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.

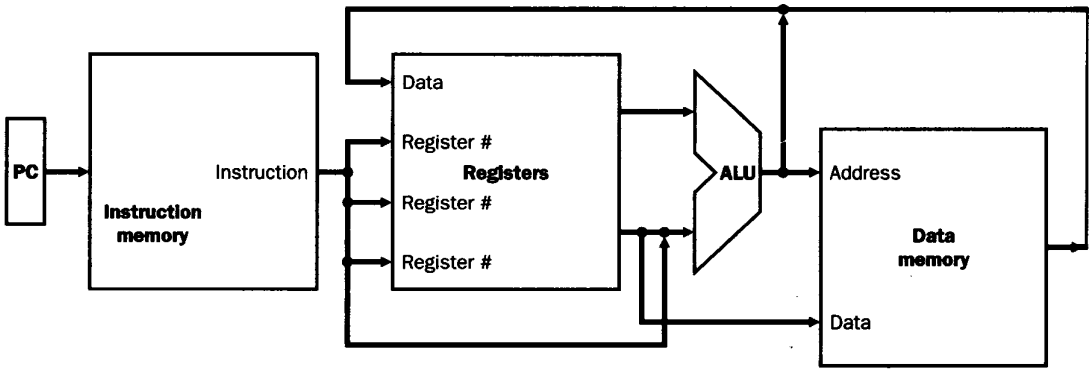
5.2 OPŠTI POGLED NA IMPLEMENTACIJU

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.

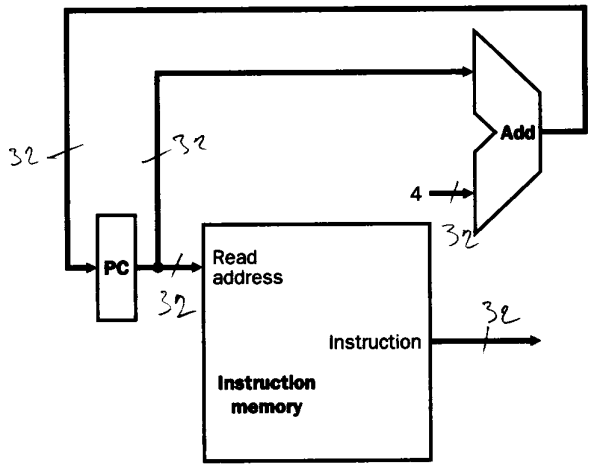
Primijetimo, odmah na strartu, da su drugi ulaz ALU i ulaz Data Registers jedinice podijeljeni (eng. *share*-ovani) za upotrebu izmedju različitih instrukcija:

- 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.

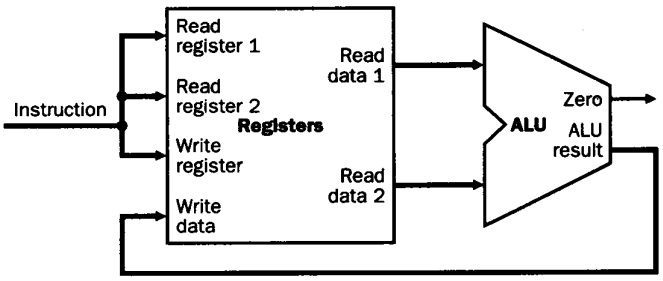
Primijetimo da je sadržaj PC registra 32-bitni, kao i kod pročitane instrukcije, u skladu sa razmatranjima iz poglavlja 3 (razmatramo, na koncu, 32-bitnu MIPS arhitekturu). Shodno 32-bitnom dizajnu sabirača, konstanta 4 koja se dovodi na njegov drugi ulaz takodje mora biti 32-bitna ($00...0100_{(2)}$). 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*)

