Vežba 3
Hardverska implementacija RISC-V procesora sa protočnom obradom

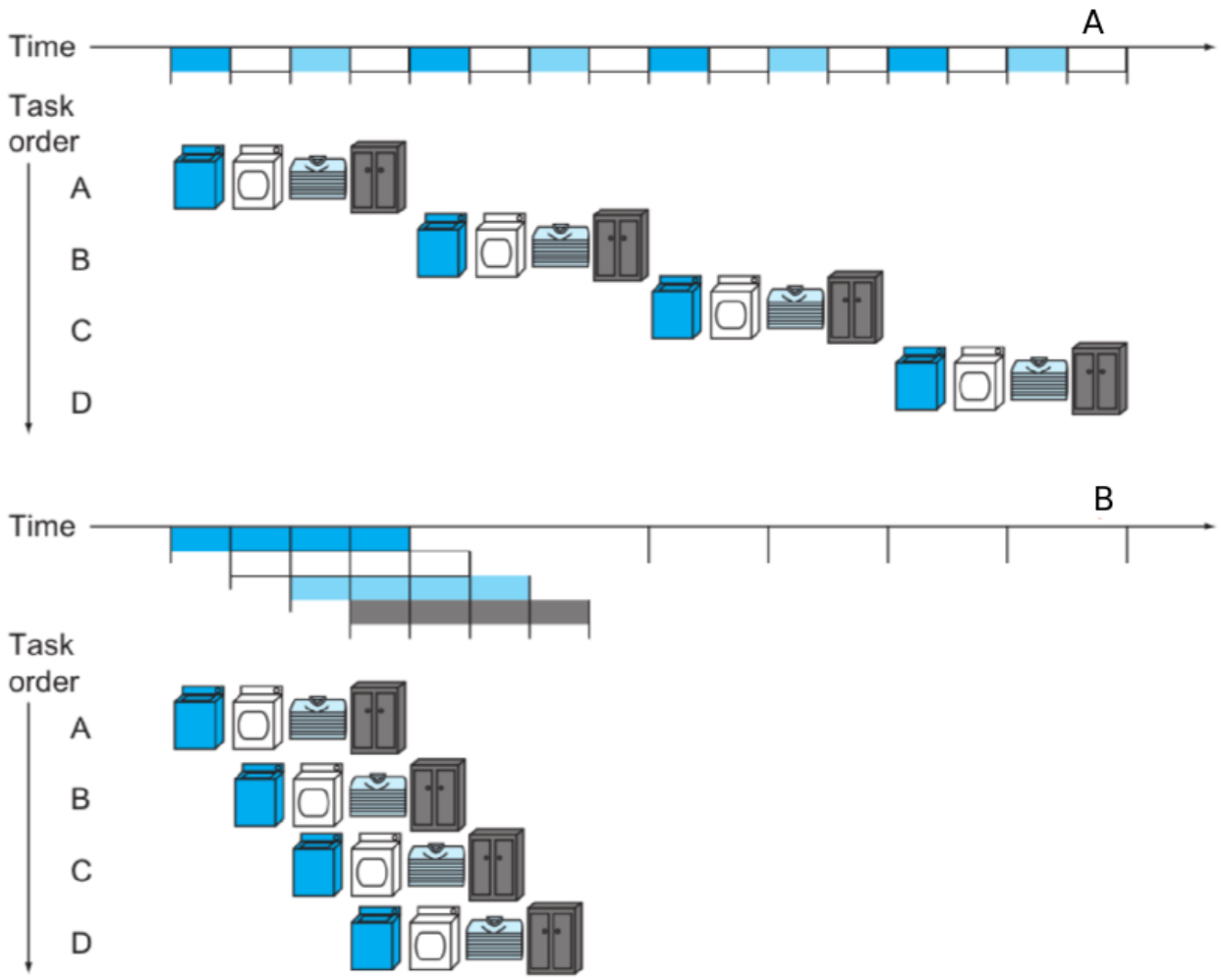
1 Uvod

Na prethodnim vežbama opisana je hardverska implementacija single cycle procesora koji podržava podskup instrukcija RISC-V arhitekture. Takva implementacija, iako funkcionalna, retko se koristi u modernim procesorima jer je neefikasna, odnosno doprinosi tome da procesor radi na niskoj frekvenciji. Razlog niske frekvencije proizilazi iz potrebe da se svaka instrukcija izvršava tačno jednu periodu takta, a perioda takta je direktno određena instrukcijom sa najvećim propagacionim kašnjenjem. U ovom slučaju to je load instrukcija, zato što koristi 5 funkcionalnih jedinica jednu za drugom (memoriju za instrukcije, registarsku banku, ALU, memoriju za podatke i opet registarstku banku).

Niska frekvencija single cycle implementacije sa fiksnim taktnim signalom se može smatrati prihvatljivom za ovako mali set instrukcija i istorijski gledano, rani kompjuteri sa malim setom instrukcija su koristili ovu tehniku implementacije procesora. Ali, ukoliko bi se set instrukcija koje procesor podržava proširio sa instrukcijama koje izvršavaju kompleksne operacije (floating point, division, ...) onda bi ova implementacija bila prespora. Iz ovog razloga u nastavku će biti analizirano korišćenje protočne obrade pri implementaciji procesora.

2 Osnovni principi protočne obrade

Protočna obrada je tehnika implementacije procesora kod koje se više instrukcija izvršava istovremeno. Kako bi se lakše razumeo koncept protočne obrade, u sledećem primeru će se napraviti analogija pranja veša i izvršavanja instrukcija od strane procesora.



Slika 1. Analogija pranja veša i protočne obrade

Pranje veša se sastoji iz sledećih faza:

1. Stavljanje prljavog veša u mašinu
2. Stavljanja mokrog veša u sušilicu nakon pranja.
3. Slaganja veša nakon sušenje.
4. Kada je slaganje završeno, reći cimeru da skloni veš

Gornja slika označena sa A, ilustruje pranje veša, pri čemu se sa narednom turom kreće tek kada prva prođe kroz sve faze. Ovde je očigledno da se prilikom korišćenja jednog resursa ostala 3 ne koriste (kada se veš slaže, mašina, sušilica i cimer se ne koriste). Slika označena sa B ilustruje efikasniji način korišćenja resursa, odnosno ilustruje princip protočne obrade gde se svi resursi paralelno koriste. Sa slike se može primetiti da čim se završi pranje veša i njegovo stavljanje u sušilicu, druga tura prljavog veša se stavlja na pranje. Kada se prva tura osuši, ona se prebacuje u fazu slaganja, druga tura opranog veša se stavlja u sušilicu i treća tura pljavog veša se stavlja u mašinu. Nakon što je prva tura veša složena cimer je preuzima, druga tura je osušena i prelazi u fazu slaganja, treća tura veša se prebacuje u sušilicu i četvrta tura pljavog veša može se staviti u mašinu. U ovom trenutku sve četiri faze protočne obrade rade paralelno i dokle god imamo razdvojene resurse za svaku fazu posao može biti paralelizovan.

Paralelizam u protočnoj obradi jeste glavni razlog njene veće efikasnosti, odnosno u istom intervalu vremena opraće se veća količina veša u odnosu na pranje bez protočne obrade. Odavde sledi da protočna obrada povećava propusnu moć, što znači da se vreme pranja, sušenja, slaganja i odlaganja jedne ture veša neće smanjiti, ali vreme potrebno da se sve ture veša operu biće smanjeno.

Ukoliko sve faze zahtevaju istu količinu vremena da se izvrše i ako postoji dovoljno posla, onda će ubrzanje biti jednako umnošku broja faza protočne obrade. U ovom slučaju taj broj je četiri (pranje, sušenje, slaganje i odlaganje).

Isti principi protočne obrade ilustrovani na prethodnom primeru mogu da se primene i pri implementaciji procesora. Ukoliko bismo posmatrali kako *single cycle* procesor izvršava instrukciju, to bi bilo jako slično pranju veša bez protočne obrade. U tom primeru pranje veša se sastoji iz više faza, te slično razmišljanje možemo primeniti i na izvršavanje instrukcijie, odnosno da se ono obavlja kroz više faza. Onda, zašto čekati da instrukcija prođe kroz sve faze, zašto ne bismo prihvatili sledeću instrukciju kada prethodna prođe kroz trenutnu fazu? Sa tom idejom implementiraćemo procesor sa protočnom obradom i prvo od čega krećemo jeste podela izvršavanja instrukcija *single cycle* procesora na faze:

- *Instruction fetch*: Faza prihvata instrukcije iz memorije.
- *Instruction decode:* Faza čitanje iz registarske banke i dekodovanja instrukcije.
- *Execute*: Faza u kojoj ALU izvršava određenu operaciju u zavisnosti od prihvaćene instrukcije.
- *Memory*: Faza pristupa memoriji za podatke.
- *Write back:* Faza upisa rezultata izvršavanja instrukcije u registarsku banku.

Ako bi se posmatrao prethodni primer ubrzanje koje bi trebalo da se ostvari je petostruko. No, to je idealan slučaj, u kome se sve faze izvršavaju istu količinu vremena i vreme potrebno da se prođe kroz sve faze je jednako vremenu da se izvrši jedna instrukcija u *single cycle* implementaciji. Sledeći primer ilustruje situaciju u kojoj nije sve idealno.

2.1 Poređenje procesora sa i bez protočne obrade

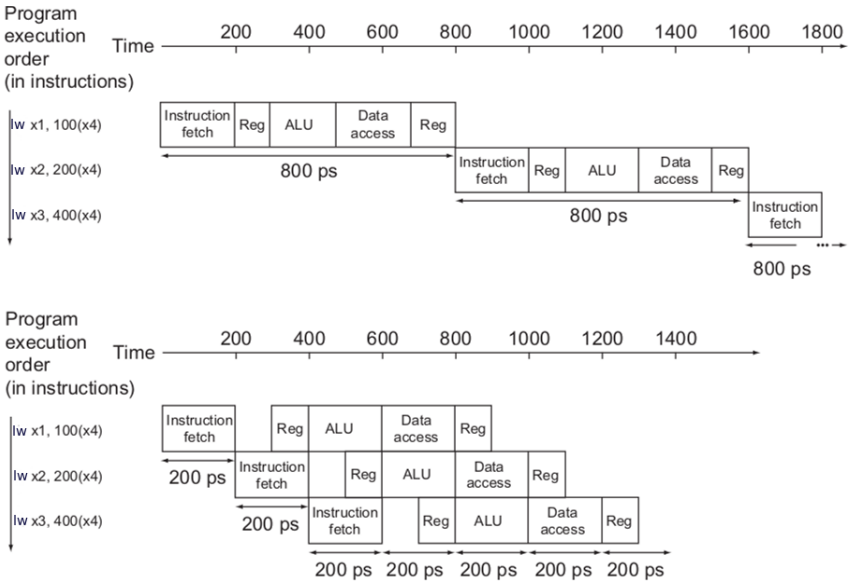
U ovom primeru će se porediti implementacija procesora sa protočnom obradom i *single cycle* procesora, pri čemu obe implementacije podržavaju sledeći set instrukcija :*lw, sw, add, sub, and, or, beq.* Sledeća tabela daje informaciju o hipotetičkim vremenima neophodnim da se svaka instrukcija izvrši:

Klasa instrukcije	*Intruction fetch phase*	*Instruction decode phase*	*Execute phase*	*Memory phase*	*Write-back phase*	*Ukupno vreme*
lw	*200 ps*	*100 ps*	*200 ps*	*200 ps*	*100 ps*	*800 ps*
sw	*200 ps*	*100 ps*	*200 ps*	*200 ps*	*0*	*700 ps*
R-format	*200 ps*	*100 ps*	*200 ps*	*0*	*100 ps*	*600 ps*
beq	*200 ps*	*100 ps*	*200 ps*	*0*	*0*	*500 ps*

Slika 2. Hipotetička vremena izvršavanja instrukcija po fazama protočne obrade

Uzimajući ova vremena u obzir, dolazi se do zaključka da perioda takta *single cycle* implementacije treba da bude 800ps, jer je perioda direktno određena vremenom propagacije instrukcije koja se najduže izvršava (lw instrukcija u ovom slučaju).

