcije.

5.7.1 Prilagodjavanje arhitekture za multitaktnu implementaciju u cilju implementacije *addi* instrukcije

Prije nego predjemo na razmatranje mogućeg prilagodjavanja postojeće arhitekture za multitaktnu implementaciju u cilju uključivanja *addi* instrukcije, podsjetimo se simboličkog koda i mašinskog formata zapisa ove instrukcije. Njen simbolički kod (pogledaj poglavlje 3.6 i tabelu 1 iz poglavlja 5.1) je:

addi $x, $y, C # $x=$y+C

gdje se konstanta (konstantni operand) C čuva unutar instrukcije i zapisuje na njenom kraju, dok $x, x=1,...,31, označava registar u koji se upusuje rezultat sabiranja operanda sadržanog u registru $y, y=0,...,31, sa konstantom C, kao što je zapisano u komentaru gore navedene instrukcije.

Mašinski format zapisa instrukcije *addi* (pogledaj poglavlje 3.6 i tabelu 1 iz poglavlja 5.1) je:

Polje:	op	rs	rt	Immediate
Br. bitova:	6	5	5	16
Sadržaj:	8	y	x	C

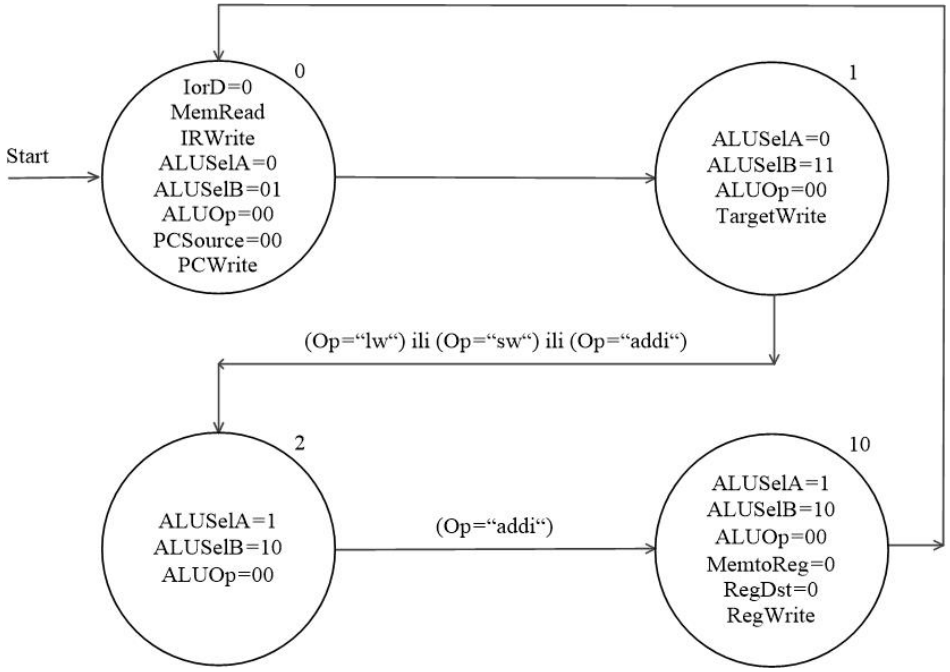
Slika 20. Mašinski format zapisa instrukcije *addi*.

Primijetimo da je, kod instrukcije *addi*, op=Instruction [31–26]=$8_{(10)}$=$001000_{(2)}$=0x8, gdje se oznakom 0x opisuje heksadekadni zapis (odnosno, 0x8≡$8_{(16)}$). U mašinskom formatu zapisa, konstanta C smještena je na najnižim bitovima instrukcije (u 16-bitnom Immediate=Instruction [15–0] polju), u rs=Instruction [25–21] polju instrukcije zapisano je obilježje (broj) registra u kome se nalazi operand ($y u simboličkoj formi), a u rt=Instruction [20–16] polju instrukcije – obilježje odredišnog registra (registar u kome se smješta rezultat sabiranja).

Drugim riječima, nakon uzimanja instrukcije, njenog dekodiranja i čitanja sadržaja registra označenog rs=IR [25–21] poljem instrukcije (u prva 2–zajednička–koraka/taktna intervala – stanja 0 i 1 sa slike 18), u cilju kompletiranja instrukcije *addi* potrebno je implementirati akciju:

Reg(IR [20–16])=Reg(IR [25–21]) + sign_extend(IR [15–0]). (14)

PRIMJEDBA: Taktni interval multitaktne implementacije, razmatrane u ovom materijalu, odredjen je vremenom pristupa Memory jedinice (pogledaj napomenu 1 u poglavlju 5.5). Medjutim, na isti način kao u slučaju instrukcija R-tipa, izvršavanje akcije (14) (i drugih pojedinačnih akcija povezanih sa aritmetičko-logičkim operacijama koje mogu biti zahtijevane od strane instrukcija R-tipa i od strane ostalih immediate instrukcija – pogledaj poglavlje 3.6) zahtijeva izračunavanje u ALU, te nakon toga pristup Registers jedinici u cilju upisa izračunatog rezultata (u slučaju immediate instrukcija, upis se vrši u registar označen rt=IR [20–16] poljem instrukcije). Odnosno, izvršavanje akcije (14) (i svih drugih akcija koje mogu biti zahtijevane instrukcijama R-tipa, sekcija 5.6.5, i immediate instrukcijama) zahtijeva vrijeme koje odgovara vremenu potrebnom za izračunavanje od strane ALU uvećanom za vrijeme pristupa Registers jedinici i koje, shodno tome, prevazilazi dužinu taktnog intervala odredjenog vremenom pristupa Memory jedinice (pogledaj napomene 3 i 4 iz sekcije 5.6.5). Drugim riječima, ukoliko bi dužina taktnog intervala bila dizajnirana tako da odgovara vremenu



Slika 21. Mašina sa konačnim brojem stanja za implemantaciju instrukcije *addi*.

zahtijevanom za izvršavanja akcije (14), ona bi bila veća od intervala odredjenog vremenom pristupa Memory jedinici (pogledaj primjer iz poglavlja 5.5), što bi impliciralo gubitke u vremenu izvršavanja koji se odnose na ostale korake, a time i gubitke u ukupnom vremenu izvršavanja pojedinačnih instrukcija. Stoga, taktni interval dizajnirane implementacije ostaje odredjen vremenom pristupa Memory jedinici, a akcija (14) izvršava se tokom 2 koraka/taktna intervala (III i IV).

Shodno navedenom, u III koraku/taktnom intervalu, izračunava se operacija zahtijevana akcijom (14), dok se u IV koraku/taktnom intervalu, izračunati rezultat upisuje u odredišni registar Reg(IR [20–16]),

1. U cilju izračunavanja operacije zahtijevane akcijom (14), na prvi ulaz ALU potrebno je dovesti sadržaj registra koji je označen rs=IR [25–21] poljem instrukcije (*ALUSelA*=1), na drugi ulaz ALU – 16-bitno immediate=IR [15–0] polje instrukcije produženo, kopiranjem znaka, do 32-bitne dužine (*ALUSelB*=10), te omogućiti ALU da izvrši operaciju sabiranja (*ALUOp*=00). Postavljanjem ovih kontrolnih signala kreira se stanje kojim se obezbjedjuje izvršavanje III koraka/taktnog intervala tokom implementacije *addi* instrukcije (stanje 2 na slici 21). Primijetimo da je ovo stanje identično stanju 2, koje je ranije kreirano u cilju izvršavanje instrukcija *lw*/*sw*, slika 18.

   NAPOMENA 1: Sabiranje koje se obavlja pod kontrolom funct polja pretpostavlja postavljanje bitova ALUOp=10 (pogledaj tabelu 7 iz sekcije 5.4.1). Medjutim, immediate instrukcije ne posjeduju funct polje (slika 20), tako da je ALUOp≠10 u slučaju immediate instrukcija, a operacija sabiranja zadaje se postavljanjem ALUOp=00, kao u slučaju instrukcija *lw*/*sw*, te u slučaju sabiranja u I i u II koraku izvršavanja instrukcija, kada takodje ne raspolažemo funct poljem.

   NAPOMENA 2: 16-bitno immediate=IR [15–0] polje instrukcije, u kome je zapisana konstanta C, produžava se, ***kopiranjem znaka***, do 32-bitne dužine da bi ostao nepromijenjen znak ove konstante, odnosno da bi se omogućio rad i sa negativnim konstantnim operandom, odnosno da bi se implementacijom *addi* instrukcije jednovremeno omogućilo izvršavanje opercije oduzimanja sa konstantnim operandom.

2. U cilju upisa izračunatog rezultata u odredišni registar (registar označen rt=IR [20–16] poljem instrukcije) potrebno je obezbijediti sljedeće:

2.1. Nepromjenljivost rezultata (iz prethodnog koraka/taktnog intervala) na izlazu ALU, zadržavanjem istih kontrolnih signala koji se odnose na funkcionisanje ALU, a postavljeni su u prethodnom koraku, odnosno *ALUSelA*=1, *ALUSelB*=10, ALUOp=00 (za razloge pogledaj napomenu 2 u sekciji 5.6.5 i paragraf koji se nalazi nakon napomene 2 u poglavlju 5.5.2).

2.2. Dovesti rezultat sa izlaza ALU na Write data ulaz Registers jedinice (*MemtoReg*=0), poljem rt=IR [20–16] instrukcije označiti/adresirati registar u koji je potrebno upisati dovedeni rezultat (*RegDst*=0), te, na koncu, dozvoliti upis dovedenog rezultata u označeni registar (*RegWrite*).

Postavljeni kontrolni signali formiraju novouvudeno (u odnosu na dijagram toka sa slike 18) stanje 10, slika 21, kojim se kompletira izvršavanje instrukcije *addi*.

Primijetimo da za implementaciju instrukcije *addi* nijesu potrebne izmjene/redizajniranje postojeće arhitekture sa slike 11 (datapath-a, ALU control ili glavne kontrolne jedinice, odnosno kreiranje bilo kog novog ili izmjena postojećeg kontrolnog signala), već samo uvodjenje novog stanja u funkcionisanje mašine sa konačnim brojem stanja kojom se implementira logika glavne kontrolne jedinice multitaktne implementacije.

Realizacija instrukcije *addi* uključuje 4 koraka/taktna intervala implementirana stanjima 0, 1, 2, 10, gdje su

- stanje 0 (čitanje/uzimanje instrukcije i njeno donošenje u proces izvršavanja) i stanje 1 (dekodiranje instrukcije i čitanje sadržaja označenih registara) – zajednička stanja za implementaciju svih instrukcija,
- stanje 2 – postojeće stanje upotrebljavano za implementaciju instrukcija *lw*/*sw*, dok je
- stanje 10 – novouvedeno stanje za implementaciju instrukcije *addi*.

NAPOMENA 3: Primijetimo razlike i sličnosti izmedju stanja 10 kojim se kompletira instrukcija *addi* i stanja 4 i 7 kojima se kompletiraju *lw* i instrukcije R-tipa (navedene instrukcije takodje zahtijevaju upis rezultata zahtijevane operacije nazad – u odredišni registar):

- Slično instrukcijama R-tipa, instrukcija *addi* zahtijeva upis **rezultata ALU** nazad u odredišni registar, tako da je *MemtoReg*=0 u stanjima 10 i 7. Suprotno tome, instrukcija *lw* zahtijeva upis **podatka pročitanog iz Memory jedinice** u odredišni registar, tako da je *MemtoReg*=1 u stanju 4,
- Slično instrukciji *lw*, odredišni registar, u koji se vrši povratni upis, označen je rt=IR [20–16] poljem instrukcije *addi*, tako da je *RegDst*=0 u stanjima 10 i 4. Suprotno tome, kod instrukcija R-tipa, odredišni registar označen je rd=IR [15–11] poljem instrukcije, tako da je *RegDst*=1 u stanju 7.