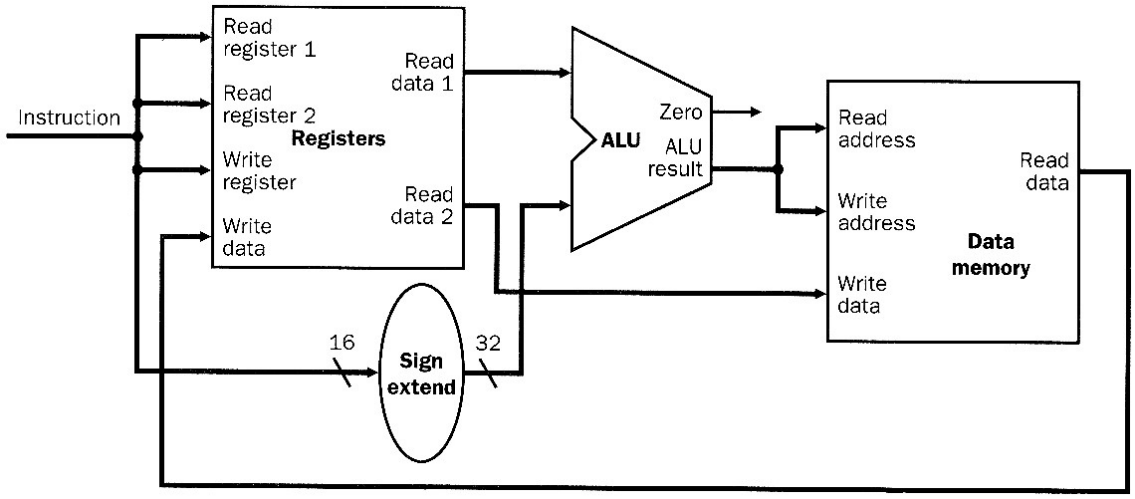
Mem[Reg(rs)+sign_extend(address)]←Reg(rt) (kod instrukcije *sw*)

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*.

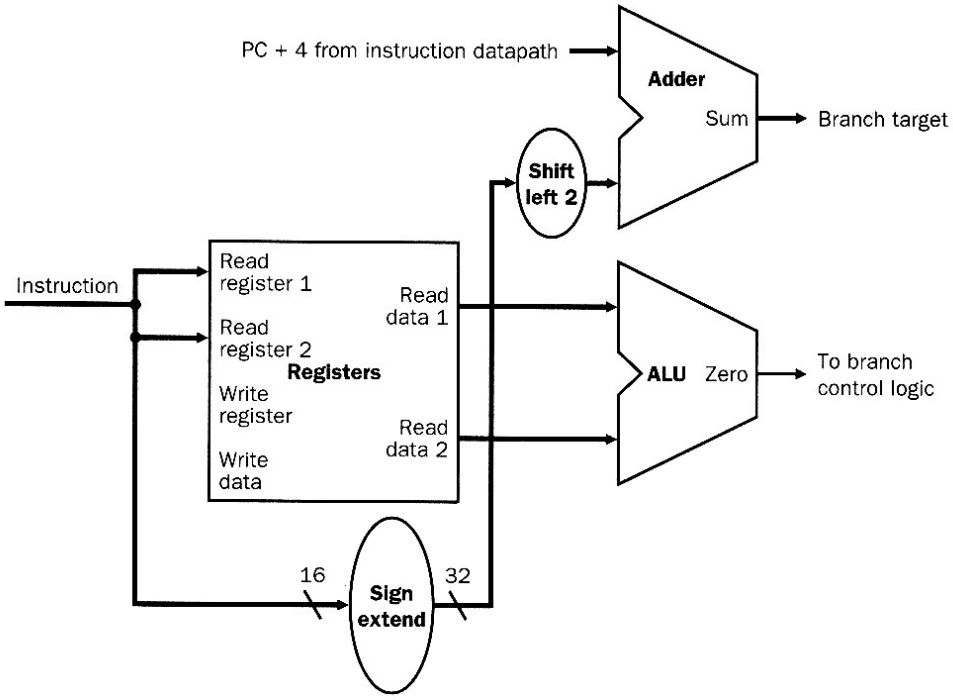
Shodno tome, u cilju implementiranja *lw*/*sw* instrukcija, na adresni ulaz Read register 1 Registers jedinice treba dovesti rs polje instrukcije, da bi se na Read register 1 izlazu iste jedinice dobio prvi operand nad kojim funkcioniše ALU. Na drugi ulaz ALU dovodi se 16-bitni sadržaj address-nog polja 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.