Kod implementacije sa protočnom obradom, perioda takta će biti srazmerna propagacionom vremenu faze protočne obrade koja se najduže izvršava (*instruction fetch*). Uzimajući ovo u obzir dolazi se do zaključka da je procesoru sa protočnom obradom potrebno 1000ps (5 perioda takta) da izvrši jednu instrukciju, a *single cycle* procesoru treba 800ps. Sledeća slika to ilustruje:



Slika 3. Poređenje rada *single cycle* procesora i procesora sa protočnom obradom

Na osnovu prethodno opisanog može se doći do zaključka da single *cycle procesor* radi brže, što i jeste tačno ako se samo gleda vreme potrebno za izvršenje jedne instrukcije. No, uvođenjem protočne obrade istovremeno se povećala i propusna moć, odnosno broj instrukcija koje mogu paralelno da se izvršavaju. Na primer, procesor bez protočne obrade sa prethodno pomenutim hipotetičkim vremenima će izvršiti tri instrukcije za 2400ps, a procesor sa protočnom obradom će to uraditi za 1400ps. Opet, ubrzanje nije veliko (samo 1,71 puta brže), u odnosu na ubrzanje pomenuto na početku, koje je jednako umnošku broja faza protočne obrade (5 puta brže u ovom primeru). Ali, ukoliko se izvrši mnogo veći broj instrukcija, na primer 1000000, onda će poboljšanje biti sledeće:

$$(1) Ubrzanje = \frac{Broj\ instrukcija * T_{single\ cyle\ perioda}}{(Broj\ instrukcija - 1) * (T_{pipeline\ perioda}) + T_{popunjavanja\ pipeline-a}}$$

$$(2)\ Ubrzanje = \frac{1000000 * 800ps}{(1000000 - 1) * 200ps + 1000} = 3.99 \simeq 4$$

U prvoj jednačini parametri imaju sledeće značenje:

- Broj instrukcija: predstavlja ukupan broj instrukcija koji procesor treba da izvrši
- $T_{\text{single cycle perioda}}$: predstavlja periodu procesora bez protočne obrade.
- $T_{\text{pipeline perioda}}$: predstavlja periodu procesora sa protočnom obradom
- $T_{\text{popunjavanja pipeline-a}}$: vreme popunjavanja pipeline-a na početku izvršavanja programa.

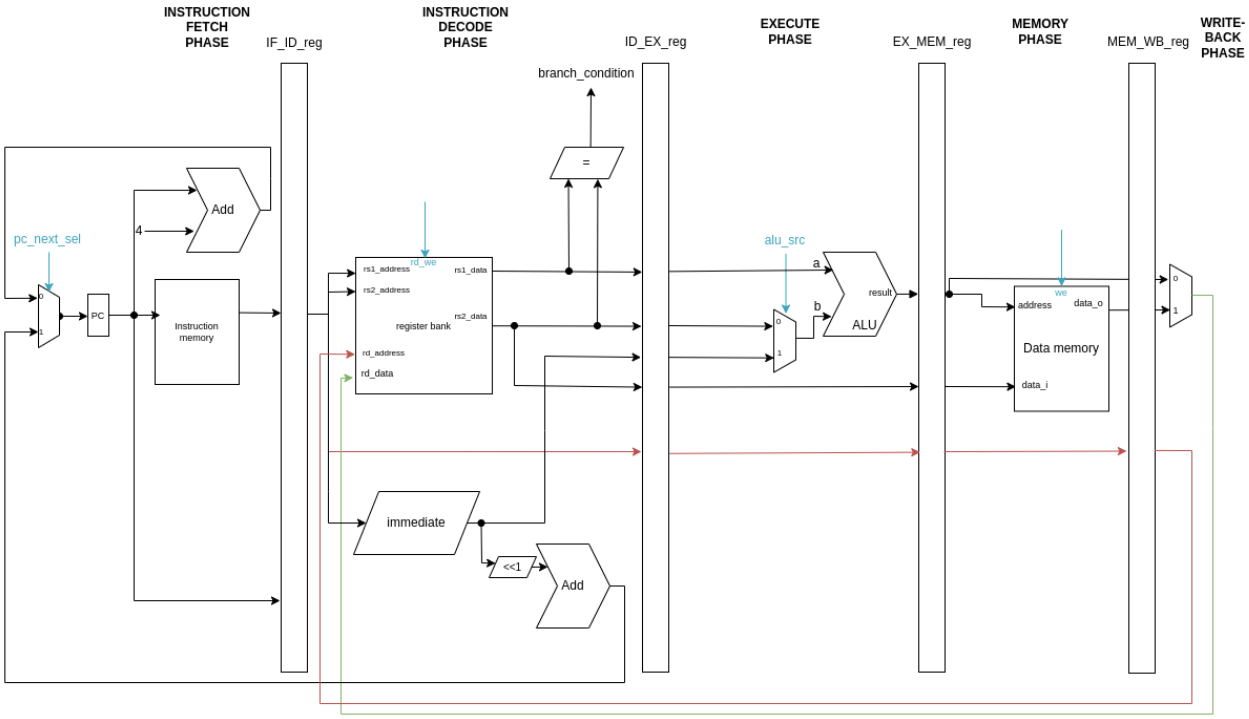
Iz prethodnog primera može se videti da što se veći broj instrukcija izvršava to je poboljšanje veće, a kako u realnoj situaciji procesor izvršava mnogo više od milion instrukcija, zaključuje se da protočna obrada umnogome ubrzava rad.

U nastavku će biti razmatrano kako da se *single cycle* procesor, implementiran na prethodnim vežbama, proširi do procesora sa protočnom obradom. Takođe biće razmatrani problemi koji nastaju uvođenjem protočne obrade, koji mogu dodatno da uspore rad procesora.

3 Uvođenje protočne obrade u *single cycle* procesor.

Na slici 4 prikazan je *single cycle datapath* koji je pomoću registara podeljen na na 5 faza protočne obrade:

- IF - *Instruction fetch.*
- ID - *Instruction decode.*
- EX - *Execute.*
- MEM - *Data memory access.*
- WB - *Write-back.*

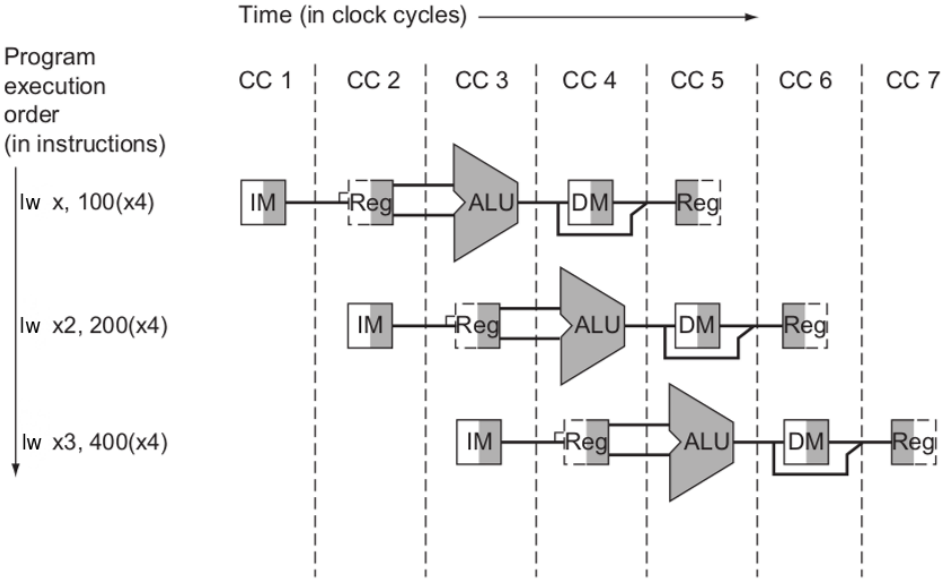


Slika 4. Podela *datapath* celine na 5 faza izvršavanja

Na ovaj način, umetanjem registara, onemogućeno je da procesor izvrši instrukciju u jednom taktu, već je neophodno da ona prođe kroz 5 faza protočne obrade.

Registri dele faze izvršavanja tako što omogućavaju da se rezultati pojedinačnih faza sačuvaju u njima na rastućoj ivici takta. Osim dodavanja registara, neophodno je kroz sve faze proslediti adresu registra u koji se instrukcija upisuje (naznačeno crvenom linijom na slici 4). Ovo je uradjeno kako bi rezultat izvršavanja instrukcije (zelena linija) i adresa registra (crvena linija) istovremeno stigli do registarske banke.

Sledeća slika ilustruje izvršavanje više instrukcija u procesoru sa protočnom obradom. Za svaku instrukciju je prikazano koje resurse koristi u zavisnosti od faze protočne obrade.



Slika 5. Izvršavanje instrukcija po fazama protočne obrade

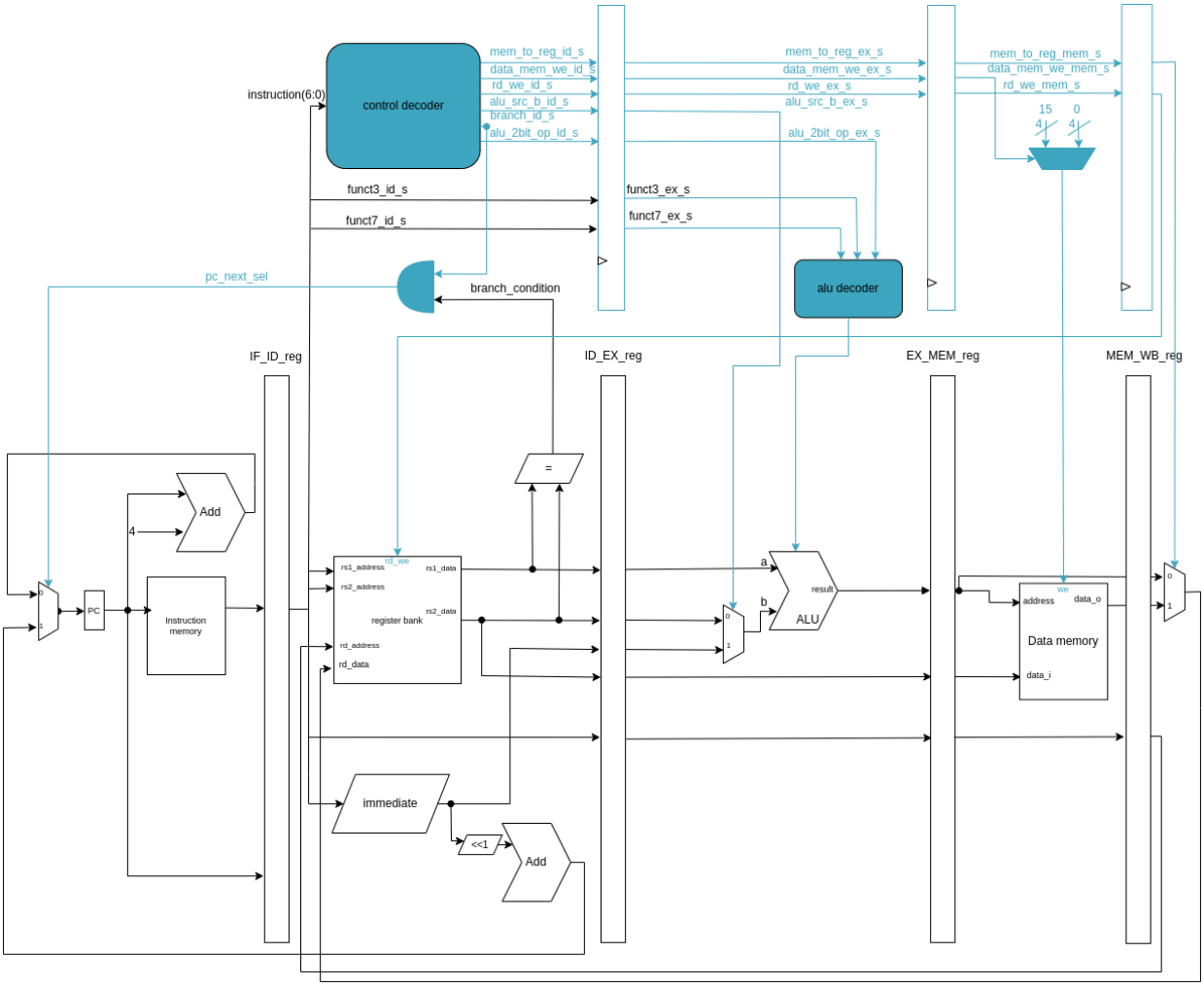
Ova reprezentacija će biti korištena u nastavku zbog lakšeg opisa rada procesora sa protočnom obradom. Na slici 5 svaka od faza protočne obrade je predstavljena sa najbitnijom funkcionalnom jedinicom koju koristi:

- IF faza je označena sa “IM” - memorija za instrukcije (*eng. Instruction memory*).
- ID faza je označena sa “Reg*”* - registarska banka.
- EX faza je označena sa “ALU” - aritmetičko logička jedinica.
- MEM faza je označena sa “DM” - memorija za podatke (*eng. Data memory*).
- WB faza je takođe označena sa “Reg” - registarska banka.

Na prethodnim vežbama je pomenuto da se upis u registarsku banku vrši na opadajuću ivicu takta, a čitanje je asinhrono. Taj način rada je predstavljen senčenjem leve polovine “Reg” faze ukoliko se vrši upis, odnosno desne ukoliko se vrši čitanje(jer se na rastuću ivicu upisuje u registar EX faze). Ako se pogleda “Reg” na slici 5, može se videti da je u drugom tatku (CC2) njegova desna strana osenčena, a u petom tatku

(CC5) osenčena leva strana. To znači da se u taktu CC2 vršilo čitanje a u taktu CC5 upis.

