J. Đorđević

Arhitektura i organizacija računara

Pipeline

Napomena: Molim da me o otkrivenim greškama obavestite na email
jdjordjevic@etf.bg.ac.yu

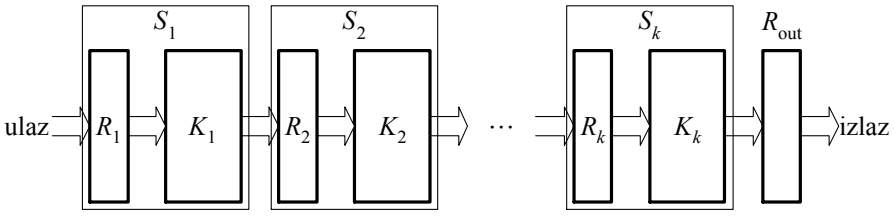
Beograd, 2007

SADRŽAJ

1. Pipeline organizacija

Pipeline je jedna od tehnika realizacije izvršavanja operacija po kojoj se preklapa izvršavanje više operacija.

Izvršavanje operacije se može razbiti na više logičkih celina koje će se nazivati fazama (čitanje instrukcije, formiranje adresa operanada, čitanje operanada, izvršavanje operacije, itd…). Za izvršavanje neke faze postoji poseban deo koji će se nazivati stepen S (slika 1). Stepen se sastoji od kombinacione mreže K i prihvatnog registra R. Kombinaciona mreža izvršava aritmetičke i logičke operacije. Prihvatni registar sadrži informacije potrebne za izvršavanje faze u datom stepenu i preostalih faza u stepenima koji slede. Faza 1 se izvršava u stepenu S_1, faza 2 u stepenu S_2, itd. Istovremeno se može naći k operacija u k različitih faza izvršavanja. Prihvatni registar uz stepen i sadrži informacije potrebne za izvršavanje faze u stepenu S_i, ali i u svim preostalim stepenima.



Slika 1 *Pipeline* organizacija

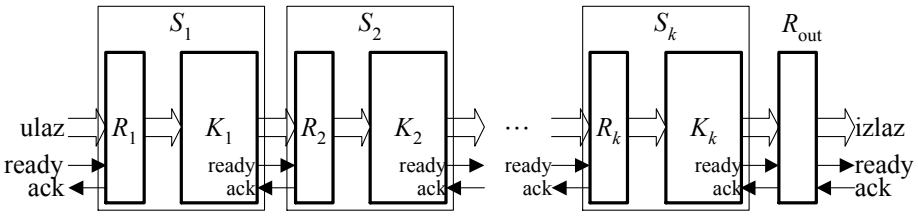
Pipeline može da bude:

- statički (linearan) i
- dinamički (nelinearan).

Kod statičkog *pipeline*-a svaka operacija prolazi kroz isti, fiksni, linearni niz stepeni *pipeline*-a. Ako se izvršavanje svake operacije može razbiti na k faza, onda se statički *pipeline* sastoji od k stepeni. Spoljašnje informacije se ubacuju u *pipeline* preko prvog stepena S_1. Obrađeni rezultati se šalju iz stepena S_i u stepen S_{i+1} za sve $i = 1, 2, \ldots, k - 1$. Finalni rezultat izlazi iz *pipeline*-a izlaskom iz stepena S_k.

U zavisnosti od toga kako se upravlja slanjem obrađenih rezultata iz stepena u stepen *pipeline*-a, statički *pipeline*-i se svrstavaju u:

- asinhrone i
- sinhrone.



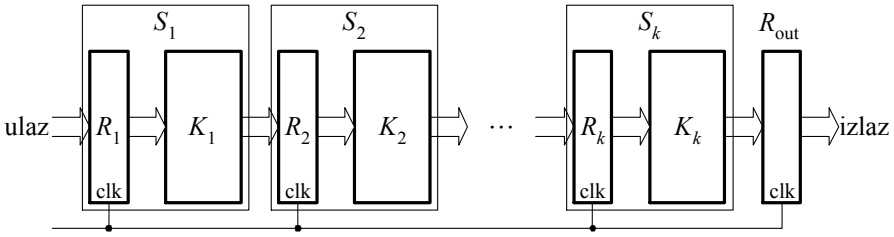
Slika 2 Asinhroni *pipeline*

U asinhronom *pipeline*-u tokom podataka između susednih stepeni se upravlja *handshaking* protokolom (slika 2). Kada je stepen S_i spreman da šalje svoj rezultat on šalje **ready** signal stepenu S_{i+1}. Pošto stepen S_{i+1} prihvati podatke u svoj prihvatni registar, on vraća signal potvrde **ack** (acknowledge) stepenu S_i.

Stepen S_{i+1} može da u svoj prihvatni registar R_{i+1} prihvati podatke iz stepena S_i, jedino ako su podaci iz stepena S_{i+1} prihvaćeni u prihvatni registar R_{i+2} stepena S_{i+2}. Prihvatanje podataka u stepenu S_{i+2} je na isti način uslovljeno od strane stepena S_{i+3}. Ovo se na isti način prenosi do stepena S_k. U suštini, stepen S_k je taj koji mora najpre da svoj rezultat prebaci u R_{out}, da bi se rezultat S_{k-1} mogao da prebaci u S_k. Tek tada rezultat S_{k-2} može da ide u S_{k-1} i tako redom do prebacivanja rezultata S_1 u S_2 i ubacivanja nove operacije u S_1.

Ovo usporava rad. Da bi se ovo ublažilo moguće je na izlaz svakog stepena staviti prihvatni registar B (buffer). Kada se u stepenu S_i obavi odgovarajuća faza, rezultat ide u prihvatni registar B_i. Ukoliko stepen S_{i+1} može da primi operaciju iz stepena S_i, prebacuje se sadržaj B_i u R_{i+1}. U suprotnom se čeka. To, međutim, ne zaustavlja prebacivanje operacije iz registra B_{i-1} stepena S_{i-1} u registar R_i stepena S_i.

Asinhroni *pipeline* se koristi tamo gde se kašnjenje u stepenu S_i razlikuje od operacije do operacije. Recimo da je stepen S_i predviđen za formiranje efektivne adrese. Tada se kašnjenje kroz stepen S_i menja u zavisnosti od načina adresiranja koji se koristi. Obično se koriste kod *message-passing* multiračunarskih sistema.

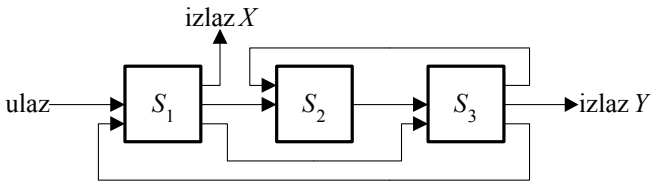


Slika 3 Sinhroni *pipeline*

U sinhronom *pipeline*-u upis u prihvatne registre svih stepeni se realizuje na isti signal takta **CLK** (slika 3). Pri njegovoj pojavi izlazi kombinacione mreže K_i stepena S_i i delovi prihvatnog registra R_i se upisuju u prihvatni registar R_{i+1} stepena S_{i+1} za $i = 1, 2, \ldots, k-1$. Izlaz stepena S_k se upisuje u izlazni prihvatni registar R_{out}, a u prihvatni registar R_1 se upisuje nova operacija. Perioda signala takta **CLK** se određuje na osnovu najvećeg kašnjenja u stepenima S_1 do S_k.

Kod dinamičkog *pipeline*-a svaka operacija ima svoj redosled prolaska kroz pojedine stepene *pipeline*-a. Stoga dinamički *pipeline* ima između svojih stepeni veze i unapred i unazad.

Dinamički *pipeline* sa tri stepena S_1, S_2 i S_3 u kome više operacija sa različitim redosledom prolaska kroz stepene *pipeline*-a može da se izvršava prikazan je na slici 4. Pored veza od S_1 do S_2 i od S_2 do S_3 kojima se ova tri stepena linearno povezuju kao kod statičkog *pipeline*-a, postoji i veza unapred od S_1 do S_3 i dve veze unazad od S_3 do S_2 i od S_3 do S_1. Sa ovakvim povezivanjem stepeni *pipeline*-a izlaz iz *pipeline*-a ne mora da bude samo iz zadnjeg stepena. U slučaju primera sa slike 4 postoje dva izlaza iz stepena S_1 i S_3, respektivno.



Slika 4 Dinamički *pipeline*

Za svaku operaciju koja se izvršava u dinamičkom *pipeline*-u mora da se zna kroz koje stepene *pipeline*-a i po kom redosledu treba da se prolazi. Kod dinamičkog *pipeline*-a to se za svaku operaciju definiše **tabelama rezervacija**. Za dinamički *pipeline* sa slike 4 **tabele rezervacija** za operacije *X* i *Y* su date na slikama 5 i 6.

→ vreme u signalima takta **CLK**

		1	2	3	4	5	6	7	8
stepeni	S_1	*X*					*X*		*X*
pipe-	S_2		*X*		*X*				
line-a	S_3			*X*		*X*		*X*	

Slika 5 Tabela rezervacije za operaciju *X*.

→ vreme u signalima takta **CLK**

		1	2	3	4	5	6
stepeni	S_1	*Y*				*Y*	
pipe-	S_2			*Y*			
line-a	S_3		*Y*		*Y*		*Y*

Slika 6 Tabela rezervacije za operaciju *Y*.

Tabela rezervacija prikazuje kako operacija u toku svog izvršavanja prolazi u ritmu signala takta kroz stepene *pipeline*-a. Broj kolona u tabeli rezervacija prikazuje u koliko perioda signala takta će se data operacija izvršiti. Broj vrsta odgovara broju stepeni u *pipeline*-u.

Svaka operacija kada ide kroz stepene dinamičkog *pipeline*-a pored ostalih informacija neophodnih za njeno izvršavanje vuče i svoju tabelu rezervacija. Uz nju mora i brojač koraka koji se inkrementira pri prelazu iz stepena u stepen i služi za selekciju kolone radi dobijanja informacije o sledećem stepenu.

Tabela rezervacija je trivijalna za statički *pipeline* i ista je za sve operacije. Tabela ima onoliko kolona koliko ima stepeni u *pipeline*-u. Za statički *pipeline* sa četiri stepena S_1 do S_4 ona bi izgledala kao na slici 7.

→ vreme u signalima takta **CLK**

		1	2	3	4
stepeni	S_1	*Z*			
pipe-	S_2		*Z*		
line-a	S_3			*Z*	
	S_4				*Z*

Slika 7 Tabela rezervacija za statički *pipeline*

Realizacija dinamičkog *pipeline*-a je dosta složena.

Dalja izlaganja će podrazumevati sinhroni statički *pipeline*.

2. Instrukcijski *pipeline*

U slučaju instrukcijskog *pipeline*-a izvršavanje instrukcije se deli na više faza i za svaku fazu postoji poseban stepen u *pipeline*-u. Instrukcija čije je izvršavanje podeljeno na k faza zahteva k stepeni u *pipeline*-u. U idealnom slučaju u *pipeline*-u se nalazi k instrukcija svaka u različitom stepenu *pipeline*-a i time u različitoj fazi izvršavanja. Ako je prva instrukcija u k-tom stepenu, druga je u $(k-1)$-om stepenu i tako redom do k-te instrukcije koja je u prvom stepenu *pipeline*-a. Ovako idealnu situaciju je nemoguće održati u *pipeline*-u zbog čitavog niza različitih razloga. Razlozi koji dovode do usporavanja *pipeline*-a, načini za ublažavanje njihovih negativnih efekata i čitav niz karakterističnih situacija koje se javljaju kod *pipeline* organizacije procesora biće ilustrovani na primeru jednog procesora. Odabrana je jednostavna arhitektura procesora da bi se izbeglo preveliko usložnjavanje *pipeline*-a. Međutim, ona sadrži dovoljno elemenata potrebnih da se prikažu osnovni principi *pipeline* organizacije procesora.

2.1 Arhitektura procesora

Arhitektura procesora je load/store tipa. Operandi se iz memorije dovlače u registre opšte namene procesora instrukcijom **load**. Operacije se specificiraju instrukcijama troadresnog formata pri čemu su dva izvorišna operanda ili dva registra opšte namene ili registar opšte namene i neposredna veličina, a odredište registar opšte namene procesora. Rezultati se vraćaju u memoriju instrukcijom **store**.

Registri

Postoje 32 32-bitna registra opšte namene za rad sa celobrojnim veličinama (GPR) označena sa $R0$ do $R31$. Postoje i 32 32-bitna registra opšte namene za rad sa veličinama u pokretnom zarezu jednostruke preciznosti na dužini 32 bita i dvostruke preciznosti na dužini 64 bita (FPR). Ukoliko se koriste za rad sa veličinama u pokretnom zarezu jednostruke preciznosti na dužini 32 bita na raspolaganju su sva 32 registra koji se tada označavaju se sa F0, F1, … , F31. Ukoliko se koriste za rad sa veličinama u pokretnom zarezu dvostruke preciznosti na dužini 64 bita na raspolaganju je 16 parova formiranih od parnih i neparnih 32 bitnih registara koji se tada označavaju sa F0, F2, … , F30. Postoje posebne **load** i **store** i aritmetičke instrukcije za rad sa registrima GPR i FPR. Takođe, postoje i instrukcije za transfer između registara GPR i FPR.

Vrednost registra R0 je uvek nula što omogućava ostvarivanje više korisnih efekata korišćenjem registra R0 u nekim instrukcijama i načinima adresiranja.

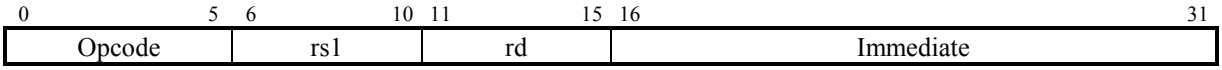
Tipovi podataka

Tipovi podataka su celobrojne veličine sa i bez znaka na dužini 8-bitnog bajta, 16-bitne polureči i 32-bitne reči i veličine u pokretnom zarezu jednostruke preciznosti na dužini 32 bita i dvostruke preciznosti na dužini 64 bita. Interno u procesoru sve operacije sa celobrojnim veličinama se izvršavaju nad 32-bitnim celobrojnim veličinama. Zbog toga se 8-bitni bajtovi i 16-bitne polureči prilikom dovlačenja u registre procesora instrukcijom **load** proširuju ili nulama u slučaju celobrojnih veličina bez znaka ili znakom u slučaju celobrojnih veličina sa znakom do dućine registra od 32 bita. Posle punjenja registara, sve operacije se izvršavaju samo nad 32-bitnim celobrojnim veličinama.

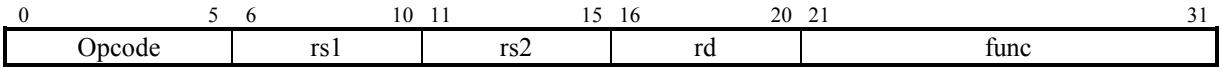
Formati instrukcija

Sve instrukcije su fiksne dužine 32 bita, pa se zbog bajtovskog adresiranja sadržaj registra *PC* pri očitavanju instrukcije uvećava za 4. Instrukcije koje se koriste u razmatranjima *pipeline* organizacije procesora imaju dva formata instrukcija i to **Tip I** i **Tip R** (slika 8).

Format instrukcija: **Tip I**



Format instrukcija: **Tip R**



Slika 8 Formati instrukcija

Format instrukcije **Tip I** koriste **load** i **store** instrukcije, **aritmetičke**, **logičke**, **relacione** i **pomeračke** instrukcije i **branch** instrukcije.

U slučaju **load i store** instrukcija sabira se sadržaj registra specificiranog poljem *rs1* i neposredna veličina specificirana poljem *immediate* da bi se dobila adresa memorijske lokacije. U slučaju instrukcije **load** sa te adrese se čita podatak i upisuje u registar opšte namene procesora specificiran poljem *rd*. U slučaju instrukcije **store** na toj adresi se upisuje podatak iz registra opšte namene procesora specificiranog poljem *rd*. Operacije **load** i **store** se specificiraju odgovarajućim vrednostima polja *opcode*.

U slučaju **aritmetičke** i **logičke** instrukcije odgovarajuća operacija se realizuje nad sadržajem registra specificiranim poljem *rs1* i neposrednom veličinom specificiranom poljem *immediate*, a dobijeni rezultat se upisuje u registar opšte namene procesora specificiran poljem *rd*. U slučaju **relacione** instrukcije operacijom se specifira relacija prema kojoj se vrši provera nad sadržajima registra specificiranim poljima *rs1* i neposrednom veličinom specificiranom poljem *immediate*, a u registar opšte namene procesora specificiran poljem *rd* upisuje vrednost l ukoliko relacija važi i vrednost 0 ukoliko relacija ne važi. U slučaju **pomeračke** instrukcije odgovarajuća operacija se realizuje nad sadržajem registra specificiranim poljem *rs1*, dok se dobijeni rezultat upisuje u registar opšte namene procesora specificiran poljem *rd*. Polje *immediate* se ne koristi. Operacije se specificiraju odgovarajućim vrednostima polja *opcode*.

U slučaju **branch instrukcije** poljem *rs1* se specificira registar čiji se sadržaj proverava da li je jednak nuli (instrukcija **beqz**) ili je različit od nule (instrukcija **bneq**). Neposredna veličina specificirana poljem *immediate* se koristi kao pomeraj. Polje *rd* se ne koristi. Operacije **beqz** i **bneq** se specificiraju odgovarajućim vrednostima polja *opcode*.

Format instrukcije **Tip R** koriste **aritmetičke, logičke** i **relacione** instrukcije. U slučaju **aritmetičke** i **logičke** instrukcije odgovarajuća operacija se realizuje nad sadržajima registara

specificiranim poljima *rs1* i *rs2*, a dobijeni rezultat se upisuje u registar opšte namene procesora specificiran poljem *rd*. U slučaju **relacione** instrukcije operacijom se specifira relacija prema kojoj se vrši provera nad sadržajima registara specificiranim poljima *rs1* i *rs2*, i u registar opšte namene procesora specificiran poljem *rd* upisuje vrednost l ukoliko relacija važi i vrednost 0 ukoliko relacija ne važi. Operacije se specificiraju odgovarajućim vrednostima polja *func*, pri čemu sve **aritmetičke, logičke** i **relacione** instrukcije ovog tipa imaju istu vrednost polja *opcode*.

Adresiranje

Memorijskim lokacijama se jedino pristupa pomoću **load** i **store** instrukcija. U slučaju obe instrukcije adresa memorijske lokacije se uvek dobija sabiranjem sadržaja jednog od registara GPR i 16-bitnog pomeraja pri čemu se adresa registra i pomeraj specificirani instrukcijom. Ovim je podržano registarsko indirektno adresiranje sa pomerajem. Kombinovanjem vrednosti registra i pomeraja mogu se realizovati i drugi načini adresiranja. Ukoliko se specificira da je pomeraj nula, dobija se registarsko indirektno adresiranje. U slučaju da je sadržaj specificiranog registra nula, a u tu svrhu se koristi registar R0, dobija se memorijsko direktno adresiranje.

U slučaju preostalih instrukcija odgovarajuće operacije se realizuju nad sadržajima dva registra ili nad sadržajem registra i neposrednom veličinom, a rezultat ide u registar. Različitim vrednostima polja koda operacije za istu operaciju se specificira da li je drugi operand registar ili neposredna veličina. Time su implicitno specificirani direktno registarsko i neposredno adresiranje.

Adresiranje je bajtovsko, jer postoje instrukcije **load** i **store** za rad sa 8-bitnim bajtom, 16-bitnom polureči i 32-bitnom reči. Adresa je dužine 32 bita. U slučaju 16-bitne, 32-bitne i 64-bitne veličine, generisana adresa je adresa najstarijeg bajta veličine, a sama veličina se nalazi u odgovarajućem broju sukcesivnih memorijskih lokacija

Skup instrukcija

Razmatrani procesor ima jednostavne instrukcije. Kod razmatranja karakterističnih situacija *pipeline* organizacije procesora pažnja će, najpre, biti usredsređena na instrukcije za rad sa celobrojnim veličinama koje se mogu grupisati u sledeće grupe:

- **load**/**store** instrukcije,
- **ALU** instrukcije i
- **branch** instrukcije.