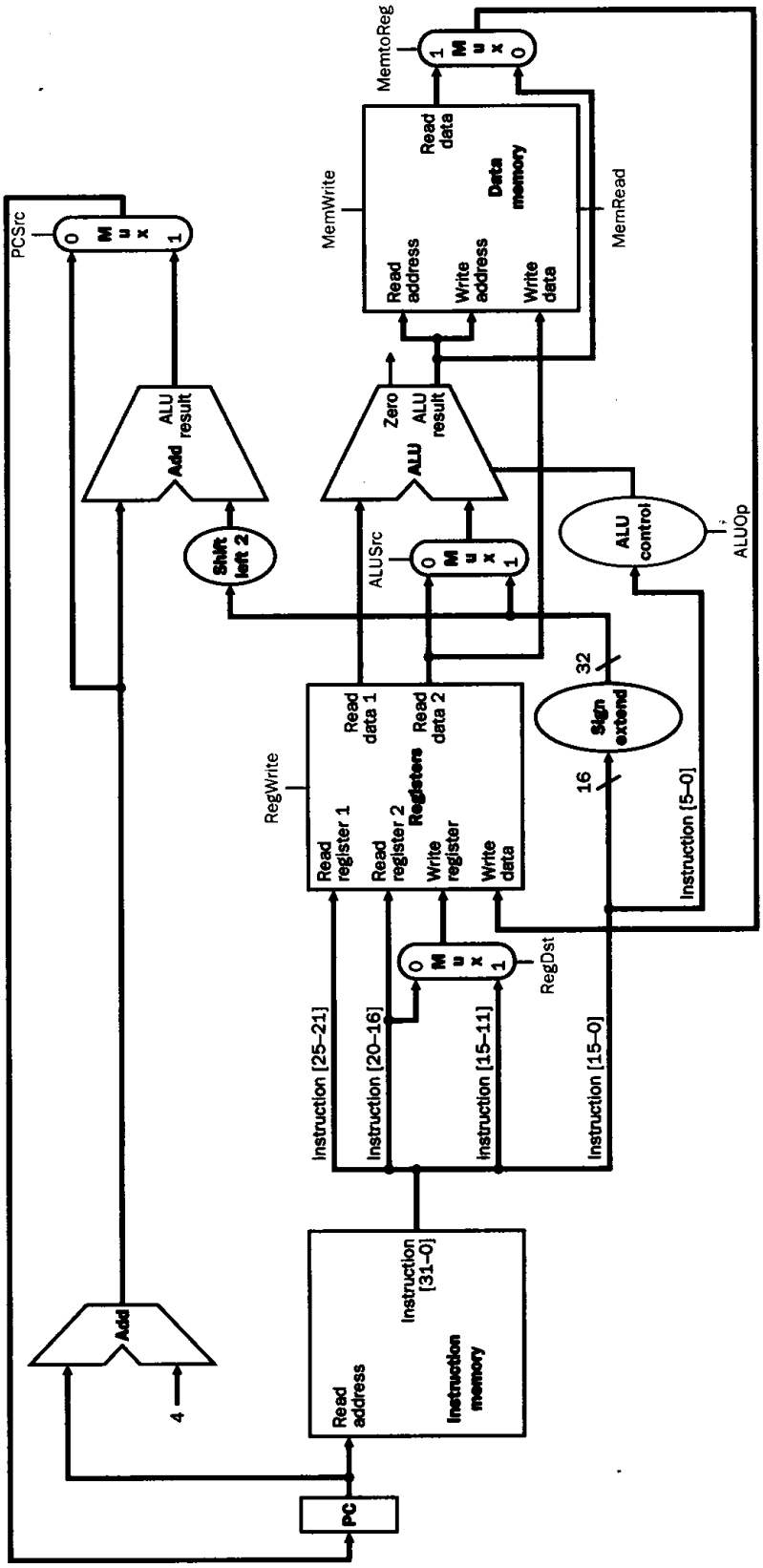
NAPOMENA 1: Tokom izvršavanja programa, instrukcije neće biti upisivane u Instruction memory, već samo čitane iz nje (instrukcije se upisuju u ovu jedinicu loaderom, i to prije početka izvršavanja programa). U Data memory, podaci mogu biti i čitani i upisivani za vrijeme izvršavanja instrukcija.