Da bi implementacija bila kompletna neophodno je ubaciti kompontente koje pripadaju *controlpath* celini i one su na sledećoj slici naznačene plavom bojom:



Slika 6. Proširenje procesora sa protočnom obradom komponentama koje pripadaju *controlpath* celini

Kao što je objašnjeno na prethodnim vežbama, kontrolni signali određuju način rada komponenti koje pripadaju *datapath* celini. Da bi to uradili na ispravan način, neke od njih je neophodno sprovesti kroz faze protočne obrade kako bi pravovremeno generisali kontrolne signale. Na primer, ukoliko se prihvati instrukcija “I” tipa, u ID fazi će kontrolni signal *alu_src_b_id_s* biti podignut na visok logički nivo. Kako on treba da kontroliše multiplekser koji se nalazi u EX fazi, njega prvo treba provući kroz ID_EX registar a zatim ga sprovesti do multipleksera.

Takođe *funct3* i *funct7* polja instrukcije treba sprovesti do *alu_decoder* jedinice kako bi ona ispravno generisala kontrolni signal koji upravlja ALU-om. Pošto se ALU nalazi u EX fazi, *funct3* i *funct7* moraju da se provuku kroz ID_EX registar, jer *alu_deocoder* komponenta treba da generiše kontrolni signal pravovremeno.

Da bi se prethodno opisano postiglo, *controlpath* celina je podeljena na faze, umetanjem registara (s tim što ona ima jednu fazu manje) i kontrolni signali su kroz njih provučeni. Ista stvar je mogla biti urađena i bez podele *contropath* celine na faze, tako što bi se kontrolni signali provukli kroz registre unutar *datapath* celine. Prva opcija omogućava jednostavniji interfejs *datapath* i *controlpath* celina prilikom implementacije u vhdl-u i zato je ona odabrana.

Ova unapređenja možda deluju kao da su dovoljna da bi procesor sa protočnom obradom ispravno funkcionisao, ali u nastavku će biti izloženi problemi koji se javljaju.

3.1 Hazardi u protočnoj obradi

3.1.1 Hazardi podataka kod aritmeticko logickih instrukcija ili instrukcija R tipa

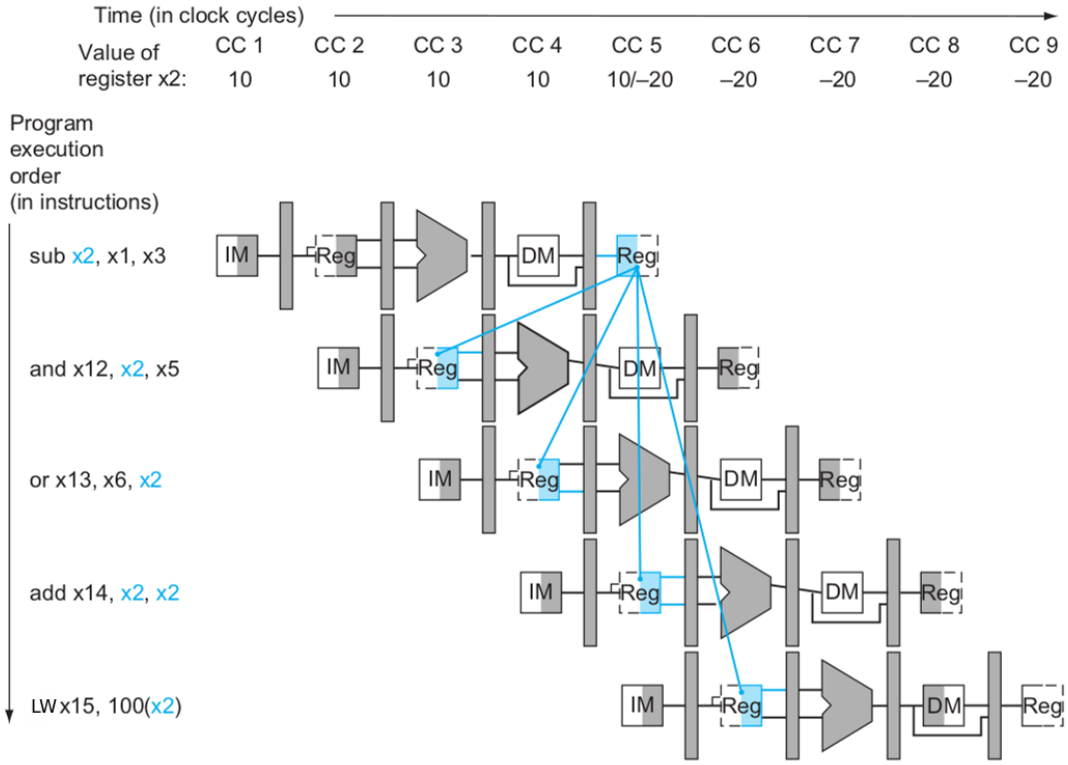
Uvođenjem protočne obrade povećana je propusna moć, ali su se pojavili i određeni problemi. Ti problemi nastaju usled zavisnosti između instrukcija koje se nalaze unutar faza protočne obrade. Prvi problem koji će biti analiziran jeste hazard podataka i sledeći primer ilustruje kada se on javlja:

```
sub x2, x1, x3
and x12, x2, x5
or x13, x6, x2
add x14, x2, x2
sw x15, 100(x2)
```

Slika 7. Primer hazarda podataka

Na slici 7 prikazane su 4 instrukcije R tipa i jedna instrukcija S tipa (*sw*). Svaka od instrukcija koristi određene registre iz registarske banke kako bi izvršila sebi specifičnu operaciju. Kod *single cycle* procesora izvršavanje ovih instrukcije ne bi predstavljalo problem, jer se sa izvršavanjem naredne instrukcije kreće tek kada se izvrši prethodna, odnosno nema zavisnosti. Kod procesora sa protočnom obradom instrukcije se izvršavaju paralelno i problem se javlja ukoliko izvršavanje određene instrukcije zavisi od rezultata prethodne. U kodu sa slike 7 prvo se izvršava “sub” instrukcija koja menja sadržaj x2 registra, a sledeća instrukcija je “and” koja čita sadržaj x2 registra. Ovde nastaje problem jer u trenutku kada “and” instrukcija čita x2 registar (ID faza), njegova vrednost nije ažurirana od strane “sub” instrukcije koja je prethodila,

jer se “sub” nalazi u EX fazi. Odavde sledi da će “and” instrukcija pročitati pogrešnu vrednost iz x2 registra i sledeća slika to ilustruje:



Slika 8. Sadržaj u fazama protočne obrade prilikom izvršavanja koda sa slike 8

Na slici 8 je naznačeno da “and” instrukcija u taktu CC3 zahteva informaciju koja će biti ažurirana tek u taktu CC5. Slična stvar važi i za “or” instrukciju, ona u tatku CC4 čita registar x2, a njegov sadržaj će biti ažuriran od strane “sub” instrukcije tek u taktu CC5. Prva instrukcija kod koje se neće javiti problem jeste “add”, jer u trenutku kada ona čita sadržaj iz registra x2 (CC5), “sub” instrukcija je već ažurirala njegovu vrednost. To je moguće jer je registarska banka napravljena tako da se u nju upisuje na opadajuću ivicu takta, a čita na rastuću.

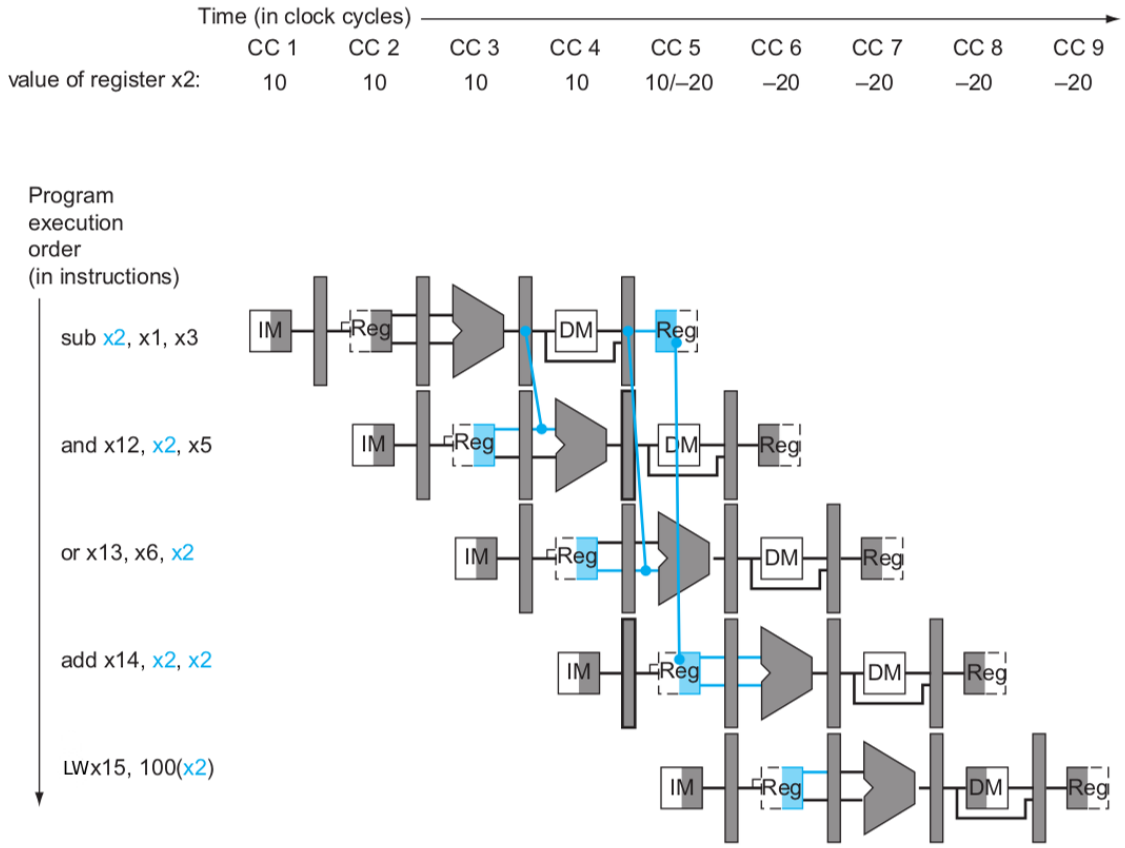
Jedan od načina da se reši ovaj problem jeste da se sačeka da “sub” instrukcija ažurira vrednost registra x2 i tek onda da se nastavi sa izvršavanjem ostalih instrukcija. Sledeći primer to ilustruje:

```
sub x2, x1, x3
nop
nop
and x12, x2, x5
or x13, x6, x2
add x14, x2, x2
sw x15, 100(x2)
```

Slika 9. Razrešavanje hazarda umetanjem nop instrukcija

Umetanjem dve “nop” instrukcije (tzv. Prazne instrukcije), odlaže se izvršenje “and” instrukcije za dva takta, što je dovoljno da “sub” ažurira vrednost u x2 registru. Očigledna mana ovakvog rešenja je zadržavanje daljeg izvršvanja programa kada god se desi sličan scenario, što diretkno smanjuje propusnu moć procesora.

Način da se ovaj problem reši jeste implementiranje tzv. jedinice za prosleđivanje podataka (*eng. Forwarding unit*). Njen smisao je ilustrovan na sledećoj slici:

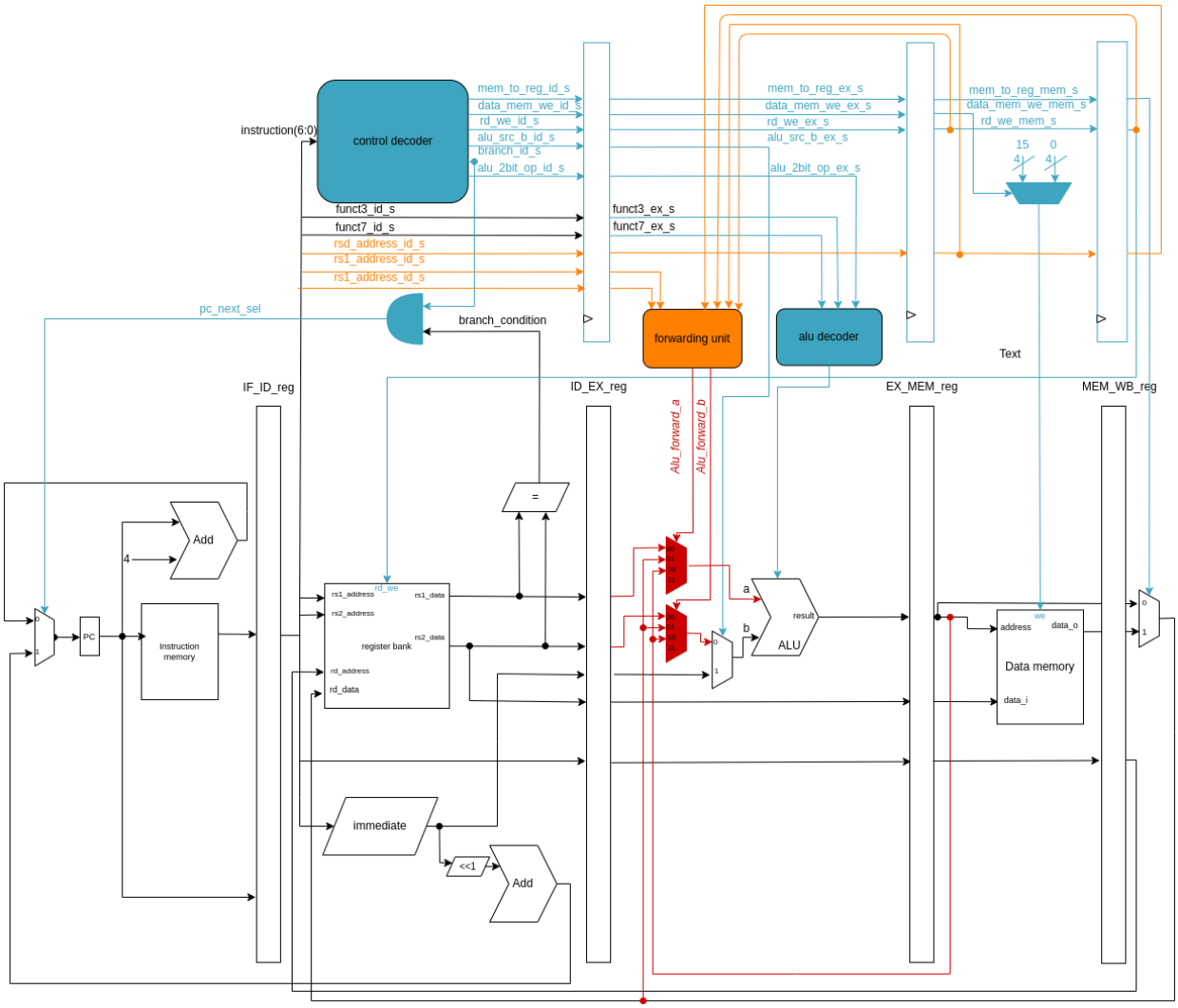


Slika 10. Ilustracija rada *forwarding* komponente

Ukoliko se desi hazard podataka, odnosno da jedna instrukcija zavisi od rezultata prethodne, ideja je da jedinica za prosleđivanje podataka prosledi rezultat izvršavanja prethodne instrukcije u trenutku kada je taj rezultat potreban trenutnoj instrukciji. U prethodnom primeru “and” instrukciji je rezultat izvršvanja “sub” instrukcije neophodan u EX fazi, jer “and” zahteva da ALU izvrši određenu operaciju sa tim rezultatom. Kako se rezultat izvršavanja “sub” instrukcije nalazi u MEM fazi, nije potrebno čekati da “sub” završi svoje izvršavanje, već je dovoljno proslediti taj rezultat nazad u EX fazu. Na slici 11 je naznačeno da “sub” instrukcija u CC4 taktu prosleđuje rezultat svog izvršavanja ALU jedinici koja treba da izvrši logičku operaciju “and”. Slična stvar je urađena i kod

“or” instrukcije, s tim što je sada rezultat izvršavanja “sub” instrukcije prosleđen iz WB faze u EX fazu.

Proširenje procesora jedinicom za prosleđivanje podataka je prikazano na sledećoj slici:



Slika 11. Proširenje procesora jedinicom za prosleđivanje podataka

Na slici 11 su naznačene modifikacije (crvenom u *datapath* celini, narandžastom u *controlpath* celini) koje je potrebno napraviti kako bi se omogućilo prosleđivanje (*eng. forwarding*) podataka. *Controlpath* celina je proširena *forwarding unit* komponentom i njena uloga je da zaključi da li postoji zavisnost između instrukcija koje se nalaze u fazama protočne obrade. Kako bi to uradila neophodne su joj sledeće informacije:

1. Adrese registara (rs1 i rs2) koji su potrebni instrukciji u EX fazi. Sa njihovim vrednostima ALU izvršava operaciju shodno prihvaćenoj instrukciji.
2. Adresu odredišnog registra (rd) instrukcije iz MEM faze.

3. Signal dozvole upisa u registarsku banku *rd_we* instrukcije iz MEM faze.
4. Adresu odredišnog registra (rd) instrukcije iz WB faze.
5. Signal dozvole upisa u registarsku banku rd_we instrukcije iz WB faze.

3. i 4. stavka su neophodne da bi se zaključilo da li instrukcije u MEM i WB fazi uopšte vrše upisivanje u registarsku banku (npr *"sw"* ne vrši upis), jer ukoliko ne vrše upis, nema zavisnosti između instrukcija i prosleđivanje nije neophodno. Ukoliko su signali iz 3. i 4. stavke na visokom logičkom nivou, odnosno instrukcije u MEM i WB fazi vrše upis u registarsku banku, potrebno je još uporediti da li su registri rs1 i rs2, koji su neophodni instrukciji u EX fazi, jednaki sa rd registrom u koji upisuju instrukcije u MEM i WB fazi. Ukoliko su jednaki, jedinica za prosleđivanje podataka će generisati kontrolne signale koji selektuju šta će se proslediti na ALU ulaze. Ti kontrolni signali su na slici 12 označeni kao *alu_forward_a* i *alu_forward_b*. To su dva dvobitna signala koji su dovedeni na selekcione ulaze multipleksera. U zavisnosti od njihove vrednosti multiplekser na svoj izlaz propušta sledeće:

- "00" - nema zavisnosti između instrukcija, propušta se sadržaj iz registarske banke.
- "01" - postoji zavisnost, neophodno je propustiti rezultat iz WB faze.
- "10" - postoji zavisnost, neophodno je propustiti rezultat iz MEM faze.

Sledeći vhdl kod opisuje šta jedinica za prosleđivanje podataka treba da radi:

```
alu_forward_a <= "00";
alu_forward_b <= "00";
if(rd_we_wb = 1 and rd_address_wb /=0) then
        if(rs1_address_ex = rd_address_wb) then
                alu_forward_a <= "01";
        end if;
        if(rs2_address_ex = rd_address_wb) then
                alu_forward_b <= "01";
        end if;
end if;
If (rd_we_mem = 1 and rd_address_mem /= 0 ) then
        if(rs1_address_ex = rd_address_mem) then
                alu_forward_a <= "10";
        end if;
        if(rs2_address_ex = rd_address_mem) then
                alu_forward_b <= "10";
        end if;
end if;
```

Slika 12. Opis rada jedinice za prosleđivanje podataka pomoću vhdl koda

U njemu se mogu primetiti dve stvari koje do sada nisu pomenute. Prva je dodatni uslov *rd_address_mem != 0*, koji nalaže da registar od koga zavisi naredna instrukcija bude različit od 0. To samo kaže da nema potrebe vršiti prosleđivanje ukoliko instrukcija u EX fazi zavisi od nultog registra, jer je njegova vrednost uvek postavljena na nulu i nijedna instrukcija ne može to da promeni (objašnjeno u prvoj vežbi).

Druga stvar je da prosleđivanje iz MEM faze ima prioritet u odnosu na WB fazu (*elsif* u prethodnom kodu). Sledeći primer ilustruje zašto je to neohpodno:

```
addi x1, x1,10
add x1, x2, 5
sub x2, x1, x3
```

Slika 13. Primer zašto prosleđivanje iz MEM ima prednost u odnosu na WB

Može se primetiti da rezultat izvršavanja “sub” instrukcije zavisi od x1 registra koji modifikuju prethodne dve instrukcije. Prvu modifikaciju vrši “addi”, dok drugu vrši “add” i da bi se “sub” izvršila kako treba neophodno je proslediti rezultat izvršavanja jedne od ove dve instrukcije. Ali koje? Kako je modifikacija koju vrši “add” instrukcija “svežija”, odgovor je da njen rezultat izvršavanja treba proslediti. Odatle sledi da prosleđivanje iz MEM faze ima prioritet, jer se tu nalazi “najsvežiji” rezultat.

3.1.2 Problem Hazarda podataka kod instrukcija uslovnog skoka

Prethodna modifikacija nije rešila problem hazarda podataka koji može da se javi prilikom izvršavanja instrukcija uslovnog skoka. Sledeći primer ilustruje situaciju u kojoj procesor neće izvršiti instrukciju uslovnog skoka na ispravan način usled prisustva hazarda podataka.

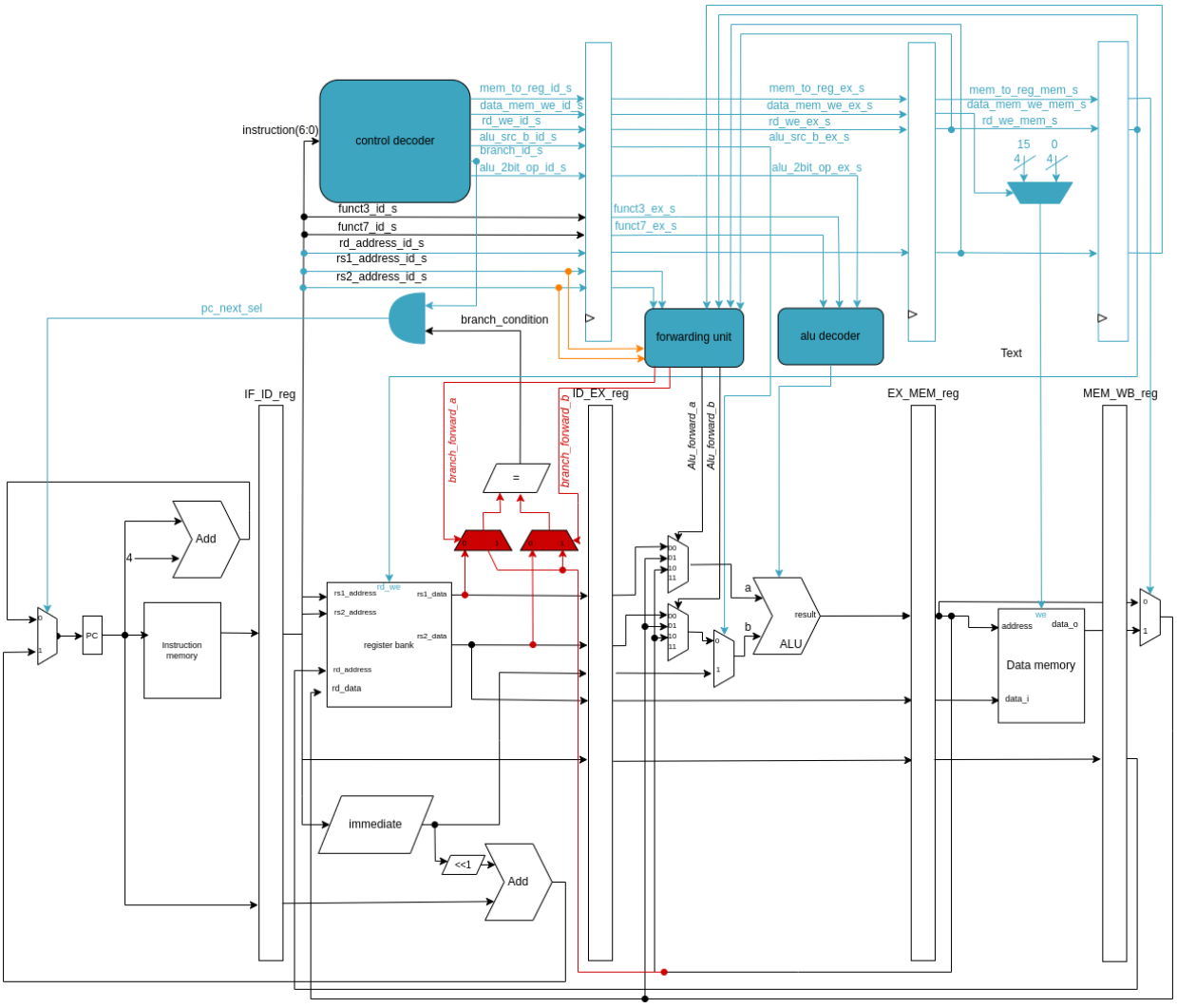
```
Adresa      Instrukcija
00:         Addi x10, x10, 10
04:         Addi x11, x11, 10
08          Addi x12, x12, 10
12:         Beq x10, x11, L1
16:         Add x12, x11, x10
20:         L1:Add x13, x12, x10
```

Slika 14. Hazard podataka priliom izvršavanja *BEQ* instrukcija

U kodu sa slike 14. pretpostavlja se da su registri x10, x11, x12 i x13 postavljeni na nultu vrednost pre izvršavanja gore navedenih instrukcija. Očekivano izvršavanje prethodnog koda jeste upisivanje konstante 10 u registre x11, x12 i x13 pomoću *addi* instrukcije, skok na adresu 20 zbog uspešnog izvršavanja “beq” instrukcije (jer su x10 i x11 registri jednaki) i izvršavanje *add* instrukcije na toj adresi.