5.7.2 Nadogradnja/redizajniranje arhitekture za multitaktnu implementaciju u cilju implementacije *slti* instrukcije

Prije nego započnemo redizajniranje postojeće arhitekture za multitaktnu implementaciju u cilju uključivanja *slti* instrukcije, podsjetimo se simboličkog koda i mašinskog formata zapisa ove instrukcije. Njen simbolički kod (pogledaj poglavlje 3.6 i tabelu 1 iz poglavlja 5.1) je:

$$slti \qquad \$x, \$y, C \qquad \# \ \$x = \begin{cases} 1, & \text{sadržaj } (\$y) < C \\ 0, & \text{sadržaj } (\$y) \geq C \end{cases}$$

gdje se konstanta (konstantni operand) C čuva unutar instrukcije i zapisuje na njenom kraju, dok $x, x=1,...,31, označava registar u koji se upusuje rezultat set-on-less-than operacije (rezultat poredjenja operanda, sadržanog u registru $y, y=0,...,31, sa konstantom C u odnosu "manji od", kao što je zapisano u komentaru gore navedene instrukcije).

Mašinski format zapisa instrukcije *slti* (pogledaj poglavlje 3.6 i tabelu 1 iz poglavlja 5.1) je:

Polje:	op	rs	rt	Immediate
Br. bitova:	6	5	5	16
Sadržaj:	10	y	x	C

Slika 22. Mašinski format zapisa instrukcije *slti*.

Primijetimo da mašinski format zapisa *slti* instrukcije odgovara mašinskom formatu zapisa instrukcije *addi* (pogledaj sliku 20 iz sekcije 5.7.1), s tom razlikom što je op=Instruction [31–26]=$10_{(10)}$ =$001010_{(2)}$=0xA u slučaju *slti* instrukcije (kod *addi* instrukcije je op=Instruction [31–26]=$8_{(10)}$ =$001000_{(2)}$=0x8), gdje se oznakom 0x opisuje heksadekadni zapis. U mašinskom formatu zapisa *slti* instrukcije, konstanta C smještena je na najnižim bitovima instrukcije (u 16-bitnom immediate=Instruction [15–0] polju), u rs=Instruction [25–21] polju instrukcije zapisano je obilježje registra u kome se nalazi operand ($y u simboličkoj formi), a u rt=Instruction [20–16] polju instrukcije – obilježje odredišnog registra (registar u kome se smješta rezultat set-on-less-than operacije).

Drugim riječima, nakon uzimanja instrukcije, njenog dekodiranja i čitanja sadržaja registra označenog rs=IR [25–21] poljem instrukcije (u prva 2–zajednička–koraka/taktna intervala – stanja 0 i 1 sa slike 18), u cilju kompletiranja instrukcije *slti* potrebno je implementirati akciju:

$$\text{Reg(IR [20–16])}=\text{Reg(IR [25–21])}\ \underline{\text{set-on-less-than}}\ \text{sign\_extend(IR [15–0])}, \qquad (15)$$

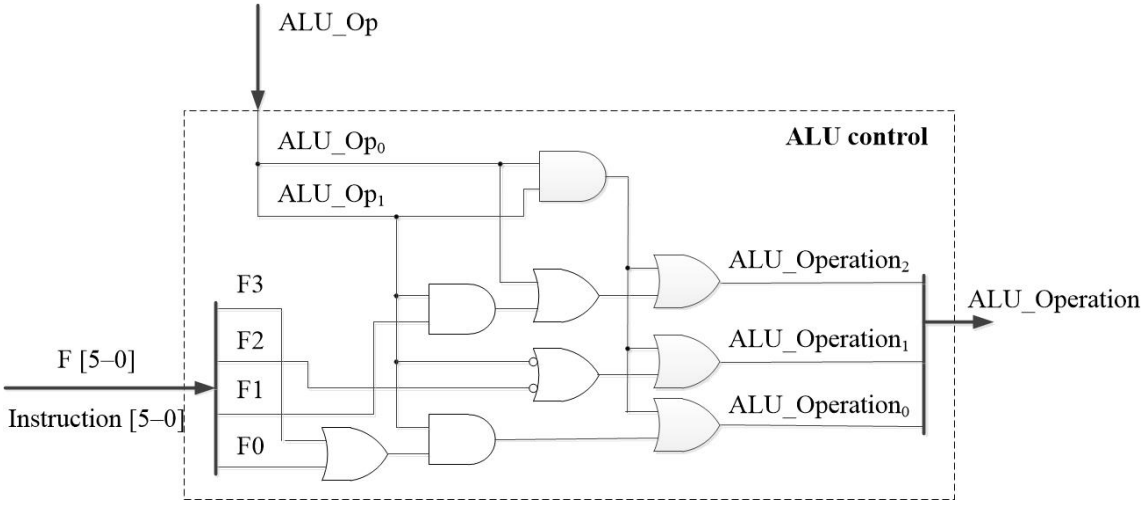
gdje je sa <u>set-on-less-than</u> simbolički zapisana set-on-less-than operacija (poredjenje u odnosu na "manji od") izmedju sadržaja registra označenog rs=IR [25–21] poljem instrukcije i 16-bitnog immediate=IR [15–0] polja instrukcije koje je produženo, kopiranjem znaka, do 32-bitne dužine.

Shodno činjenicama detaljno razmotrenim u primjedbi iz sekcije 5.7.1, akcija (15) zahtijeva izvršavanje u 2 koraka/taktna intervala (III i IV). U III koraku/taktnom intervalu, izračunava se operacija set-on-less-than, zahtijevana akcijom (15), dok se u IV koraku/taktnom intervalu, izračunava rezultat upisuje u odredišni registar Reg(IR [20–16]).

1. U cilju izračunavanja operacije zahtijevane akcijom (15), na prvi ulaz ALU potrebno je dovesti sadržaj registra koji je označen rs=IR [25–21] poljem instrukcije (*ALUSelA*=1), na drugi ulaz ALU – 16-bitno immediate=IR [15–0] polje instrukcije produženo, kopiranjem znaka, do 32-bitne dužine (*ALUSelB*=10), te omogućiti ALU da izvrši operaciju set-on-less-than. Medjutim, postavlja se pitanje kako omogućiti ALU da izvrši ovu operaciju?

   POJAŠNJENJE: ALU je dizajnirana za izvršavanje operacije set-on-less-than (vidji tabelu 5 iz sekcije 5.4.1), a izvršavanje ove operacije zadaje se postavljanjem kontrolnih signala $\text{ALU\_Operation}_{(2,1,0)}$=(1,1,1). Dakle, nepotrebne su izmjene u dizajnu ALU. Potrebno je samo natjerati ALU da, u cilju implementacije *slti* instrukcije, izvrši operaciju set-on-less-than. Medjutim, u slučaju set-on-less-than operacije, postavljanje ALU_Operation signala (od strane ALU control jedinice) suštinski se vrši pod kontrolom funct=Instruction [5–0] polja instrukcije. Naime, tada je ALUOp=10 (pogledaj tabele 7 i 8 u sekciji 5.4.1), ali ne samo za slučaj set-on-less-than operacije, već za svih 5 operacija zahtijevanih instrukcijama R-tipa, a sve one ne mogu jednoznačno biti definisane sa jednom vrijednošću ALUOp signala. Stoga se, u slučaju ALUOp=10, kontrola postavljanja ALU_Operation signala prepušta funct polju, kojim se jednoznačno definišu zahtijevane operacije ALU. Medjutim, immediate instrukcije (time i instrukcija *slti*) ne raspolažu funct poljem. Shodno tome, u slučaju *slti* immediate instrukcije, potrebno je pronaći alternativni način da se ALU zada da obavi operaciju set-on-less-than, a da to nije pod kontrolom funct polja.

   RJEŠENJE: ALU control jedinica postavlja ALU_Operation kontrolne signale na osnovu ALUOp bitova, koje generiše glavna kontrolna jedinica, i funct=Instruction [5–0] polja instrukcije, ali na osnovu funct polja samo kada je ALUOp=10. Kombinacija ALUOp=00 obezbjedjuje izvršavanje operacije sabiranja, kombinacija ALUOp=01 obezbjedjuje izvršavanje operacije oduzimanja, dok kombinacija ALUOp=10 prepušta kontrolu funkcionisanja ALU – funct polju instrukcije, kojim, ponovimo, ne raspolažu immediate instrukcije. Dakle, ne može se upotrijebiti niti jedna od do sada upotrebljavanih kombinacija ALUOp bitova u cilju kontrole izvršavanja set-on-less-than operacije za potrebe instrukcije



Slika 23. ALU control jedinica, redizajnirana u cilju generisanja ALU_Operation signala potrebnih za kontrolu izvršavanja operacija zahtijevanih instrukcijama iz tabele 7, ali i dodatom *slti* instrukcijom. Elementi, dodati u cilju redizajniranja (u odnosu na dizajn sa slike 7), istaknuti su sijenčenjem.

Tabela 14. Funkcionalna tabela ALU control jedinice sa uključenom operacijom koju zahtijeva izvršavanje instrukcije *slti*.

Instr.	ALUOp		Funct (Instruction [5–0])						ALU_Operation		
	1	0	F5	F4	F3	F2	F1	F0	2	1	0
lw, *sw*	0	0	×	×	×	×	×	×	0	1	0
beq	0	1	×	×	×	×	×	×	1	1	0
add	1	0	×	×	0	0	0	0	0	1	0
sub	1	0	×	×	0	0	1	0	1	1	0
and	1	0	×	×	0	1	0	0	0	0	0
or	1	0	×	×	0	1	0	1	0	0	1
slt	1	0	×	×	1	0	1	0	1	1	1
slti	1	1	×	×	×	×	×	×	1	1	1

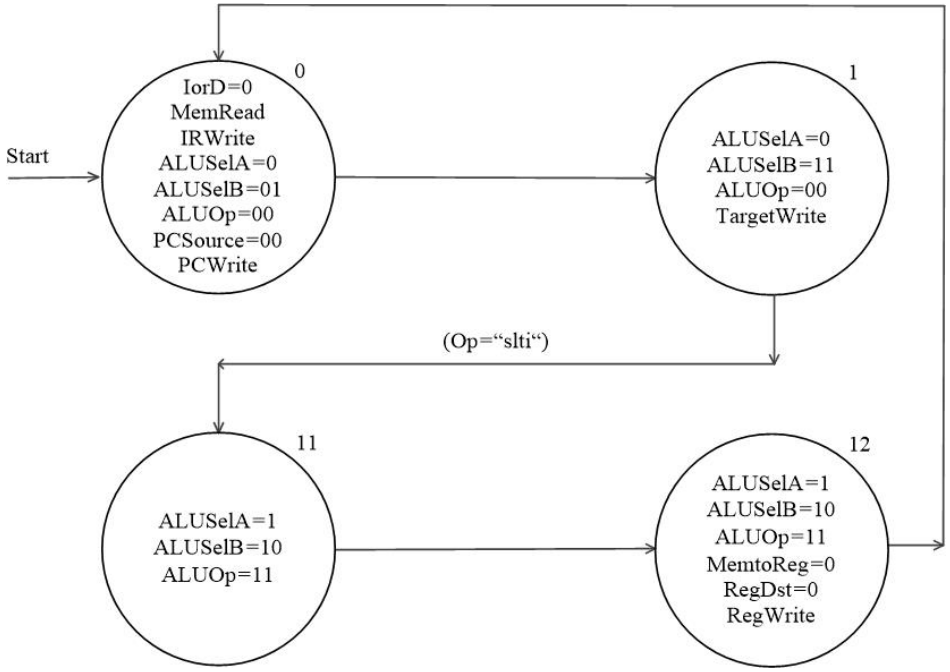
slti. Medjutim, do sada nije upotrijebljena kombinacija ALUOp=11. Upotrijebit ćemo je sada na način da se ALU zada da bezuslovno izvrši operaciju set-on-less-than kada je ALUOp=11, kao što je prikazano u tabeli 14. Drugim riječima, u cilju implementacije *slti* instrukcije potrebno je redizajnirati ALU control jedinicu iz sekcije 5.4.1 tako da obezbijedi generisanje $\text{ALU\_Operation}_{(2,1,0)}=(1,1,1)$ za ALUOp=11. Primijetimo da funkcionalna tabela 14, u poredjenju sa tabelama 7 i 8 iz sekcije 5.4.1, sadrži samo jedan dodatni red više (za kombinaciju bitova ALUOp=11 – red odvojen debljom isprekidanom linijom), tako da je:

$$\text{ALU\_Operation}_{(2,1,0)} = \text{ALU\_Operation}^{\text{staro}}_{(2,1,0)} + \text{ALUOp}_1 \wedge \text{ALUOp}_0$$

gdje kontrolni signali $\text{ALU\_Operation}^{\text{staro}}_{(2,1,0)}$ odgovaraju dizajnu ALU control jedinice sa slike 7 i već su izvedeni u sekciji 5.4.1 i dati izrazima (1)–(3). ALU control jedinica, redizajnirana u skladu sa prethodno izvedenim izrazima, prikazana je na slici 23.