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

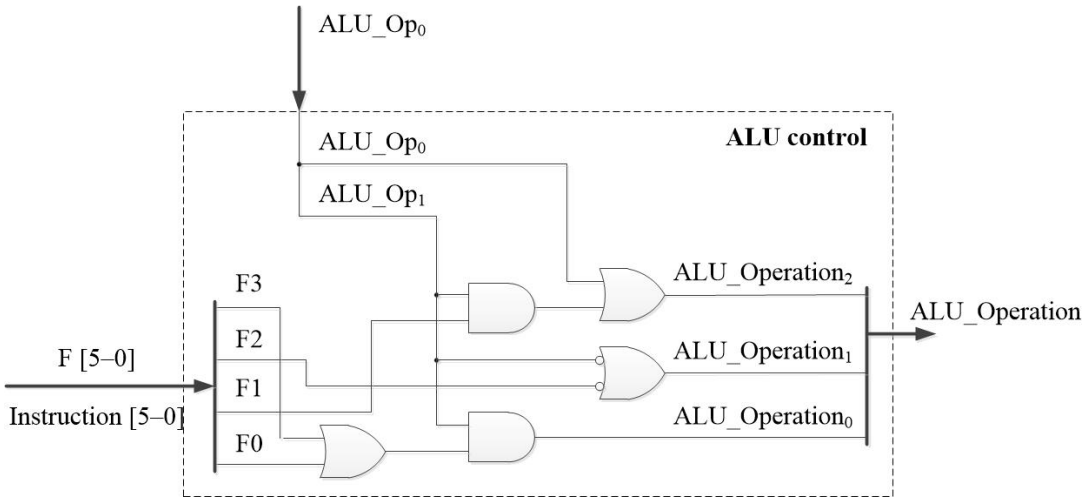
Na ovaj način, stvoreni su uslovi za pronalaženje minimalnih izraza za $ALU\_Operation_{(2,1,0)}$.

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,

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

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



Slika 7. ALU control jedinica.

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

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

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

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

5.4.2 Dizajniranje glavne kontrolne jedinice jednotaktne arhitekture

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

- Po 1 kontrolni signal na selekcionim ulazima svakog od 4 upotrijebljena multipleksora 2/1:
  RegDst signal multipleksora na Write register adresnom ulazu Registers jedinice,
  MemtoReg signal multipleksora na Write data ulazu Registers jedinice,
  ALUSrc signal multipleksora na drugom ulazu ALU,
  PCSrc signal multipleksora na ulazu PC registra.

- 3 kontrolna signala memorijskih elemenata:
  RegWrite kontrolni signal upisivanja podatka/rezultata ALU u Regitars jedinicu,