Implementacija procesora kakva je sada (slika 12) neće uspešno izvršiti prethodni kod, odnosno neće uspešno izvršiti instrukciju uslovnog skoka. Problem nastaje usled zavisnosti “beq” instrukcije od rezultata izvršavanja prethodne. Prilikom njenog izvršavanja, u ID fazi se vrši poređenje registara na adresama rs1 i rs2 (u ovom primeru to su registri x10 i x11) registarske banke, i ukoliko su oni jednaki, desiće se skok. U kodu sa slike 14, pre “beq” instrukcije izvršavaju se dve “addi” instrukcije koje vrše modifikaciju registara x10 i x11. Usled postojanja protočne obrade, u trenutku kada “beq” stigne u ID fazu, nova vrednost možda još uvek neće biti upisana u x10 i x11 registre. Prva “addi” instrukcija (addi x10, x10, 10) nalaziće se u WB fazi u trenutku kada je “beq” u ID fazi i ona upisuje rezultat svog izvršavanja u registarsku banku na opadajuću ivicu takta, što znači da će “beq” instrukcija u ID fazi imati vremena do sledeće rastuće ivice takta da pročita ispravnu vrednost iz x10 registra (čitanje iz registarske banke je asinhrono). Druga “addi” instrukcija (addi x11, x11, 10) nalaziće se u MEM fazi kada je “beq” u ID fazi i ona će predstavljati problem, jer još uvek nije stigla da upiše rezultat svog izvršavanja u registar x11. To će prouzrokovati da “beq” ne izvrši skok, pošto vrednosti u registrima x10 i x11 nisu jednake (u x10 registar je upisana vrednost 10, a u x11 vrednost 0).

Način da se ovaj problem razreši jeste uvođenje dodatne logike za prosleđivanje podatata, koja je prikazana na sledećoj slici:



Slika 15. Razrešenje hazarda podataka prilikom izvršavanja instrukcija uslovnog skoka

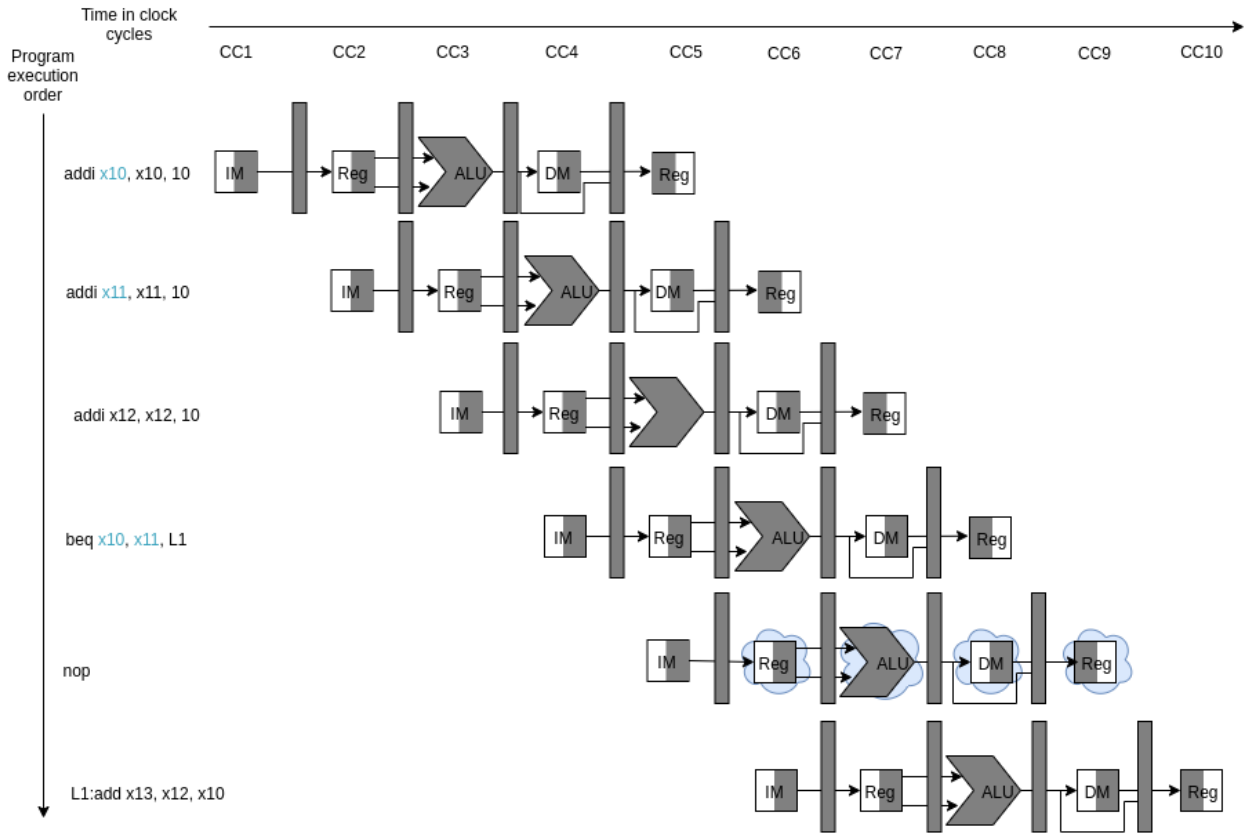
Kao i kod razrešavanja prethodnog hazarda, crvenom bojom su naznačene promene u *datapath* celini, a narandžastom u *controlpath* celini. Jedinica za prosleđivanje podataka (*eng. Forwarding unit*) je proširena sa dodatna dva ulaza na koje se dovode *rs1_address_id i rs2_address_id* i dva izlaza (*branch_forward_a* i *branch_forward_b*) koji predstavljaju kontrolne signale. Ukoliko instrukcija koja se nalazi u MEM fazi rezultat svog izvršavanja upisuje u registarsku banku i ukoliko je rs1_address_id = rd_address_mem izvršiće se prosleđivanje podatka iz MEM faze u ID fazu, odnosno *branch_forward_a* će biti na visokom logičkom nivou. Isto važi i ukoliko je rs2_address_id = rd_address_mem, s tim što će sada branch_forward_b biti na visokom logičkom nivou. Prethodno opisano je predstavljno preko vhdl koda na slici 16.

```
Branch_forward_a <= 0;
Branch_forward_b <= 0;
if(rd_we = 1 and rd_address_mem /= 0) then
        if(rs1_address_id = rd_address_mem) then
                Branch_forward_a <= 1;
        End if;
        if(rs2_address_id = rd_address_mem) then
                Branch_forward_b <= 1;
        End if;
End if;
```

Slika 16. Vhdl kod generisanja kontrolnih signala za prosleđivanja iz MEM u ID fazu

Takođe, kao i u prethodnom vhdl kodu (slika 13) ukoliko instrukcija u MEM fazi pokuša da upiše u registar na nultoj adresi registarske banke, nije potrebno vršiti prosleđivanje, jer nijedna instrukcija ne može da izvrši modifikovanje tog registra, odnosno neće se desiti problem zavisnosti izvršavanja jedne instrukcije od rezultata izvršvanja druge.

Na slici 15, **obratiti pažnju** na liniju obojenu u zeleno, koja je dovedena na IF_ID_reg registar. Njena uloga je da, ukoliko se desi uslovni skok, izvrši *flush* tog registra, jer naredna instrukcija treba da bude preskočena i ne sme da uđe u ID fazu protočne obrade (treba da se izvrši njeno odbacivanje). *Flush* se radi tako što se svi izlazi registra resetuju, čime se u fazu protočne obrade ubacuje “nop” (prazna) instrukcija kao što je prikazano na sledećoj slici:

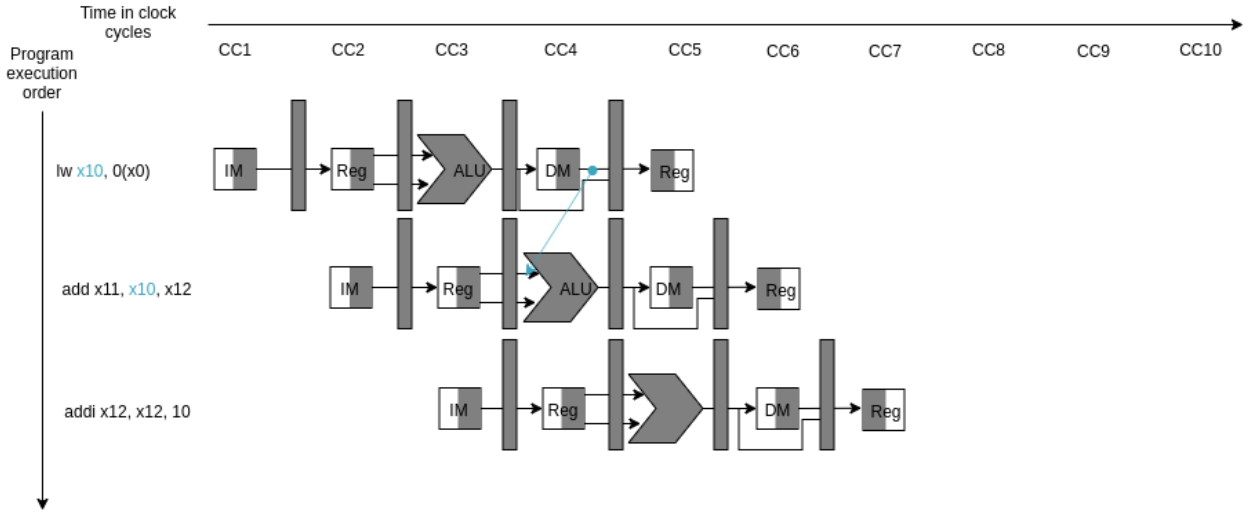


Slika 17. Ilustracija umetanja nop instrukcije nakon što je izvršen flush *IF_ID_reg* registra

“Nop” instrukcija umanjuje propusnu moć procesora, jer ona mora da propagira kroz sve faze protočne obrade.

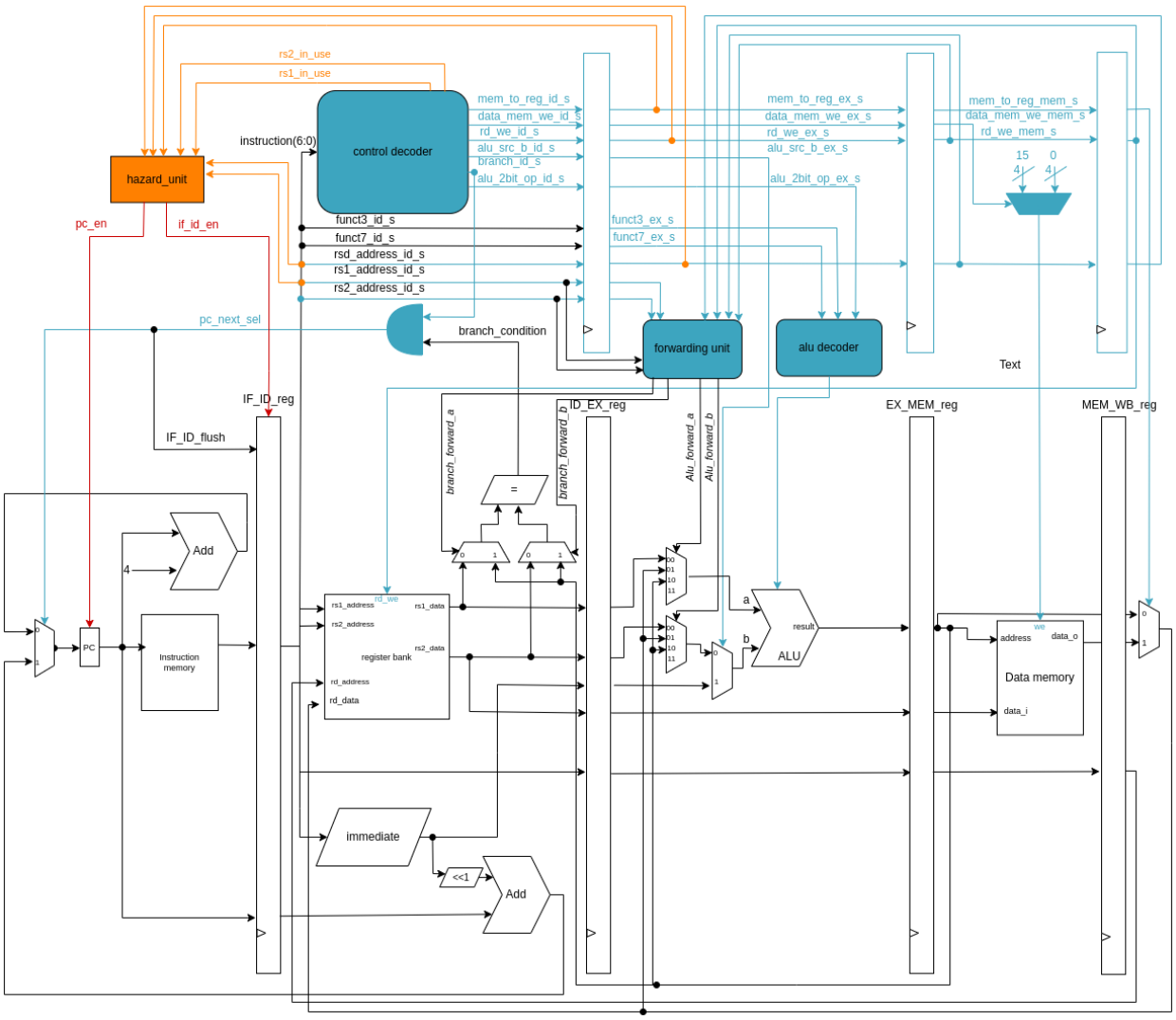
3.1.3 Hazardi podataka i zadržavanje (*eng. stall*) procesora