Sumarno, postavljanjem kontrolnih signala *ALUSelA*=1, *ALUSelB*=10, ALUOp=11 kreira se novouvedeno (u odnosu na dijagram toka sa slike 18) stanje kojim se obezbjedjuje izvršavanje III koraka/taktnog intervala tokom implementacije *slti* instrukcije (stanje 11 na slici 24).

NAPOMENA: 16-bitno immediate=IR [15–0] polje instrukcije, u kome je zapisana konstanta C, produžava se, ***kopiranjem znaka***, do 32-bitne dužine da bi ostao nepromijenjen znak ove konstante, odnosno da bi se omogućio poredjenje u odnosu "manji od" i sa negativnim konstantnim operandom.



Slika 24. Mašina sa konačnim brojem stanja za implementaciju instrukcije *slti*.

2. U cilju upisa izračunatog rezultata u odredišni registar (registar označen rt=IR [20–16] poljem instrukcije) potrebno je obezbijediti sljedeće:

   2.1. Nepromjenljivost rezultata (iz prethodnog koraka/taktnog intervala) na izlazu ALU, zadržavanjem istih kontrolnih signala koji se odnose na funkcionisanje ALU, a postavljeni su u prethodnom koraku, odnosno *ALUSelA*=1, *ALUSelB*=10, ALUOp=11 (za razloge pogledaj napomenu 2 u sekciji 5.6.5 i paragraf koji se nalazi nakon napomene 2 u poglavlju 5.5.2).

   2.2. Dovesti rezultat sa izlaza ALU na Write data ulaz Registers jedinice (*MemtoReg*=0), poljem rt=IR [20–16] instrukcije označiti/adresirati registar u koji je potrebno upisati dovedeni rezultat (*RegDst*=0), te, na koncu, dozvoliti upis dovedenog rezultata u označeni registar (*RegWrite*).

   Postavljeni kontrolni signali formiraju novouvudeno (u odnosu na dijagram toka sa slike 18) stanje 12, slika 24, kojim se kompletira izvršavanje instrukcije *slti*.

Primijetimo da su za implementaciju instrukcije *slti* neophodne izmjene/redizajniranje postojeće arhitekture sa slike 11 u dijelu kontrole funkcionisanja ALU (ALU control jedinice – slika 23 i tabela 14), kao i uvodjenje novih stanja u funkcionisanje mašine sa konačnim brojem stanja kojom se implementira logika glavne kontrolne jedinice multitaktne implementacije.

5.7.3 Nadogradnja/redizajniranje arhitekture za multitaktnu implementaciju u cilju implementacije *ori* instrukcije

Podsjetimo se, naprije, simboličkog koda i mašinskog formata zapisa *ori* instrukcije. Njen simbolički kod (pogledaj poglavlje 3.6 i tabelu 1 iz poglavlja 5.1) je:

ori $x, $y, C # $x=$y | C

gdje se konstanta (konstantni operand) C čuva unutar instrukcije i zapisuje na njenom kraju, dok $x, x=1,...,31, označava registar u koji se upusuje rezultat logičke ILI (eng. OR) operacije izmedju operanda sadržanog u registru $y, y=0,...,31, i konstante C, kao što je zapisano u komentaru gore navedene instrukcije.

Mašinski format zapisa instrukcije *ori* (pogledaj poglavlje 3.6 i tabelu 1 iz poglavlja 5.1) je:

Polje:	op	rs	rt	Immediate
Br. bitova:	6	5	5	16
Sadržaj:	13	y	x	C

Slika 25. Mašinski format zapisa instrukcije *ori*.

Primijetimo da je, kod instrukcije *ori*, op=Instruction [31–26]=$13_{(10)}$=$001101_{(2)}$=0xD, gdje se oznakom 0x opisuje heksadekadni zapis. U mašinskom formatu zapisa, konstanta C smještena je na najnižim bitovima instrukcije (u 16-bitnom immediate=Instruction [15–0] polju), u rs=Instruction [25–21] polju instrukcije zapisano je obilježje registra u kome se nalazi operand ($y u simboličkoj formi), a u rt=Instruction [20–16] polju instrukcije – obilježje odredišnog registra (registar u kome se smješta rezultat logičke OR (ILI) operacije).

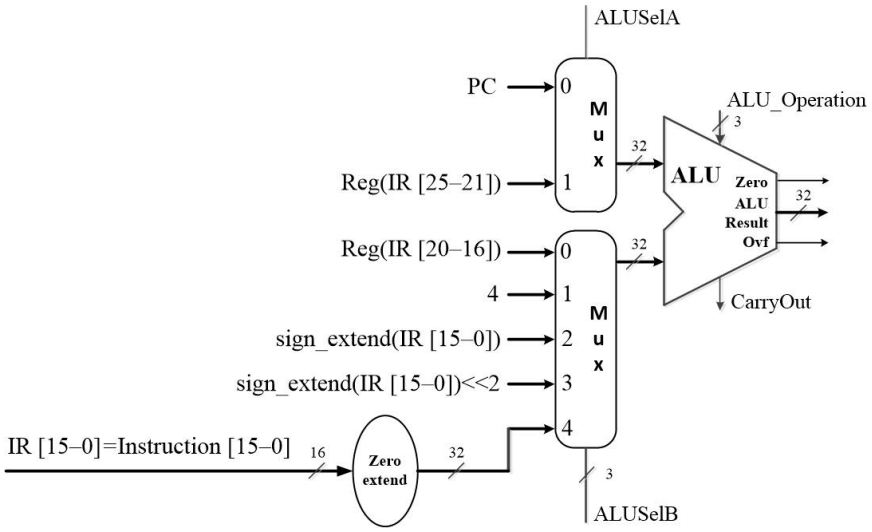
Drugim riječima, nakon uzimanja instrukcije, njenog dekodiranja i čitanja sadržaja registra označenog rs=IR [25–21] poljem instrukcije (u prva 2–zajednička–koraka/taktna intervala – stanja 0 i 1 sa slike 18), u cilju kompletiranja instrukcije *ori* potrebno je implementirati akciju:

$$\text{Reg(IR [20–16])}=\text{Reg(IR [25–21])}\ \underline{\text{OR}}\ \text{zero\_extend(IR [15–0])}, \tag{16}$$

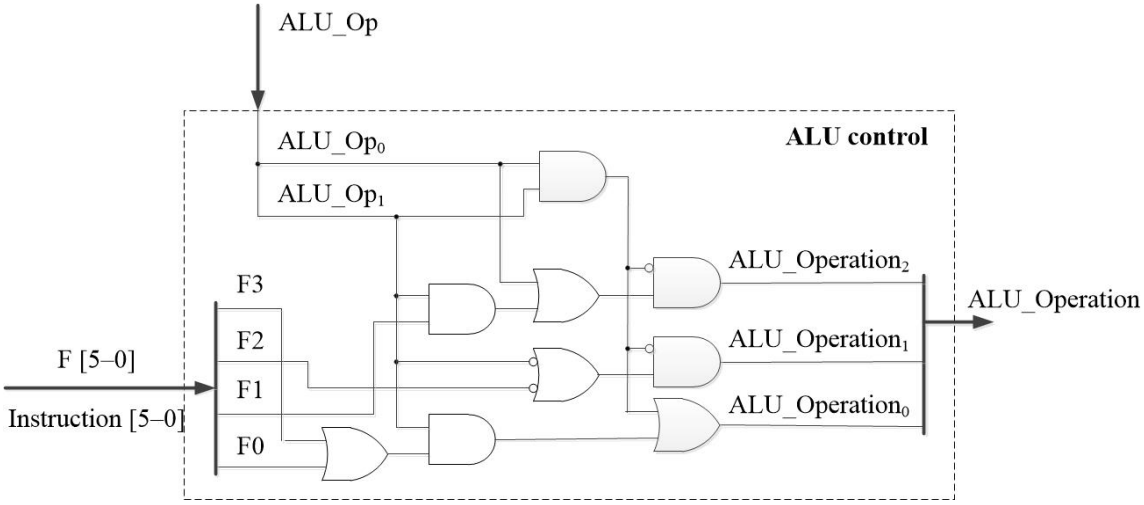
gdje je sa OR označena logička OR (ILI) operacija izmedju sadržaja registra označenog rs=IR [25–21] poljem instrukcije i 16-bitnog immediate=IR [15–0] polja instrukcije koje je produženo, kopiranjem nule, do 32-bitne dužine.

NAPOMENA 1: Logičke operacije su bit-by-bit, odnosno izvršavaju se nad svakim od bitova operanada pojedinačno i nezavisno od ostalih bitova. Drugim riječima, operandi logičkih operacija tretiraju se kao neoznačeni (unsigned) brojevi i njihovo produžavanje do odredjene dužine vrši se kopiranjem nule (eng. zero-extend – pogledaj "važno zapažanje" iz poglavlja 4), a implementira se zero-extend jedinicom.

POSLJEDICA: U cilju implementacije immediate instrukcija koje zahtijevaju izvršavanja logičkih operacija, neophodno je proširiti multipleksor na II ulazu ALU da bi se na njegov dodati ulaz doveo sadržaj 16-bitnog immediate=IR [15–0] polja instrukcije produžen, **kopiranjem nule (upotrebom Zero extend jedinice)**, do 32-bitne dužine, kao što je prikazano na slici 26. Shodno tome, prošireni multipleksor treba da ima 3 selekciona ulaza, na koje se dovodi 3-bitni *ALUSelB* kontrolni signal, te je sa ovom činjenicom (da je *ALUSelB* kontrolni signal multipleksora na II ulazu ALU – 3-bitni signal) potrebno uskladiti postavljanje ovog kontrolnog signala u prva 2–zajednička–koraka/taktna intervala. Primijetimo, uz to, da prva 4 ulaza ovog multipleksora ostaju nepromijenjena u odnosu na sliku 11, samo što se, u izmijenjenoj arhitekturi, adresiraju/selektiraju 3-bitnim kontrolnim signalom *ALUSelB*.



Slika 26. Izmjene arhitekture za multitaktnu implementaciju sa slike 11 na drugom ulazu ALU u cilju implementacije immediate instrukcija koje zahtijevaju izvršavanje logičkih operacija.



Slika 27. ALU control jedinica, redizajnirana u cilju generisanja ALU_Operation signala potrebnih za kontrolu izvršavanja operacija zahtijevanih instrukcijama iz tabele 7, ali i dodatom *ori* instrukcijom. Elementi, dodati u cilju redizajniranja (u odnosu na dizajn sa slike 7), istaknuti su sijenčenjem.

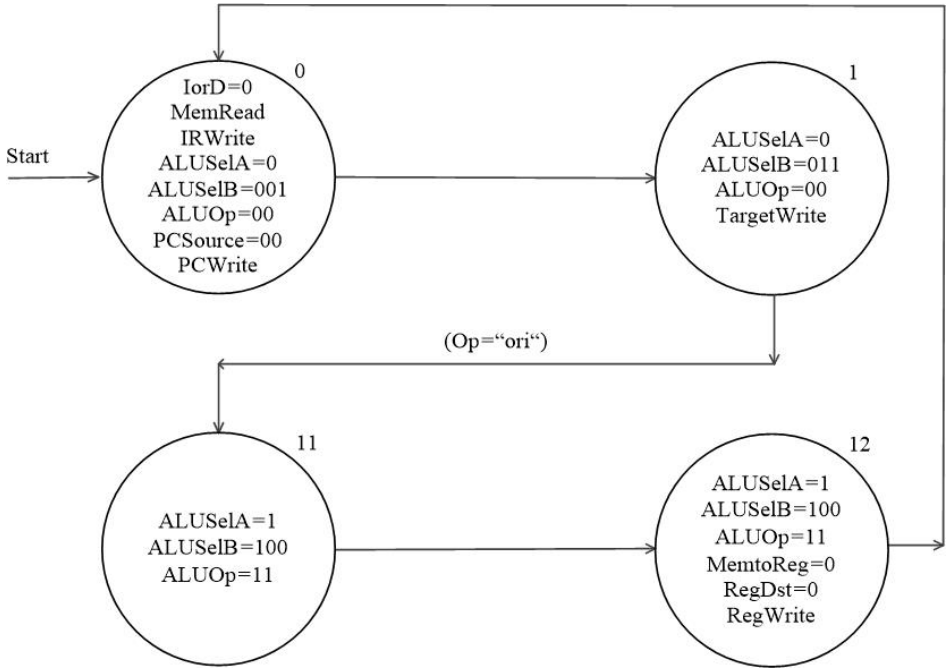
Tabela 15. Funkcionalna tabela ALU control jedinice sa uključenom operacijom koju zahtijeva izvršavanje instrukcije *ori*.