Neke od instrukcija za svaku od ovih grupa su date na slici 9.

Značenja oznaka sa slike 9 su:

- Indeks se dodaje uz znak $\leftarrow$ da bi se specificirala dužina podatka koji se prenosi. Tako, na primer, sa $\leftarrow_n$ je označen prenos podatka dužine *n* bita.
- Indeks se koristi da označi selekciju određenog bita iz nekog polja, pri čemu su bitovi označeni da najstariji bit počinje od oznake 0. Indeksi mogu da se koriste da označe ili jedan bit ili opseg bitova. Tako, na primer, $Regs[R4]_0$ je oznaka za najstariji razred registra R4, dok je $Regs[R4]_{24\ldots31}$ oznaka za najmlaći bajt registra R4.
- Izložilac se koristi za umnožavanje (replicate) nekog polja. Tako, na primer, 0^{24} je oznaka za polje popunjeno nulama na dužini 24 bita.
- Simbol ## se koristi za spajanje (concatenate) dva polja.

Kao ilustracija ovih oznaka, pretpostavljajući da su R8 i R10 32-bitni registri, izraz

$Regs[R10]_{16...31} \leftarrow_{16} (Mem[Regs[R8]]_0)^8$ ## $Mem[Regs[R8]]$

označava da je bajt podatka očitan iz memorije sa adrese određene sadržajem registra R8 proširen znakom (sign extended) na 16-bitnu veličinu i potom je ta veličina upisana u donjih 16 razreda registra R10, dok je 16 gornjih razreda ostalo nepromenjeno.

1. load i **store** instrukcije:

lw	*R*1, *displ*(*R*2)	load word	$R1 \leftarrow_{32} M[displ + R2]$
sw	*displ*(*R*4), *R*3	store word	$M[displ + R4] \leftarrow_{32} R3$

2. ALU instrukcije:

2.1 aritmetičke instrukcije**:**

add	*R*1, *R*2, *R*3	add	$R1 \leftarrow R2 + R3$
addi	*R*1, *R*2, *immed*	add immediate	$R1 \leftarrow R2 + immed$

2.2 logičke instrukcije**:**

and	*R*1, *R*2, *R*3	and	$R1 \leftarrow R2 \ \& \ R3$
andi	*R*1, *R*2, *immed*	and immediate	$R1 \leftarrow R2 \ \& \ immed$

2.3 relacione instrukcije**:**

slt	*R*1, *R*2, *R*3	set less than	**if** ($R2 < R3$)	**then** $R1 \leftarrow 1$ **else** $R1 \leftarrow 0$
slti	*R*1, *R*2, *immed*	set less than immediate	**if** ($R2 < immed$)	**then** $R1 \leftarrow 1$ **else** $R1 \leftarrow 0$

2.4 pomeračke instrukcije**:**

sll	*R*1, *R*2	shift left logical	$R1 \leftarrow$ shift $R2$

3. branch instrukcije:

beqz	*R*4, *displ*	branch equal zero	**if** ($R4 = 0$) **then** $PC \leftarrow PC + displ$
bneq	*R*4, *displ*	branch not equal zero	**if** ($R4 \neq 0$) **then** $PC \leftarrow PC + displ$

Slika 9. Skup instrukcija

U okviru grupe **load**/**store** instrukcija postoje posebne instrukcije za rad sa 8-bitnim, 16-bitnim i 32-bitnim veličinama sa znakom i bez znaka. Ove instrukcije koriste registre R0, R1, …, R31. U ovoj grupi su posebne **load**/**store** instrukcije za rad sa 32-bitnim i 64-bitnim veličinama u pokretnom zarezu. Ove instrukcije koriste registre F0, F1, …, F31.

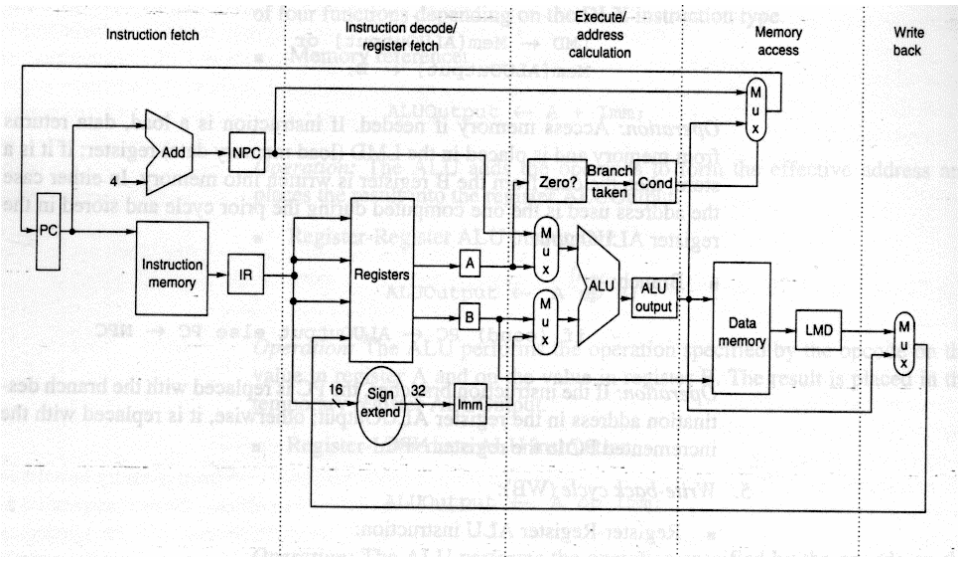
U okviru grupe **ALU** instrukcija postoje artimetičke, logičke, relacione i pomeračke instrukcije. Aritmetičke, logičke i relacione instrukcije koriste oba formata instrukcija, dok pomeračke koriste samo format **Tip I**. Od aritmetičkih operacija postoje operacije sabiranja i oduzimanja. Operacija sabiranja u formatu **Tip I** gde je jedan izvorišni operand registar R0 a drugi neposredna veličina, moće se koristiti za punjenje registra konstantom. Takođe, operacija sabiranja u formatu **Tip R**, gde je jedan izvorišni operand registar R0, može se koristiti za prenos sadržaja iz jednog registra u drugi. Od logičkih operacija postoje operacije logičko I, logičko ILI i logičko ekskluzivno ILI. Od relacionih operacija postoji 6 operacija za 6 relacija, i to: jednako, nije jednako, veće, veće ili jednako, manje i manje ili jednako. Od

pomeračkih operacija postoje aritmetičko pomeranje, logičko pomeranje i rotiranje za jedno mesto ulevo ili udesno.

U okviru grupe **branch** instrukcija postoje samo dve instrukcije koje realizuju relativni skok ako je u jednom slučaju sadržaj specificiranog registra nula, a u drugom ukoliko nije nula.

2.2 Organizacija procesora bez *pipeline*-a

Da bi se došlo do *pipeline* organizacije procesora, najpre se razmatra organizacija procesora bez *pipeline*-a. Između više mogućih organizacija procesora namerno je odabrana ona organizacija iz koje prirodno proizlazi *pipeline* organizacija. Zbog toga odabrana organizacija nije ni najekonomičnija ni sa najboljim performansama, među mogućim organizacijama bez *pipeline*-a. Usvojeno je da se izvršavanje instrukcija podeli na najviše pet faza i da izvršavanje jedne faze traje jednu periodu signala takta (slika 10). Tih pet faza su: IF (instruction fetch), ID (instruction decode and register fetch), EX (execute and effective address calculation), MEM (memory access and branch completion) i WB (write back). Sa slike 10 se vidi kako teče izvršavanje instrukcija. Na kraju svake periode signala takta, vrednost sračunata za vreme date periode signala takta, a potrebna za vreme neke kasnije periode signala takta bilo te ili neke sledeće instrukcije, se upisuje u memoriju, neki od registara opšte namene, programski brojač PC ili u neki od pomoćnih registara, kao na primer LMD, Imm, A, B, IR, NPC, ALUOUT ili cond. Pomoćni registri drže vrednosti između perioda signala takta iste instrukcije, dok memorijske lokacije, registri opšte namene i programski brojač PC, kao programski vidljivi registri, drže vrednosti između instrukcija. Stoga izvršavanje jedne instrukcije može da traje najviše pet perioda signala takta.



Slika 10 Organizacija procesora bez *pipeline*-a

U svakoj od pet faza za svaku grupu instrukcija realizuju se operacije date u daljem tekstu. U datim operacijama Instruction memory i Data memory se označavaju sa Mem, a registarski fajl Registers sa Regs.

1) IF—instruction fetch

$IR \leftarrow Mem[PC]$;

$NPC \leftarrow PC + 4$;

Operacija: U ovoj fazi registar *PC* se koristi za čitanje instrukcije iz memorije Instruction Memory i očitana instrukcija se smešta u prihvatni registar instrukcije *IR*. Pored toga, programski brojač *PC* se uvećava za 4 u jedinici Add i upisuje u registar *NPC*. Prihvatni registar instrukcije *IR* se koristi za čuvanje očitane instrukcije u narednih nekoliko perioda signala takta u kojima se izvršavaju preostale faze instrukcije. Registar *NPC* sadrži adresu prve sledeće sekvencijalne instrukcije.

2) ID—instruction decode and register fetch

$A \leftarrow Regs[IR_{6..10}]$;
$B \leftarrow Regs[IR_{11..15}]$;
$Imm \leftarrow ((IR_{16})^{16} \# \# IR_{16..31})$;

Operacija: U ovoj fazi se dekoduje instrukcija i pristupa registarskom fajlu Registers radi čitanja dva izvorišna registra koji se, potom, upisuju u pomoćne registre *A* i *B*. Paralelno sa tim nižih 16 razreda registra *IR*, koji sadrže 16 nižih bitova neposredne veličine, se u jedinici Sign extend proširuju znakom na 32-bitnu veličinu i smeštaju pomoćni registar *Imm*. Pomoćni registri *A*, *B* i *Imm* se koriste u sledećim fazama. Dekodovanje instrukcije, čitanje registara i formiranje 32-bitne neposredne veličine se realiziju paralelno. To je moguće da se realizuje zbog toga što su odgovarajuća polja na fiksnim mestima u formatu instrukcije. Treba uočiti da se u nekim instrukcijama u kasnijim fazama neki od registara *A*, *B* i *Imm* neće koristiti. Međutim, to što se u sva tri registra uvek upisuju neke vrednosti ne nanosi nikakvu štetu onda kada to nije potrebno.

3) EX—execution and effective address calculation

U ovoj fazi jedinica ALU izvršava u zavisnosti od instrukcije u registru *IR* jednu od pet funkcija za pet tipova instrukcija nad operandima pripremljenim u prethodnoj fazi i koji se nalaze u registrima *A*, *B* i *Imm*. Rezultat jedinice ALU se upisuje u registar *ALUOUT*. Paralelno sa tim jedinica Zero? vrši proveru da li je sadržaj registra *A* nula. Rezultat jedinice Zero? se upisuje u jednorazredni pomoćni registar cond samo u slučaju **branch** instrukcija, dok se za preostale instrukcije u registar cond uvek upisuje nula.

3.1) load i **store** instrukcije

$ALUOUT \leftarrow A + Imm$;
$cond \leftarrow 0$;

Operacija: U jedinici ALU se sabira sadržaj registra i pomeraj iz instrukcije koji se nalaze u registrima *A* i *Imm*, respektivno, da bi se formirala efektivna adresa koja se upisuje u registar *ALUOUT*.

3.2) Aritmetičke, logičke i relacione instrukcije **Tipa R**

$ALUOUT \leftarrow A$ func B;
$cond \leftarrow 0$;

Operacija: U jedinici ALU se izvršava operaciju specificirana kodom operacije func nad sadržajima dva registra koji se nalaze u registrima *A* i *B* i upisuje rezultat u registar *ALUOUT*.

3.3) Aritmetičke, logičke i relacione instrukcije **Tipa I**

ALUOUT←*A* op *Imm*;
cond←0;

Operacija: U jedinici ALU se izvršava operaciju specificirana kodom operacije *opcode* nad sadržajem registra i neposrednom veličinim koji se nalaze u registrima *A* i *Imm*, respektivno, i upisuje rezultat u registar *ALUOUT*.

3.4) Pomeračke instrukcije

ALUOUT←op *A*;
cond←0;

Operacija: U jedinici ALU se izvršava operacija specificirana kodom operacije *opcode* nad sadržajem registra koji se nalazi u registru *A* i upisuje rezultat u registar *ALUOUT*.

3.5) branch instrukcije

ALUOUT←*NPC* + *Imm*;
cond←(*A* op 0);

Operacija: U jedinici ALU se sabira sadržaj PC-ja date instrukcije uvećan za 4 i pomeraj iz instrukcije, koji se nalaze u registrima *NPC* i *Imm*, respektivno, da bi se sračunala adresa skoka. Paralelno sa tim jedinica Zero? proverava da li je sadržaj registra *A* nula ili ne i na osnovu toga upisuje u jednorazredni registar *cond* vrednost 1 ili 0. Operator *op* je određen na osnovu toga da li je kod operacije za **beqz** ili **bneq**. U slučaju beqz upisuje se 1 u cond, ako je sadržaj nula, a u slučaju bneq upisuje se 1 u cond, ako sadržaj nije nula.

4) MEM—memory access and branch completion

U ovoj fazi se izvršavaju dve vrste operacija u zavisnosti od instrukcije u registru *IR*, i to prva vrsta samo za **load** i **store** instrukcije i druga vrsta za sve instrukcije.

4.1) load/store instrukcije

LMD←*Mem*[*ALUOUT*]; za **load**
Mem[*ALUOUT*]←*B*; za **store**

Operacija: U ovoj fazi se pristupa Data memoriji pa se za instrukciju **load** čita iz memorije i upisuje u registar *LMD*, a za instrukciju **store** sadržaj iz registra *B* se upisuje u memoriju. U oba slučaja koristi se adresa sračunata u prethodnoj fazi koja se nalazi u registru *ALUOUT*.

4.2) Sve instrukcije

if (*cond*) **then** *PC*←*ALUOUT* **else** *PC*←*NPC*

Operacija: U slučaju svih instrukcija sem **branch** instrukcija je u prethodnoj fazi u registar *cond* upisana vrednost nula, pa se u registar *PC* iz registra *NPC* kao adresa sledeće instrukcije upisuje adresa tekuće instrukcije uvećana za 4 i time produžava sa sekvencijalnim izvršavanjem programa. U slučaju **branch** instrukcija u prethodnoj fazi je u registar *cond* upisana ili vrednost nula ili jedan. Ako je u registru *cond* vrednost nula onda se u registar *PC* iz registra *NPC* kao adresa sledeće instrukcije upisuje adresa tekuće instrukcije uvećana za 4 i time produžava sa sekvencijalnim izvršavanjem programa. Ako je u registru *cond* vrednost jedan onda se u registar *PC* iz registra *ALUOUT* kao adresa sledeće instrukcije upisuje sračunata adresa skoka.

5) WB—write back

5.1) ALU instrukcije **Tipa R**

$Regs[IR_{16..20}] \leftarrow ALUOUT$;

5.2) ALU instrukcije **Tipa I**

$Regs[IR_{11..15}] \leftarrow ALUOUT$;

5.3) load instrukcija

$Regs[IR_{11..15}] \leftarrow LMD$;

Operacija: U ovoj fazi se vrši upis u registarski fajl Registers i to očitana vrednost iz memorije Data Memory koja se nalazi u registru *LMD* za instrukciju **load** i rezultat jedinice *ALU* koji se nalazi u registru *ALUOUT* za ALU instrukcije.

Uz ovako usvojenu organizaciju procesora, jedan pristup bi bio da se svaka instrukcija izvršava u pet taktova pri čemu bi perioda signala takta odgovarala trajanju najsporije faze. U tom slučaju u fazi WB ne bu bilo nikakvih aktivnosti za **branch** i **store** instrukcije, a u fazi MEM za ALU operacije. Upravljačka jedinica bi mogla da se realizuje korišćenjem standardnih tehnika kao što su, na primer, ožičena ili neka od mikroprogramskih realizacija. Tada bi gore pomenute grupe instrukcija mogle da se realizuju u četiri takta. Pored toga određene uštede bi i u organizaciji procesora mogle da se naprave. Tako, na primer, umesto posebnih Instruction Memory i Data Memory, kojima se pristupa u različitim taktovima, mogla bi da se koristi samo jedna memorija. Takođe, uvećavanje vrednosti registra PC u jedinici Add i izvršavanje neke od ALU operacija ili sračunavanje adrese memorijske lokacije ili adrese instrukcije na koju se skače u jedinici ALU se realizuju u različitim taktovima. Stoga bi jedinica Add mogla da se izbaci i da se uvećavanje vrednosti registra PC realizuje u jedinici ALU. Međutim, usvojena organizacija procesora sa slike 10 je dobra osnova za prelazak na *pipeline* organizaciju procesora, pa pomenute izmene neće biti uvedene.

Kao alternativa ovom rešenju sa pet taktova po instrukciji, moguće je i rešenje kod koga bi bio jedan takt po instrukciji pet puta dužeg trajanja od trajanja takta u prethodno razmatranoj realizaciji. U tom slučaju pomoćni registri IR, NPC, A, B, Imm, ALUOUT, Cond i LMD bi bili izbačeni, jer nema potrebe za pamćenjem rezultata izvršavanja jedne faze instrukcije i njihovim prenošenjem u deo procesora u kome se realizuje sledeća faza. Sada se cela instrukcija izvršava u jednom pet puta dužem taktu, pri čemu se dobijeni rezultati upisuju na kraju tog takta u Data Memory, Registers i registar PC.

Sva dalja razmatranja biće usmerena na potrebnu nadgradnju procesora sa slike 10, da bi se dobila organizacija procesora sa *pipeline*-om.

2.3 Organizacija procesora sa *pipeline*-om

U slučaju organizacije procesora sa *pipeline*-om svakoj od pet faza kroz koje prolazi instrukcija tokom izvršavanja odgovara poseban stepen u sinhronom statičkom *pipeline*-u. Na svaki signal takta ubacuje se nova instrukcija u *pipeline*. Posle pet signala takta u pet stepeni *pipeline*-a izvršava se pet različitih faza za instrukcije i do $i + 4$ (slika **11**). Tih pet stepeni *pipeline*-a imaju iste nazive kao i odgovarajuće faze koje se u njima izvršavaju.

Za izvršavanje svake instrukcije sada je potrebno pet perioda signala takta, ali za vreme svake periode signala takta pet stepeni *pipeline*-a izvršava pet različitih faza pet različitih

instrukcija. Tako, na primer, u periodi 5 signala takta, instrukcija *i* izvršava fazu WB, instrukcija $i + 1$ fazu MEM, itd. Program se kod ovakvog načina izvršavanja instrukcija izvršava pet puta brže nego kada bi instrukcija $i + 1$ počela sa svojom fazom IF tek u periodi 6 signala takta i to tek pošto u periodi 5 signala takta instrukcija *i* izvrši svoju fazu WB.

instrukcija	signal takta								
	1	2	3	4	5	6	7	8	9
i	IF	ID	EX	MEM	WB				
$i + 1$		IF	ID	EX	MEM	WB			
$i + 2$			IF	ID	EX	MEM	WB		
$i + 3$				IF	ID	EX	MEM	WB	
$i + 4$					IF	ID	EX	MEM	WB

Slika 11. Preklapanjem izvršavanja pet faza, pet instrukcija u pet stepeni *pipeline*-a

Izvršavanje jedne instrukcije kod organizacije procesora sa *pipeline*-om se sada čak i produžava u odnosu na organizaciju procesora bez *pipeline*-a. Kod procesora bez *pipeline*-a vreme izvršavanja jedne instrukcije je suma pojedinačnih trajanja pet faza instrukcije. Kod procesora sa *pipeline*-om vreme izvršavanja jedne instrukcije je proizvod 5 puta trajanje najsporije faze. Situacija je još nepovoljnija posmatrano sa aspekta samo jedne instrukcije, jer trajanje najsporije faze treba produžiti za *set-up* vreme prihvatnih registara i *clock skew* vreme koje se javlja pri razvođenju centralizovanog signala takta na prihvatne registre stepeni *pipeline*-a. Međutim, iako se pojedinačno instrukcije sporije izvršavaju, program kao celina se izvršava brže.