Implementiranjem logike za prosleđivanje (*eng. Forwarding*) rešeno je nekoliko problema, ali ne i svi. Za neke hazarde podataka prosleđivanje nije dovoljno i tada je neophodno izvršti zadržavanje (*eng. stall*) procesora. U nastavku je prikazan primer u kome zavisnost između instrukcija zahteva takvo rešenje:



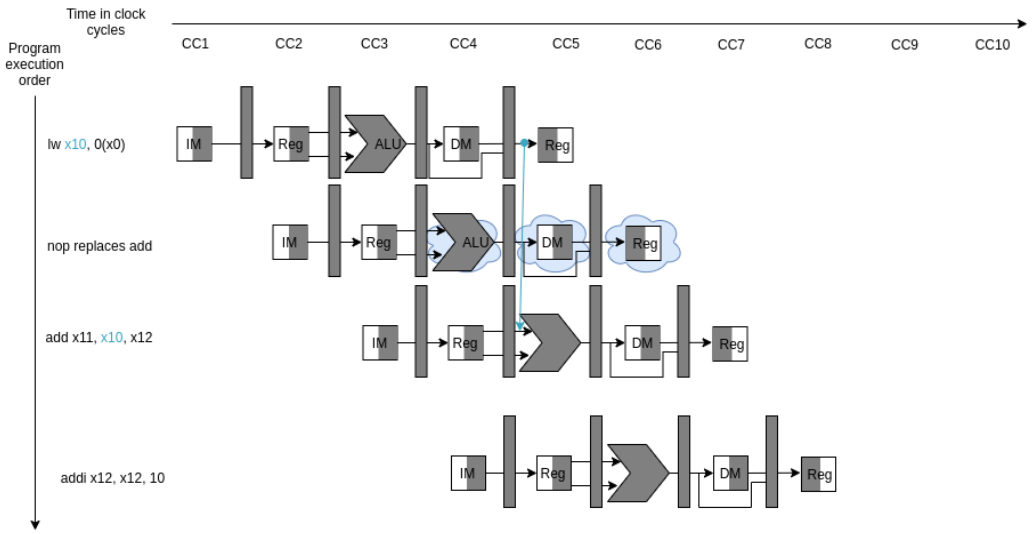
Slika 18. Sekvenca instrukcija koja zahteva *stall* procesora

Instrukcija “lw” vrši čitanje podatka iz memorije za podatke i smešta ga u registar x10 registarske banke, dok naredna instrukcija (add x11, x10, x12) vrši sabiranje registara x10 i x12 i smešta rezultat u x11. Ovde se odmah vidi problem, a to je da u trenutku kada “add” vrši sabiranje, njoj je potreban rezultat iz registra x10 koji “lw” još uvek nije “ažurirala”. Jedinica za prosleđivanje podataka u ovoj situaciji ne pomaže jer “lw” mora prvo da pročita podatak iz memorije podataka i on će tek iz WB faze moći da bude prosleđen. Način da se ovo reši jeste da se uvede dodatna logika koja će biti zadužena da zaustavi dalje izvršavanje procesora, sve dok “lw” ne pročita podatak iz memorije i taj podataka se ne pojavi u WB fazi. Na slici 19 je prikazano proširenje trenutne implementacije procesora dodatnom logikom zaduženom za **zadržavanje** (*eng. stall*) procesora kada za tim postoji potreba.



Slika 19. Procesor proširen *hazard unit* komponentom

Ukoliko postoji zavisnost između instrukcija, kao u primeru sa slike 18, *hazard unit* komponenta zaustavlja prihvat novih instrukcija pomoću tri kontrolna signala: *pc_en if_id_en* i *controll_pass.* Prvi kontroliše *pc* registar, te ukoliko je on na niskom logičkom nivou pc registar se zaustavlja, te adresa memorije za instrukcije ostaje ista. Drugi kontroliše *if_id_reg* registar i ukoliko je na niskom logičkom nivou naredna instrukcija iz IF faze ne može biti propuštena u ID fazu. *Control_pass* resetuje ID_EX_reg registar u *controlpath* celini, odnosno resetuje sve kontrolne signale u EX fazi, što je ekvivalento umetanju “nop” instrukcije u EX fazu.To je neophodno, jer se u slučaju *stall-a* procesora instrukcija mora zadržati u ID fazi, a to je jedino moguće ukoliko se umesto nje u EX fazu pusti “nop” instrukcija.



Slika 20. *Stall* procesora.

Da bi se kontrolni signali (*pc_en*, *if_id_en* i control pass*)* generisali pravovremeno i kada za tim postoji potreba, *hazard unit* komponenti su neophodne sledeće informacije:

1. Adrese registara u registarskoj banci kojima pristupa instrukcija u ID fazi (rs1_address_id i rs2_address_id).
2. Adresu odredišnog registra instrukcije iz EX faze (*Rd_address_ex*).
3. Da li instrukcija iz EX fazi vrši upis u registarsku banku (*Rd_we_ex*).
4. Da li instrukcija iz EX fazi vrši čitanje podatka iz memorije i smešta ga u registarsku banku (*Mem_to_reg_mem*) .
5. Da li instrukcija u ID fazi koristi registar iz registarske banke na adresama rs1_address ili rs2_address (*Rs1_in_use_id i rs2_in_use_id.*). Npr. instrukcije “I” tipa, u EX fazi, na ulaze “a” i “b” ulaze ALU jedinice prosleđuju sadržaj registra na adresi *rs1_address* i *immediate_extended* respektivno. Odnosno, registar na adresi *rs2_address* se ne koristi i nije potrebno zaustavljati procesor, ukoliko se npr. *“*lw*”* instrukcija izvršava neposredno pre instrukcije “I” tipa.

Sledeći vhdl kod ilustruje šta hazard unit komponenta treba da uradi:

```
en <= '1';
if(((rs1_address_id_i = rd_address_ex_i and rs1_in_use_i = '1') or
        (rs2_address_id_i = rd_address_ex_i and rs2_in_use_i = '1')) and
        mem_to_reg_ex_i = '1' and rd_we_ex_i = '1')then -- load instrukcija je u EX fazi
        en_s <='0';
end if;
```

Slika 20. Vhdl kod hazard unit komponente

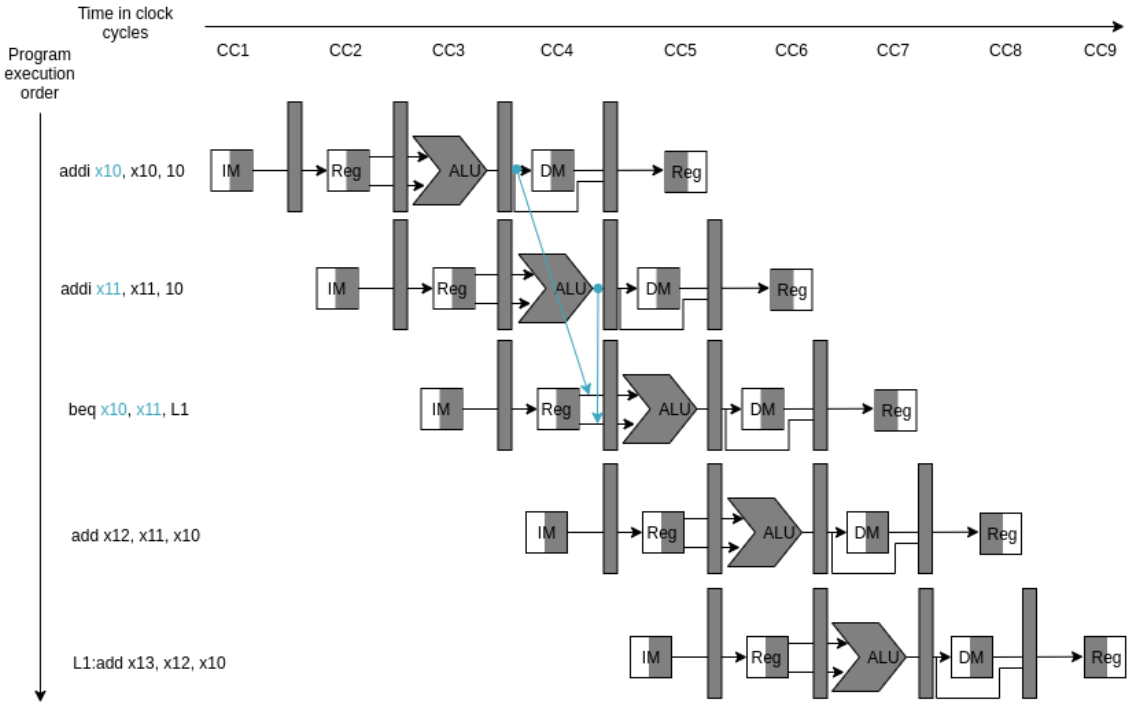
Takođe, potrebno je proširiti *ctrl_decoder* komponentu sa dodatna dva izlaza, koji predstavljaju *rs1_in_use_id* i *rs2_in_use_id* kontrolne signale, neophodne *hazard unit* komponenti. Tako da sliku 21, sa vežbe 2, koja ilustruje kako *ctrl_decoder* komponenta generiše kontrolne signale na osnovu *opcode* polja instrukcije, treba proširiti na sledeći način:

	Ime signala	R-format	lw	sw	beq	addi
Ulazi	opcode(6)	0	0	0	1	0
	opcode(5)	1	0	1	1	0
	opcode(4)	1	0	0	0	1
	opcode(3)	0	0	0	0	0
	opcode(2)	0	0	0	0	0
	opcode(1)	1	1	1	1	1
	opcode(0)	1	1	1	1	1
Izlazi	Alu_src	0	1	1	x	1
	Mem_to_reg	0	1	x	0	0
	rd_we	1	1	0	0	1
	data_mem_we	0	0	1	0	0
	branch	0	0	0	1	0
	Alu_2bit_op(1)	1	0	0	0	1
	Alu_2bit_op(0)	0	0	0	1	1
	rs1_in_use	1	1	1	1	1
	rs2_in_use	1	0	1	1	0

Slika 21. Veza Kontrolnih signala i opcode polja instrukcije

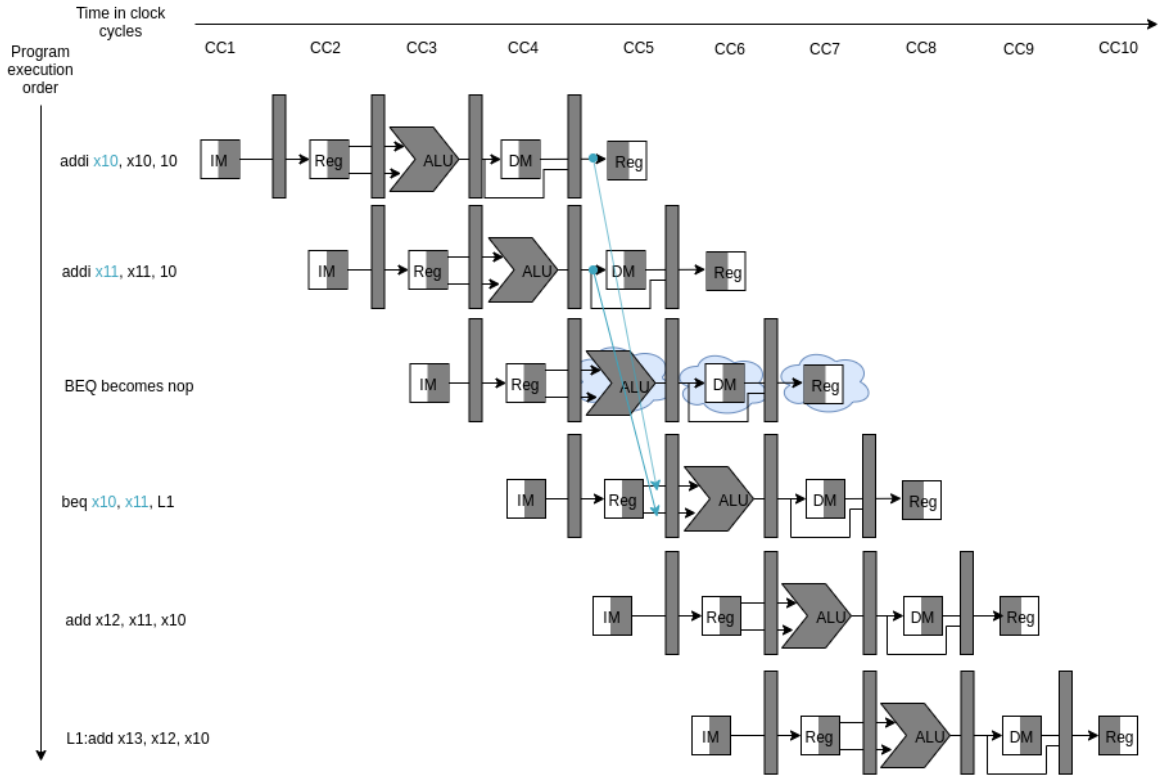
3.2 Upravljački hazardi

Uvođenjem protočne obrade komplikuje se izvršavanje instrukcija uslovnog skoka i problemi koji nastaju u tim situacijama se nazivaju upravljački hazardi. Na slici 22. prikazana je sekvenca instrukcija koja izaziva takav scenario:



Slika 22. Sekvenca instrukcija koja izaziva upravljački hazard

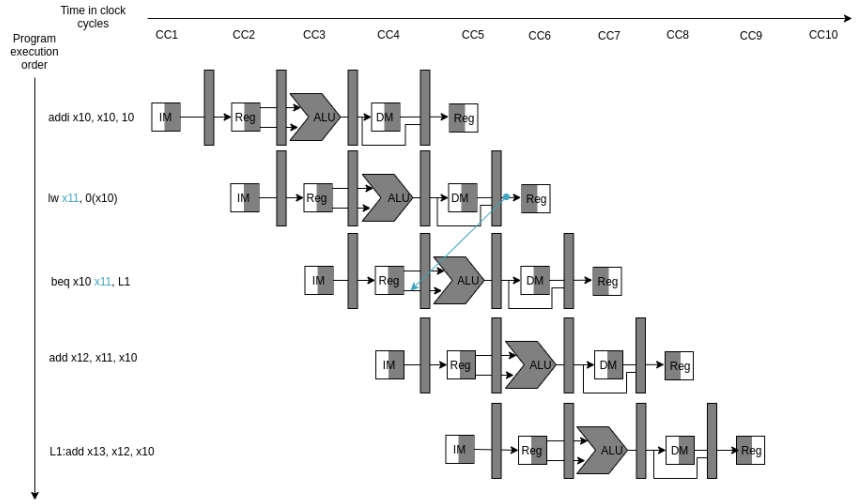
Prve dve instrukcije (*addi x10, x10, 10* i *addi x11, x11, 10*) modifikuju registar x10, odnosno x11 i u njih smeštaju konstantu 10. Nakon njih se izvršava “beq” instrukcija, koja vrši poređenje sadržaja u registrima x10 i x11 i ukoliko su jednaki skače na labelu L1. U trenutku kada se vrši poređenje (takt CC4), prve dve instrukcije, koje modifikuju registre x10 i x11, se još nisu izvršile, odnosno još uvek nisu upisale rezultat svog izvršavanja u registarsku banku. Iz tog razloga, kao i u prethodnim slučajevima, trebalo bi da se vrši prosleđivanje kako bi “beq” instrukcija izvršila poređenje sa ispravnim vrednostima. Ali, ako se pogleda slika 22, prosleđivanje vrednosti x11 registra ne može da se izvrši u tom taktu, jer u trenutku kada je taj rezultat potreban (CC4 takt), on se tek proračunava u EX fazi. U ovoj situaciji jedino rešenje jeste da se zaustavi izvršavanje procesora i da instrukcija u ID fazi sačeka da vrednost koja treba da bude upisana u x11 registar dođe u MEM fazu, kada će prosleđivanje moći da se uradi. Situacija koja će se desiti prikazana je na sledećoj slici:



Slika 23. *Stall* (zadržavanje) procesora umetanjem nop instrukcije

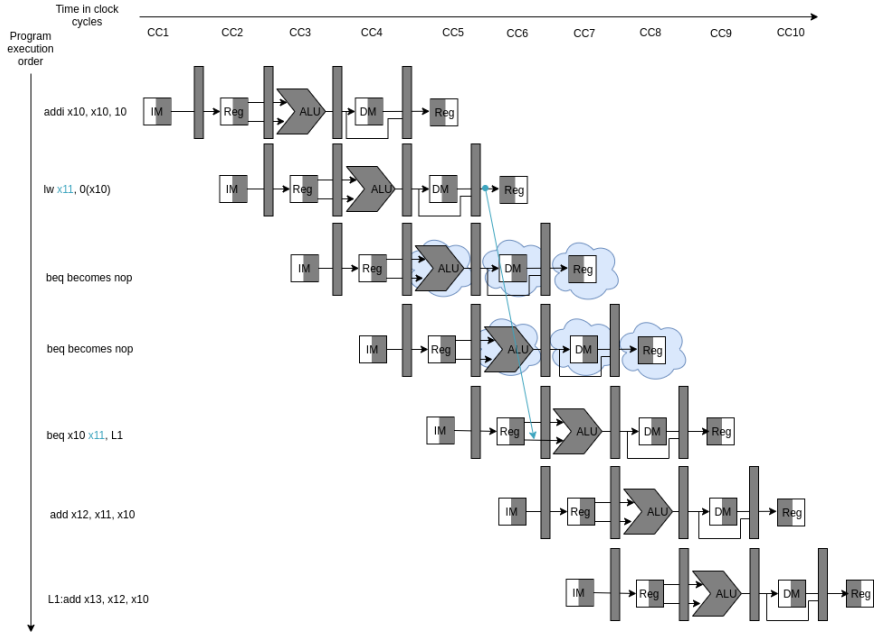
Čekanje da *addi x11, x11, 10* instrukcija dođe u MEM fazu, odakle rezultat njenog izvršavanja može da se prosledi, radi se umetanjem “nop” (prazne) instrukcije i zadržavanjem “beq” instrukcije u ID fazi. Na slici 23. je “oblačićima” naznačeno u kojoj fazi će se naći “nop” instrukcija. Tu se vidi da “beq” dolazi do ID faze, nakon čega se zaustavlja i umesto nje se u EX fazu umeće nop instrukcija, koja propagira kroz sve faze.

Prethodno opisani upravljački hazard može da se pogorša ukoliko se pojavi sledeća sekvenca instrukcija:



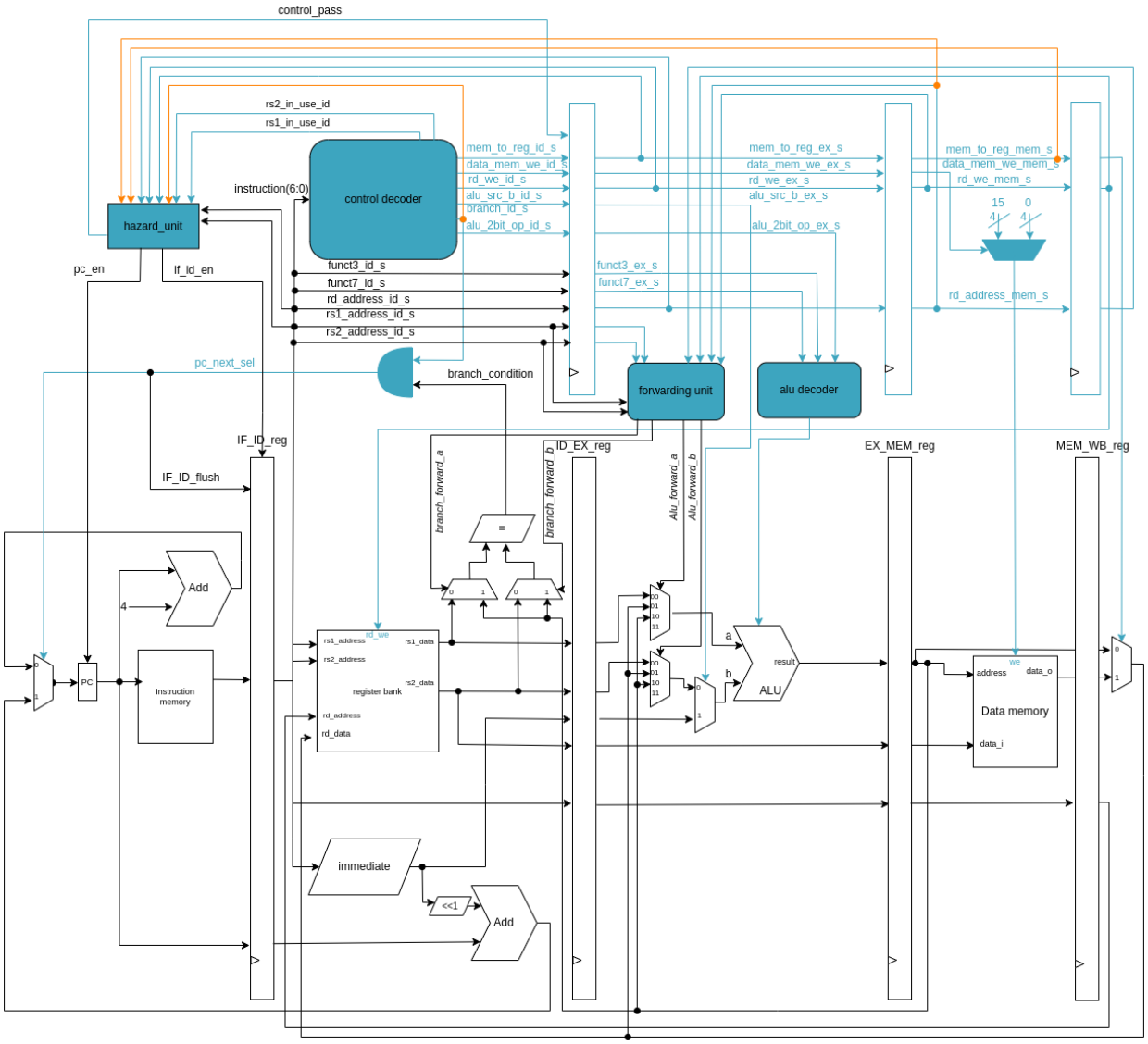
Slika 24. upravljački hazard koji zahteva *stall* dužine dva takta.

Sekvenca instrukcija u ovom primeru je slična kao na slici 22, stim što je umesto “addi” instrukcije stavljena “lw” instrukcija. Ona takođe modifikuje registar x11, od koga zavisi “Beq” instrukcija. U taktu CC4, kada je “Beq” instrukciji potreban taj podatak, “lw” se nalazi u EX fazi, te se rezultat ne može proslediti, jer “lw” još nije pročitala podatak iz memorije. Prvi trenutak kada je to moguće jeste takt CC6, odnosno neophodno je izvršiti *stall* u trajanju od dva takta , kako bi se “lw” instrukciji dalo dovoljno vremena da pročita podatak iz memorije (pogledati sliku 25).



Slika 25. Stall u trajanju od 2 takta

Slika 26 ilustruje proširenje trenutne implementacije procesora dodatnom logikom kako bi se prethodno opisani upravljački hazardi rešili:



Slika 26. Proširenje implementacije procesora logikom koja rešava kontrolne hazarde

Hazard unit komponenta je proširena sa dodatna 3 ulazna porta:

1. *Branch_id*, sadrži informaciju da li je primljena instrukcija instrukcija uslovnog skoka. Potreba za ovom informacijom proizilazi iz činjenice da ukoliko instrukcija u ID fazi nije instrukcija uslovnog skoka, *stall* će biti dugačak maksimalno 1 takt, a u suprotnom će možda biti dva.
2. *Rd_address_mem*, sadrži informaciju o tome kom registru pristupa instrukcija koja se nalazi u MEM fazi.

3. *Mem_to_reg_mem*, govori da li instrukcija u MEM fazi vrši čitanje iz memorije za podatke i smešta rezultat čitanja u registarsku banku.

Portovi *rd_address_mem i mem_to_reg_mem* su neophodni jer na osnovu njih *hazard unit* komponenta zna da li treba da generiše *stall* u trajanju od jednog takta ili dva takta. Sledeći vhdl kod predstavlja proširenje funkcionalnosti *hazard unit* komponente prikazano na slici 20:

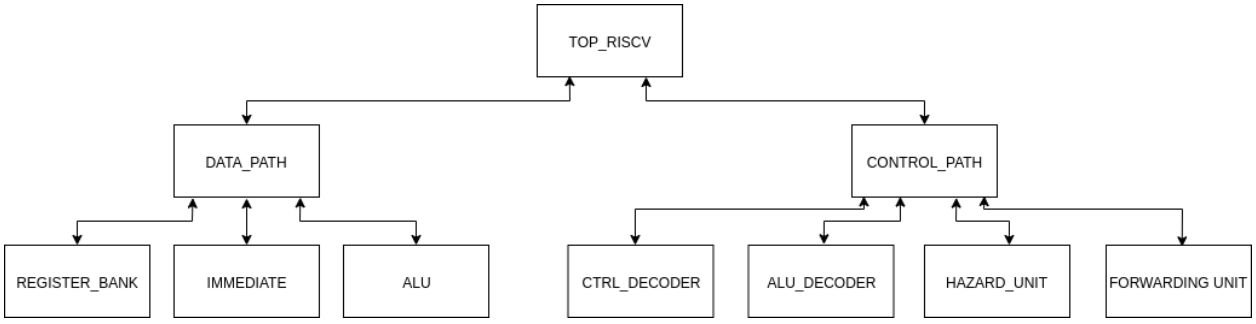
```
en_s <= '1';
    if (branch_id_i = '0') then -- instrukcija u ID fazi nije skok
      if(((rs1_address_id_i = rd_address_ex_i and rs1_in_use_i = '1') or
        (rs2_address_id_i = rd_address_ex_i and rs2_in_use_i = '1')) and
        mem_to_reg_ex_i = '1' and rd_we_ex_i = '1')then -- load instrukcija je u EX fazi
        en_s <='0';
      end if;
    elsif(branch_id_i = '1')then -- instrukcija u ID fazi je uslovni skok (branch)
      if((rs1_address_id_i = rd_address_ex_i or rs2_address_id_i = rd_address_ex_i)
        and rd_we_ex_i = '1')then -- load ili R-tip u EX fazi
        en_s <='0';
      elsif((rs1_address_id_i = rd_address_mem_i or rs2_address_id_i = rd_address_mem_i)
        and mem_to_reg_mem_i = '1')then -- load u MEM fazi
        en_s <='0';
      end if;
    end if;
```

Slika 27. Vhdl kod koji opisuje hazard unit komponentu.