Instr.	ALUOp		Funct (Instruction [5–0])						ALU_Operation		
	1	0	F5	F4	F3	F2	F1	F0	2	1	0
lw, *sw*	0	0	×	×	×	×	×	×	0	1	0
beq	0	1	×	×	×	×	×	×	1	1	0
add	1	0	×	×	0	0	0	0	0	1	0
sub	1	0	×	×	0	0	1	0	1	1	0
and	1	0	×	×	0	1	0	0	0	0	0
or	1	0	×	×	0	1	0	1	0	0	1
slt	1	0	×	×	1	0	1	0	1	1	1
ori	1	1	×	×	×	×	×	×	0	0	1

Shodno činjenicama detaljno razmotrenim u primjedbi iz sekcije 5.7.1, akcija (16) zahtijeva izvršavanje u 2 koraka/taktna intervala (III i IV). U III koraku/taktnom intervalu, izračunava se logička OR (ILI) operacija, zahtijevana akcijom (16), dok se u IV koraku/taktnom intervalu, izračunati rezultat upisuje u odredišni registar Reg(IR [20–16]).

1. U cilju izračunavanja operacije zahtijevane akcijom (16), na prvi ulaz ALU potrebno je dovesti sadržaj registra koji je označen rs=IR [25–21] poljem instrukcije (*ALUSelA*=1), na drugi ulaz ALU – 16-bitno immediate=IR [15–0] polje instrukcije produženo, **kopiranjem nule**, do 32-bitne dužine (*ALUSelB*=100), te omogućiti ALU da izvrši operaciju OR (ILI).

   Primijetimo da je ALU dizajnirana za izvršavanje operacije OR/ILI (drugim riječima, nije je potrebno redizajnirati), ali da je, u aktuelnoj implementaciji ALU control jedinice iz sekcije 5.4.1, postavljanje ove oparacije kontrolisano kombinacijom bitova ALUOp=10, odnosno funct=IR [5–0] poljem instrukcije (pogledaj "pojašnjenje" iz sekcije 5.7.2), kojim ne raspolažu immediate instrukcije (time ni *ori* instrukcija). Drugim riječima, za izvršavanje akcije (16), potrebno je obezbijediti izvršavanje logičke OR/ILI operacija koje neće biti kontrolisano funct=IR [5–0] poljem. U tom cilju, upotrebljava se kombinacija bitova ALUOp=11, kao što je prikazano u tabeli 15, koja nije upotrijebljena prilikom dizajniranja ALU control jedinice u sekciji 5.4.1. (Naravno, ova kombinacija ALUOp bitova može biti upotrijebljena isključivo pod pretpostavkom da nije ranije upotrijebljena – prilikom implementacije instrukcije *slti*. Mogućnost redizajniranja postojeće arhitekture u cilju istovremene implementacije instrukcija *slti* i *ori* razmatrana je u napomeni 2 iz ove sekcije).



Slika 28. Mašina sa konačnim brojem stanja za implementaciju instrukcije *ori*. Primijetimo da je u stanjima 0 i 1 uskladjeno postavljanje *ALUSelB* kontrolnog signala sa 3-bitnom strukturom ovog signala koja je posljedica redizajniranja arhitekture prikazanog na slici 26.

U skladu sa upotrebom kombinacije ALUOp=11 u svrhu kontrole operacije zahtijevane *ori* instrukcijom, ALU control jedinica iz sekcije 5.4.1 mora biti redizajnirana tako da obezbijedi generisanje $\text{ALU\_Operation}_{(2,1,0)}=(0,0,1)$ za ALUOp=11. U tom cilju, primijetimo da funkcionalna tabela 15 ALU control jedinice u odnosu na tabele 7 i 8 iz sekcije 5.4.1 sadrži samo jedan dodatni red više (za kombinaciju bitova ALUOp=11 – red odvojen debljom isprekidanom linijom), tako da je:

$$\text{ALU\_Operation}_{(2,1)} = \text{ALU\_Operation}_{(2,1)}^{\text{staro}} \wedge \overline{\text{ALUOp}_1 \wedge \text{ALUOp}_0}$$

$$\text{ALU\_Operation}_0 = \text{ALU\_Operation}_0^{\text{staro}} + \text{ALUOp}_1 \wedge \text{ALUOp}_0$$

gdje kontrolni signali $\text{ALU\_Operation}_{(2,1,0)}^{\text{staro}}$ odgovaraju dizajnu ALU control jedinice sa slike 7 i već su izvedeni u sekciji 5.4.1 i dati izrazima (1)–(3). ALU control jedinica, redizajnirana u skladu sa prethodno izvedenim izrazima, prikazana je na slici 27.

Sumarno, postavljanjem kontrolnih signala *ALUSelA*=1, *ALUSelB*=100, ALUOp=11 kreira se novouvedeno (u odnosu na dijagram toka sa slike 18) stanje kojim se obezbjedjuje izvršavanje III koraka/taktnog intervala tokom implementacije *ori* instrukcije (stanje 11 na slici 28).

NAPOMENA 2: Redizajniranje arhitekure sa slike 11 u cilju pojedinačne implementacije instrukcija *slti* i *ori* izvršeno je uz kontrolu funkcionisanja ALU sa istom kombinacijom ALUOp signala (ALUOp=11). Naravno, ovo je moguće samo pod pretpostavkom pojedinačne implementacije svake od ovih instrukcija (kao da se ona druga instrukcija ne implementira). Redizajniranje arhitekture sa slike 11, ali u cilju jednovremene implementacije *slti* i *ori* instrukcija, koje obje zahtijevaju upotrebu dodatne kombinacije ALUOp bitova, zahtijevalo bi uvodjenje dodatnog ALUOp_2 bita, tako da bi se kontrola funkcionisanja ALU obavljala 3-bitnim ALUOp signalom. U tom slučaju bi, na primjer, kombinacija bitova ALUOp=011 mogla biti rezervisana za kontrolu bezuslovnog izvršavanja (od strane ALU) OR/ILI logičke operacije, a kombinacija ALUOp=100 – za kontrolu bezuslovnog izvršavanja set-on-less-than operacije. Naravno, kombinacije bitova ALUOp=000, ALUOp=001 i ALUOp=010 odgovarale bi ranije utvrdjenim namjenama (bezuslovnom sabiranju, bezuslovnom

oduzimanju i izvršavanju operacija zahtijevanih instrukcijama R-tipa – pogledaj poglavlje 5.4.1), dok bi kombinacije ALUOp=101, ALUOp=110 i ALUOp=111 mogle biti upotrijebljene za bezuslovno izvršavanje neke nove ili neke od preostalih operacija ALU koje su zahtijevane od strane instrukcija R-tipa (na primjer logičke AND operacije – sekcija 5.7.4).

2. U cilju upisa izračunatog rezultata u odredišni registar (registar označen rt=IR [20–16] poljem instrukcije) potrebno je obezbijediti sljedeće:

   2.1. Nepromjenljivost rezultata (iz prethodnog koraka/taktnog intervala) na izlazu ALU, zadržavanjem istih kontrolnih signala koji se odnose na funkcionisanje ALU, a postavljeni su u prethodnom koraku, odnosno *ALUSelA*=1, *ALUSelB*=100, ALUOp=11 (za razloge pogledaj napomenu 2 u sekciji 5.6.5 i paragraf koji se nalazi nakon napomene 2 u poglavlju 5.5.2).

   2.2. Dovesti rezultat sa izlaza ALU na Write data ulaz Registers jedinice (*MemtoReg*=0), poljem rt=IR [20–16] instrukcije označiti/adresirati registar u koji je potrebno upisati dovedeni rezultat (*RegDst*=0), te, na koncu, dozvoliti upis dovedenog rezultata u označeni registar (*RegWrite*).

   Postavljeni kontrolni signali formiraju novouvudeno (u odnosu na dijagram toka sa slike 18) stanje 12, slika 28, kojim se kompletira izvršavanje instrukcije *ori*.

Primijetimo da su za implementaciju instrukcije *ori* neophodne izmjene/redizajniranje postojeće arhitekture sa slike 11 i to

- proširivanje multipleksora koji se nalazi na drugom ulazu ALU (slika 26),
- uključivanje u dizajn Zero extend jedinice (slika 26) i
- izmjene u dijelu kontrole funkcionisanja ALU (ALU control jedinice – slika 27 i tabela 15),

kao i uvodjenje novih stanja u funkcionisanje mašine sa konačnim brojem stanja kojom se implementira logika glavne kontrolne jedinice multitaktne implementacije.

5.7.4 Nadogradnja/redizajniranje arhitekture za multitaktnu implementaciju u cilju implementacije *andi* instrukcije

Podsjetimo se, naprije, simboličkog koda i mašinskog formata zapisa *andi* instrukcije. Njen simbolički kod (pogledaj poglavlje 3.6 i tabelu 1 iz poglavlja 5.1) je:

andi $x, $y, C # $x=$y & C

gdje se konstanta (konstantni operand) C čuva unutar instrukcije i zapisuje na njenom kraju, dok $x, x=1,...,31, označava registar u koji se upusuje rezultat logičke I (eng. AND) operacije izmedju operanda sadržanog u registru $y, y=0,...,31, i konstante C, kao što je zapisano u komentaru gore navedene instrukcije.

Mašinski format zapisa instrukcije *andi* (pogledaj poglavlje 3.6 i tabelu 1 iz poglavlja 5.1) je:

Polje:	op	rs	rt	Immediate
Br. bitova:	6	5	5	16
Sadržaj:	12	y	x	C

Slika 29. Mašinski format zapisa instrukcije *andi*.

Primijetimo da je, kod instrukcije *andi*, op=Instruction [31–26]=$12_{(10)}$=$001100_{(2)}$=0xC, gdje se oznakom 0x opisuje heksadekadni zapis. U mašinskom formatu zapisa, konstanta C smještena je na najnižim bitovima instrukcije (u 16-bitnom immediate=Instruction [15–0] polju), u rs=Instruction [25–21] polju instrukcije zapisano je obilježje registra u kome se nalazi operand ($y u simboličkoj formi), a u rt=Instruction [20–16] polju instrukcije – obilježje odredišnog registra (registar u kome se smješta rezultat logičke AND operacije).

Drugim riječima, nakon uzimanja instrukcije, njenog dekodiranja i čitanja sadržaja registra označenog rs=IR [25–21] poljem instrukcije (u prva 2–zajednička–koraka/taktna intervala – stanja 0 i 1 sa slike 18), u cilju kompletiranja instrukcije *andi* potrebno je implementirati akciju:

$$\text{Reg(IR [20–16])}=\text{Reg(IR [25–21])}\ \underline{\text{AND}}\ \text{zero\_extend(IR [15–0])}, \qquad (17)$$

gdje je sa AND označena logička AND operacija izmedju sadržaja registra označenog rs=IR [25–21] poljem instrukcije i 16-bitnog immediate=IR [15–0] polja instrukcije koje je produženo, kopiranjem nule, do 32-bitne dužine.

NAPOMENA: Kao i OR operacija, AND je takodje logička bit-by-bit operacija (izvršava se nad svakim od bitova operanada pojedinačno i nezavisno od ostalih bitova). Drugim riječima, operandi se tretiraju kao neoznačeni (unsigned) brojevi i njihovo produžavanje do odredjene dužine vrši se kopiranjem nule (eng. zero-extend – pogledaj "važno zapažanje" iz poglavlja 4, kao i napomenu 1 iz sekcije 5.7.3), a implementira se zero-extend jedinicom.