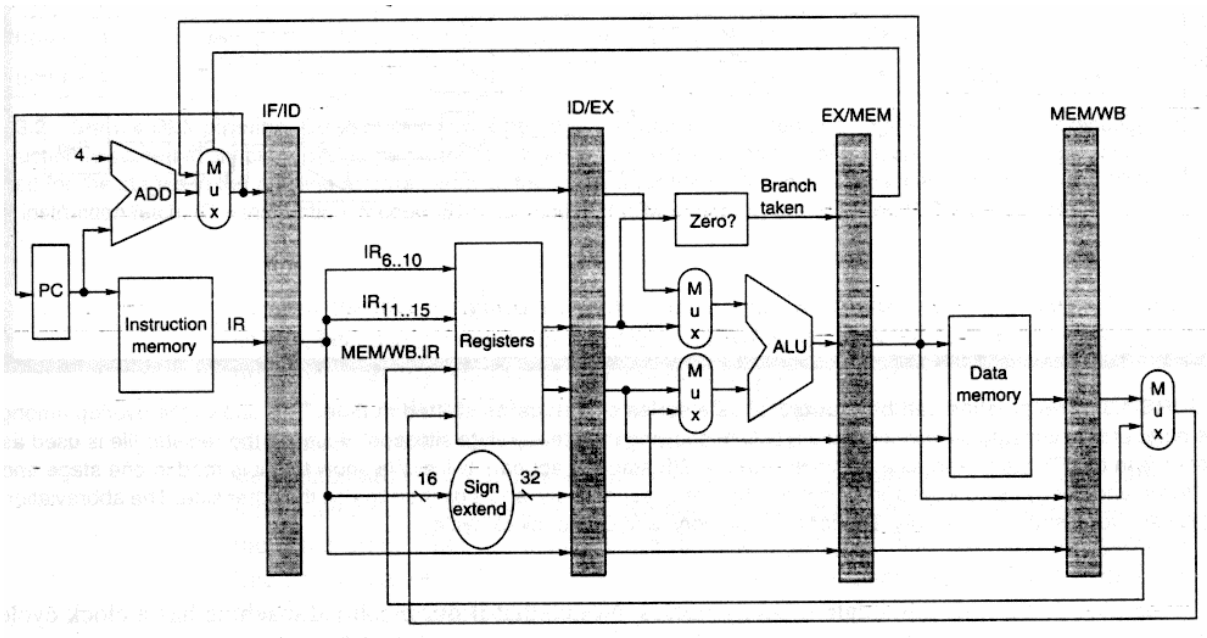
Problemi koji mogu da nastanu u *pipeline*-u i rešenja koja treba preduzeti se razmatraju u daljem tekstu.

Prvi mogući problem se odnosi na organizaciju memorije. U organizaciji procesora sa slike 10 već postoje posebne Instruction Memory i Data Memory. One se obično realizuju kao posebne keš memorije za instrukcije i podatke. Korišćenjem posebnih keš memorija eliminiše se konflikt koji bi nastajao u slučaju postojanja samo jedne kad god bi se u stepenu MEM našla **load** ili **store** instrukcija. Tada bi iz stepena MEM bilo obraćanje radi čitanja ili upisa podatka, a iz stepena IF radi čitanja instrukcije.

Drugi problem se odnosi na pristup registar fajlu Registers. Sa slike 10 se vidi da se iz stepena ID čitaju dva registra, a iz stepena WB vrši upis. Ove operacije su skoro stalno prisutne, pa se za realizaciju registarskog fajla obično koriste takvi memorijski elementi koji omogućavaju istovremeno čitanje dva registra i upis trećeg.

Treći problem se odnosi na ažuriranje vrednosti registra PC. Da bi se startovala nova instrukcija na svaki takt, vrednost registra PC mora da se uveća za 4 i ta vrednost upiše u registar PC na svaki takt. Formiranje vrednosti PC uvećane za 4 na izlazu jedinice Add se realizuje paralelno sa čitanjem instrukcije prema vrednosti registra PC. Na signal takta se istovremeno upisuju u registar PC uvećana vrednost PC-ja i u registar IR očitana instrukcija. Problem nastaje zbog **branch** instrukcija koje takođe menjaju vrednost registra PC, ali tek u stepenu MEM. Kada **branch** instrukcija dođe u stepen MEM u kome eventualno skok treba da se realizuje u slučaju da je uslov za skok ispunjen javljaju se dva problema. Prvi je da skok treba napraviti relativno u odnosu na vrednost registra PC **branch** instrukcije uvećan za 4, a tada je u registru *NPC* vrednost PC-ja branch instrukcije uvećan za 12. Drugi je da se u stepenima IF, ID i EX nalaze pogrešne instrukcije. Ovi problemi koji nastaju kod **branch** instrukcija posebno se kasnije razmatraju.

U slučaju preklapanja izvršavanja različitih faza više instrukcija u stepenima *pipeline*-a, svaki stepen *pipeline*-a je aktivan na svaku periodu signala takta. Zbog toga sve operacije u svim stepenima *pipeline*-a moraju da se kompletiraju u toku trajanja jedne periode signala takta. Pored toga svaki stepen *pipeline*-a mora da ima svoj registar u kome će biti sve ono što je neophodno za izvršavanje odgovarajuće faze u njemu, kao i za izvršavanje preostalih faza date instrukcije u sledećim stepenima *pipeline*-a. Uvođenjem tih registara, koji će se nazivati *pipeline* registri, u procesor sa slike 10, dolazi se do organizacije procesora sa slike 12. Ovi registri su označeni imenima stepena koje povezuju, pa je sa IF/ID označen registar između stepena IF i stepena ID, koji drži sve informacije neophodne za realizaciju faze ID. Isto važi i za registre ID/EX, EX/MEM i MEM/WB i faze i stepene EX, MEM i WB, respektivno. Registar PC se, takođe, može tretirati kao *pipeline* registar jer on drži informacije potrebne za fazu i stepen IF. Treba imati u vidu i činjenicu da se u nekom *pipeline* registru ne nalaze samo informacije potrebne za stepen kome on pripada, već i za preostale stepene. Stoga će po isteku perioda takta ne samo rezultat faze ID biti upisan u registar ID/EX, već i neke informacije iz registra IF/ID. Kao primer ovoda se može uzeti polje koje specificira odredišni registar za ALU instrukciju. Ovo polje mora da se prenosi zajedno sa instrukcijom do stepena WB u kome se vrši upis u registar. Kao adresa odredišnog registra koristi se vrednost ovog polja iz MEM/WB registra. Slična situacija je i sa vrednošću registra PC uvećanom za 4 koja se prenosi sa svojom instrukcijom sve do stepena EX u kome se koristi za sračunavanje adrese skoka, ako je ta instrukcija neka od **branch** instrukcija.



Slika 12 *Pipeline* organizacija procesora

Procesor sa slike 12 se razlikuje od procesora sa slike 10 i po tome što je multiplekser za selekciju vrednosti za upis u registar PC prebačen iz stepena MEM u stepen IF. Time je osigurano da se u registar PC upisuje samo iz jednog stepena i to stepena IF. Da nije bilo ovog prebacivanja javljale bi se situacije kada bi se iz stepena IF i stepena MEM istovremeno upisivale dve različite vrednosti. Zbog *pipeline* organizacije i potrebe da se u ritmu takta ubacuju nove instrukcije u *pipeline*, stepen IF mora na svaki takt da u registar PC upiše novu vrednost PC-ja uvećanu za 4. Međutim, kada se u stepenu MEM nađe **branch** instrukcija za koju je uslov za skok ispunjen, onda bi iz stepena MEM trebalo u registar PC da se upiše

sračunata adresa skoka. Tada stepeni IF i MEM pokušavaju da na isti takt u registar PC upišu dve različite vrednosti.

Treba uočiti da je najčešći tok informacija kroz stepene *pipeline*-a sleva udesno od IF preko ID, EX i MEM do WB. Pored njega, međutim, postoji i tok informacija sdesna ulevo. To je u slučaju kada iz stepena WB informacija ide u stepen ID radi upisa u neki registar registar fajla, ili iz stepena MEM u stepen IF radi upisa adrese skoka u registar PC u slučaju **branch** instrukcije. Tok informacija sdesna ulevo u *pipeline*-u stvara određene konfliktne situacije zbog kojih se usporava kretanje instrukcija kroz stepene *pipeline*-a. Zbog toga će kasnije biti razmatrane tehnike za eliminisanje tih konfliktnih situacija ili bar ublažavanje negativnih efekata koji nastaju zbog njih.

Operacije koje se izvršavaju u stepenima *pipeline*-a date su na slici 13. U stepenu IF se na osnovu sadržaja registra PC čita instrukcija i sračunava vrednost PC-ja uvećana za 4. Ukoliko je vrednost jednorazrednog registra cond koja dolazi iz stepena MEM nula, to znači ili da se u stepenu MEM nalazi instrukcija koja nija **branch** instrukcija, ili da je **branch** instrukcija, ali da uslov za skok nije ispunjen i da treba produžiti sa sekvencijalnim očitavanjem instrukcija. Stoga se kroz multiplekser u stepenu IF propušta uvećana vrednost PC-ja a ne vrednost koja dolazi iz stepena MEM. Na signal takta uvećana vrednost PC-ja se upisuje u registar PC. Na isti signal takta uvećana vrednost PC-ja i očitana instrukcija se upisuju u registre IR i NPCC, respektivno, *pipeline* registra IF/ID. Time se očitana instrukcija prebacije iz stepena IF u stepen ID. Sa instrukcijom ide i uvećana vrednost PC-ja, da bi se eventualno u stepenu EX, ukoliko se utvrdi da je to **branch** instrukcija, koristila za sračunavanje adrese skoka. Ukoliko je vrednost jednorazrednog registra cond, koja dolazi iz stepena MEM jedinica, to znači da se u stepenu MEM nalazi **branch** instrukcija i da je uslov za skok ispunjen, pa se u stepenima *pipeline*-a EX, ID i IF nalaze instrukcije koje su čitane sekvencijalno iza instrukcije **branch** i koje su nekorektne. Tada se kroz multiplekser u stepenu IF propušta vrednost koja dolazi iz stepena MEM i koja predstavlja adresu instrukcije koju treba izvršiti posle **branch** instrukcije i instrukcije u stepenima EX, ID i IF pretvaraju u instrukcije bez dejstva. Na signal takta se u registar PC upisuje adresa instrukcije na koju se skače, pa se sada u stepenu IF vrši njeno očitavanje, dok se u *pipeline* registre IF/ID, ID/EX i EX/MEM stepeni ID, EX i MEM, respektivno, ubacuju instrukcije bez dejstva.

U stepenu ID se čitaju iz registarskog fajla Registers sadržaji dva registra i u jedinici Sign extend formira 32-bitna neposredna veličina, pa se na signal takta upisuju u registre A i B i Imm *pipeline* registra ID/EX. Pored toga registri IR i NPC se samo prenose kroz stepen ID iz *pipeline* registra IF/ID u *pipeline* registar ID/EX.

U stepenu EX jedinica ALU formira adresu memorijske lokacije sabiranjem sadržaja registara A i Imm za instrukcije **load** i **store**, i izvršava odgovarajuću operaciju nad sadržajima ili registara A i B ili A i Imm za ALU operacije il formira adresu instrukcije skoka sabiranjem sadržaja registara NPC i Imm za slučaj **branch** instrukcije. Paralelno sa ovim jedinica Zero? za **branch** instrukcije vrši proveru sadržaja registra A i formira vrednost jedan ako je uslov za skok ispunjen, odnosno formira vrednost nula za sve ostale instrukcije. Na signal takta sadržaj sa izlaza jedinice ALU i jedinice Zero se upisuje u registre ALUOUT i cond *pipeline* registra EX/MEM. Pored toga registri IR i B se samo prenose kroz stepen EX iz *pipeline* registra ID/EX u *pipeline* registar EX/MEM. Registar B se prenosi u stepen MEM za slučaj da je to **store** instrukcija koja u ovom stepenu treba da upiše u Data memory sadržaj ovog registra.

U stepenu MEM se u slučaju **load** ili **store** instrukcija vrši čitanje iz ili upis u Data Memory. U oba slučaja adresa je u registru ALUOUT *pipeline* registra EX/MEM. U slučaju instrukcije **store** upisuje se sadržaj registra EX/MEM, a u slučaju **load** instrukcije očitana

vrednost se upisuje u registar LMD *pipeline* registra EX/MEM. Pored toga registri IR i ALUOUT se samo kroz stepen MEM iz *pipeline* registra EX/MEM upisuju u *pipeline* registar MEM/WB. Registar ALUOUT se prenosi iz stepena WB za slučaj da je to ALU operacija koja u ovom stepenu treba da upiše u registarski fajl Registers sadržaj ovog registra. U toku MEM faze signal cond iz *pipeline* registra EX/MEM će kroz multiplekser stepena IF propustiti ili vrednost PC-ja uvećanu za 4 sa izlaza jedinice Add ili adresu instrukcije na koju treba da se skoči iz registra ALUOUT *pipeline* registra EX/MEM. Selektovana vrednost se na signal takta upisuje u registar PC. Time se obezbeđuje da se produži sa sekvencijalnim očitavanjem instrukcija, ukoliko je registar cond *pipeline* registra EX/MEM bio nula, odnosno realizuje skok ukoliko je njihova vrednost jedan.

stepen	**instrukcija**
IF	**sve instrukcije** IF/ID.IR ← Mem[PC]; IF/ID.NPC, PC ← (if EX/MEM.cond then {EX/MEM.ALUOUT} else {PC + 4});
ID	**sve instrukcije** ID/EX.A ← Regs[IF/ID.$IR_{6...10}$]; ID/EX.B ← Regs[IF/ID.$IR_{11...15}$]; ID/EX.NPC ← IF/ID.NPC; ID/EX.IR ← IF/ID.IR; ID/EX.Imm ← (IF/ID.IR_{16})^{16} ## IF/ID.$IR_{16...31}$;
EX	**ALU instrukcije** EX/MEM.IR ← ID/EX.IR; EX/MEM.ALUOUT ← ID/EX.A func ID/EX.B; ili EX/MEM.ALUOUT ← ID/EX.A op ID/EX.Imm; EX/MEM.cond ← 0;
	load/store instrukcije EX/MEM.IR ← ID/EX.IR; EX/MEM.ALUOUT ← ID/EX.A + ID/EX.Imm; EX/MEM.cond ← 0; EX/MEM.B ← ID/EX.B;
	branch instrukcije EX/MEM.ALUOUT ← ID/EX.NPC + ID/EX.Imm; EX/MEM.cond ← (ID/EX.A op 0);
MEM	**ALU instrukcije** MEM/WB.IR ← EX/MEM.IR; MEM/WB.ALUOUT ← EX/MEM.ALUOUT;
	load/store instrukcije MEM/WB.IR ← EX/MEM.IR; MEM/WB.LMD ← Mem[EX/MEM.ALUOUT] ili Mem[EX/MEM.ALUOUT] ← EX/MEM.B;
WB	**ALU instrukcije** Regs[MEM/WB.$IR_{16...20}$] ← MEM/WB.ALUOUT; ili Regs[MEM/WB.$IR_{11...15}$] ← MEM/WB.ALUOUT;
	load instrukcije Regs[MEM/WB.$IR_{11...15}$] ← MEM/WB.LMD;

Slika 13 Operacije koje se izvršavaju u stepenima *pipeline*-a

U stepenu WB se vrši upis u registar registarskog fajla Registers ukoliko je reč o instrukciji **load** ili nekoj od ALU instrukcija. U slučaju instrukcije **load** upisuje se sadržaj registra LMD iz *pipeline* registra MEM/WB, a u slučaju ALU instrukcija sadržaj registra ALUOUT iz *pipeline* regostra MEM/WB. Adresa registra registarskog fajla Registers je specificirana

bitovima 16 do 20 registra IR *pipeline* registra MEM/WB za slučaj ALU operacija tipa R, odnosno bitovima 11 do 15 istog registra za slučaj ALU operacija tipa I i **load** instrukcije.

Treba zapaziti da se vrednost registra IR prenosi kroz sve stepene *pipeline*-a pri čemu je taj registar u stepenu ID označen sa IF/ID.IR, u stepenu EX sa ID/EX.IR itd. To se čini zbog toga što su informacije iz ovog registra potrebne u svim stepenima *pipeline*-a. Zbog jednostavnosti je uzeto da se od stepena IF do stepena WB prenosi ceo registar, mada sa napredovanjem instrukcije kroz stepene *pipeline*-a sve manje i manje informacija iz registra IR je potrebno.

Za korektan to podataka kroz stepene *pipeline*-a treba generisati odgovarajuće vrednosti upravljačkih signala za četiri multipleksera sa slike 12. Dva multipleksera u stepenu EX kontrolišu vrednosti koje se pripuštaju na dva ulaza jedince ALU u zavisnosti od registra IR u *pipeline* registru ID/EX. Gornji multiplekser propušta registar NPC iz *pipeline* registra ID/EX ako je **branch** instrukcija, a u ostalim situacijama registar A iz *pipeline* registra ID/EX. Donji multiplekser propušta registar B iz *pipeline* registra ID/EX ako je ALU instrukcija tipa R, a u ostalim situacijama registar Imm iz *pipeline* registra ID/EX. Multiplekser u stepenu IF propušta ili vrednost registra PC iz stepena IF uvećanu za 4 ili sračunaru adresu skoka koja dolazi iz stepena MEM i to registra ALUOUT *pipeline* registra EX/MEM. Ovim multiplekserom upravlja jednorazredni registar cond iz *pipeline* registra EX/MEM. Multiplekser u stepenu WB propušta ili registar LMD iz *pipeline* registra MEM/WB, ako je u WB stepenu **load** instrukcija, ili ALUOUT registar iz *pipeline* registra MEM/WB, ako je u WB stepenu neka od ALU instrukcija. Vrsta instrukcije u stepenu WB je određena sadržajem registra IR *pipeline* registra MEM/WB.

U stepenu WB se nalazi još jedan multiplekser koji nije prikazan na slici 12. Taj multiplekser kao adresu odredišnog registra registarskog fajla Registers propušta ili rezrede 16 do 20 registra IR *pipeline* registra MEM/WB za slučaj ALU instrukcija tipa R ili razrede 11 do 15 registra IR *pipeline* registra MEM/WB za slučaj ALU instrukcija tipa I ili **load** instrukcije.

Takođe, uz svaku instrukciju se iz jednog stepena u drugi stepen *pipeline*-a prenosi vrednost registra IR, pri čemu je taj registar u stepenu ID označen sa IF/ID.IR, u stepenu EX sa ID/EX.IR, itd.

3. Hazardi u *pipeline*-u

Prilikom *pipeline* izvršavanja instrukcija u *pipeline*-u mogu da se stvore takve situacije da za neku instrukciju ne može da se izvrši faza predviđena stepenom *pipeline*-a u kome se ona nalazi. Ove situacije se nazivaju hazardima. Postoje tri vrste hazarda:

- strukturalni hazardi,
- hazardi podataka i
- upravljački hazardi.

Strukturalni hazardi nastaju zbog potrebe da se istovremeno pristupi istom resursu od strane instrukcija koje se nalaze u različitim stepenima *pipeline*-a. Hazardi podataka nastaju kada je pristup nekom podatku od strane neke instrukcije u nekom stepenu *pipeline*-a uslovljeno prethodnim pristupom tom istom podatku od strane neke prethodne instrukcije iz nekog drugog stepena *pipeline*-a. Upravljački hazardi nastaju zbog skokova i drugih instrukcija koje menjaju vrednost programskog brojača *PC*.

U slučaju hazarda instrukcija se zaustavlja u nekom stepenu *pipeline*-a onoliko perioda signala takta koliko je potrebno da se uzrok hazarda otkloni. Zaustavljanje neke instrukcije u nekom stepenu *pipeline*-a zaustavlja i instrukcije koje su iza nje u *pipeline*-u. Time se zaustavlja i ubacivanje novih instrukcija u *pipeline*. Instrukcijama koje su ispred nje u *pipeline*-u može se dozvoliti da produže sa izvršavanjem. Zbog toga je direktna posledica pojavljivanja hazarda to da je ubrzavanje izvršavanja programa u *pipeline* procesoru lošije od onog koje bi teoretski moglo da se postigne.