Ukoliko je *branch_id = ‘0’* hazard unit komponenta se ponaša kao njena prethodna implementacija opisana na slici 20. U suprotnom, *hazard unit* proverava da li postoji zavisnost instrukcije uslovnog skoka u ID fazi i instrukcija u EX i MEM fazama. Ukoliko zavisnost postoji generiše se *stall* signal. Npr. ukoliko je u ID fazi “Beq” instrukcija, a u EX fazi “lw” instrukcija i ukoliko postoji zavisnost između te dve instrukcije, neophodno je generisati *stall* u trajanju od 2 takta. Odnosno prvi *stall* se generiše u trenutku kada je lw u EX fazi, a drugi u trenutku kada dođe u MEM fazu.

4 VHDL implementacija

Hijerarhija procesora sa protočnom obradom je slična kao kod *single cycle* procesora i prikazana je na sledećoj slici. U narednim sekcijama biće napisani interfejsi svih blokova sa slike 22. **Prilikom realizovanja svake od komponenti procesora ne modifikovati interfejs ukoliko to nije neophodno**!



Slika 28. Hierarhija procesora sa protočnom obradom.

U nastavku su dati prikazani intefersji komponenti u kojima postoji promena u odnosu na *single cycle* implementaciju.

4.1 DATA_PATH

Sledeći kodni listing predstavlja proširenje interfejsa *datapath* celine *single cycle* procesora opisanog na prethodnom vežbama.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.datapath_signals_pkg.all;


entity data_path is
   port(
      -- sinhronizacioni signali
      clk                : in  std_logic;
      reset              : in  std_logic;
      -- interfejs ka memoriji za instrukcije
      instr_mem_address_o : out std_logic_vector (31 downto 0);
      instr_mem_read_i    : in  std_logic_vector(31 downto 0);
      instruction_o       : out std_logic_vector(31 downto 0);
      -- interfejs ka memoriji za podatke
      data_mem_address_o  : out std_logic_vector(31 downto 0);
      data_mem_write_o    : out std_logic_vector(31 downto 0);
      data_mem_read_i     : in  std_logic_vector (31 downto 0);
      -- kontrolni signali
      mem_to_reg_i        : in  std_logic;
      alu_op_i            : in  std_logic_vector (4 downto 0);
```

```
      alu_src_b_i         : in  std_logic;
      pc_next_sel_i       : in  std_logic;
      rd_we_i             : in  std_logic;
      branch_condition_o  : out std_logic;
      -- kontrolni signali za prosledjivanje operanada u ranije faze protocne
obrade
      alu_forward_a_i     : in  std_logic_vector (1 downto 0);
      alu_forward_b_i     : in  std_logic_vector (1 downto 0);
      branch_forward_a_i  : in  std_logic;
      branch_forward_b_i  : in  std_logic;
      -- kontrolni signal za resetovanje if/id registra
      if_id_flush_i       : in  std_logic;
      -- kontrolni signali za zaustavljanje protocne obrade
      pc_en_i             : in  std_logic;
      if_id_en_i          : in  std_logic);

end entity;
```

4.2 CTRL_DECODER

Jedina promena *ctrl_decoder* komponente jeste dodavanje *rs1_in_use* i *rs2_in_use* kontrolnih signala.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity ctrl_decoder is
   port (
      -- opcode instrukcije
      opcode_i      : in  std_logic_vector (6 downto 0);
      -- kontrolni signali
      branch_o      : out std_logic;
      mem_to_reg_o  : out std_logic;
      data_mem_we_o : out std_logic;
      alu_src_b_o   : out std_logic;
      rd_we_o       : out std_logic;
      rs1_in_use_o  : out std_logic;
      rs2_in_use_o  : out std_logic;
      alu_2bit_op_o : out std_logic_vector(1 downto 0)

      );
end entity;
```

4.3 FORWARDING UNIT

Sledeći kodni listing predstavlja interfejs forwarding unit komponente.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;


entity forwarding_unit is
   port (
      -- ulazi iz ID faze
      rs1_address_id_i   : in  std_logic_vector(4 downto 0);
      rs2_address_id_i   : in  std_logic_vector(4 downto 0);
      -- ulazi iz EX faze
      rs1_address_ex_i   : in  std_logic_vector(4 downto 0);
      rs2_address_ex_i   : in  std_logic_vector(4 downto 0);
      -- ulazi iz MEM faze
      rd_we_mem_i        : in  std_logic;
      rd_address_mem_i   : in  std_logic_vector(4 downto 0);
      -- ulazi iz WB faze
      rd_we_wb_i         : in  std_logic;
      rd_address_wb_i    : in  std_logic_vector(4 downto 0);
      -- izlazi za prosledjivanje operanada ALU jedinici
      alu_forward_a_o    : out std_logic_vector (1 downto 0);
      alu_forward_b_o    : out std_logic_vector(1 downto 0);
      -- izlazi za prosledjivanje operanada komparatoru za odredjivanje uslova
skoka
      branch_forward_a_o : out std_logic;
      branch_forward_b_o : out std_logic);
end entity;
```

4.4 HAZARD UNIT

Sledeći kodni listing predstavlja interfejs *hazard unit* komponente.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity hazard_unit is
   port (
      -- ulazni signali
      rs1_address_id_i : in std_logic_vector(4 downto 0);
      rs2_address_id_i : in std_logic_vector(4 downto 0);
      rs1_in_use_i     : in std_logic;
      rs2_in_use_i     : in std_logic;
      branch_id_i      : in std_logic;
```

```
      rd_address_ex_i  : in std_logic_vector(4 downto 0);
      mem_to_reg_ex_i  : in std_logic;
      rd_we_ex_i       : in std_logic;
      rd_address_mem_i : in std_logic_vector(4 downto 0);
      mem_to_reg_mem_i : in std_logic;
      -- izlazni kontrolni signali
      -- pc_en_o je signal dozvole rada za pc registar
      pc_en_o          : out std_logic;
      -- if_id_en_o je signal dozvole rada za if/id registar
      if_id_en_o       : out std_logic;
      -- control_pass_o kontrolise da li ce u execute fazu biti prosledjeni
      --   kontrolni signali iz ctrl_decoder-a ili sve nule
      control_pass_o   : out std_logic
      );
end entity;
```

4.5 CONTROL_PATH

Slično kao *datapath* i *controlpath* je pretrpeo izmene koje su neophodne kako bi se podržala protočna obrada. Sledeći kodni listing prikazuje interfejs izmenjene *controlpath* jedinice, onako kako je to objašnjeno u koraku 3 ovih vežbi:

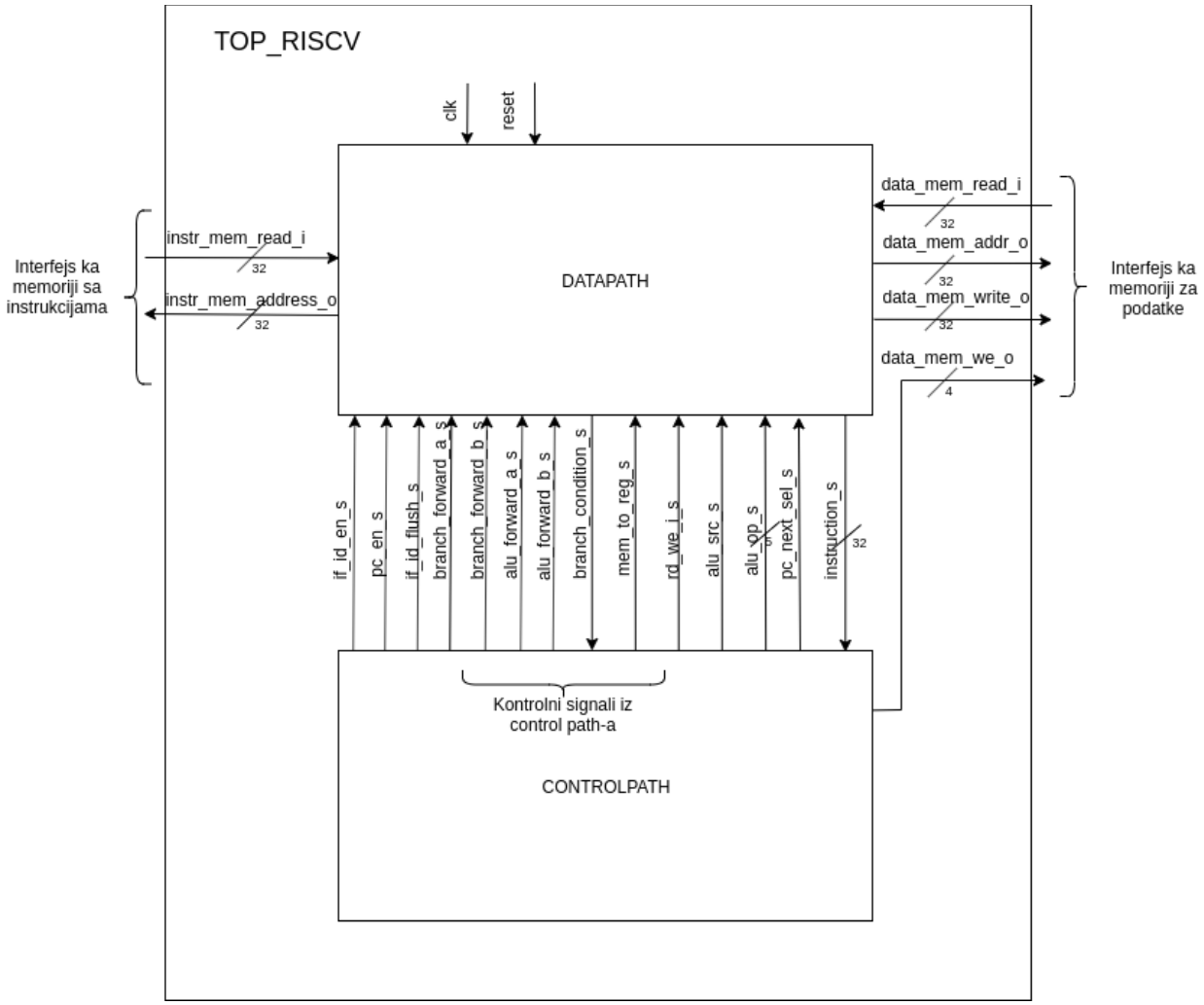
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.controlpath_signals_pkg.all;

entity control_path is
   port (
      -- sinhronizacija
      clk               : in  std_logic;
      reset             : in  std_logic;
      -- instrukcija dolazi iz datapah-a
      instruction_i     : in  std_logic_vector (31 downto 0);
      -- Statusni signaln iz datapath celine
      branch_condition_i : in  std_logic;
      -- kontrolni signali koji se prosledjiuju u datapath
      mem_to_reg_o      : out std_logic;
      alu_op_o          : out std_logic_vector(4 downto 0);
      alu_src_b_o       : out std_logic;
      rd_we_o           : out std_logic;
      pc_next_sel_o     : out std_logic;
      data_mem_we_o     : out std_logic_vector(3 downto 0);
      -- kontrolni signali za prosledjivanje operanada u ranije faze protocne
obrade
      alu_forward_a_o   : out std_logic_vector (1 downto 0);
```

```
      alu_forward_b_o    : out std_logic_vector (1 downto 0);
      branch_forward_a_o : out std_logic;  -- mux a
      branch_forward_b_o : out std_logic;   -- mux b
      -- kontrolni signal za resetovanje if/id registra
      if_id_flush_o      : out std_logic;
      -- kontrolni signali za zaustavljanje protocne obrade
      pc_en_o            : out std_logic;
      if_id_en_o         : out std_logic
      );
end entity;
```

4.6 TOP_RISCV

Na samom kraju neophodno je instancirati i povezati DATA_PATH i CONTROL_PATH i to je urađeno na sledeći način:



Slika 29. Modul na najvišem hijerarhijskom nivou

Sledeći kodni listing predstavlja njegov interfejs, koji je isti kao kod single cycle implementacije.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity TOP_RISCV is
   port(
      -- Globalna sinhronizacija
      clk                 : in  std_logic;
      reset               : in  std_logic;
      -- Interfejs ka memoriji za podatke
      instr_mem_address_o : out std_logic_vector(31 downto 0);
      instr_mem_read_i    : in  std_logic_vector(31 downto 0);
      -- Interfejs ka memoriji za instrukcije
      data_mem_address_o  : out std_logic_vector(31 downto 0);
      data_mem_read_i     : in  std_logic_vector(31 downto 0);
      data_mem_write_o    : out std_logic_vector(31 downto 0);
      data_mem_we_o       : out std_logic_vector(3 downto 0));
end entity;
```

4.8 Verifikaciono okruženje

Verifikaciono okruženje je u potpunosti isto kao kod *single cycle* procesora, te ovde neće biti razmatrano.