POSLJEDICA: U cilju implementacije immediate instrukcija koje zahtijevaju izvršavanja logičkih operacija, time i *andi* instrukcije, neophodno je proširiti multipleksor na II ulazu ALU da bi se na njegov dodati ulaz doveo sadržaj 16-bitnog immediate=IR [15–0] polja instrukcije produžen, **kopiranjem nule (upotrebom Zero extend jedinice)**, do 32-bitne dužine, kao što je prikazano na slici 26. Shodno tome, prošireni multipleksor treba da ima 3 selekciona ulaza, na koje se dovodi 3-bitni *ALUSelB* kontrolni signal, te je sa ovom činjenicom (da je *ALUSelB* kontrolni signal multipleksora na II ulazu ALU – 3-bitni signal) potrebno uskladiti postavljanje ovog kontrolnog signala u prva 2–zajednička–koraka/taktna intervala. Primijetimo, uz to, da prva 4 ulaza ovog multipleksora ostaju nepromijenjena u odnosu na sliku 11, samo što se, u izmijenjenoj arhitekturi, adresiraju/selektiraju 3-bitnim kontrolnim signalom *ALUSelB*.

Shodno činjenicama detaljno razmotrenim u primjedbi iz sekcije 5.7.1, akcija (17) zahtijeva izvršavanje u 2 koraka/taktna intervala (III i IV). U III koraku/taktnom intervalu, izračunava se logička AND operacija, zahtijevana akcijom (17), dok se u IV koraku/taktnom intervalu, izračunati rezultat upisuje u odredišni registar Reg(IR [20–16]).

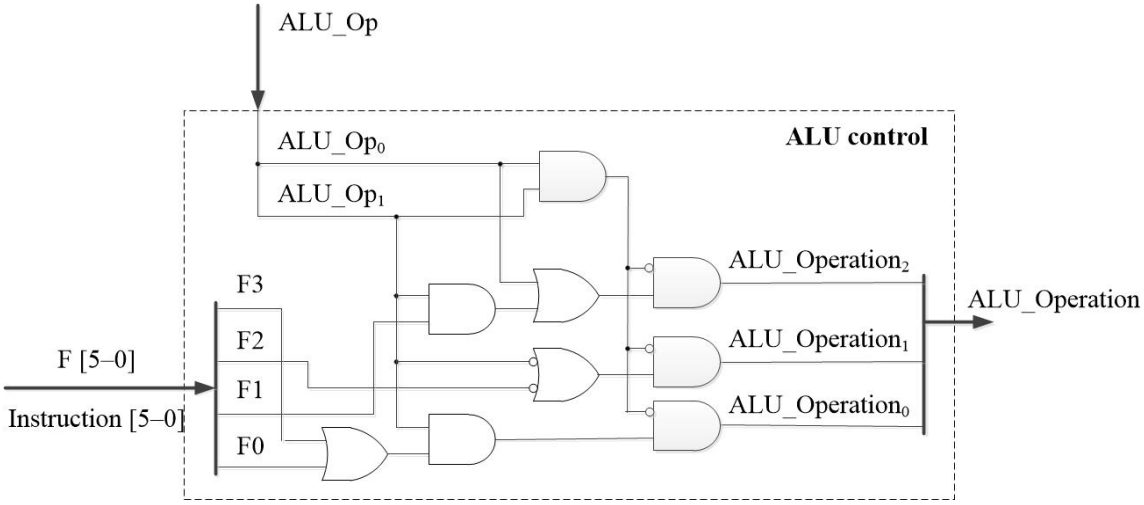
1. U cilju izračunavanja operacije zahtijevane akcijom (17), na prvi ulaz ALU potrebno je dovesti sadržaj registra koji je označen rs=IR [25–21] poljem instrukcije (*ALUSelA*=1), na drugi ulaz ALU – 16-bitno immediate=IR [15–0] polje instrukcije produženo, **kopiranjem nule**, do 32-bitne dužine (*ALUSelB*=100), te omogućiti ALU da izvrši operaciju AND.

   Primijetimo da je ALU dizajnirana za izvršavanje operacije AND (dakle, nije je potrebno redizajnirati), ali da je, u aktuelnoj implementaciji ALU control jedinice iz sekcije 5.4.1, postavljanje ove oparacije kontrolisano kombinacijom bitova ALUOp=10, odnosno funct=IR [5–0] poljem instrukcije (pogledaj "pojašnjenje" iz sekcije 5.7.2), kojim ne raspolažu immediate instrukcije (time ni *andi* instrukcija). Drugim riječima, za izvršavanje akcije (17), potrebno je obezbijediti izvršavanje logičke AND operacija koje neće biti kontrolisano funct=IR [5–0] poljem. U tom cilju, upotrebljava se kombinacija bitova ALUOp=11, kao što je prikazano u tabeli 16, koja nije upotrijebljena prilikom dizajniranja ALU control jedinice u sekciji 5.4.1. (Naravno, ova kombinacija ALUOp bitova može biti upotrijebljena isključivo pod pretpostavkom da nije ranije upotrijebljena – prilikom implementacije instrukcija *slti* i *ori*. Mogućnost redizajniranja postojeće arhitekture u cilju istovremene implementacije instrukcija *slti*, *ori* i *andi* razmatrana je u napomeni 2 iz sekcije 5.7.3).

   U skladu sa upotrebom kombinacije ALUOp=11 u svrhu kontrole operacije zahtijevane *andi* instrukcijom, ALU control jedinica iz sekcije 5.4.1 mora biti redizajnirana tako da obezbijedi generisanje $\text{ALU\_Operation}_{(2,1,0)}=(0,0,0)$ za ALUOp=11. U tom cilju, primijetimo da funkcionalna tabela 16 ALU control jedinice u odnosu na tabele 7 i 8 iz sekcije 5.4.1 sadrži samo jedan dodatni red više (za kombinaciju bitova ALUOp=11 – red odvojen debljom isprekidanom linijom), tako da je:

$$\text{ALU\_Operation}_{(2,1,0)} = \text{ALU\_Operation}^{\text{staro}}_{(2,1,0)} \wedge \overline{\text{ALUOp}_1 \wedge \text{ALUOp}_0}$$

   gdje kontrolni signali $\text{ALU\_Operation}^{\text{staro}}_{(2,1,0)}$ odgovaraju dizajnu ALU control jedinice sa slike 7 i već su izvedeni u sekciji 5.4.1 i dati izrazima (1)–(3). ALU control jedinica, redizajnirana u skladu sa prethodno izvedenim izrazima, prikazana je na slici 30.



Slika 30. ALU control jedinica, redizajnirana u cilju generisanja ALU_Operation signala potrebnih za kontrolu izvršavanja operacija zahtijevanih instrukcijama iz tabele 7, ali i dodatom *andi* instrukcijom. Elementi, dodati u cilju redizajniranja (u odnosu na dizajn sa slike 7), istaknuti su sijenčenjem.

Tabela 16. Funkcionalna tabela ALU control jedinice sa uključenom operacijom koju zahtijeva izvršavanje instrukcije *andi*.

Instr.	ALUOp		Funct (Instruction [5–0])						ALU_Operation		
	1	0	F5	F4	F3	F2	F1	F0	2	1	0
lw, *sw*	0	0	×	×	×	×	×	×	0	1	0
beq	0	1	×	×	×	×	×	×	1	1	0
add	1	0	×	×	0	0	0	0	0	1	0
sub	1	0	×	×	0	0	1	0	1	1	0
and	1	0	×	×	0	1	0	0	0	0	0
or	1	0	×	×	0	1	0	1	0	0	1
slt	1	0	×	×	1	0	1	0	1	1	1
andi	1	1	×	×	×	×	×	×	0	0	0

Sumarno, postavljanjem kontrolnih signala *ALUSelA*=1, *ALUSelB*=100, ALUOp=11 kreira se novouvedeno (u odnosu na dijagram toka sa slike 18) stanje kojim se obezbjedjuje izvršavanje III koraka/taktnog intervala tokom implementacije *andi* instrukcije (stanje 11 na slici 31).

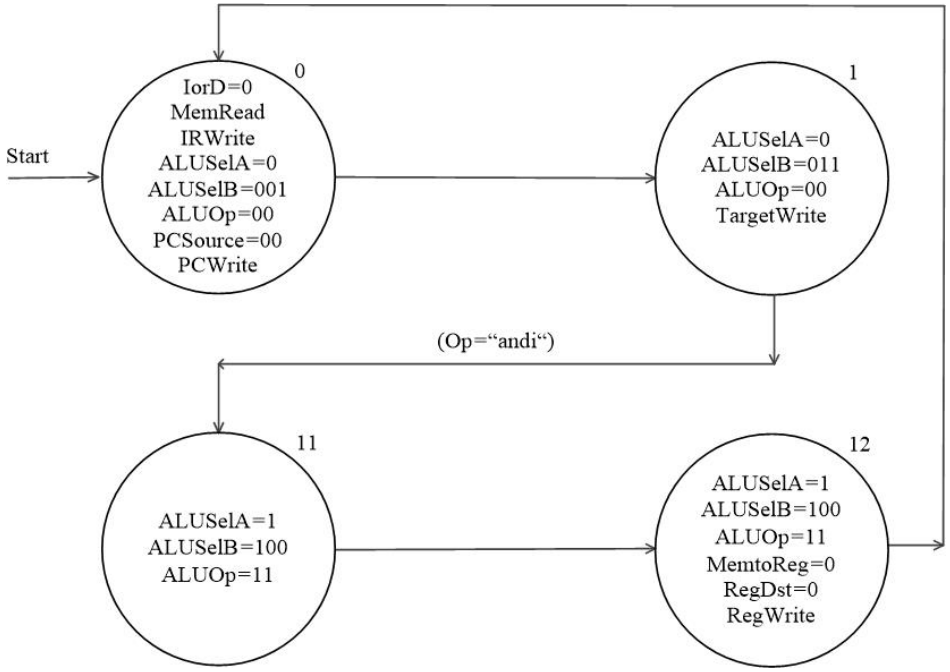
2. U cilju upisa izračunatog rezultata u odredišni registar (registar označen rt=IR [20–16] poljem instrukcije) potrebno je obezbijediti sljedeće:

   2.1. Nepromjenljivost rezultata (iz prethodnog koraka/taktnog intervala) na izlazu ALU, zadržavanjem istih kontrolnih signala koji se odnose na funkcionisanje ALU, a postavljeni su u prethodnom koraku, odnosno *ALUSelA*=1, *ALUSelB*=100, ALUOp=11 (za razloge pogledaj napomenu 2 u sekciji 5.6.5 i paragraf koji se nalazi nakon napomene 2 u poglavlju 5.5.2).

   2.2. Dovesti rezultat sa izlaza ALU na Write data ulaz Registers jedinice (*MemtoReg*=0), poljem rt=IR [20–16] instrukcije označiti/adresirati registar u koji je potrebno upisati dovedeni rezultat (*RegDst*=0), te, na koncu, dozvoliti upis dovedenog rezultata u označeni registar (*RegWrite*).

   Postavljeni kontrolni signali formiraju novouvudeno (u odnosu na dijagram toka sa slike 18) stanje 12, slika 31, kojim se kompletira izvršavanje instrukcije *andi*.

Primijetimo, na koncu, da su za implementaciju instrukcije *andi* neophodne izmjene/redizajniranje postojeće arhitekture sa slike 11 i to



Slika 31. Mašina sa konačnim brojem stanja za implementaciju instrukcije *andi*. Primijetimo da je u stanjima 0 i 1 uskladjeno postavljanje ALUSelB kontrolnog signala sa 3-bitnom strukturom ovog signala koja je posljedice redizajniranja arhitekure prikazane na slici 30.

- proširivanje multipleksora koji se nalazi na drugom ulazu ALU (slika 26),
- uključivanje u dizajn Zero extend jedinice (slika 26) i
- izmjene u dijelu kontrole funkcionisanja ALU (ALU control jedinice – slika 30 i tabela 16),

kao i uvodjenje novih stanja u funkcionisanje mašine sa konačnim brojem stanja kojom se implementira logika glavne kontrolne jedinice multitaktne implementacije.