3.1 Strukturalni hazardi

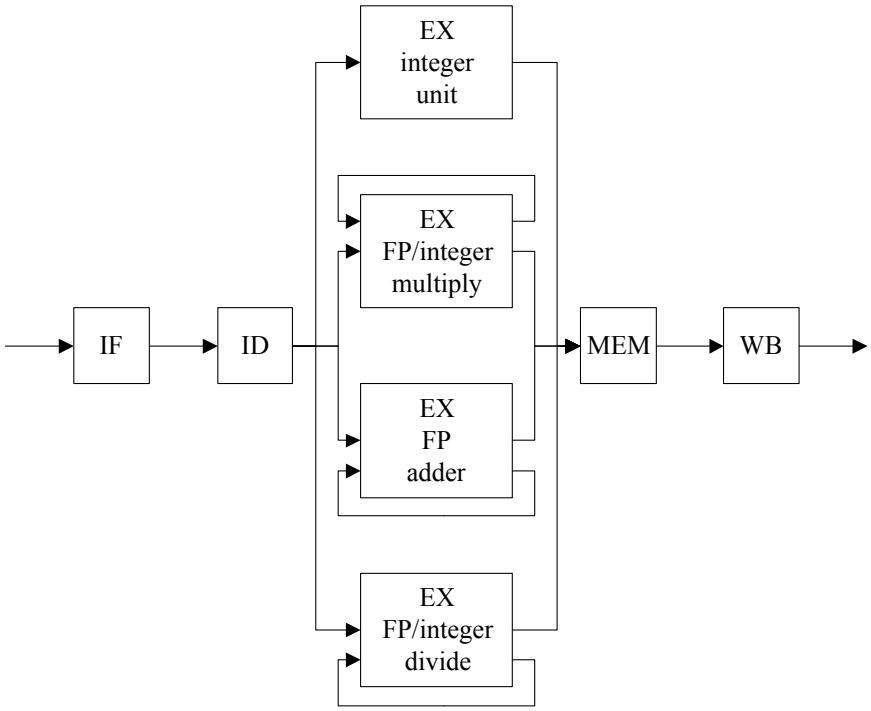
Strukturalni hazard u *pipeline* procesoru nastaje kada dve instrukcije koje se nalaze u različitim stepenima *pipeline*-a treba da pristupe istom resursu. Tipičan primer za ovo je sistem u kome se koristi ista memorija i za instrukcije i za podatke. Ovaj hazard bi mogao da se javi u slučaju razmatranog procesora ukoliko ne bi postojale posebne memorije za instrukcije i podatke. Hazard bi se desio kada bi **load** ili **store** instrukcija došla u stepen MEM u kome bi se čitao ili upisivao podatak u memoriju (slika 14). Tada očitavanje nove instrukcije i ubacivanje u stepen IF ne bi moglo da se realizuje i bilo bi odloženo za jednu periodu signala takta što je na slici 14 označeno sa *stall*. Sve ostale instrukcije bi se normalno izvršavale. Način za kompletno eliminisanje ovog strukturalnog hazarda je podela memorije na posebne memorije za instrukcije i podatke, što je i urađeno za razmatrani procesor sa slike 12.

Mogu se javiti strukturalni hazardi i kod pristupa drugim resursima. Tako, na primer, procesor koji se razmatra je dosta jednostavan, jer se ne razmatra aritmetika u pokretnom

zarezu, kao ni množenje i deljenje celobrojnih veličina. Zbog toga je stepen EX jednostavan i u njemu se za svaku instrukciju faza EX izvršava u jednom taktu. U slučaju pomenutih operacija treba više taktova. Da to ne bi kočilo *pipeline* stepen EX može da se realizuje kao paralelna veza više funkcionalnih jedinica (slika 15). Kada neka instrukcija dođe do svoje faze EX šalje se u odgovarajuću EX funkcionalnu jedinicu u kojoj se izvršava u onoliko taktova koliko joj je potrebno. Sada je moguće u ritmu takta, bez zadrške u *pipeline*-u, instrukcije iz stepena ID slati u različite EX funkcionalne jedinice radi izvršavanja faze EX. Čim se faza EX neke od instrukcija završi u nekoj od funkcionalnih jedinica instrukcija ide kroz stepene MEM i WB radi kompletiranja izvršavanja.

	broj perioda signala takta								
Instruction	1	2	3	4	5	6	7	8	9
load instruction	IF	ID	EX	MEM	WB				
instruction *i* + 1		IF	ID	EX	MEM	WB			
instruction *i* + 2			IF	ID	EX	MEM	WB		
instruction *i* + 3				stall	IF	ID	EX	MEM	WB
instruction *i* + 4					stall	IF	ID	EX	MEM

Slika 14 Situacija u *pipeline*-u kao posledica strukturalnog hazarda



Slika 15 Stepen EX realizovan kao paralelna veza više funkcionalnih jedinica

Ovo funkcioniše lepo sve dok za instrukcije koje dolaze postoje slobodne EX funkcionalne jedinice. Strukturalni hazard se ovde javlja kada naiđe instrukcija koju treba poslati u neku EX funkcionalnu jedinicu, koja nije raspoloživa jer prethodna instrukcija još uvek nije završila svoju fazu EX u njoj. Zbog toga se ova instrukcija zaustavlja u stepenu ID *pipeline*-a gde čeka da se odgovarajuća EX funkcionalna jedinica oslobodi. Neki drugi problemi koji se javljaju kod *pipeline*-ova sa ovako povezanim EX funkcionalnim jedinicama razmatraju se u odeljku ******.

U zavisnostio od toga koliko se često javlja neki strukturalni hazard procenjuje se da li se isplati ubacivanje nekog rešenja kojim se ovaj hazard izbegava ili smanjuje. U nekim situacijama zaustavljanje *pipeline*-a zbog hazarda može da bude prihvatljivo rešenje.

3.2 Hazardi podataka

Hazard podataka se javlja kod procesora sa *pipeline* organizacijom zbog izmenjenog redosleda pristupa podacima u odnosu na redosled pristupa podacima kod procesora bez *pipeline* organizacije. Ovo je posledica činjenice da se kod *pipeline* procesora preklapa izvršavanje različitih faza više instrukcija. Kod procesora koji nemaju *pipeline* organizaciju instrukcije se izvršavaju sekvencijalno, pa se tek po izvršavanju svih faza jedne instrukcije kreće sa izvršavanjem faze prve sledeće. Kao ilustracija hazarda podataka koji može da nastane kod *pipeline* izvršavanja instrukcija može se uzeti sledeći primer:

add R1, R2, R3
sub R4, R5, R1
and R6, R1, R7
or R8, R1, R9
xor R10, R1, R11

Sve instrukcije posle instrukcije **add** koriste rezultat instrukcije **add** koji se nalazi u registru R1. Situacija u stepenima *pipeline*-a prilikom izvršavanja pet instrukcija prikazana je na slici 16. Vidi se da instrukcija **add** upisuje podatak u *R*1 u stepenu WB, dok instrukcija **sub** čita podatak u stepenu ID. Instrukcija **add** završava upis tek posle tri periode signala takta u odnosu na trenutak kad instrukcija **sub** počinje čitanje. Ovakva situacija se naziva hazard podataka. Ukoliko se nešto ne preduzme da se ovakva situacija izbegne, instrukcija **sub** će očitati pogrešnu vrednost i koristiće je. Ova vrednost čak i ne mora da bude uvek postavljena od iste instrukcije. Nekada to može da bude neka od instrukcija pre instrukcije **add**. Ukoliko, pak, stigne prekid između instrukcija **add** i **sub**, a obrada prekida je tako realizovana da se skače na prekidnu rutinu po kompletiranju instrukcije **add**, u *R*1 će biti vrednost koju je postavila instrukcija **add**.

	1	2	3	4	5	6	7	8	9
add R1, R2, R3	IF	ID	EX	MEM	WB*				
sub R4, R5, R1		IF	ID**	EX	MEM	WB			
and R6, R1, R7			IF	ID**	EX	MEM	WB		
or R8, R1, R9				IF	ID**	EX	MEM	WB	
xor R10, R1, R11					IF	ID***	EX	MEM	WB

Legenda:
*—**add** upisuje u R1
—sub**, **and**, **or** čitaju pogrešnu vrednost iz R1
***—**xor** čita korektnu vrednost iz R1

Slika 16 Situacija u *pipeline*-u sa prisutnim hazardom podataka

Instrukcija **and** je takođe pogođena hazardom podataka. Sa slike 16 se vidi da instrukcija **add** tek na kraju perioda 5 signala takta kompletira upis u registar R1. Zbog toga instrukcija **and** koja čita registar R1 u periodi 4 signala takta dobija pogrešnu vrednost.

Slična je situacija i sa instrukcijom **or**. Ova instrukcija čita registar R1 za vreme periode 5 signala takta. S obzirom da instrukcija **add** tek na kraju periode 5 signala takta kompletira upis u registar R1, i instrukcija **or** dobija pogrešnu vrednost registra R1.

Tek instrukcija **xor** dobija korektnu vrednost registra R1, jer instrukcija **add** u toku periode 5 signala takta upisuje u registar R1, a instrukcija **xor** ga čita u periodi 6 signala takta.

Ovo je samo jedna i to dosta česta vrsta hazarda podataka. Hazarda podataka ima više i oni se mogu svrstati u tri grupe u zavisnosti od toga po kom redosledu bi trebalo čitati i upisivati od strane instrukcija u *pipeline*-u da bi se sačuvalo regularno izvršavanje programa. Posmatraće se dve instrukcije *i* i *j*, pri čemu se instrukcija *i* javlja pre instrukcije *j*. Mogući hazardi podataka su:

1) RAW (read after write)—instrukcija *j* pokušava da čita podatak pre nego što instrukcija *i* upiše, tako da instrukcija *j* dobija nekorektno staru vrednost. Ovo je najčešći tip hazarda podataka koji je do sada i jedino razmotren.
2) WAR (write after read)—instrukcija *j* pokušava da upiše pre nego što instrukcija *i* čita, pa instrukcija *i* nekorektno dobija novu vrednost. U posmatranom *pipeline*-u ovo ne može da se desi zato što se čitanje obavlja u jednom od početnih stepeni *pipeline*-a i to stepenu ID, a upis u jednom od poslednjih stepeni *pipeline*-a i to stepenu WB.
3) WAW (write after write)—instrukcija *j* pokušava da upiše neki podatak pre nego što je on najpre upisan instrukcijom *i*. Ovde se upisivanje izvršava po pogrešnom redosledu pa u operandu ostaje vrednost upisana instrukcijom *i* umesto instrukcijom *j*. Ovaj hazard postoji u *pipeline*-ovima u kojima se može upisivati iz više stepeni, što nije slučaj u posmatranom *pipeline*-u u kome se može upisivati samo iz stepena WB.

Treba zapaziti da slučaj RAR (read after read) nije hazard.

Zbog toga što se RAW hazard najčešće javlja, a jedino je prisutan hazard podataka u usvojenoj *pipeline* organizaciji procesora, dalja razmatranja će biti usmerena na tehnike kojima se obezbeđuje korektno izvršavanje instrukcija u *pipeline*-u sa prisutnim RAW hazardom podataka.

3.2.1 Korektno izvršavanje instrukcija zaustavljanjem *pipeline*-a

Najjednostavniji način da se hazard podataka prisutan u primeru sa slike 16 reši je zaustavljanje instrukcije **sub** u *pipeline*-u za tri periode signala takta, da bi se instrukcija **add** normalno izvršavala (slika 17). Time bi se instrukciji **sub** omogućilo da pređe u stepen ID u kome čita podatak iz registra R1 tek pošto instrukcija **add** završi fazu WB u kojoj upisuje podatak u registar R1. Instrukcije **and**, **or** i **xor** bi, takođe, čitale korektnu vrednost registra R1 pri prolasku kroz stepen ID *pipeline*-a.

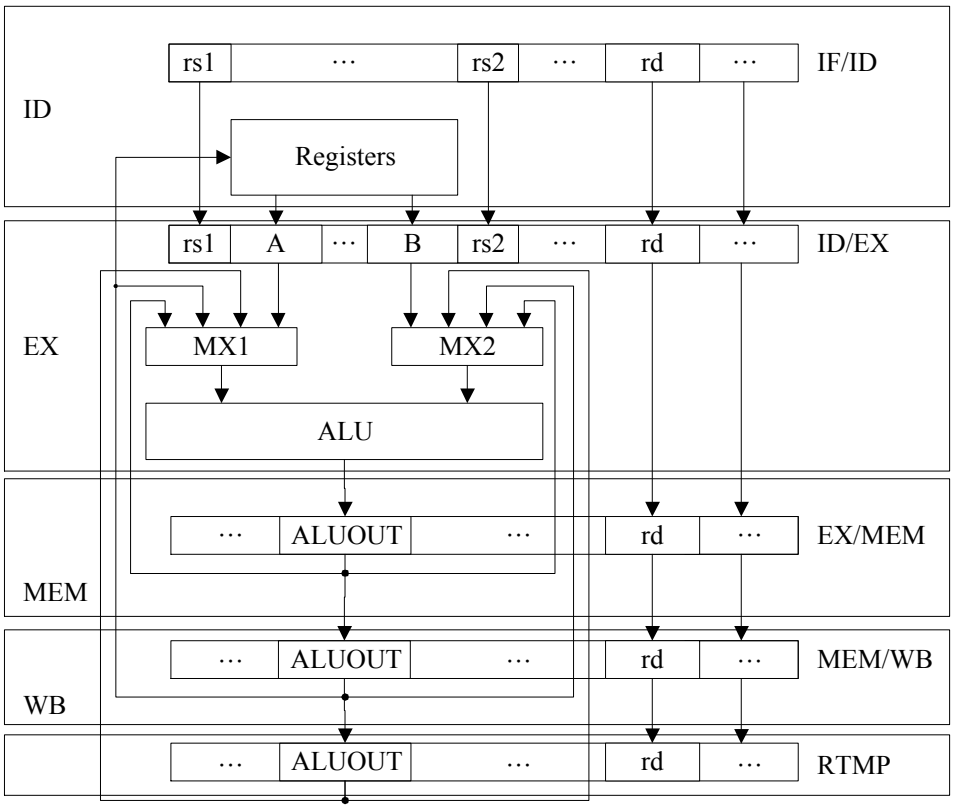
	1	2	3	4	5	6	7	8	9	10	11	12
add R1, R2, R3	IF	ID	EX	MEM	WB							
sub R4, R5, R1		IF	*stall*	*stall*	*stall*	ID	EX	MEM	WB			
and R6, R1, R7						IF	ID	EX	MEM	WB		
or R8, R1, R9							IF	ID	EX	MEM	WB	
xor R10,R1, R11								IF	ID	EX	MEM	WB

Slika 17 Situacija u *pipeline*-u kao posledica hazarda podataka

3.2.2 Korektno izvršavanje instrukcija bez zaustavljanjem *pipeline*-a prosleđivanjem

Hazard tipa RAW se može hardverski eliminisati tehnikom prosleđivanja (*forwarding*, *bypassing*, *short-circuiting*). Rezultat sa izlaza jedinice ALU stepena EX se zajedno sa još nekim informacijama iz *pipeline* registra *ID/EX* stepena EX na signal takta šalje u *pipeline* registar *EX/MEM* stepena MEM (slika 18). Izlaz registra *EX/MEM.ALUOUT* se vodi ne samo

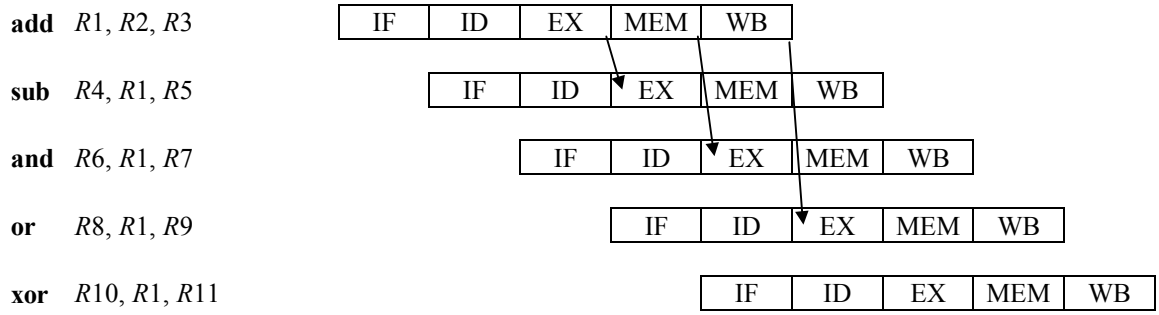
na ulaze *pipeline* registra *MEM/WB.ALUOUT* stepena WB već se preko multipleksera *MX*1 i *MX*2 vodi i na ulaze *ALU*. Ukoliko hardver za prosleđivanje otkrije da instrukcija koja je trenutno u stepenu MEM treba da svoj rezultat upiše u registar koji je izvorišni za instrukciju koja se nalazi u stepenu EX, on selektuje kao operand za *ALU* ne vrednost očitanu iz registarskog fajla Registers i koja se nalazi u registru *ID/EX.A* već vrednost prosleđenu od prethodne instrukcije preko registra *EX/MEM.ALUOUT*.



Slika 18 Hardver za prosleđivanje rezultata tri instrukcije

Međutim, situacija se usložnjava u slučajevima sličnim onim iz programa sa slike 16 u kome instrukcija **add** postavlja registar *R*1, a sledeće četiri instrukcije **sub**, **and**, **or** i **xor** ga koriste. Tada rezultat instrukcije **add** treba proslediti ne samo prvoj instrukciji iza nje (**sub**), već i drugoj (**and**) i trećoj (**or**). Tek četvrta instrukcija (**xor**) može da čita korektnu vrednost registra *R*1. Sa slike 19 se vidi situacija u *pipeline*-u tokom izvršavanja posmatranih instrukcija. Dok je instrukcija **add** u stepenu EX instrukcija **sub** koja je prva iza nje bi trebalo da u stepenu ID očita registar *R*1. Dok je instrukcija **add** u stepenu MEM instrukcija **and**, koja je druga iza nje, bi trebalo u stepenu ID da očita registar *R*1. Dok instrukcija **add** u stepenu WB upisuje u registar *R*1, instrukcija **or** koja je treća iza nje bi trebalo u stepenu ID da očita registar *R*1. Zbog toga treba preko multipleksera MX1 i MX2 (slika 18) na ulaze jedinice ALU prosleđivati i izlaze registara *EX/MEM.ALUOUT*, *MEM/WB.ALUOUT* i *RTMP.ALUOUT*. Instrukcija koja se nalazi u stepenu EX će, ako joj to treba, iz registra *EX/MEM.ALUOUT* stepena MEM učitati rezultat prve instrukcije ispred sebe, iz registra *MEM/WB.ALUOUT* stepena WB učitati rezultat druge instrukcije ispred sebe i iz registra *RTMP.ALUOUT* izlaznog registra *RTMP* rezultat treće instrukcije ispred sebe. Tek kada instrukcija **xor** dođe u stepen ID u registru *R*1 će se nalaziti rezultat instrukcije **add**, pa će ga tek ova instrukcija uzimati iz registarskog fajla Registers. Stoga instrukcija koja se nalazi u

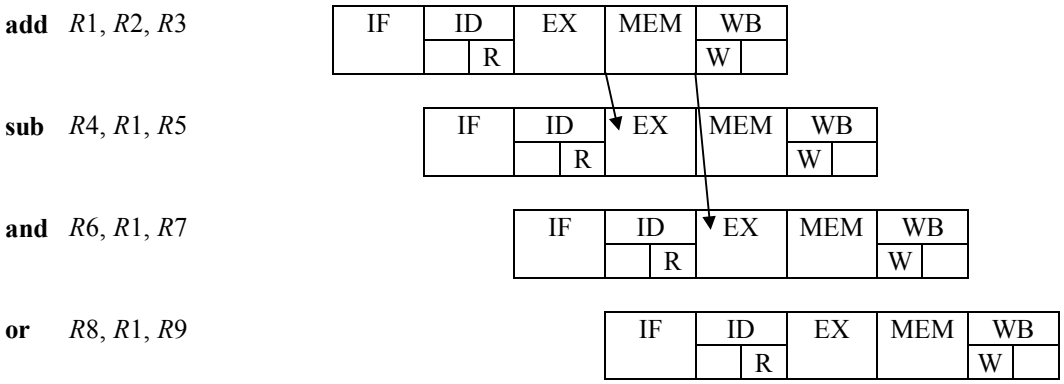
stepenu EX treba da proverava da li je neko od njenih izvorišta odredište za vrednosti iz registara *EX/MEM.ALUOUT*, *MEM/WB.ALUOUT* i *RTMP.ALUOUT* koje su nastale izvršavanjem prve, druge i treće instrukcije ispred nje, respektivno. Ako je to slučaj, onda hardver za prosleđivanje kao operand koristi *EX/MEM.ALUOUT*, *MEM/WB.ALUOUT* i *RTMP.ALUOUT*, a ne vrednost očitanu iz registarskog fajla koja se nalazi u registru *ID/EX.A*. Ovo važi za oba izvorišna operanda instrukcije koja se nalazi u stepenu EX.