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

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

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

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

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

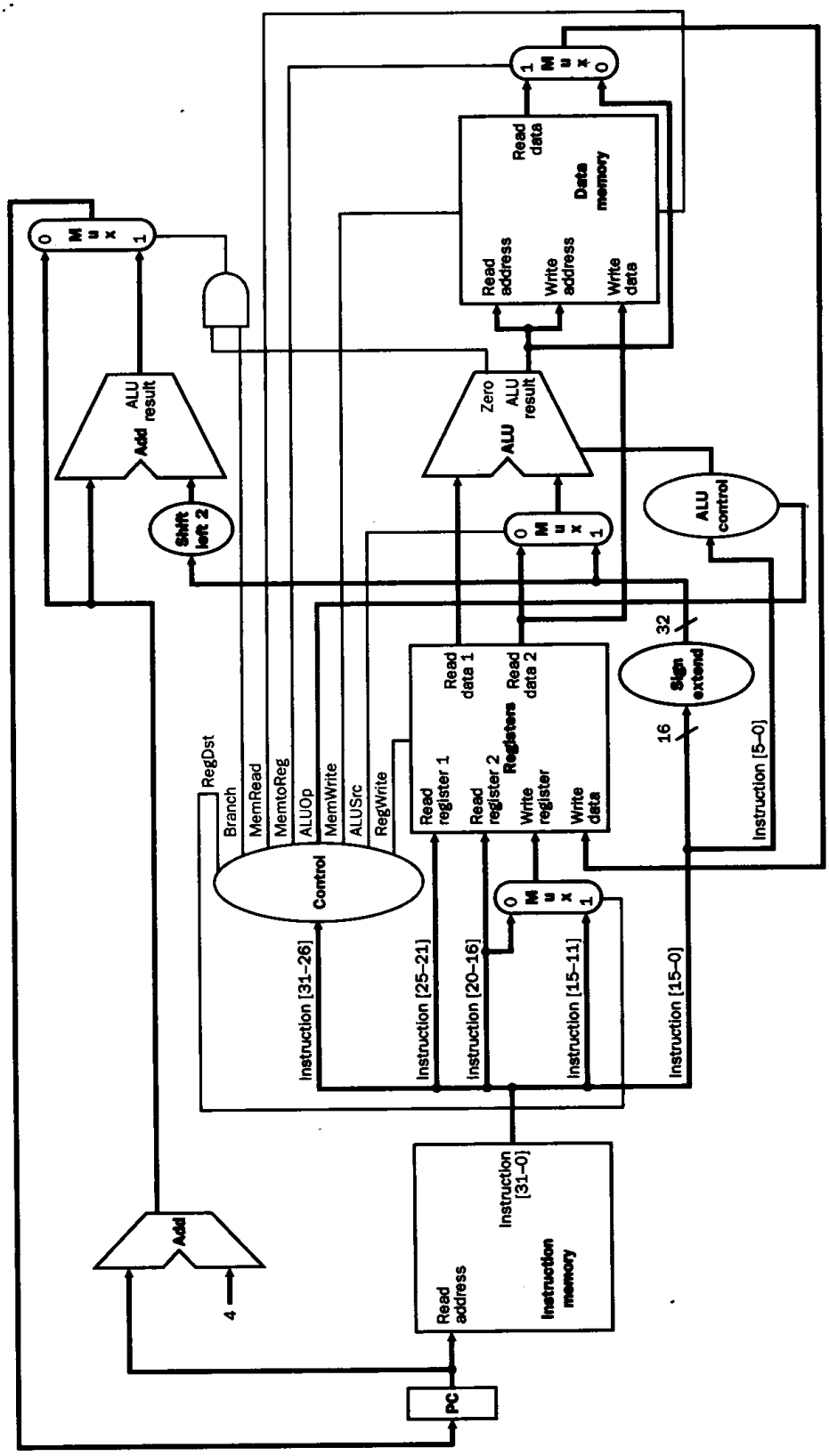
(*PCSrc* signal odlučuje da li će se u PC registar upisati ciljna adresa grananja Target ili PC+4).

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

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

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

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



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

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

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

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

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

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

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

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

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

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

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

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

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

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.

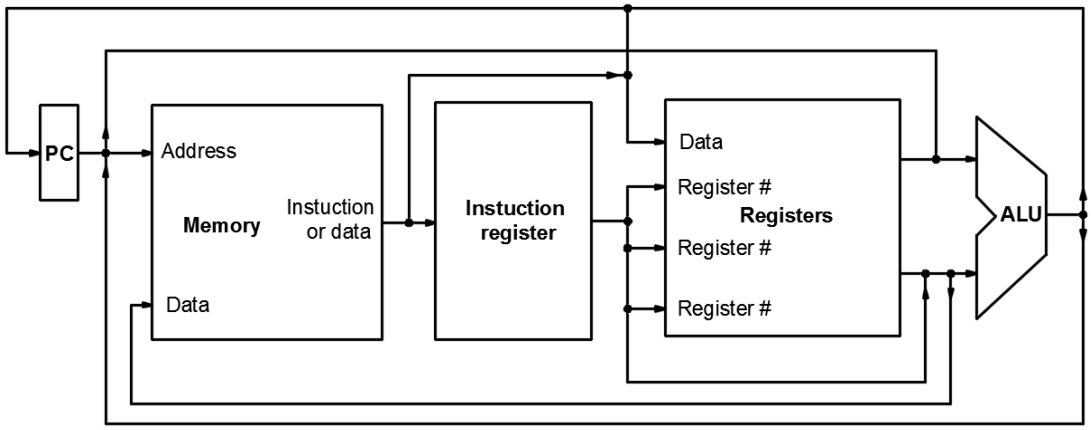
Shodno tome, multitaktna implementacija predstavlja praktično aplikabilan dizajn koji se upotrebljava u svim modernim računarskim sistemima. Stoga će joj u nastavku biti posvećena potpuna pažnja.