VJEŽBE

ODGOVORENA PITANJA ZA PROVJERU ZNANJA I RIJEŠENI ISPITNI ZADACI

NAPOMENA: Prilikom rješavanja pitanja i zadataka iz ove oblasti, studentima je na završnom ispitu dostupna sljedeća literatura: šema kompletne arhitekture za multitaktnu implementaciju (slika 11), dijagrami stanja (slika 18), spisak svih MIPS instrukcija, zajedno sa sintaksom i prikazanim formatom mašinskog zapisivanja, i sažeta funkcionalna tabela ALU control jedinice (tabela 7).

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

Odgovor:

Izvršava se instrukcija *beq*. Shodno tome, multitaktna implemenmtacija će tokom izvršavanja ove instrukcije proći kroz stanja 0, 1, 8 (pogledaj sliku 18). Sadržaj PC registra tokom svakog od ovih stanja je:

Stanje 0: **PCWrite**=1, **PCSource**=00 → PC=PC+4.
Stanje 1: isto – sadržaj PC registra ostaje nepromijenjen
Stanje 8: **PCWriteCond**=1, **Zero**=0 (sadržaji registara $4 i $5 su različiti), **PCWrite**=0
⇒ isto – sadržaj PC registra ostaje nepromijenjen.

Dakle, nakon izvršavanja zadate instrukcije, u registar PC biće upisano PC+4, gdje je PC adresa instrukcije *beq*.

2. Čemu služi **Shift left 2** jedinica na posljednjem ulazu multipleksora koji se nalazi na drugom ulazu ALU i u toku izvršavanja kojih instrukcija ona dolazi do izražaja?

Odgovor:

Shift left 2 jedinica implementira množenje sa konstantom 4 i koristi se prilikom izračunavanja adrese uslovnog skoka/grananja 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 vrši se u II koraku/taktnom intervalu, u toku izvršavanja svih instrukcija (II kotak/taktni interval, odnosno stanje 1 je zajedničko za sve instrukcije). Mašinski kod *jal* instrukcije zapisuje se u J-formatu (pogledaj tabelu 1 iz poglavlja 5.1), odnosno adresa 2000 zapisuje se u odgovarajućem 26-bitnom address polju, sa

IR [25–0]=00 0000 0000 0000 0111 1101 0000.

Primijetimo da se prilikom izračunavanja adrese koja se upisuje u Target registar upotrebljavaju bitovi koji odgovaraju 16-bitnom address/immediate/offset polju instrukcije (pogledaj tabelu 2 iz poglavlja 5.2 i akciju (5) iz sekcije 5.6.2), odnosno bitovi IR [15–0]=0000 0111 1101 0000=$2000_{(10)}$. Shodno tome, sadržaj Target registra, tokom izvršavanja instrukcije ***jal* 2000**, postavlja se (u II koraku – stanje 1) na PC+4+4*2000, gdje je PC adresa instrukcije ***jal* 2000**.

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

Odgovor:

Epilog nakon izvršavanja svake od instrukcija biće zapisan u komentarima instrukcija.

***slti* $8,$0,1** # PC=PC+4, a u registru $8 je 1 nakon izvršavanja instrukcije *slti*
***bne* $8,$0,111** # u stanju 0: PC=PC+4, u stanju 8: uslov grananja je zadovoljen ⇒ PC=PC+4*111.

Dakle, u registar PC biće upisano PC+8+4*111, gdje je PC adresa instrukcije *slti*.

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:

Tokom izvršavanja instrukcije *lw*, izračunvaanje adrese sa koje se preuzima podatak (iz Memory jedinice) za upis u odgovarajući registar obavlja ALU i to u stanju 2 (slika 18), **ALU result**=IR[25–21] + sign_extend(IR [15–0]). Signal **ALU result** dovodi se na ulaz 1 multipleksora koji se nalazi na **Read address** ulazu **Memory** jedinice. Signal **ALU result** predstavlja adresu memorijske lokacije sa koje je potrebno pročitati podatak koji je kasnije potrebno upisati u registar $20. U tom cilju, da bi **ALU result** pristupila Read address ulazu Memory jedinice, potrebno je postaviti signal *IorD*=1.

6. MIPS procesor izvršava instrukciju ***bne* $13, $0, 130**, a prije nje je izvršena instrukcija ***addi* $13, $0, 0**. Što će biti upisano u PC registru nakon izvršavanja pomenute *bne* instrukcije? Objasniti!

Odgovor:

Nakon izvršavanja instrukcije ***addi* $13, $0, 0**, u registru $13 upisana je nula, a PC=PC+4. S' obzirom da, u toku izvršavanja instrukcije ***bne* $13, $0, 130**, neće doći do skoka (sadržaji registara $0 i $13 nijesu različiti, odnosno nije zadovoljen uslog grananja), tako da će sadržaj PC registra, i tokom izvršavanja instrukcije *bne*, biti uvećan za 4.

Dakle, nakon izvršavanja instrukcije *bne*, u PC registru biće upisano PC+8, gdje je PC adresa instrukcije *addi*.

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

Odgovor:

Izvršava se instrukcija *beq*. Shodno tome, multitaktna implementacija će tokom izvršavanja ove instrukcije proći kroz stanja 0, 1, 8 (pogledaj sliku 18). Sadržaj PC registra nakon svakog od ovih stanja je:

Stanje 0: PC=PC+4 (**PCWrite**=1, **PCSource**=00)
Stanje 1: isto – sadržaj PC registra ostaje nepromijenjen, Target=PC+4*1021
Stanje 8: PC←Target=PC+4*1021 (**PCWriteCond**=1, **Zero**=1, **PCSource**=01).

Dakle, nakon izvršavanja zadate instrukcije, u registar PC biće upisano PC+4+4*1021, gdje je PC adresa instrukcije *beq*.

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

Odgovor:

Epilog nakon izvršavanja svake od instrukcija biće zapisan u komentarima instrukcija.

***slti* $8, $0, 1021** # PC=PC+4, a u registru $8 je 1 nakon izvršavanja instrukcije *slti*
***bne* $8, $0, 1021** # u stanju 0: PC=PC+4, u stanju 8: uslov grananja je zadovoljen ⇒
⇒ PC=PC+4*1021

Dakle, u registar PC biće upisano PC+8+4*1021, gdje je PC adresa instrukcije *slti*.

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 bitovima rezultata ALU. Ovaj signal učestvuje prilikom implementacije instrukcija uslovnog skoka/grananja. Shorno tome, nema nikakvu ulogu prilikom izvršavanja instrukcije ***lw* $3, 1021($10)**.

10. Od MIPS instrukcija koje su implementirane datom šemom (slike 11), koja se instrukcija zahtijeva najkraće vrijeme za svoje izvršavanje, a koja najduže, i koliko traje njihovo izvršavanje izraženo brojem zahtijevanih taktnih intervala (clock ciklusa)?

Odgovor:

Najduže vrijeme zahtijeva ***lw*** instrukcija – 5 taktnih intervala, a najkraće (po 3 taktna intervala) – instrukcije uslovnog i bezuslovnog skoka ***beq*** i ***j***.

11. Zbog čega je potrebno uvesti **Target** registar u arhitekturu za multitaktnu implementaciju?

Odgovor:

Za implementaciju pojedinih instrukcija, ALU se višestruko upotrebljava (u više taktnih intervala). U II koraku – stanju 1 (zajedničko stanje za sve implementirane instrukcije), ALU se upotrebljava za izračunavanje ciljne adrese grananja na koju se usmjerava tok izvršavanja programa u slučaju instrukcija uslovnog skoka/grananja, ali samo ukoliko je uslov grananja zadovoljen. Izračunata adresa čuva se (od prebrisavanja) u **Target** registru. Nakon toga, ALU se može koristiti za potpunu implementaciju date instrukcije. Kada ciljna adresa grananja, izračunata u II koraku, ne bi bila sačuvana u Target registru, već u sljedećem koraku bi bila prebrisana novim rezultatom izračunavanja na izlazu ALU, čak i u slučaju izvršavanja instrukcija uslovnog skoka/grananja.

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/selektovanje da li će u određeni registar **Registers** jedinice biti upisan rezultat neke aritmetičko-logičke operacije doveden sa **ALU result** izlaza ALU (instrukcije R-tipa), ili, podatak pročitan iz memorije (instrukcija ***lw***).

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

Odgovor:

Tokom izvršavanja instrukcije *lw*, izračunavanje adrese memorijske lokacije u koju će biti upisan sadržaj registra označen rt=IR [20–16] poljem instrukcije (u ovom zadatku registar $21), obavlja ALU i to u stanju 2 (slika 18), **ALU result**=IR [25–21] + sign_extend(IR [15-0]). Signal **ALU result** dovodi se na **Write address** ulaz **Memory** jedinice, kao adresa memorijske lokacije u 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. Što će tačno biti upisano u **Target** registar u toku izvršenja ove instrukcije i u kojem taktnom intervalu? Objasniti! Koristi li se sadržaj ovog registra tokom izvršavanja zadate instrukcije?

Odgovor:

Najprije je potrebno odrediti instrukciju koja se izvršava. Ona se odredjuje na osnovu op=Instruction [31–26] polja mašinskog koda zapisivanja ove instrukcije.

op=Instruction [31–26]=$101011_{(2)}$=$43_{(10)}$=0x2B ⇒ instrukcija ***sw***
address=Instruction [15–0]=0000 0100 0000 $0010_{(2)}$=$1026_{(10)}$.

Dakle, izvršava se instrukcija *sw*. Shodno tome, multitaktna implemenmtacija ove instrukcije zahtijeva 4 taktna intervala i tokom izvršavanja proći će kroz stanja 0, 1, 2, 5 (pogledaj sliku 18). Sadržaj PC i Target registra se mijenja tokom prva 2 taktna intervala (stanja 0 i 1):

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 se nalazi na ulazu PC registra i u toku izvršavanja kojih instrukcija se upotrebljava ova jedinica?

Odgovor:

Arhitektura za multitaktnu implementaciju instrukcija sa slike 11 upotrebljava dvije **Shift left 2** jedinice. **Shift left 2** jedinica, odredjena tekstom zadatka, upotrebljava se u cilju množenja sa 4, prilikom postavljanja adrese bezuslovnog skoka (Jump address na slici 11). Ova **Shift left 2** jedinica se upotrebljava kod implementacije instrukcija bezuslovnog skoka (instrukcije J-tipa).

16. U toku je izvršavanje instrukcije ***beq* $21, $19, 160**. Koja je memorijska adresa na koju se "prelazi/skače" i od čega zavisi da li će doći do skoka?

Odgovor:

U II koraku/taktnom intervalu – stanje 1, izračunava se ciljna adresa grananja i, nakon izračunavanja, upisuje se u Target registar, **Target**=PC+4+4*160.

Da li će doći do "skoka" na ciljnu adresu grananja zavisi od sadržaja registara $21 i $19, odnosno od vrijednosti **Zero** signala. Naime, ukoliko su sadržaji registara $21 i $19 jednaki, na **ALU result** izlazu ALU biće nula (stanje 8). **Zero** signal u tom slučaju ima vrijednost 1. S' obzirom da je u stanju 8: **PCWriteCond**=1 i **PCSource**=01, u PC registar će, u slučaju jednakosti sadržaja registara $21 i $19, biti upisana vrijednost sačuvana u **Target** registru. Kao posljedica, tok izvršavanja programa usmjerava se na instrukciju koja je u memoriji računara zapisana na adresi PC+4+4*160 (u ovom slučaju kažemo da je došlo do "skoka"). Ukoliko su sadržaji registara $21 i $19 različiti, na **ALU result** izlazu ALU biće vrijednost različita od nule. **Zero** signal u tom slučaju ima vrijednost 0. Drugim riječima, u slučaju nejednakosti sadržaja registara $21 i $19, u PC registru bi ostala vrijednost PC+4 (u ovom slučaju kažemo da nije došlo do "skoka", već da se izvršavanje programa nastavlja od instrukcije koja je u memoriji računara zapisana odmah nakon instrukcije *beq*).