Slika 19. Situacija u *pipeline*-u pri korišćenju hardvera za prosleđivanje rezultata jedinice ALU iz registara EX/MEM.ALUOUT (**sub**), MEM/WB.ALUOUT (**and**) i RTMP.ALUOUT (**or**) na ulaz jedinice ALU za tri instrukcije

Hardver za prosleđivanje pored registara *EX/MEM.ALUOUT*, *MEM/WB.ALUOUT* i *RTMP.ALUOUT*, ima i registre *EX/MEM.rd, MEM/WB.rd* i *RTMP.rd* za čuvanje adresa registara registarskog fajla Registers u koje treba upisati *EX/MEM.ALUOUT*, *MEM/WB.ALUOUT* i *RTMP.ALUOUT*, registre *ID/EX.rs1* i *ID/EX.rs2* za čuvanje adresa izvorišnih registara čiji se sadržaji koriste u instrukciji koja je u stepenu EX, tri para komparatora za upoređivanje *ID/EX.rs1* i *ID/EX.rs2* sa *EX/MEM.rd*, *MEM/WB.rd* i *RTMP.rd* i multipleksere *MX*1 i *MX*2 sa odgovarajućom upravljačkom logikom.

Radi smanjivanja složenosti hardvera procesora poželjno je smanjiti broj instrukcija koje se na ovakav način prosleđuju. Taj broj bi se mogao smanjiti sa tri na dva ukoliko bi se u toku trajanja jedne periode signala takta realizovao i upis i čitanje iz registar fajla Registers. Uzeće se da se u prvoj polovini faze WB upisuje u registarski fajl Registers, a u drugoj polovini faze ID čita iz registarskog fajla Registers. Uz ovakvu realizaciju upisivanja i čitanja registarskog fajla Registers situacija u *pipeline*-u za program sa slike 19 se može predstaviti kao na slici 20. Instrukcije **sub** i **and** još uvek zahtevaju da im se rezultat instrukcije **add**, koji će se upisati u registar *R*1, prosledi u stepen EX. Međutim, u ovom slučaju, za razliku od malopre, to nije potrebno uraditi i za instrukciju **or** koja je treća iza instrukcije **add**. Instrukcija **add**, koja se nalazi u stepenu WB, će u prvoj polovini periode signala takta upisati svoj rezultat u registar *R*1, dok će instrukcija **or**, koja se nalazi u stepenu ID, u drugoj polovini te iste periode signala takta, očitati sadrđaj registra *R*1. Za ovakvu realizaciju treba sa slike 18 izbaciti registre *RTMP.ALUOUT* i *RTMP.rd* i par komparatora. U svim daljim razmatranjima će se podrazumevati da se upis i čitanje iz registarskog fajla Registers realizuju u dve polovine signala takta i da se instrukciji u stepenu EX prosleđuju rezultati instrukcija koje se nalaze u stepenima MEM i WB.



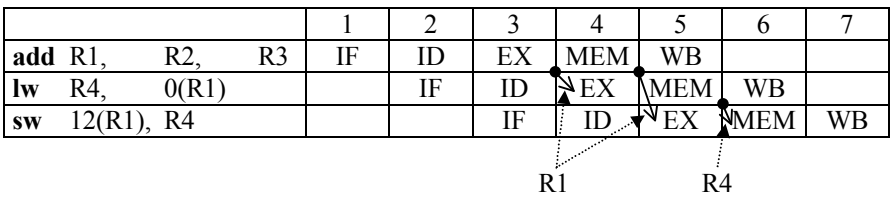
Slika 20. Situacija u *pipeline*-u pri korišćenju hardvera za prosleđivanje rezultata jedinice ALU iz registara EX/MEM.ALUOUT (**sub**) i MEM/WB.ALUOUT (**and**) na ulaz jedinice ALU za dve instrukcije

Ovo je bio samo jedan od više mogućih slučajeva prosleđivanja. Ovde je prosleđivan rezultat jedinice ALU na ulaz iste jedinice. Prosleđivanje se može generalizovati i na slučajeve kada druge jedinice procesora svoj rezultat prosleđuju na svoj ulaz kao i na slučajeve kada se prosleđuje rezultat jedne jedinice na ulaz druge jedinice.

Situacija kada jedinica Data memory treba da prosledi rezultat sa svog izlaza na svoj ulaz se može ilustrovati sledećom sekvencom instrukcija:

```
add R1,      R2,      R3
lw  R4,      0(R1)
sw  12(R1),  R4
```

Ovde bi bez prosleđivanja, a da bi se izbegao RAW hazard podataka, moglo da dođe do zaustavljanja *pipeline*-a iz dva razloga. Prvi je da je rezultat ALU jedinice iz stepena EX, koji nastaje kao rezultat izvršavanja instrukcije **add** i koji treba da se upiše u registar R1, potreban istoj toj jedinici u instrukcijama **lw** i **sw** koje slede za njom. Drugi je da je rezultat jedinice Data Memory iz stepena MEM, koji nastaje kao rezultat izvršavanja instrukcije **lw** i koji treba da se upiše u registar R4, potreban istoj toj jedinici u instrukciji **sw** koja sledi za njom radi upisa u Data memory. Na slici 21 su prikazana potrebna prosleđivanja da bi se radi eliminisanja RAW hazarda podataka izbeglo zaustavljanje *pipeline*-a. Ovde je potrebno na ulaze jedinice ALU zbog instrukcija **lw** i **sw** i registra R1 vraćati sadržaje registara EX/MEM.ALUOUT i MEM/WB.ALUOUT, respektivno, a na ulaz jedinice Data Memory zbog instrukcije **sw** i registra R4 vraćati sadržaj registra MEM/WB.LMD i koristiti umesto EX/MEM.B.



Slika 21. Situacija u *pipeline*-u pri korišćenju hardvera za prosleđivanje rezultata jedinice Data memory iz registara MEM/WB.LMD (**sw**) na ulaz jedinice Data memory

Slučaj kada treba proslediti izlaz jedne jedinice na ulaz druge jedinice da bi se radi eliminisanja RAW hazarda podataka izbeglo zaustavljanje *pipeline*-a može se ilustrovati sledećom sekvencom instrukcija:

add R1, R2, R3
sw 0(R4), R1

Na slici 22 su prikazana potrebna prosleđivanja da bi se radi eliminisanja RAW hazarda podataka izbeglo zaustavljanje *pipeline*-a. U ovom slučaju je instrukciji **sw** za jedinicu Data memory u stepenu MEM potreban rezultat instrukcije **add** iz jedinice ALU stepena EX. Zbog toga na ulaze jedinice Data Memory stepena MEM treba voditi sadržaj registra MEM/WB.ALUOUT i koristiti za upis umesto sadržaja registra EX/MEM.B na adresi određenoj sadržajem registra EX/MEM.ALUOUT.

	1	2	3	4	5	6
add R1, R2, R3	IF	ID	EX	MEM	WB	
sw 0(R4), R1		IF	ID	EX	MEM	WB

Slika 22. Situacija u *pipeline*-u pri korišćenju hardvera za prosleđivanje rezultata jedinice ALU iz registra MEM/WB.ALUOUT (**sw**) na ulaze jedinice Data memory

Da bi se u razmatranom procesoru *pipeline* organizacije pri prisutnom RAW hazardu podataka izbeglo zaustavljanje *pipeline*-a a sam hazard eliminisao, treba obezbediti prosleđivanje ne samo na ulaze jedinica ALU i Data Memory, već i na ulaz jedinice Zero?, koja se nalazi u stepenu EX. Na njen ulaz treba prosleđivati izlaze jedinica ALU i Data Memory. U slučaju **branch** instrukcija u jedinici Zero? se vrši provera sadržaja registra ID/EX.A ili na vrednost nula ili na vrednost različito od nule i postavlja jednorazredni registar EX/ME.cond. Slučajevi kada treba proslediti rezultat jedinice ALU na ulaz jedinice Zero? dati su na slikama 23 i 24. U slučaju primera sa ovih slika na ulaz jedinice Zero? treba proslediti sadržaje registara EX/MEM.ALUOUT i MEM/WB.ALUOUT, respektivno, i koristiti umesto sadržaja registra ID/EX.A.

	1	2	3	4	5	6
add R1, R2, R3	IF	ID	EX	MEM	WB	
beqz R1, 50		IF	ID	EX	MEM	WB

Slika 23. Situacija u *pipeline*-u pri korišćenju hardvera za prosleđivanje rezultata jedinice ALU iz registra EX/MEM.ALUOUT (**beqz**) na ulaz jedinice Zero?

	1	2	3	4	5	6	7
add R1, R2, R3	IF	ID	EX	MEM	WB		
sub R4, R5, R6		IF	ID	EX	MEM	WB	
bneq R1, 50			IF	ID	EX	MEM	WB

Slika 24. Situacija u *pipeline*-u pri korišćenju hardvera za prosleđivanje rezultata jedinice ALU iz registra MEM/WB.ALUOUT (**bneq**) na ulaz jedinice Zero?

Slučaj kada, radi eliminisanja RAW hazarda bez zaustavljanja *pipeline*-a, treba proslediti rezultat jedinice Data Memory iz stepena MEM na ulaz jedinice Zero? stepena EX, dat je na slici **25**. U ovom slučaju na ulaz jedinice Zero? treba proslediti sadržaj registra MEM/WB.LMD i koristiti umesto sadržaja registra ID/EX.A.

	1	2	3	4	5	6	7
lw R1, 0(R2)	IF	ID	EX	MEM	WB		
sub R3, R4, R5		IF	ID	EX	MEM	WB	
beqz R1, 50			IF	ID	EX	MEM	WB

Slika 25. Situacija u *pipeline*-u pri korišćenju hardvera za prosleđivanje rezultata jedinice Data memory iz registra MEM/WB.LMD (**beqz**) na ulaz jedinice Zero?

Hazard podataka se javlja kada nekom podatku pristupaju dve instrukcije koje su dovoljno blizu da se preklapanjem njihovog izvršavanja zbog *pipeline*-a menja redosled pristupa tom podatku od strane te dve instrukcije. Do sada razmotreni hazardi podataka se odnose na pristup registrima. Međutim, hazardi podataka postoje i kod pristupa memorijskim lokacijama. U razmatranom procesoru *pipeline* organizacije oni ne mogu da se jave jer *pipeline* organizacija ne menja redosled pristupa memorijskim lokacijama u odnosu na onaj koji bi bio za slučaj procesora sa sekvencijalnim izvršavanjem instrukcija.

Napomena: Pogledati sledeće sekvence:

add *R*1, *R*2, *R*3
sub *R*5, *R*6, *R*7
sw 0(*R*8), *R*1

i

lw *R*1, 0(*R*2)
sw 0(*R*5), *R*6
sw 0(*R*8), *R*1

3.2.3 Zaustavljanje *pipeline*-a kao jedini način obezbeđivanja korektnog izvršavanja instrukcija (*pipeline stall*, *pipeline interlocks*)

Postoje neke situacije RAW hazarda podataka kada je jedini način da se obezbedi korektno izvršavanje instrukcija u *pipeline*-u zaustavljanje *pipeline*-a. Posmatra se sledeća sekvenca instrukcija:

lw *R*1, 32(*R*6)
add *R*4, *R*1, *R*7
sub *R*5, *R*1, *R*8
and *R*6, *R*1, *R*7

Vrednost koja se čita iz memorije instrukcijom **lw** i upisuje u registar *R*1 koristi se u sve tri instrukcije iza instrukcije **lw**. Situacija u stepenima *pipeline*-a za slučaj ovog programa bez zaustavljanja *pipeline*-a prikazana je na slici 26.

lw *R*1, 32(*R*6)	IF	ID	EX	MEM	WB
add *R*4, *R*1, *R*7		IF	ID	EX	MEM
sub *R*5, *R*1, *R*8			IF	ID	EX
and *R*6, *R*1, *R*7				IF	ID

Slika 26 Situacija u *pipeline*-u sa prisutnim hazardom podataka pri čitanju iz memorije

Instrukcija **lw** ima podatak u registru *MEM/WB.LMD* tek na kraju svoje faze MEM. Ova faza se izvršava u istoj periodi signala takta kao i faza EX instrukcije **add**. Međutim, instrukciji **add** je podatak od instrukcije **lw** potreban na početku faze EX. Vidi se da ne postoji način da on stigne za instrukciju **add** makar se koristila tehnika prosleđivanja sadržaja registra *MEM/WB.LMD* na ulaze jedinice *ALU*. Tek za instrukciju **sub** podatak od instrukcije **lw** se može iz registra *MEM/WB.LMD* proslediti na ulaz jedinice *ALU*. Instrukcija **and** ga dobija preko registarskog fajla Registers, jer ga instrukcija **lw** upisuje u registar *R*1 u prvoj polovini faze WB, a instrukcija **and** čita u drugoj polovini faze ID.

Jedini način da se obezbedi korektno izvršavanje programa je zaustavljanje *pipeline*-a (*pipeline* interlock, *pipeline* stall). Izgled *pipeline*-a pri zaustavljanju *pipeline*-a je dat na slici 27. Sve instrukcije počev od instrukcije **add** su zakašnjene za jednu periodu signala takta. Sada se rezultat instrukcije **lw** iz registra *MEM/WB.LMD* stepena WB direktno prosleđuje instrukciji **add** na ulazu jedinice ALU u stepenu EX. Instrukcija **lw** u prvoj polovini stepena WB upisuje u *R*1, a instrukcija **sub** čita u drugoj polovini stepena ID.

bilo koja instrukcija	IF	ID	EX	MEM	WB					
lw *R*1, 32(*R*6)		IF	ID	EX	MEM	WB				
add *R*4, *R*1, *R*7			IF	ID	stall	EX	MEM	WB		
sub *R*5, *R*1, *R*8				IF	stall	ID	EX	MEM	WB	
and *R*6, *R*1, *R*7					stall	IF	ID	EX	MEM	WB

Slika 27. Zaustavljanje *pipelinei-a* radi izbegavanja hazarda podataka pri čitanju vrednosti iz memorije koja je potrebna jedinici ALU

Slična situacija se javlja i u sekvenci instrukcija sa slike 28. Na ulaze jedinice Zero? treba proslediti sadržaj registra MEM/WB.LMD iz stepena WB, ali tek pošto se instrukcija **beqz** zaustavi u *pipeline*-u za jednu periodu signala takta i time sačeka da se operacija čitanja iz jedinice Data Memory u stepenu MEM kompletira.

	1	2	3	4	5	6	7
lw R1, 0(R2)	IF	ID	EX	MEM	WB		
beqz R1, 50		IF	ID	*stall*	EX	MEM	WB

Slika 28. Zaustavljanje *pipeline*-a radi izbegavanja hazarda podataka pri čitanju vrednosti iz memorije koja je potrebna jedinici Zero?

3.2.4 Zakašnjeno punjenje (delayed load)

Ima više situacija kada je jedini način da se eliminiše RAW hazard podataka zaustavljanje *pipeline*-a. Jedan od tipičnih primera za to je izvršavanje instrukcija generisanih na uobičajeni način za izraz $A = B + C$. Ovde se zahteva zaustavljanje *pipeline*-a zbog druge instrukcije **lw**, koja sadržaj *C* puni u registar *R*2 (slika 29). Iako je obezbeđeno prosleđivanje rezultata izvršavanja druge instrukcije **lw** iz registra MEM/WB.LMD stepena WB na ulaz jedinice ALU stepena EX, instrukcije **add** i **sw** se moraju zaustaviti za jednu periodu signala takta u *pipeline*-u. Instrukcija **sw** ne treba da se zaustavlja, jer se vrednost koju instrukcija **add** treba da upiše u registar *R*3, a koja je potrebna za instrukciju **sw**, prosleđuje iz registra MEM/WB.ALUOUT stepena WB na ulaz jedinice Data memory stepena MEM gde se koristi za upis u jedinicu Data Memory umesto sadržaja registra EX/MEM.B.

lw	R1, *b*	IF	ID	EX	MEM	WB				
lw	R2, *c*		IF	ID	EX	MEM	WB			
add	R3, R1, R2			IF	ID	stall	EX	MEM	WB	
sw	*a*, R3				IF	stall	ID	EX	MEM	WB

Slika 29 Tipičan izgled *pipeline*-a pri izračunavanju izraza $A = B + C$

Jedan od načina da se ova vrsta hazarda podataka izbegne bez zaustavljanja *pipeline*-a je da prevodilac preuredi generisane instrukcije na takav način da se izbegne da iza instrukcije **lw** bude odmah instrukcija koja kao izvorište ima registar koji je odredište za instrukciju **lw**. Jedna ovako generisana sekvenca instrukcija za sračunavanje izraza

$A = B + C;$
$D = E + F;$

je:

lw R_b, b
lw R_c, c
lw R_e, e
add R_a, R_b, R_c } ove dve instrukcije su zamenile mesta
lw R_f, f
sw a, R_a } ove dve instrukcije su zamenile mesta
add R_d, R_e, R_f
sw d, R_d

Dve moguće situacije zaustavljanja *pipeline*-a zbog instrukcije **lw** i to:

lw R_c, c i **add** R_a, R_b, R_c i
lw R_f, f i **add** R_d, R_e, R_f

su izbegnute. U prvom slučaju je između instrukcija **lw** R_c, c i **add** R_a, R_b, R_c ubačena instrukcija **lw** R_e, e iz drugog izraza, a u drugom slučaju je između instrukcija **lw** R_f, f i **add** R_d, R_e, R_f ubačena instrukcija **sw** a, R_a iz prvog izraza. Pretpostavlja se da se rezultati stepena MEM za instrukcije **lw** R_c, c i **lw** R_f, f prosleđuju na ulaze stepena EX za instrukcije **add** R_a, R_b, R_c i **add** R_d, R_e, R_f, respektivno, kao i da se rezultat stepena EX za instrukciju **add** R_d, R_e, R_f prosleđuje u stepen MEM za instrukciju **sw** d, R_d i da zbog toga nema zaustavljanja *pipeline*-a.

Mehanizam kojim se, softverskim putem izbegavanjem stavljanja iza instrukcije **lw** koja koristi kao izvorišni registar odredišni registar instrukcije **lw**, sprečava zaustavljanje *pipeline*-a pri izvršavanju instrukcije **lw** se naziva zakašnjeno punjenje (*delayed load*). Ova tehnika je sasvim dobra, tako da neki procesori umesto hardverski softverski se štite od ovog hazarda podataka. Ako ne može da se nađe ni jedna druga instrukcija, onda se kod takvih procesora radi sprečavanja ovog hazarda podataka stavlja instrukcija **nop**.

3.3 Upravljački hazardi

Upravljački hazardi predstavljaju situacije koje se javljaju u *pipeline* procesorima prilikom izvršavanja instrukcija uslovnog skoka, kada treba zaustaviti *pipeline* određen broj signala takta dok se ne odluči da li će biti skok ili ne i time se dobije vrednost *PC*-ja sa koje treba očitati sledeću instrukciju. Broj perioda signala takta koji se mora sačekati da bi se utvrdila nova vrednost *PC*-ja se naziva **branch delay**.