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

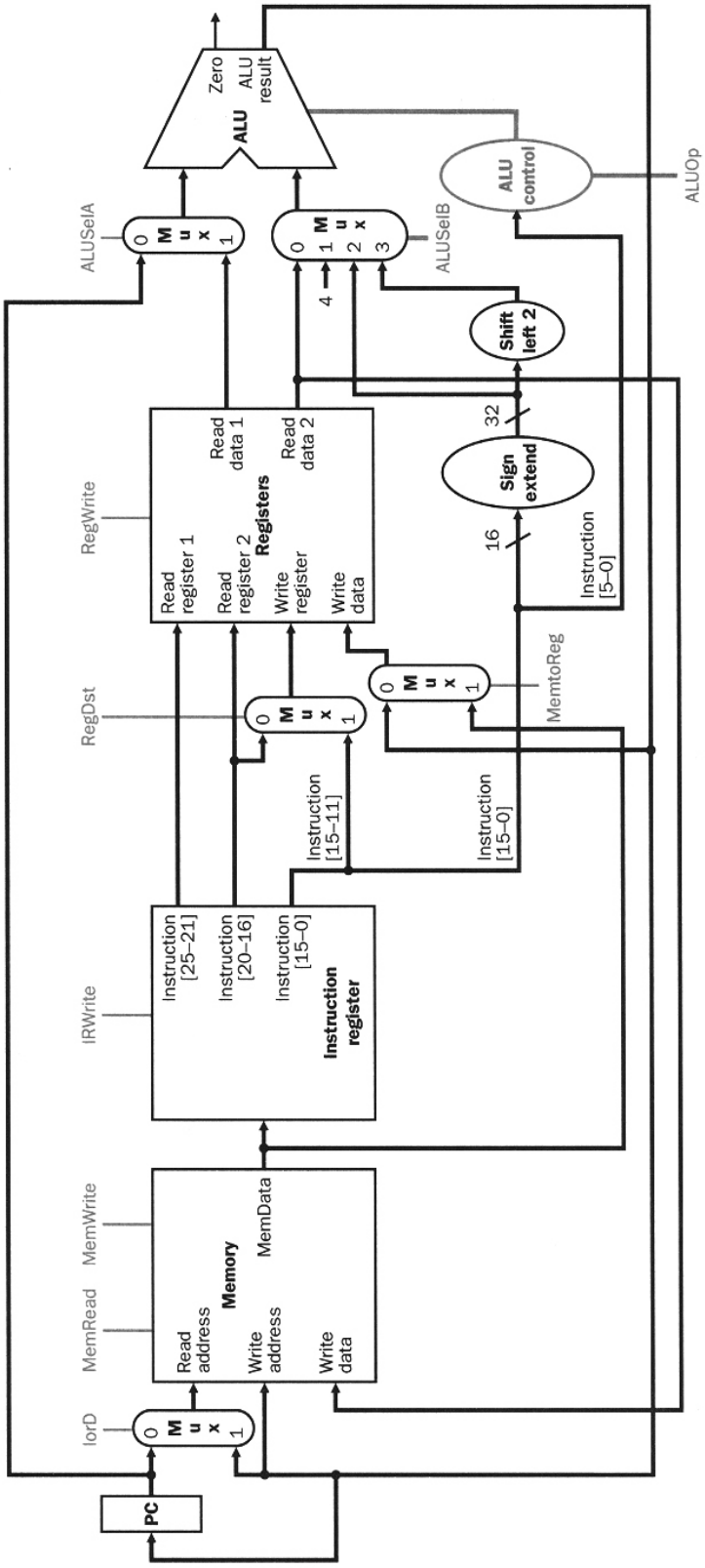
  - Na prvi ulaz ovog multipleksora dovodit će se sadržaj PC registra, u cilju obezbjedjivanja uslova za potencijalno uzimanje/čitanje instrukcije sa odgovarajuće lokacije,
  - Na drugi ulaz ovog multipleksora dovodit će se adresa izračunata od strane ALU (ALUOut), u cilju obezbjedjivanja uslova za potencijalno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