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 vrši se u II koraku/taktnom intervalu, u toku izvršavanja svih instrukcija (II kotak/taktni interval, odnosno stanje 1 je zajedničko za sve instrukcije). Tokom izvršavanja zadate instrukcije ***beq* $13, $17, 221**, u II koraku/taktnom intervalu (stanje 1) sadržaj Target registra se mijenja, **Target**=PC+4+4*221, gdje je PC adresa instrukcije *beq*.

18. Modifikovati datapath i controlpath da bi se mogla implementirati instrukcija *jal*. Prikazati modifikaciju dijagrama stanja za implementiranje ove instrukcije.

Rješenje:

Prilikom realizacije instrukcija koje nijesu realizovane postojećom multitaktnom arhitekturom, prvi korak je da prepoznamo tip instrukcije koju je potrebno realizovati, i da nađemo njoj najsličniju instrukciju, a čija realizacija postoji.

Zadata *jal* instrukcija pripada *j* tipu instrukcija. U izučavanoj arhitekturi postoji realizacija *j* instrukcije, odnosno osnovne instrukcije skoka. Posmatrajmo dijagram stanja. *j* instrukcija se izvršava u 3 takta (stanja 0, 1 i 9).

Uočimo da su stanja 0 i 1 zajednička za sve relizovane instrukcije. Stoga, i kada budemo relizovali nove instrukcije, ova stanja ćemo zadržavati, tj. **u njima neće biti promjena!** Takodje, prilikom realizacije novih instrukcija, **ne smije** se izvršiti promjena u datapath/controlpath/dijagramu stanja koja uzrokuje da neka od postojećih, već realizovanih funkcija ne radi ispravno!

Po čemu se razlikuju instrukcije *j* i *jal*?

I jedan i druga instrukcija vrše skok na određenu labelu (adresu). Međutim, osim skoka instrukcija *jal* vrši i „povezivanje" (*jal – jump and link*), odnosno pamti adresu sljedeće instrukcije (one koja se vrši neposredno nakon nje) i smješta tu adresu u $31. Podsjećanja radi, upotrebom *jal* instrukcije se vrši skok na proceduru, a činjenica da se pamti adresa sljedeće instrukcije nam omogućava povratak nazad na tačnu lokaciju nakon izvršenja pozvane procedure.

Šta ovo znači u smislu realizacije *jal* instrukcije? Osim skoka koji vrši već realizovana *j* instrukcija, potrebno je obezbjediti smještanje sljedeće adrese u $31.

Ko čuva informaciju o sljedećoj adresi? To je PC (Program Counter) registar. Dakle, sadržaj ovog registra moramo proslijediti u $31. Da bismo mogli (bilo koji) podatak smjestiti u $31, moramo omogućiti upis $31.

Obratite pažnju na Registers jedinicu u izučavanoj višetaktnoj realizaciji. Posljednja dva ulaza u Registers jedinicu su *Write register* i *Write data*. Na ulaz *Write register* se dovodi registar u koji se želi izvršti upis, a na ulaz *Write data* se dovodi podatak čiji upis se vrši. Dakle, u našem slučaju na ulaz *Write register* je potrebno dovesti $31, a na ulaz *Write data* je potrebno dovesti PC. Na koji način stižu podaci na ova dva ulaza Registers jedinice? Uočavamo da stižu iz izlaza dva multipeksora. Oba multipleksora su MUX 2/1 (podsjetite se principa rada MUXeva) i oba ova MUXa su iskorišćena u punom kapacitetu. To znači da na njihovim ulazima već postoje određeni podaci. Mogli bismo neke od tih podataka zamijeniti željenim podacima ($31 i PC), ali bismo u tom slučaju onemogućili rad jedne ili više standarnih instrukcija, što **nije dozvoljeno**. Preostaje nam da proširimo uočene MUXeve, tako da osim postojećih podataka mogu primiti i podatke koji su nama od interesa. Postojeći MUXevi imaju po dva ulaza, a da bi primili potrebne podatke neophodna su im po tri ulaza. To znači da ćemo umjesto MUXeva 2/1 koristiti MUXeve 4/1. Uočite da novi MUXevi 4/1 neće biti iskorišćeni u punom kapacitetu (od 4 ulaza nama će biti potrebna samo 3, i to 2 za postojeće podatke i 1 za novi podatak). Osim toga, MUX 4/1 ima 2 kontrolna/selekciona signala (za razliku od MUX 2/1 koji ima jedan kontrolni/selekcioni signal). Dakle, signali koji kontrolišu posmatrana dva MUX 2/1 (signali *RegDst* i *MemtoReg*) će umjesto dosadašnjih jednobitnih vrijednosti uzimati dvobitne vrijednosti.

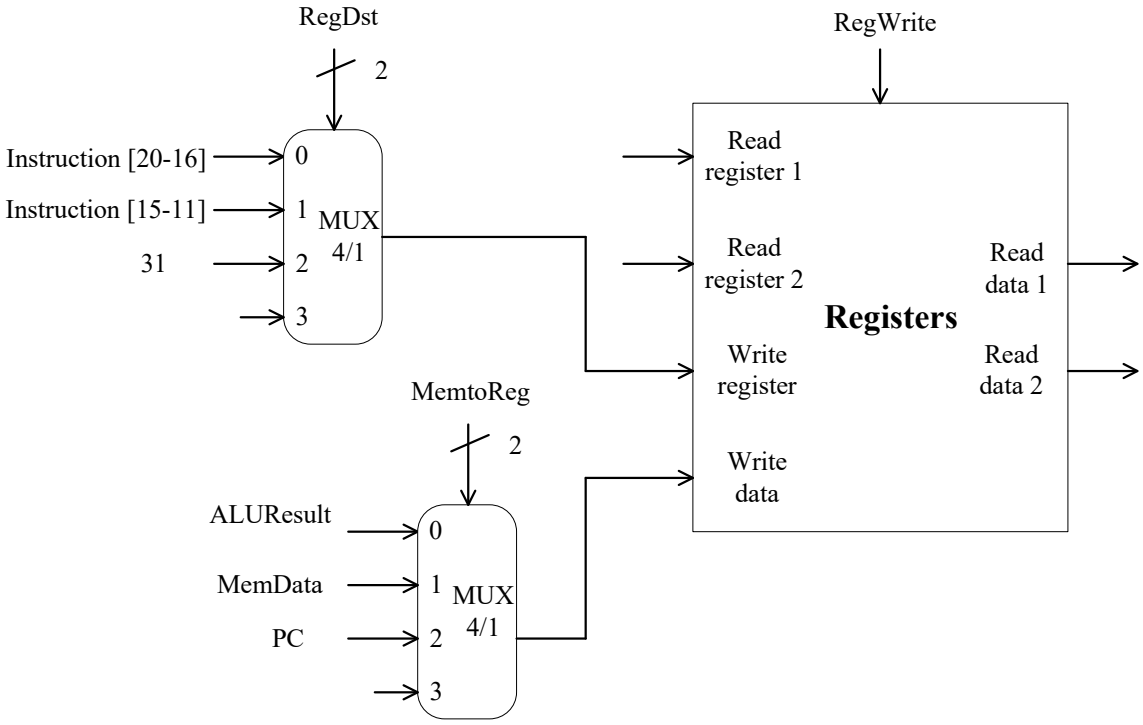
U nastavku je data grafički objašnjena izmjena u postojećoj arhitekturi.

Ostatak arhitetkure nije mijenjan, te stoga nije ni predstavljen na slici.

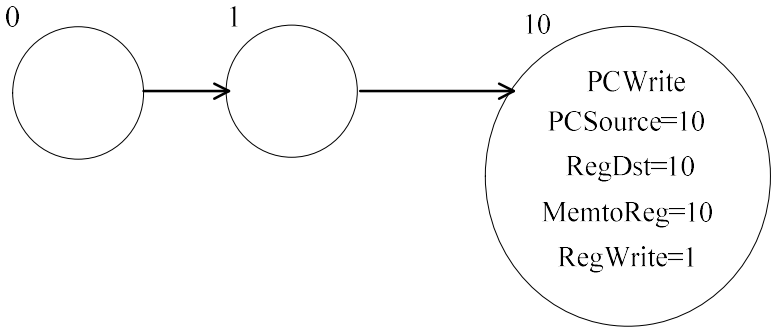
Šta se dešava sa dijagramom stanja?

Kao što je već rečeno, stanja 0 i 1 su ista za sve instrukcije, i u njima nema promjena. Treći takt će sadržati sve ono što sadrži treći takt instrukcije *j* (stanje 9), jer nam se na taj način omogućava funkcija skoka. Uz to, sadržaće kontrolne signale koje smo koristili da bismo obezbijedili „povezivanje". To su signali *RegDst* i *MemtoReg*, koji će uzimati vrijednosti 10. Zašto ove vrijednosti? Zato što je

neophodno propustiti ulaze 2 (binarno 10) sa posmatranih MUXeva na odgovarajuće izlaze. Postoji još jedan signal koji moramo setovati (postaviti na 1). To je signal *RegWrite*, koji omogućava upotrebu Registers jedinice. Kako registers jedinica nije bila potrebna u realizaciji *j* instrukcije, to se ovaj signal nije upotrebljavao. Međutim, prilikom realizacije *jal* instrukcije intenzivno koristimo Registers jedinicu, te nam je neophodno setovati *RegWrite*, i na taj način dozvoliti upisivanje u istu.



Dijagram stanja *jal* instrukcije će stoga izgledati ovako:



Obratite pažnju na numeraciju stanja. Posljednje stanje smo označili kao stanje 10, zato što je u pitanju novo stanje, i svojstveno samo intrukciji *jal*. Pogrešno bi bilo označiti ga kao 9, jer bi na taj način oštetili dijagram stanja instrukcije *j*. Kako nema izmjena u stanju 0 i 1, za njih koristimo iste oznake (i nije potrebno prepisivati njihove sadržaje).

Napomena: Ako je u dijagramu stanja neki signal naveden, a ne piše koja mu je vrijednost, podrazumijeva se da mu je vrijednost 1.

19. Modifikovati datapath i controlpath da bi se mogla implementirati instrukcija *nor*. Šta se dešava sa dijagramom stanja za implementiranje ove instrukcije?

Rješenje:

nor je instrukcija R tipa, pa ćemo se prilikom njene relizacije osloniti na instrukcije R tipa. Sve instrukcije R tipa se realizuju upotrebom ALU. Potrebno je znati strukturu jednobitne ALU. Jednobitna ALU realizuje 4 instrukcije (*and, or, add* i *slt*), te je MUX 4/1 koji ona upotrebljava iskorišćen u maksimalnom kapacitetu. Pošto tražena instrukcija *nor* nije realizovana, a potrebna nam je ALU za njenu realizaciju, proširićemo MUX 4/1, tako da na ulaz možemo dovesti izlaz iz osnovnog NILI (nor) kola. Kada proširujemo MUX 4/1 da bi dobili potrebni peti ulaz, jasno je da ćemo morati upotrijebiti MUX 8/1 (prvi sljedeći po veličini). To znači da ćemo imati tri neiskorišćena ulaza (što nam nije bitno), ali i da ćemo morati dodati još jedan selekcioni/kontrolni signal (MUX 4/1 je kontrolisan sa 2, a MUX 8/1 sa 3 selekciona/kontrolna signala). Na koji način će biti definisan dodatni kontrolni signal? Da bismo to riješili, pogledajmo najprije tabelu signala koji figurišu u ALU (ova tabela je sastavni dio dostupne literature):