3.3.1 Korektno izvršavanje instrukcija zaustavljanjem *pipeline*-a

U slučaju usvojenog procesora *pipeline* organizacije tek kada **branch** instrukcija stigne u stepen MEM se zna da li će biti skoka ili ne i koju vrednost signalom takta na kraju faze MEM treba upisati u registar PC. To je ili sračunata adresa instrukcije na koju se skače, koja se nalazi u registru EX/MEM.ALUOUT, ili adresa prve sledeće instrukcije iza instrukcije skoka, koja se nalazi u registru EX/MEM.NPC. Selekcija jedne od ove dve vrednosti se vrši signalom EX/MEM.cond (slike 12 i 13). U opštem slučaju za svaku **branch** instrukciju treba sačekati njeno kompletiranje faze MEM, pa tek onda krenuti sa ubacivanjem novih instrukcija u stepen *pipeline*-a. To znači zaustavljanje *pipeline*-a za tri takta. Zbog toga kada **branch** instrukcija stigne u stepen MEM, u stepenima EX, ID i IF nema instrukcija. Signalom takta na kraju faze MEM **branch** instrukcije u registar PC se upisuje korektna adresa instrukcije sa kojom treba produžiti izvršavanje instrukcija posle **branch** instrukcije, pa se tek tada u stepen IF *pipeline* ubacuje nova instrukcija. Slika 30 prikazuje zaustavljanje *pipeline*-a za tri signala takta zbog upravljačkog hazarda.

U zavisnosti od toga da li se kao rezultat izvršavanja **branch** instrukcije u registar PC upisuje adresa instrukcije na koju se skače ili adresa prve sledeće instrukcije posle **branch** instrukcije, kaže se da je ili **branch taken** (skok napravljen) ili **branch not taken** (skok nije napravljen), respektivno.

Međutim ono što se stvarno dešava u *pipeline*-u izgleda nešto malo drugačije i dato je na slikama 31 i 32 za slučajeve kada je **branch taken** i **branch not taken**, respektivno. Za posmatrani procesor *pipeline* organizacije instrukcija iza **branch** instrukcije se očitava jer se za *i*-tu instrukciju otkriva da je **branch** tek kada dođe u stepen ID. Tada se očitana instrukcija zaustavlja dok se u stepenu MEM za instrukciju **branch** ne odluči da li je **branch taken** ili **branch not taken**. Ako je **branch taken**, očitana instrukcija u stepenu IF se ignoriše, jer je očitana sa pogrešne adrese, pa se ponovo očitava instrukcija sa adrese skoka (slika 31). Ako je **branch not taken** nema potrebe ponovo očitavati instrukciju jer je dobra instrukcija u stepenu IF, pa ona može da ide u stepen ID (slika 32). Kao rezultat toga u usvojenom procesoru *pipeline* organizacije kod **branch** instrukcija se gube ili tri takta, ako je **branch taken**, ili dva takta, ako je **branch not taken**.

instr. *i* (**branch**)	IF	ID	EX	MEM	WB					
instr. *i* + 1 ili instr. sa adr. skoka		stall	stall	stall	IF	ID	EX	MEM	WB	
instr. *i* + 2 ili instr. sa adr. skoka + 1			stall	stall	stall	IF	ID	EX	MEM	WB
instr. *i* + 3 ili instr. sa adr. skoka + 2				stall	stall	stall	IF	ID	EX	MEM
instr. *i* + 4 ili instr. sa adr. skoka + 3					stall	stall	stall	IF	ID	EX
instr. *i* + 5 ili instr. sa adr. skoka + 4						stall	stall	stall	IF	ID
instr. *i* + 6 ili instr. sa adr. skoka + 5							stall	stall	stall	IF

Slika 30 Situacija u *pipeline*-u kao posledica upravljačkog hazarda

instr. *i* (**branch**)	IF	ID	EX	MEM	WB					
instr. sa adr. skoka		IF	stall	stall	IF	ID	EX	MEM	WB	
instr. sa adr. skoka + 1			stall	stall	stall	IF	ID	EX	MEM	WB
instr. sa adr. skoka + 2				stall	stall	stall	IF	ID	EX	MEM
instr. sa adr. skoka + 3					stall	stall	stall	IF	ID	EX
instr. sa adr. skoka + 4						stall	stall	stall	IF	ID
instr. sa adr. skoka + 5							stall	stall	stall	IF

Slika 31 Situacija u *pipeline*-u kao posledica upravljačkog hazarda za slučaj da je **branch taken**

instr. *i* (**branch**)	IF	ID	EX	MEM	WB					
instr. *i* + 1		IF	stall	stall	ID	EX	MEM	WB		
instr. *i* + 2			stall	stall	IF	ID	EX	MEM	WB	
instr. *i* + 3				stall	stall	IF	ID	EX	MEM	WB
instr. *i* + 4					stall	stall	IF	ID	EX	MEM
instr. *i* + 5						stall	stall	IF	ID	EX
instr. *i* + 6							stall	stall	IF	ID

Slika 32 Situacija u *pipeline*-u kao posledica upravljačkog hazarda za slučaj da je **branch not taken**

3.3.2 Izbor stepena u *pipeline*-u u kome se realizuje skok

Gubitak tri ili dva takta za svaki **branch taken** značajno utiče na usporavanje *pipeline*-a. Broj izgubljenih taktova u ovakvim situacijama se može smanjiti ako bi se utvrđivanje da li je za **branch** instrukciju **branch taken** ili **branch not taken** realizovalo u nekom od ranijih stepeni *pipeline*-a i ukoliko bi se sračunavanje adrese instrukcije na koju se skače, takođe, realizovalo u nekom od ranijih stepeni *pipeline*-a. U opštem slučaju poboljšanja su veća ukoliko je to urađeno što je moguće ranije u *pipeline*-u.

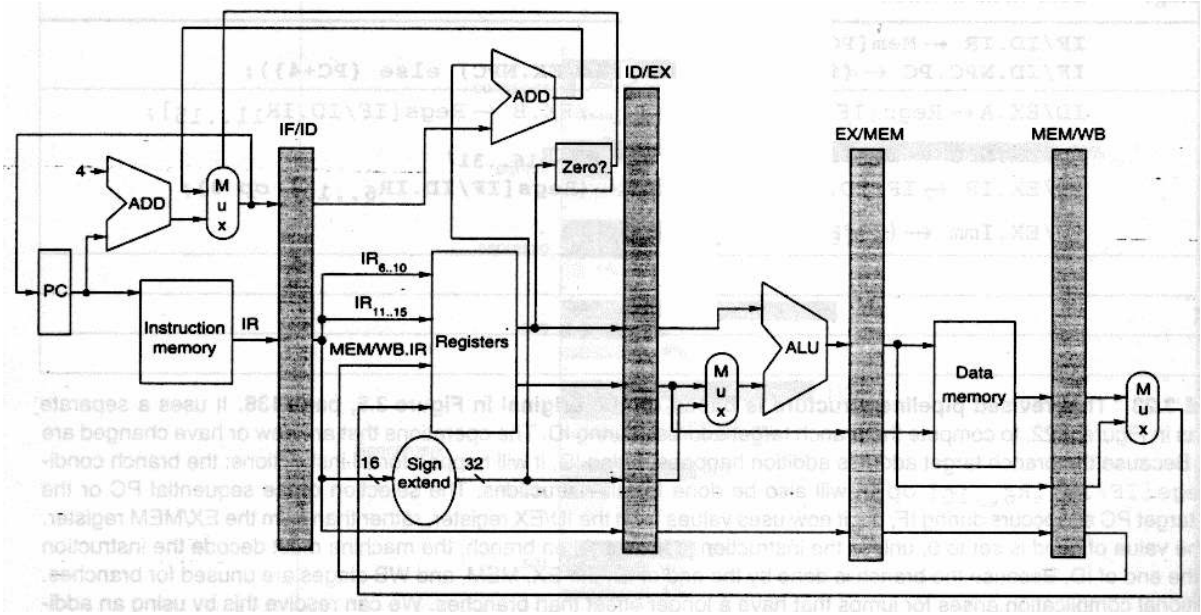
Utvrđivanje da li je za **branch** instrukciju **branch taken** ili **branch not taken** se realizuje na osnovu provere u jedinici Zero? stepena EX da li je sadržaj specificiranog registra nula ili ne. To može najranije da se realizuje u stepenu ID u kome se prvi put u registru IF/ID.IR pojavljuje očitana instrukcija i utvrđuje se da je reč o **branch** instrukciji. Pored toga u stepenu ID se iz registarskog fajla Registers čita registar koji koristi jedinica Zero?. Stoga bi ušteda jednog takta kod **branch** instrukcija bila postignuta ukoliko bi se jedinica Zero? prebacila iz stepena EX u stepen ID. Zajedno sa ovim u stepenu ID treba utvrditi adresu skoka. Ona se može utvrditi u stepenu ID u kome je pomeraj potreban za sračunavanje adrese skoka raspoloživ na izlazu jedinice Sign extend. Sračunavanje adrese skoka zahteva dodatni sabirač u stepenu ID, zbog toga što se jedinica ALU, koja je između ostalog bila korišćena za sračunavanje adrese skoka, nalazi u stepenu EX. Zbog ovoga *pipeline* organizaciju procesora sa slike 12 treba modifikovati na opisani način što dovodi do *pipeline* organizacije procesora kao što je dato na slici 33.

U *pipeline* organizaciji procesora sa slike 33 je u odnosu na *pipeline* organizaciju procesora sa slike 12 urađena još jedna stvar koja donosi uštedu za još jedan takt pri skokovima. U procesoru sa slike 12 izlazi jedinica Zero? i ALU se, najpre, na takt upisuju u *pipeline* registar EX/MEM, čime se **branch** instrukcija prebacuje u stepen MEM, pa se tek na sledeći takt vrši upis odgovarajuće vrednosti iz stepena MEM u registar PC. Da je to isto urađeno i za procesor sa slike 33, izlazi jedinica Zero? i Add iz stepena ID bi se, najpre, na takt upisivali u *pipeline* registar ID/EX, čime bi se **branch** instrukcija prebacila u stepen EX, pa bi se tek na sledeći takt izvršio upis odgovarajuće vrednosti iz stepena EX u registar PC. Time bi se postigla

ušteda jednog takta. Umesto toga u procesoru sa slike 33 izlazi jedinica Zero? i Add iz stepena ID u kome se nalazi **branch** instrukcija se vode direktno u stepen IF, pa se odmah na prvi takt vrši upis odgovarajuće vrednosti u registar PC. Time se postiže ušteda još jednog takta. Saglasno tome sliku 13 sa operacijama koje se realizuju za pojedine stepene *pipeline*-a treba modifikovati u delovima koji se odnose na instrukcije skoka, što dovodi do slike 34.

Treba zapaziti da će jedinica Zero? u svim instrukcijama, sem u **branch** instrukciji, na svom izlazu davati vrednost nula. Time se na takt u registar PC upiše vrednost registra PC uvećana za 4, čime se obezbeđuje sekvencijalno očitavanje instrukcija. Isto važi i za **branch** instrukciju ako je na izlazu jedinice Zero? nula. Treba napomenuti da se tada, s obzirom da je to slučaj **branch not taken**, na isti signal takta iz stepena IF u stepen ID prebacuje korektno očitana instrukcija i da nema zaustavljanja *pipeline*-a. Ukoliko za **branch** instrukciju u stepenu ID jedinica Zero? da na izlazu vrednost jedan, na takt u registar PC se upisuje adresa instrukcije na koju se skače. S obzirom da je to slučaj **branch taken**, u stepenu ID se nalazi očitana nekorektna instrukcija, pa se na isti takt u stepen ID ubacuje instrukcija bez dejstva. Efekat ovoga je zaustavljanje *pipeline*-a za jedan takt.

U modifikovanoj *pipeline* organizaciji procesora sa slike 33, skok se realizuje u fazi ID, tako da **branch** instrukcija ne koristi stepene EX, MEM i WB. Za dati procesor pri **branch** instrukcijama gubi se jedan takt ako je **branch taken**, a nema gubitka taktova ako je **branch not taken**.



Slika 33. Modifikovana *pipeline* organizacija procesora

Rešenje da se skok realizuje u stepenu ID *pipeline*-a na način prikazan na slici 33 umesto u stepenu MEM na način prikazan na slici 12 ima i dobre i loše strane. Dobre strane ovog rešenja su prikazane u prethodnom tekstu i svode se na redukovanje broja taktova za koliko treba pri instrukciji skoka zaustaviti *pipeline* sa tri ili dva takta na jedan ili nula taktova.

Ovim se redukuju upravljački hazardi i smanjuje broj taktova za koje zbog njih treba zastaviti *pipeline* ali se javljaju novi hazardi podataka. Ima situacija kada je jedino rešenje za eliminaciju novonastalih hazarda podataka zaustavljanje *pipeline*-a. Dve situacije koje mogu da izazovu hazarde podataka u zbog toga zaustavljanje *pipeline*-a date su u daljem tekstu.

Prva situacija se javlja pri izvršavanju sledeće sekvence instrukcija:

add R1, R2, R3
beqz R1, 50

U slučaju *pipeline* organizacije procesora sa slike 12 situacija u *pipeline*-u je prikazana na slici 23. Rezultat instrukcije **add** koji treba da se upiše u registar R1 je potreban instrukciji **beqz**. Prosleđivanjem rezultata jedinice ALU iz stepena MEM na ulaz jedinice Zero? izbegnuto je zaustavljanje *pipeline*-a.

U slučaju modifikovane *pipeline* organizacije procesora sa slike 33, situacija je prikazana na slici 35. Instrukcija **beqz** koja se nalazi u stepenu ID *pipeline*-a ne može da se kompletira u taktu 3, već mora da se zaustavi *pipeline* jedan takt. Ova instrukcija se tek u taktu 4 prosleđivanjem rezultata jedinice ALU iz registra EX/MEM.ALUOUT na ulaz jedinice Zero? može kompletirati.

stepen	**instrukcija**
IF	IF/ID.IR ← Mem[PC]; IF/ID.NPC, PC ← (if (((IF/ID.IR$_{0...5}$ eql beql) and (Regs(IF/ID.IR$_{6...10}$ eql 0)) or ((IF/ID.IR$_{0...5}$ eql bneq) and (Regs(IF/ID.IR$_{6...10}$ neq 0))) then {NPC + ((IF/ID.IR$_{16}$)^{16} ## IF/ID.IR$_{16...31}$)} else {PC + 4});
ID	ID/EX.A ← Regs[IF/ID.IR$_{6...10}$]; ID/EX.B ← Regs[IF/ID.IR$_{11...15}$]; ID/EX.IR ← IF/ID.IR; ID/EX.Imm ← (IF/ID.IR$_{16}$)^{16} ## IF/ID.IR$_{16...31}$;
EX	**ALU instrukcije** EX/MEM.IR ← ID/EX.IR; EX/MEM.ALUOUT ← ID/EX.A func ID/EX.B; ili EX/MEM.ALUOUT ← ID/EX.A func ID/EX.Imm;
	load/store instrukcije EX/MEM.IR ← ID/EX.IR; EX/MEM.ALUOUT ← ID/EX.A + ID/EX.Imm; EX/MEM.B ← ID/EX.B;
MEM	**ALU instrukcije** MEM/WB.IR ← EX/MEM.IR; MEM/WB.ALUOUT ← EX/MEM.ALUOUT;
	load/store instrukcije MEM/WB.IR ← EX/MEM.IR; MEM/WB.LMD ← Mem[EX/MEM.ALUOUT] ili Mem[EX/MEM.ALUOUT] ← EX/MEM.B;
WB	**ALU instrukcije** Regs[MEM/WB.IR$_{16...20}$] ← MEM/WB.ALUOUT; ili Regs[MEM/WB.IR$_{11...15}$] ← MEM/WB.ALUOUT;
	store instrukcija Regs[MEM/WB.IR$_{11...15}$] ← MEM/WB.LMD;

Slika 34. Operacije koje se izvršavaju u stepenima *pipeline*-a modifikovanog procesora

	1	2	3	4	5	6
add R1, R2, R3	IF	ID	EX	MEM	WB	
beqz R1, 50		IF	ID	*stall*		

Slika 35. Situacija u *pipeline*-u pri korišćenju hardvera za prosleđivanje rezultata jedinice ALU na ulaz jedinice Zero?

Druga još gora situacija se javlja pri izvršavanju sledeće sekvence instrukcija:

```
lw      R1,   0(R2)
beqz    R1,   50
```

U slučaju *pipeline* organizacije procesora sa slike 12 situacija u *pipeline*-u je prikazana na slici 28. Rezultat izvršavanja instrukcije **lw** koji treba da se upiše u registar R1 je potreban instrukciji **beqz**. Uz prosleđivanje rezultata jedinice Data memory iz stepena WB na ulaz jedinice Zero? instrukcija **beqz** je zaustavljena u *pipeline*-u samo jedan takt.

U slučaju modifikovane *pipeline* organizacije procesora sa slike 33, situacija u *pipeline*-u je prikazana na slici 36. Instrukcija **beqz** koja se nalazi u stepenu ID *pipeline*-a ne može da se kompletira ne samo u taktu 3, već ni u taktu 4. Ova instrukcija se tek u taktu 5 prosleđivanjem rezultata jedinice Data memory iz registra MEM/WB.LMD na ulaz jedinice Zero može kompletirati.

	1	2	3	4	5	6
lw R1, 0(R2)	IF	ID	EX	MEM	WB	
beqz R1, 50		IF	ID	*stall*	*stall*	

Slika 36. Situacija u *pipeline*-u pri korišćenju hardvera za prosleđivanje rezultata jedinice Data memory na ulaz jedinice Zero?

Iz ovoga se vidi da ukoliko se odluka o skoku donosi u nekom od kasnijih stepeni *pipeline*-a povećavaju se negativni efekti upravljačkog hazarda. Ukoliko se odluka o skoku donosi u nekom od ranijih stepeni *pipeline*-a javljaju se negativni efekti hazarda podataka. Pri projektovanju treba naći kompromis između ovih kontradiktornih zahteva.

3.3.3 Smanjivanje posledica zbog skokova u *pipeline*-u

Ima više metoda za smanjivanje posledica koje mogu da nastanu u *pipeline*-u zbog skokova. Tim metodama se pravi predviđanje kako će se program ponašati kod instrukcija skoka. Na osnovu toga se kao prva instrukcija posle instrukcije skoka očitava ili instrukcija sa adrese skoka ili prva sledeća instrukcija. Metode predviđanja se svrstavaju u statičke i dinamičke. U slučaju statičkih metoda predviđanje za svaki skok je uvek isto za vreme kompletnog izvršavanja programa. U slučaju dinamičkih metoda predviđanje se menja u toku izvršavanja programa, na osnovu prethodnog ponašanja programa.

3.3.3.1 Statičko predviđanje

U ovom poglavlju se razmatraju četiri statičke metode ponašanja pri skokovima u *pipeline*-u: zaustavljanje, predviđanje nema skoka (not taken), predviđanje ima skoka (taken) i zakašnjen skok (delayed branch). Prve tri metode su hardverske, dok je četvrta metoda softverska.

3.3.3.1.1 Zaustavljanje

Najjednostavnija metoda ponašanja pri skoku je zaustavljanje *pipeline*-a čime se očitavanje prve sledeće instrukcije iza instrukcije skoka zaustavlja onoliko perioda signala takta koliko je potrebno da se utvrdi sledeća vrednost *PC*-ja. Ova metoda je primamljiva jer je veoma jednostavna za realizaciju. Situacija u *pipeline*-u za procesor sa slike 12 u slučaju korišćenja ove metode data je na slikama 30, 31 i 32.

3.3.3.1.2 Predviđanje nema skoka (not taken)

Moguća metoda ponašanja pri skoku je da se uvek predvidi da neće biti skoka. Kod ove metode hardver produžava sa očitavanjem i izvršavanjem instrukcija iza instrukcije skoka kao da skoka neće biti, pri čemu se mora voditi računa o tome da instrukcije iza instrukcije skoka ne smeju da menjaju stanje u procesoru sve dok ishod instrukcije skoka nije poznat. U slučaju da nema skoka produžava se sa izvršavanjem očitanih instrukcija, jer se u *pipeline*-u nalaze korektne instrukcije. U slučaju da ima skoka treba zaustaviti *pipeline* i isprati ga (flush) od pogrešnih instrukcija. To se postiže restartovanjem *pipeline*-a od koraka IF instrukcije na koju se skočilo. U slučaju modifikovanog procesora *pipeline* organizacije (slika 33), stanje u *pipeline*-u za tehniku predviđanja nema skoka u slučaju kada nije bilo skoka dato je na slici 37, a za slučaj kada je bilo skoka na slici 38.