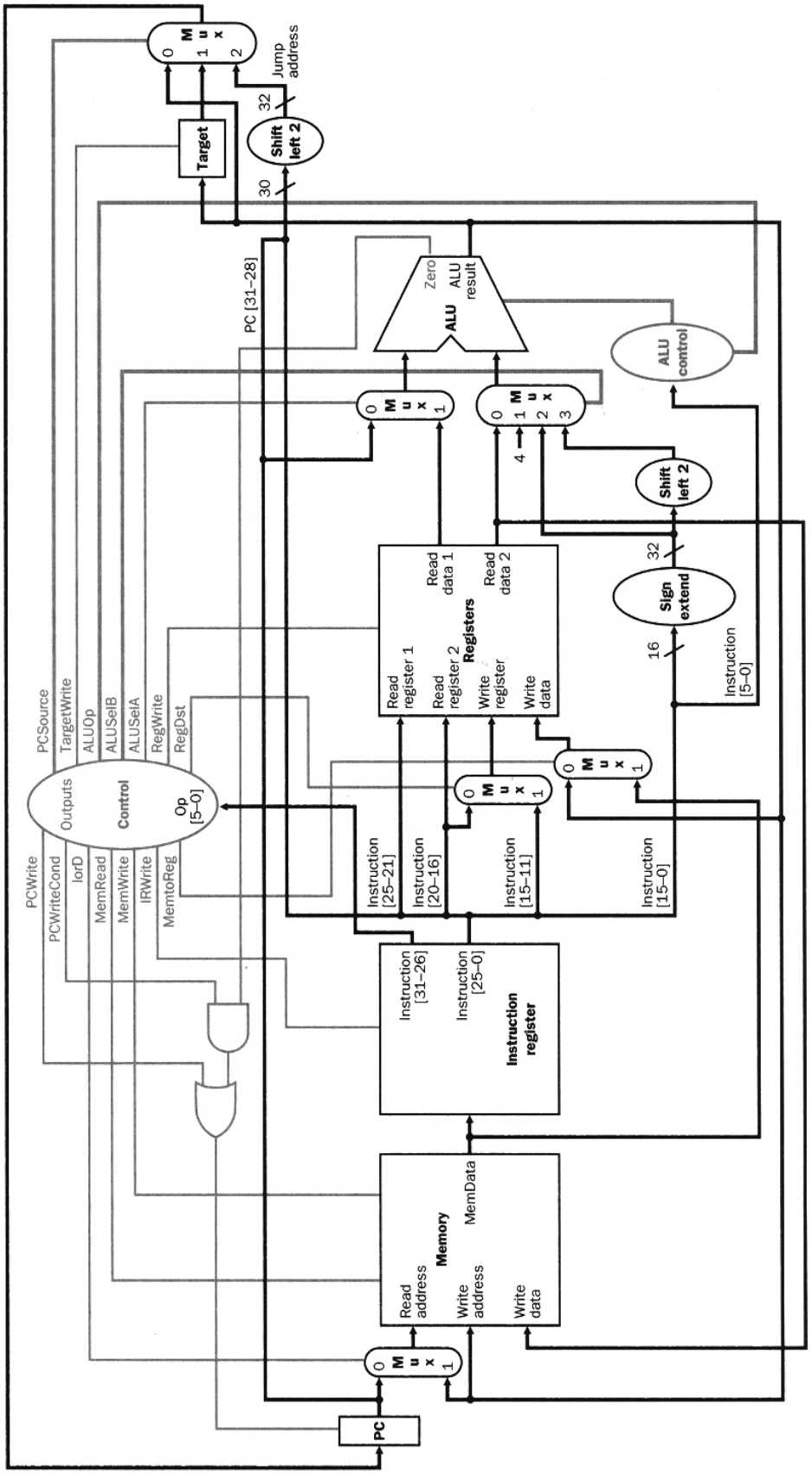
NAPOMENA 4: Na slici 11, prikazana je i realizacije instrukcije bezuslovnog skoka *j*. Realizuje se formiranjem 32-bitne adrese bezuslovnog skoka (Jump address na slici 11) i njenim dovodjenjem na ulaz br. 2 multipleksora koji se nalazi na ulazu PC registra (pogledaj napomenu 2 iz sekcije 5.5.1). Jump 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.