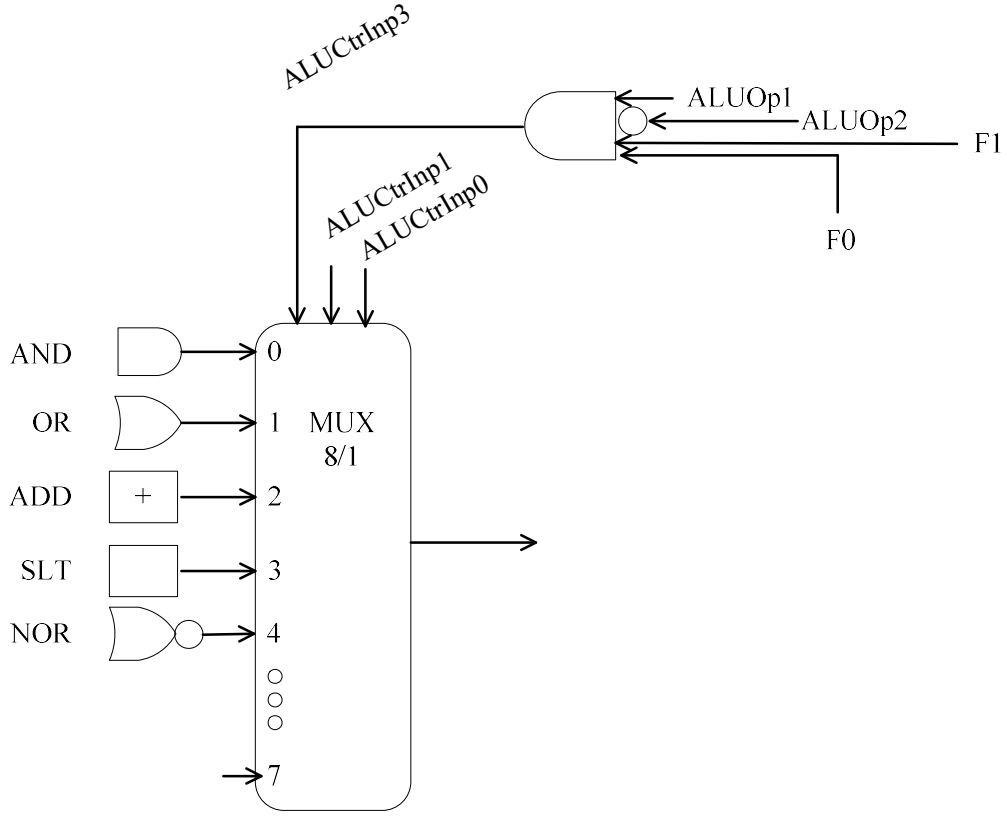
	ALUop		Function						ALU Control Inputs		
	ALUop1	ALUop2	F5	F4	F3	F2	F1	F0	2	1	0
lw, sw	0	0	x	x	x	x	x	x	0	1	0
beq	0	1	x	x	x	x	x	x	1	1	0
add	1	0	1	0	0	0	0	0	0	1	0
sub	1	0	1	0	0	0	1	0	1	1	0
and	1	0	1	0	0	1	0	0	0	0	0
or	1	0	1	0	0	1	0	1	0	0	1
slt	1	0	1	0	1	0	1	0	1	1	1

Pojašnjenje tabele:

Sa krajnje lijeve strane su navedene instrukcije koje koriste ALU. Uočićete to nijesu samo intrukcije R tipa. Posmatrajmo njihova *ALUop* polja: za sve instrukcije R tipa *ALUop* polja imaju vrijednost 10. To znači da će i naša instrukcija *nor* imati vrijednost 10 na poziciji ovih polja. *Function* polja su polja po kojem se sve instrukcije R tipa međusobno razlikuju, i postoje samo kod ovih instrukcija (primjetićete u tabeli da instrukcije *lw, sw, beq* nemaju vrijednosti na ovim poljima, te je stoga upisano x). Koje vrijednosti se upisuju na ovim pozicijama? Pogledajte kompletan spisak instrukcija. Uočićete npr. da *add* instrukcija ima funkcijsko polje 0x20. Oznaka 0x znači da je podatak nakon nje napisan u heksadecimalnom brojnom sistemu. Kada ovu vrijednost pretvorimo u binarni brojni sistem dobijamo: $24_{(16)}=10000_{(2)}$. Ovo su upravo biti iz gornje tabele. Instrukcija *and* ima funkcijsko polje 0x24, odnosno binarno 100100. Slično, naša instrukcija *nor* ima funkcijsko polje 0x27, odnosno njeni funkcijski biti su 100111. Preostale bite iz tabele, *ALU Control Inputs* bite tumačimo na sljedeći način: bit sa oznakom 2 označava da li u se u navedenoj operaciji dešava oduzimanje (1 ako se dešava, 0 u suprotnom); biti sa oznakama 1 i 0 su biti koji predstavljaju selekcione/kontrolne signale MUXa u posmatranoj ALU. Dakle, prikom proširenja MUXa sa MUX 4/1 na MUX 8/1, biće nam potreban još jedan ovakav bit (bit sa oznakom 3), kako bismo selekcione/kontrolne priključke setovali na 100, i omogući da ono što je na ulazu 4 (izlaz is nor kola) proslijedimo na izlaz MUXa 8/1, odnosno kompletne ALU.

	ALUop		Function						ALU Control Inputs			
	ALUop1	ALUop2	F5	F4	F3	F2	F1	F0	3	2	1	0
nor	1	0	1	0	0	1	1	1	1	0	0	0

Zašto smo bit sa oznakom 2 stavili na 0? Zato što u operaciji nor nema oduzimanja. Dakle, novododati bit 3 će uzeti vrijednost 0 za sve ostale instrukcije iz tabele, osim za našu instrukciju *nor* kada će imati vrijednost 1. Kako ćemo postići vrijednost 1 za novododato polje sa oznakom 3?

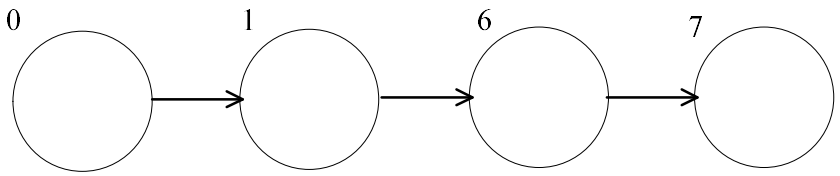


$$ALUCtrlInp3 = ALUop1 * \overline{ALUop2} * F5 * \overline{F4} * \overline{F3} * F2 * F1 * F0$$

Ili jednostavnije – možemo uočiti iz gornje tabele da ni jedna druga instrukcija (osim naše) nema na poljima F1 i F0 istovremeno 1. To znači da će ta kombinacija biti dovoljna da razlikuje novouvedeni signal od ostalih, odnosno:

$$ALUCtrlInp3 = ALUop1 * \overline{ALUop2} * F1 * F0$$

Dakle, proširenjem ALU i podešavanjem odgovarajućih signala smo postigli realizaciju instrukcije *nor*. Pri tome smo dodali samo jedan signal, *ALUCtrlInp3* koji nije sastavni dio dijagrama stanja. To znači da nam je dijagram stanja ostao isti kao za sve instrukcije R tipa, odnosno:



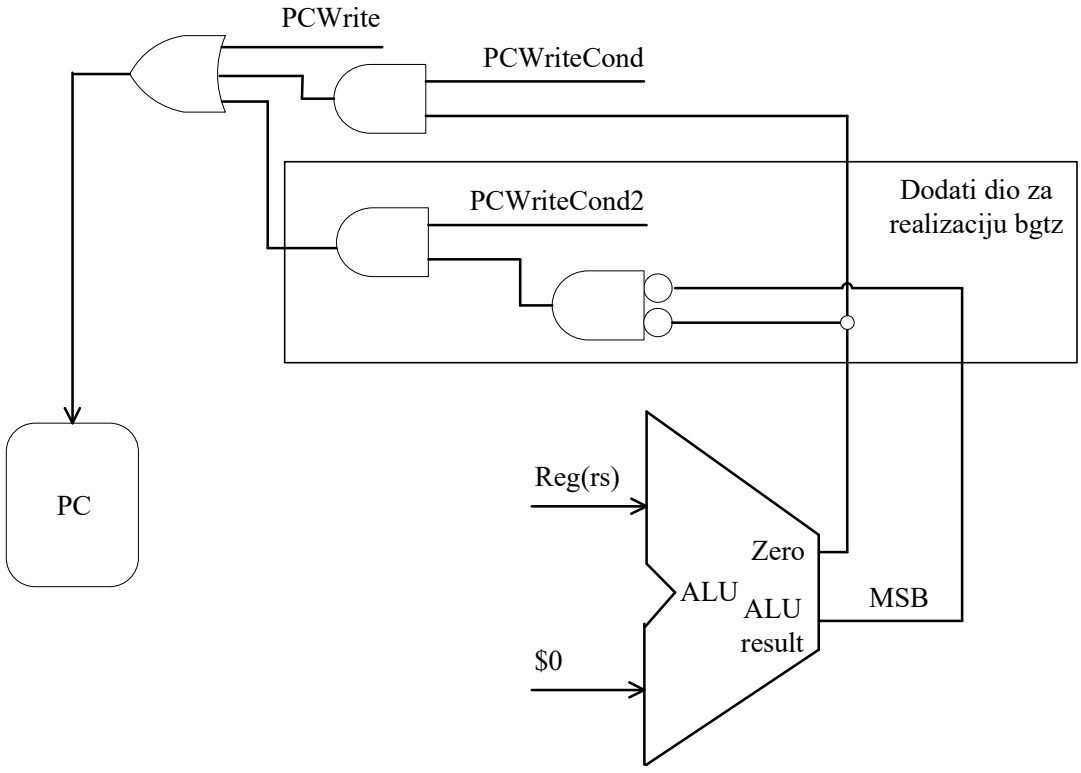
20. Modifikovati datapath i controlpath da bi se mogla implementirati instrukcija *bgtz*. Prikazati modifikaciju dijagrama stanja za implementiranje ove instrukcije.

Rješenje:

bgtz je branch instrukcija. Kod branch instrukcija se poređenje dvije vrijednosti obavlja tako što se one oduzmu (upotrebom ALU), a potom se provjeri rezultat. ALU ima dva izlaza (pogledajte šemu izučavane multitaktne realizacije). To su *ALU result* i *Zero*. *ALU Result* (očekivano) prikazuje rezultat operacije koju je obavljala ALU. Zero izlaz uzima jednu od moguće dvije vrijednosti: 1, ukoliko je rezultat oduzimanja jednak 0, i 0 ako rezultat oduzimanja nije jednak nuli. Kod instrukcija *beq* i *bne* za provjeru da li će doći do skoka na naznačenu labelu ili ne dovoljan nam je izlaz *Zero*. Međutim, za instrukciju *bgtz* (Be Greater Then Zero) to će nam biti potreban, ali ne i dovoljan podatak. Dakle, *Zero* signal će nam reći da li su dvije vrijednosti koje se porede jednake ili različite, ali ne i da li je prva vrijednost veća od nule. Za to ćemo morati iskoristiti *ALU result*. Međutim, nije nam potrebna kompletna vrijednost *ALU resulta*, već će nam biti dovoljan samo njen prvi bit (*MSB*). Zašto? Ukoliko je rezultat oduzimanja pozitivan, zasigurno će prva vrijednost biti veća, a ako to nije slučaj (odnosno ako je rezultat oduzimanja negativan), druga vrijednost će biti veća. Da sumiramo:

- Ako je *Zero*=0 → vrijednosti koje se porede su različite
- Ako je *MSB*=0 → prva vrijednost je veća od druge vrijednosti

Dakle, potrebni uslovi da bi prva vrijednost bila veća od druge vrijednosti (nule u našem slučaju) je da *Zero*=0 i da *MSB*=0. Ove uslove na neki način treba realizovati i unijeti u arhitekturu.



Imajući u vidu da je jedina realizovana branch naredba u izučavanoj arhitekturi beq oslonićemo se na nju, ali pri tome voditi računa da je izmjenama koje su nam potrebne za realizaciju *bgtz*, ne oštetimo/onemogućimo njen rad.

Na slici je prikazana promjena arhitekture (uokvireni dio se odnosni na našu instrukciju *bgtz*). Zašto smo uveli novi signal *PCWriteCond2*? (ime je potpuno proizvoljno) Signal *PCWriteCond* koristi

instrukcija *beq*, te je setovan samo kada se izvršava *beq*. Stoga, nijesmo mogli uzeti postojeći signal a da ne oštetimo realizovanu istrukciju. Novouvedeni signal *PCWriteCond2* za instrukciju *bgtz* ima istu ulogu kao *PCWriteCond* za instrukciju *beq*, a to je da je setovan kada je ispunjen ispitivani uslov.

Izmjene u dijagramu stanja će biti minimalne: umjesto *PCWriteCond* ćemo upisati naš novi signal, koji nam omogućava obavljanje nove istrukcije.