Odluka o skoku se za modifikovani procesor *pipeline* organizacije donosi u stepenu ID. Ako nema skoka u stepenu IF je korektna instrukcija pa se izvršavanje produžava (slika 37). Ako ima skoka u stepenu IF je pogrešna instrukcija, pa se korak IF ponavlja, ali sada za instrukciju na koju se skočilo (slika 38). Da je odlučivanje o skoku ostalo u stepenu MEM, onda bi se u slučaju kada nema skoka ubacivala dva a u slučaju kada ima skoka tri idle perioda takta).

	1	2	3	4	5	6	7	8	9
instr. *i* (**branch not taken**)	IF	ID	EX	MEM	WB				
instr. *i* + 1		IF	ID	EX	MEM	WB			
instr. *i* + 2			IF	ID	EX	MEM	WB		
instr. *i* + 3				IF	ID	EX	MEM	WB	
instr. *i* + 4					IF	ID	EX	MEM	WB

Slika 37 Stanje u *pipeline*-u za tehniku predviđanja nema skoka ako nije bilo skoka

	1	2	3	4	5	6	7	8	9
instr. *i* (**branch taken**)	IF	ID	EX	MEM	WB				
instr. *i* + 1		IF	*idle*	*idle*	*idle*	*idle*			
instr. sa adr. skoka			IF	ID	EX	MEM	WB		
instr. sa adr. skoka + 1				IF	ID	EX	MEM	WB	
instr. sa adr. skoka + 2					IF	ID	EX	MEM	WB

Slika 38 Stanje u *pipeline*-u za tehniku predviđanja nema skoka ako je bilo skoka

3.3.3.1.3 Predviđanje ima skoka (taken)

Moguća metoda ponašanja pri skoku je da se uvek predvidi da će biti skoka. Kod ove metode čim se dekoduje instrukcija skoka i sračuna adresa skoka, predviđa se da će biti skok, pa se počinje sa očitavanjem i izvršavanjem instrukcija od instrukcije na koju se predviđa da će se skočiti. Kada se utvrdi da li je zaista skok napravljen ili ne izvršavanje se produžava dvojako. Ako je skok napravljen u *pipeline*-u su korektne instrukcije, pa treba produžiti sa očitavanjem i izvršavanjem instrukcija. Ako skok nije napravljen, treba očistiti *pipeline* od instrukcija iza instrukcije skoka, jer su pogrešne, i krenuti sa korakom IF prve instrukcije iza instrukcije skoka.

U slučaju modifikovanog procesora *pipeline* organizacije (slika 33) ova metoda nema smisla, jer se u stepenu ID dekoduje instrukcija skoka, utvrđuje adresa skoka i odlučuje da li treba napraviti skok ili ne. Ova metoda nema smisla ni kod prvobitne varijante razmatranog procesora *pipeline* organizacije (slika 12). Kod nje se u stepenu ID dekoduje instrukcija skoka, a adresa skoka utvrđuje u stepenu MEM. Međutim, u istom stepenu se i odlučuje da li ima ili nema skoka, pa opet ova metoda predviđanja nema smisla.

Ova metoda bi imala smisla ukoliko bi se, na primer, u stepenu ID ne samo dekodovala instrukcija skoka, već i utvrđivala adresa skoka, a tek u stepenu MEM odlučivalo da li ima ili nema skoka. U slučaju takvog procesora *pipeline* organizacije, stanje u *pipeline*-u za tehniku predviđanja ima skoka u slučaju kada je bilo skoka dato je na slici 39, a za slučaj kada nije bilo skoka na slici 40. U oba slučaja se u stepenu ID utvrđuje da je reč o instrukciji skoka i sračunava se adresa skoka. Za to vreme u stepenu IF se čita instrukcija koja je fizički u programu iza nje. S obzirom da je uvek predviđanje da će biti skok, instrukcija koja se čita u stepenu IF ne odgovara predviđanju. Stoga se na takt iz stepena ID u registar PC stepena IF upisuje adresa instrukcije na koju se skače, a instrukcija očitana u stepenu IF kod prebacivanja, na isti takt, u stepen ID pretvara u instrukciju bez dejstva. U oba slučaja je situacija u *pipeline*-u ista do kompletiranja faze MEM instrukcije skoka, uz izgubljen jedan takt. Posle toga situacija u *pipeline*-u se razlikuje za slučajeve kada je bilo skoka (slika 39) i kada nije bilo skoka (slika 40). U slučaju da je bilo skoka, pošto je u predviđanje bilo da ima skoka, u *pipeline*-u su korektne instrukcije, pa se produžava sa sekvencijalnim očitavanjem instrukcija bez dodatnih gubitaka taktova. U slučaju da nije bilo skoka, u stepenu EX je instrukcija bez dejstva, a u stepenima ID i IF pogrešne instrukcije. Stoga se i instrukcije u stepenima ID i IF pretvaraju u instrukcije bez dejstva, i time čisti (flush) *pipeline* od instrukcija koje se nalaze počevši od adrese skoka, čime se gube još dva takta. Iz stepena MEM se u registar PC u stepenu IF upisuje adresa prve sledeće instrukcije iza instrukcije skoka koja je **not taken** i u taktu 5 se u stepenu IF vrši njeno očitavanje.

instr. *i* (**branch taken**)	IF	ID	EX	MEM	*idle*				
instr. *i* + 1		IF	*idle*	*idle*	*idle*	*idle*			
instr. sa adr. skoka			IF	ID	EX	MEM	WB		
instr. sa adr. skoka + 1				IF	ID	EX	MEM	WB	
instr. sa adr. skoka + 2					IF	ID	EX	MEM	WB

Slika 39. Stanje u *pipeline*-u za tehniku predviđanja ima skoka ako je bilo skoka

instr. *i* (**branch not taken**)	IF	ID	EX	MEM	*idle*					
instr. *i* + 1		IF	*idle*	*idle*	*idle*	*idle*				
instr. sa adr. skoka			IF	ID	*idle*	*idle*	*idle*			
instr. sa adr. skoka + 1				IF	*idle*	*idle*	*idle*	*idle*		
instr. *i* + 1					IF	ID	EX	MEM	WB	
instr. *i* + 2						IF	ID	EX	MEM	WB

Slika 40. Stanje u *pipeline*-u za tehniku predviđanja ima skoka ako nije bilo skoka

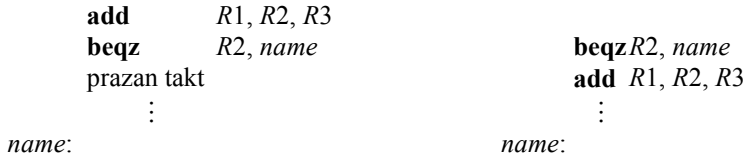
Efekat tehnike predviđanja ima skoka za slučaj procesora *pipeline* organizacije kod koga bi se u stepenu ID dekodovala instrukcija skoka i utvrđivala adresa skoka, a u stepenu MEM odlučivalo o skoku, je jedan izgubljen takt ako ima skoka i tri izgubljena takta ako nema skoka.

3.3.3.1.4 Zakašnjen skok (delayed branch)

Kod nekih procesora negativni efekti upravljačkog hazarda se ublažavaju ili eliminišu korišćenjem softverske metode zakašnjeni skok (*delayed branch*). Kod ove metode se tokom prevođenja iza instrukcije skoka stavlja onoliko instrukcija koliko bi perioda signala takta trebalo zaustaviti *pipeline* dok se ne donese odluka o tome koja će se sledeća instrukcija iza instrukcije skoka izvršavati. To moraju da budu instrukcije koje treba izvršiti bez obzira na to da li je kao rezultat izvršenja instrukcije skoka skok napravljen ili ne. Ova metoda, slično kao i tehnika zakašnjenog punjenja (*delayed load*), zahteva da u vreme prevođenja prevodilac nađe instrukcije koje može da smesti iza instrukcije skoka. U slučaju modifikovakog

procesora *pipeline* organizacije (slika 33) odlučivanje o skoku se realizuje u stepenu ID, pa bi zaustavljanje *pipeline*-a bilo jedna ili nijedna perioda signala takta. Stoga bi pri korišćenju ove metode trebalo ubaciti jednu instrukciju. U slučaju nemodifikovanog procesora *pipeline* organizacije (slika 12) odlučivanje o skoku bi se realizovalo u stepenu MEM, pa bi zaustavljanje *pipeline*-a bilo tri ili dve periode signala takta. Stoga bi bi pri korišćenju ove metode trebalo ubaciti tri instrukcije.

Jedan primer sekvence instrukcija bez i sa primenom ove tehnike dat je na slici 41.



Slika 41 Zakašnjen skok jedne periode signala takta sa nezavisnom instrukcijom

Ovo je najjednostavniji slučaj kada postoji nezavisna instrukcija čije izvršavanje ne utiče na uslovni skok. Instrukciju **add**, stoga, može prevodilac prebaciti iza instrukcije **beqz**. Ona treba uvek da se izvršava i izvršavaće se bez zaustavljanja *pipeline*-a zbog instrukcije skoka **beqz** bez obzira da li će biti skoka ili ne.

Ako je nemoguće naći nezavisnu instrukciju, onda se koriste i druge strategije nalaženja instrukcija.

Primeri a) from target b) from fall through Uporediti ih sa komentarima

3.3.3.2 Dinamičko predviđanje

Dinamičko predviđanje skokova zahteva postojanje posebne jedinice koja će dinamički predviđati da li će skok biti napravljen ili ne. Za razliku od hardverskih tehnika koje to rade statički, uvek predviđajući da će skok biti napravljen (taken) ili da neće biti napravljen (not taken), jedinica za dinamičko predviđanje skokova menja svoje predviđanje u toku samog izvršavanja programa. Najčešće korišćene jedinice za dinamičko predviđanje razlikuju se u samoj realizaciji jedinica i korišćenim šemama za predviđanje.

3.3.3.2.1 Jedinica za predviđanje

Razmotriće se dve najčešće korišćene realizacije jedinice za dinamičko predviđanje i to jedinica sa baferom predviđanja (**branch prediction buffer**) i jedinica sa kešom predviđanja (**branch target cache**). Ove jedinice se razlikuju po složenosti, po stepenu *pipeline*-a u kome treba da budu smeštene i po efikasnosti.

Jedinica sa baferom predviđanja

Jedinica sa baferom predviđanja sadrži memoriju u koju se ulazi samo za instrukcije skoka i to na osnovu nekoliko najmlađih bitova adrese instrukcije skoka. Memorija u svakoj lokaciji sadrži samo jedan bit koji se naziva bit predviđanja i koji vrednostima 1 i 0 određuje da li je pri poslednjem izvršavanju instrukcije skoka na toj adresi skok napravljen ili ne. Ako je vrednost 1 očitavanje i izvršavanje instrukcija se produžava sa adrese skoka. U suprotnom slučaju očitavanje i izvršavanje instrukcija se produžava sekvencijalno. Ako se kasnije u

stepenu *pipeline*-a u kome se odlučuje da li treba napraviti skok ili ne utvrdi da je predviđanje bilo pogrešno, sve instrukcije u *pipeline*-u iza instrukcije skoka se ispiraju i kreće se sa očitavanjem korektne instrukcije. Pritom se bit predviđanja u memoriji invertuje.

Korišćenje jedinice sa baferom predviđanja je moguće u stepenu *pipeline*-a u kome se dekoduje instrukcija, jer se na osnovu toga što je dekodovana instrukcija skoka utvrđuje da treba ići u bafer predviđanja. U slučaju razmatranog procesora *pipeline* organizacije to je stepen ID. Potrebno je, dalje, da može da se u tom istom stepenu formira adresa skoka, da bi se, u slučaju predviđenog skoka, na sledeći takt produžilo sa izvršavanjem programa od instrukcije na koju se skače. U slučaju modifikovanog procesora *pipeline* organizacije adresa skoka se, takođe, zna u stepenu ID. Međutin, primena ove tehnike kod njega nema smisla, jer se u stepenu ID i zna da li skok treba napraviti ili ne.

Ova tehnika bi imala smisla ukoliko bi se odluka o skoku donosila u nekom kasnijem stepenu *pipeline*-a, na primer MEM, a instrukcija dekodovala i adresa skoka utvrđivala u nekom ranijem stepenu, na primer ID. U slučaju originalnog procesora *pipeline* organizacije odluka o skoku se donosi u stepenu MEM, a instrukcija dekoduje u stepenu ID. Potrebno je u stepen ID staviti i sabirač da bi se u njemu utvrđivala i adresa skoka. U stepen ID se stavlja i jedinica sa baferom predviđanja. Očitavanje instrukcija se sada realizuje na sledeći način. Na svaki signal takta nova instrukcija dolazi iz stepena IF u stepen ID. U stepenu ID instrukcija se dekoduje i na uobičajeni način čitaju operandi. Međutim, paralelno sa tim proverava se rezultat dekodovanja instrukcije. Ukoliko dekodovana instrukcija nije instrukcija skoka, produžava se sa sekvencijalnim očitavanjem instrukcija. Ukoliko je dekodovana instrukcija instrukcija skoka, ulazi se u bafer predviđanja i čita bit predviđanja. U slučaju da je bit predviđanja nula, produžava se sa sekvencijalnim očitavanjem instrukcija, dok se u slučaju da je bit predviđanja jedan, produžava sa očitavanjem instrukcija od adrese skoka. Situacije u *pipeline*-u su identične sa situacijama za statička predviđanja **branch not taken** i **branch taken**, respektivno. Treba uočiti da se sa instrukcijom skoka od stepena ID, preko stepena EX do stepena MEM vuku i bit predviđanja i adresa instrukcije skoka. U stepenu MEM se za instrukciju skoka utvrđuje da li skoka ima ili ne, pa se na osnovu upoređivanja ishoda instrukcije skoka i bita predviđanja utvrđuje da li je predviđanje bilo dobro ili ne. Ukoliko je predviđanje bilo dobro, ne preduzima se nikakva akcija. U suprotnom slučaju preduzimaju se dve vrste akcija. Prvo, instrukcije u stepenima EX, ID i IF su nekorektne, pa se ovi stepeni ispiraju tako što se instrukcije u njima pretvaraju u instrukcije bez dejstva. Drugo, iz stepena MEM se ulazi u bafer predviđanja sa nekoliko najmlađih bitova adrese instrukcije skoka i invertuje bit predviđanja.

Treba zapaziti da je, zbog toga što se u bafer predviđanja ulazi na osnovu nekoliko najmlađih bitova adrese instrukcije skoka, moguće da različite instrukcije skoka, ukoliko se nalaze na adresama koje imaju iste najmlađe bitove koji se koriste za ulaz u bafer predviđanja, koriste isti bit predviđanja. Zato se može desiti da se predviđanje za neku instrukciju skoka donosi na osnovu vrednosti bita predviđanja koji je postavila neka druga instrukcija skoka. Treba napomenuti da efekat ovoga može samo da bude nekorektno predviđanje i ispiranje *pipeline*-a, ali ne i nekorektno izvršavanje instrukcija. Ovo bi, međutim, trebalo, uz pažljivo dimenzionisanje bafera predviđanja, dosta retko da se javlja.

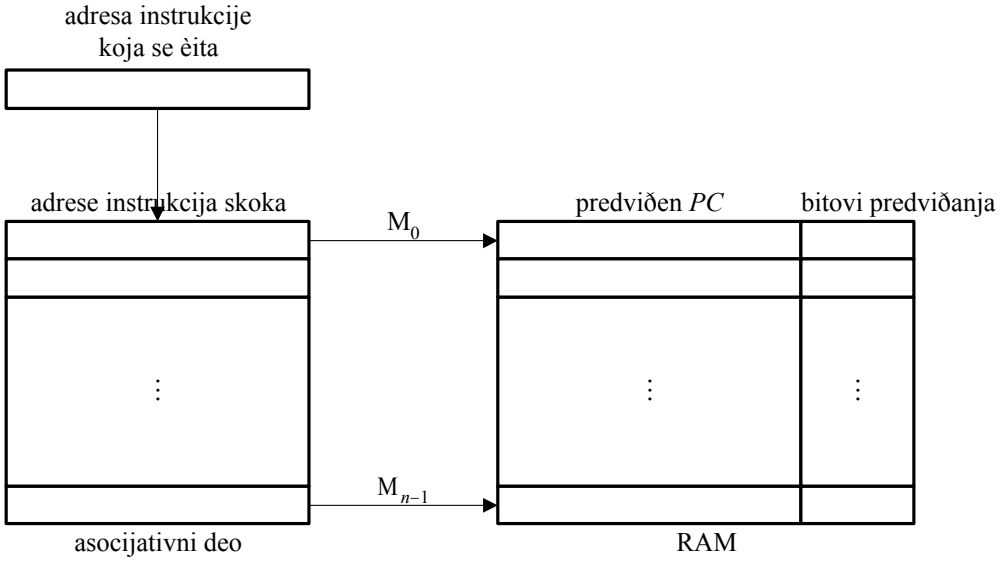
Nedostatak tehnika predviđanja sa baferom predviđanja je da se odlučivanje na osnovu bafera predviđanja može realizovati tek kada se utvrdi da je reč o instrukciji skoka. To je moguće uraditi tek u stepenu *pipeline*-a u kome se dekoduje instrukcija. Ako je stepen dekodovanja instrukcije prvi stepen (ID) iza stepena čitanja instrukcije (IF), onda se u slučaju

da je bit predviđanja jedan, sa očitavanjem instrukcije od adrese skoka kasni jednu periodu signala takta.

Da bi se ovo izbeglo potrebno je još dok se instrukcija skoka čita u stepenu IF utvrditi da li se čita instrukcija skoka, napraviti predviđanje da li će se skok realizovati ili ne, i ako je predviđanje da će se skok realizovatim utvrditi adresu skoka. Time će biti omogućeno da se na isti takt, kojim se očitana instrukcija skoka prebacuje iz stepena IF u stepen ID u kome se tek vrši njeno dekodovanje, krene u stepenu IF sa očitavanjem instrukcije sa adrese prema predviđenom ishodu skoka.

Jedinica sa kešom predviđanja

Jedinica sa kešom predviđanja ima strukturu koja je jako slična sa strukturom keš memorije i može da bude realizovan u tehnici asocijativnog, direktnog ili set asocijativnog preslikavanja. Funkcionisanje jedinice sa kešom predviđanja je dato za slučaj asocijativnog preslikavanja, a razlike u slučaju realizacija sa direktnim i set asocijativnim preslikavanjima su samo u delu u kome se utvrđuje da li ima saglasnosti. U slučaju asocijativne realizacije keša predviđanja u asocijativnom delu se čuvaju adrese izvršavanih instrukcija skokova (slika 42). U odgovarajućem ulazu RAM dela nalazi se predviđena vrednost za PC i bit predviđanja. Predviđena vrednost za PC je adresa prve sledeće instrukcije iza instrukcije skoka ukoliko je bit predviđanja nula i adresa skoka ukoliko je bit predviđanja jedan.



Slika 42 Keš za predikciju skokova asocijativne realizacije

Funkcionisanje jedinice sa kešom predviđanja se daje za originalni procesor *pipeline* organizacije kod koga se odluka o skoku donosi u stepenu MEM. Jedinica se stavlja u stepen IF u kome se vrši očitavanje instrukcije. Očitavanje instrukcija se sada realizuje na sledeći način.

Kad god se u stepenu IF čita neka instrukcija adresa te instrukcije se vodi na ulaze asocijativnog dela keš memorije i vrši provera da li se u nekom od ulaza asocijativnog dela nalazi ta vrednost. Moguće su sledeće realizacije:

1) Ako se ne nalazi, to znači ili da nije reč o instrukciji skoka ili je reč o instrukciji skoka koja do sada nije izvršavana, pa se produžava sa sekvencijalnim očitavanjem instrukcija
2) Ako se nalazi to znači da je instrukcija koja se čita instrukcija skoka koja je već bila izvršavana, pa za nju postoji predviđanje da li će biti skoka ili ne. Tada se iz RAM dela

keša dobijaju predviđena vrednost za PC i bit predviđanja. Na isti signal takta se očitana instrukcija prebacuje iz stepena IF u stepen ID i predviđena vrednost za PC upisuje u PC. Treba uočiti da se sa instrukcijom skoka od stepena IF, preko stepena ID i EX do stepena MEM vuku i bit predviđanja i adresa instrukcije skoka. U stepenu MEM se za instrukciju skoka utvrđuje da li skoka ima ili ne. Tada

1) treba ažurirati keš memoriju i
2) videti da li su instrukcije iza te instrukcije dobre.

Ažuriranje keš memorije može da zahteva:

- ako nema tog ulaza—unošenje kompletnih informacija,
- ako ima tog ulaza—ažuriranje bita predviđanja i možda predviđene vrednosti za *PC* ako dođe do promene predviđanja.

Videti da li su instrukcije iza te instrukcije dobre zahteva:

- ako za tu instrukciju nije bilo ulaza u keš memoriji pa nije bilo predviđanja:
  - skok napravljen—ispire se,
  - skok nije napravljen—u redu;
- ako je za tu instrukciju bio ulaz u keš memoriji pa je bilo predviđanja:
  - pogrešno predviđeno—ispire se,
  - dobro predviđeno—u redu.

Treba napomenuti da se može javiti i situacija da se u stepenu MEM nađe neka instrukcija skoka koja se do tog trenutka nije pojavljivala, pa se ne nalazi u kešu za predviđanje, i da su svi ulazi keša za predviđanje popunjeni. Tada se primenom nekog od algoritama zamene iz keša predviđanja izbacuje neka od instrukcija skoka i time stvara prostor za ubacivanje date instrukcije skoka.

U razmotrenom kešu za predviđanje nalaze se izvršene instrukcije skoka bez obzira na to da li je skok napravljen ili ne. Moguća je i varijanta realizacije keša za predviđanje kod koje se u kešu predviđanja nalaze instrukcije skoka samo ukoliko je skok napravljen. To zahteva da se instrukcije skoka ubacuju u keš za predikciju kad god je skok napravljen i da se izbacuju iz za predikciju kad god skok nije napravljen.

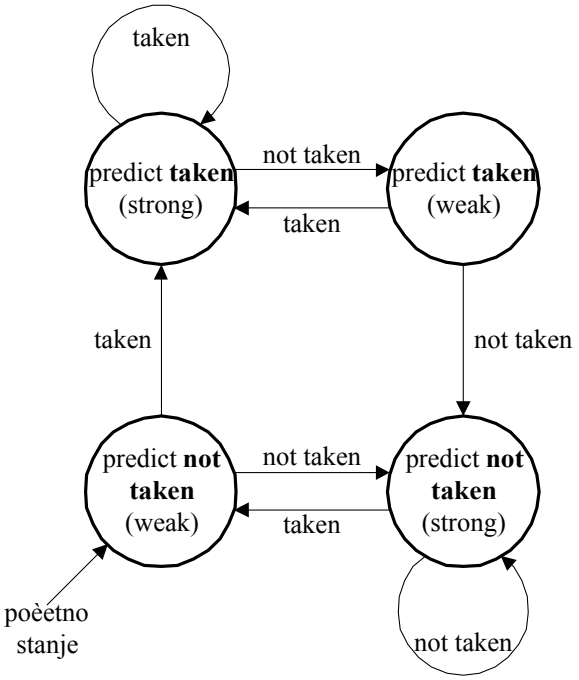
3.3.3.2.2 Šeme za predviđanje

Predviđanje sa jednim bitom je najjednostavija šema za predviđanje. Ona, međutim, ima nedostatak u petljama gde se skok, sem kada se izlazi iz petlje, stalno pravi. Iako se ovde skok samo jedanput ne pravi, ova tehnika će dva puta pogrešno predvideti. Kod prvog prolaska kroz petlju kada se prođe kroz instrukciju skoka pogrešno će se predvideti da neće biti skoka a napraviće se skok i postaviće se bit predviđanja. Kod svakog sledećeg prolaska postavljen bit predviđanja korektno će predvideti skok. Kod poslednjeg prolaska kroz petlju predvideće se pogrešno skok, jer se tada izlazi iz petlje, i invertovaće se bit predviđanja. Kod novog ulaska u petlju, i to kod prvog prolaska, ponovo će biti predviđanje da neće biti skoka. Da bi se ovakve situacije efikasnije rešavale veoma često se koristi šema predviđanja sa 2 umesto sa 1 bitom. Ažuriranje dvobitne vrednosti i korišćenje tih vrednosti za predikciju su dati na slici 43.

Sa slike se vidi da će se posle dva ili više napravljenih skokova (**taken**) snažno preporučivati (predict **taken**—strong) da se sledeći put napravi skok. Posle prvog nenapravljenog skoka (not taken) još uvek će se ali nešto slabije (predict **taken**—weak) preporučivati da se sledeći put napravi skok. Posle dva uzastopna nenapravljena skoka (**not taken**) sledeći put će se snažno preporučivati (predict **not taken**—strong) da se sledeći put ne

pravi skok. Posle prvog napravljenog skoka još uvek će se ali slabije preporučivati (predict **not taken**—weak) da se sledeći put ne pravi skok. Posle dva uzastopna napravljena skoka ponovo će se snažno preporučivati (predict **taken**—strong) da se sledeći put pravi skok.

Korišćenje bafera predviđanja ili keša predviđanja sa 2 bita, trenutak ažuriranja, situacije koje mogu da nastanu u *pipeline*-u i način njihovog rešavanja su isti kao i kod bafera predviđanja ili keša za predikciju sa 1 bitom.



Slika 43 Stanja i prelazak iz stanja u dvobitnoj šemi predviđanja

Dvobitna šema za predviđanje je najjednostavniji slučaj jedne opštije šeme za predviđanje koja ima n-bitni brojač (saturating counter) za svaki ulaz bilo bafera predviđanja bilo keša predviđanja.U slučaju n-bitnog brojača, brojač može da ima vrednosti u opsegu 0 do 2^n-1. Kada brojač ima vrednost veću ili jednaku polovini maksimalne vrednosti (2^{n-1}), predviđanje je **taken**, dok je u suprotnom slučaju **not taken**. Kao i u slučaju 2-bitne šeme, brojač se inkrementira na svaki napravljen skok i dekrementira na svaki nenapravljen skok. Studije n-bitnih šema za predikciju ukazuju da one daju neznatno bolje rezultate nego 2-bitne šeme za predviđanje. Zbog toga se danas 2-bitne šeme za predviđanje daleko više koriste nego ooopštije n-bitne šeme za predikciju.

Posebne šeme za predviđanje su korelisane šeme za predviđanje koje mogu da budu ugrađene i u jedinice sa baferom predviđanja i u jedinice sa kešom predviđanja. Jedna od tih šema se zove (1,1) šema za predviđanje. Ova šema koristi ponašanje na zadnjoj instrukciji skoka za izbor jedne od dve jedinice za predviđanje sa 1-bitnom šemom za predviđanje. Jedna jedinica se koristi za slučaj da na zadnjoj izvršenoj instrukciji skoka nije napravljen skok, a druga za slučaj da je skok napravljen. Ovde mora da postoji i poseban flip-flop koji vrednostima nula i jedan ukazuje da li na zadnjoj izvršenoj instrukciji skoka skok nije napravljen ili je napravljen, respektivno. Kada se sada čita instrukcija skoka na osnovu vrednosti flip-flopa se za predviđanje koristi odgovarajuća jedinica za predviđanje. Ukoliko ishod instrukcije skoka odgovara predviđanju, ništa se ne dira, dok se u suprotnom slučaju ažurira jedinica za predviđanje na osnovu koje je i napravljeno predviđanje. Na osnovu ishoda instrukcije skoka se postavlja i flip-flop, da bi se na sledećoj instrukciji skoka na osnovu

njegove vrednosti za predviđanje birala jedna od dve jedinice sa 1-bitnom šemom za predviđanje.

U opšem slučaju (m,n) šema za predviđanje koristi ponašanje na zadnjih m instrukcija skoka za izbor jedne od 2^m jedinica za predviđanje, pri čemu jedinice koriste n-bitne šeme za predviđanje. Ovde je potreban i jedan m-bitni šift registar koji beleži ponašanje na zadnjih m instrukcija skokova, pri čemu svaki bit vrednostima nula i jedan određuje da li skok nije napravljen ili je napravljen. Trenutna vrednost ovog šift registra se koristi za selekciju jedne od 2^m jedinica za predviđanje.

3.3.4 Prekid

Programski dostupni registri procesora su registri opšte namene R0 do R31, programski brojač PC i programska statusna reč ISCR (*Interrupt Status and Control Register*). Svi registri su dužine 32 bita. Registri R0, R29, R30 i R31 imaju posebne uloge. Registar R0 uvek ima vrednost 0, registar R29 se koristi kao ukazivač na vrh steka, u registru R30 se čuva adresa povratka prilikom poziva potprograma i u registru R31 se čuva adresa povratka prilikom pokretanja opsluživanja prekida. U registru ISCR (slika 1) bitovi 31 do 10 su nule, dok bitovi 9 do 0 imaju sledeće značenje: GM – globalna maska za dozvolu ili zabranu svih prekida, INI – indikator inicijalizacije procesora, TRAP – indikator prekida izazvanog instrukcijom **`trap`**, UOE – indikator prekida izazvanog pogrešnim kodom operacije, ARE – indikator prekida izazvanog prekoračenjem pri aritmetičkoj operaciji, NMI – indikator zahteva za nemaskirajući prekid, MI – indikator zahteva za maskirajući prekid, MASK – maska za dozvolu ili zabranu maskirajućih prekida, TRM – indikator režima prekid posle svake instrukcije i INC – indikator da prekidna rutina treba da inkrementira adresu povratka.

31	...	10	9	8	7	6	5	4	3	2	1	0
0	...	0	GM	INI	TRAP	UOE	ARE	NMI	MI	MASK	TRM	INC

Slika 1 Struktura registra ISCR

Instrukcije skoka se svrstavaju u četiri grupe, i to instrukcije uslovnog skoka, instrukcija bezuslovnog skoka, instrukcija skoka na potprogram i instrukcija prekida. Instrukcije uslovnog skoka **`beqz`** *rs1*, *imm* i **`bnez`** *rs1*, *imm* realizuju relativni skok sa pomerajem *imm* u odnosu na registar PC, ukoliko je sadržaj registra opšte namene *rs1* jednak ili nije jednak nuli, respektivno. Instrukcija bezuslovnog skoka **`jr`** *rs1*, *imm* realizuje skok sa pomerajem *imm* u odnosu na registar *rs1*. Ova instrukcija se koristi i za povratak iz potprograma i povratak iz prekidne rutine. Instrukcija skoka na potprogram **`jsr`** *rd*, *rs1*, *imm* prebacuje sadržaj registra PC u registar *rd* i realizuje skok sa pomerajem *imm* u odnosu na registar *rs1*. Preporučuje se da *rd*, u slučaju ove instrukcije, bude registar opšte namene R30. Instrukcija prekida **`trap`** *rs1*, *imm* programskim putem izaziva pokretanje opsluživanja prekida i prebacuje sadržaj registra PC u registar R31 i realizuje skok sa pomerajem *imm* u odnosu na registar *rs1*.

Instrukcijama za pristup registru ISCR **`trm`** *imm*, **`intm`** *imm* i **`intgm`** *imm* se u bitove 1,2 i 9 registra ISCR upisuje najmlađi bit neposredne veličine *imm* i time aktivira/deaktivira režim rada sa prekidom posle svake instrukcije, dozvoljava/zabranjuje opsluživanje maskirajućih prekida i dozvoljava/zabranjuje opsluživanje svih prekida, respektivno. Instrukcijom **`movs2i`** *rd* se sadržaj registra ISCR upisuje u registar opšte namene *rd*. Instrukcijom **`movi2s`** *rs2* se sadržaj registra opšte namene *rs2* upisuje u registar ISCR. Prilikom ovog upisa ne menjaju se bitovi 7, 6, 5 i 0 (TRAP, UOE, ARE i INC) registra ISCR.

Mehanizam prekida je realizovan na takav način da se vrednost programskog brojača PC, instrukcije u kojoj se prihvata prekid i koja predstavlja adresu povratka iz prekidne rutine, upisuje u registar opšte namene R31 i adresa prekidne rutine, koja je fiksna ili data adresnim delom instrukcije **`trap`**, upisuje u programski brojač PC. Prekidi koji se mogu javiti su: prekid izazvan instrukcijom **`trap`**, prekid zbog pogrešnog koda operacije, prekid zbog prekoračenja pri aritmetičkoj operaciji, nemaskirajući prekid, maskirajući prekid i prekid zbog zadatog režima rada sa prekidom posle svake instrukcije, pri čemu se njihova pojava evidentira postavljanjem indikatora TRAP, UOE, ARE, NMI, MI i TRM u registru ISCR, respektivno (slika 1). Prekidi TRAP, UOE, ARE i TRM su unutrašnji, a NMI i MI spoljašnji. Prekidi TRAP i TRM su izazvani programski, a UOE i ARE problemima pri izvršavanju instrukcija. Prekid NMI izaziva uređaj koji kontroliše ispravnost rada računarskog sistema, a MI ulazno/uzlazni uređaji. Prekidi imaju prioritete, pri čemu najviši prioritet ima TRAP, zatim redom UOE, ARE, NMI, MI do najnižeg TRM. Svi prekidi mogu da budu maskirani bitom globalne maske GM u registru ISCR i samo maskirajući prekid MI može da bude maskiran bitom maske MASK u registru ISCR, dok se pojedinačni prekidi od ulazno/izlaznih uređaja, koji se svi šalju kao prekid MI, mogu pojedinačno maskirati na samim uređajima.

U svakom taktu se vrši analiza indikatora prekida i bitova maski u registru ISCR i na osnovu njihovih vrednosti, pokreće ili ne pokreće opsluživanje prekida. Ako se utvrdi da treba pokrenuti opsluživanje prekida, u jednom taktu se vrednost programskog brojača PC, instrukcije u kojoj se prihvata prekid i koja predstavlja adresu povratka iz prekidne rutine, upisuje u registar opšte namene R31, adresa prekidne rutine, koja je fiksna ili data adresnim delom instrukcije **`trap`**, upisuje u programski brojač PC i bit globalne maske GM registra ISCR postavlja na neaktivnu vrednost (slika 1). Pored toga, tom prilikom se i bit INC registra ISCR postavlja na aktivnu ili neaktivnu vrednost. Adresa povratka iz prekidne rutine je za sve instrukcije, sem instrukcije skoka, adresa sledeće instrukcije u odnosu na instrukciju tokom čijeg izvršavanja se prihvata prekid, pa se u tim slučajevima u registar opšte namene R31 upisuje vrednost programskog brojača PC za instrukciju tokom čijeg izvršavanja se prihvata prekid i bit INC registra ISCR se postavlja na aktivnu vrednost. Međutim, ako se prekid prihvata u toku izvršavanja instrukcije skoka, adresa povratka iz prekidne rutine može da bude adresa instrukcije na koju bi se skočilo da nije bilo prekida, pa se u tom slučaju pamti adresa skoka, a bit INC registra ISCR postavlja na neaktivnu vrednost.

Tokom izvršavanja prvih nekoliko instrukcija prekidne rutine prekidi nisu dozvoljeni, jer je bit globalne maske GM registra ISCR postavljen na neaktivnu vrednost. Zbog toga se pretpostavlja da je u tom delu prekidna rutina pažljivo napisana i da ne može dovesti do pojave unutrašnjih prekida. Sa prvih nekoliko instrukcija prekidne rutine se na steku čuvaju registar ISCR i registri opšte namene čije se vrednosti menjaju u prekidnoj rutini. Pri tome se obavezno čuva registar R31, u kome se nalazi adresa povratka iz prekidne rutine, a kao ukazivač na vrh steka se koristi registar R29. U ovom delu se, po potrebi, može izvršiti i instrukcija kojom se u indikator režima rada sa prekidom posle svake instrukcije TRM registra ISCR upisuje neaktivna vrednost. Potom se, po opadajućim prioritetima, vrši provera indikatora TRAP, UOE, ARE, NMI, MI i TRM u registru ISCR, da bi se utvrdio prekid najvišeg prioriteta i skočilo na odgovarajuću prekidnu rutinu. Sada je moguće, u zavisnosti od željenog režima izvršavanja prekidne rutine, izvršiti instrukcije kojima se u bitove GM i TRP registra ISCR upisuju aktivne vrednosti i time dozvoliti prekide i zadati režim rada sa prekidom posle svake instrukcije.

Sa zadnjih nekoliko instrukcija prekidne rutine realizuje se povratak u prekinuti program. Na početku ovog dela se, najpre, izvršava instrukcija kojom se u bit globalne maske GM registra ISCR upisuje neaktivna vrednost i time ponovo maskiraju svi prekidi. Potom se sa

steka restauriraju sadržaji onih registara opšte namene čiji su sadržaji na početku prekidne rutine sačuvani na steku i registra ISCR. Zatim se, ukoliko je bit INC registra ISCR postavljen na aktivnu vrednost, inkrementira sadržaj registra opšte namene R31, koji sadrži adresu povratka. Na kraju se, instrukcijom bezuslovnog skoka **`jr`** *rs1*, *imm*, u kojoj je *rs1* registar R31, a pomeraj *imm* nula, realizuje povratak u prekinuti program.

Prekidi, takođe, izazivaju upravljačke hazarde. S obzirom na to da se posle utvrđivanja zahteva za prekid prelazi na izvršavanje instrukcija prekidne rutine, instrukcije u stepenima *pipeline*–a posle instrukcije u stepenu *pipeline*–a u kome se utvrđuje prekid su pogrešne i potrebno ih je isprati. Kako je usvojeno da se prisustvo prekida utvrđuje u stepenu MEM *pipeline*–a, neophodno je isprati instrukcije u stepenima IF, ID i EX *pipeline*–a. Zbog toga prekidi, kao i instrukcije skoka, izazivaju gubitak od tri periode signala takta. Međutim, u odnosu na instrukcije skoka situacija je nešto složenija, jer je, pre upisa adrese prekidne rutine u registar PC, potrebno vrednost PC–ja, instrukcije u toku čijeg izvršavanja je utvrđen prekid, upisati u registar opšte namene R31.

Spoljašnji nemaskirajući i maskirajući zahtevi za prekid se, na svaki signal takta, upisuju u razrede NMI i MI registra ISCR, koji se nalazi u stepenu MEM. Prekid, zbog režima rada sa prekidom posle svake instrukcije, se zadaje izvršavanjem instrukcije **`trm`**, koja u stepenu MEM postavlja na aktivnu vrednost bit TRM registra ISCR. Prekid, zbog instrukcije prekida **`trap`**, se otkriva sa dekodovanjem operacija, ali se utvrđuje tek u stepenu MEM *pipeline*–a. Zbog toga se ova instrukcija prenosi do stepena MEM, gde, kao indikacija ove vrste prekida, služi bit TRAP *pipeline* registra R_{MEM}. Slična je situacija i sa prekidom zbog pogrešnog koda operacije, koji se otkriva sa dekodovanjem operacija, ali se prenosi do stepena MEM, u kome se utvrđuje prekid i gde, kao indikacija ove vrste prekida, služi bit UOE *pipeline* registra R_{MEM}. Prekid, zbog prekoračenja, se otkriva u stepenu EX, ali se instrukcija, tokom čijeg izvršavanja je otkriven ovaj prekid, prenosi do stepena MEM, u kome se utvrđuje prekid i gde, kao indikacija ove vrste prekida, služi bit ARE *pipeline* registra R_{MEM}.

Bitovi zahteva za prekid TRAP, UOE i ARE *pipeline* registra R_{MEM} i NMI, MI i TRM registra ISCR, kao i bitovi za maskiranje prekida GM i MASK registra ISCR, hardverski se u stepenu MEM analiziraju u svakom taktu. Ako se utvrdi da treba pokrenuti opsluživanje prekida, hardverski se adresa povratka iz prekidne rutine upisuje u registar opšte namene R31, postavlja se bit INC registra ISCR na odgovarajuću vrednost, zabranjuju se svi prekidi postavljanjem bita GM registra ISCR na neaktivnu vrednost, prepisuju se bitovi TRAP, UOE i ARE iz *pipeline* registra R_{MEM} u odgovarajuće bitove registra ISCR, ispiraju se instrukcije u stepenima *pipeline*-a i poziva prekidna rutina, upisivanjem adrese prekidne rutine u programski brojač PC.

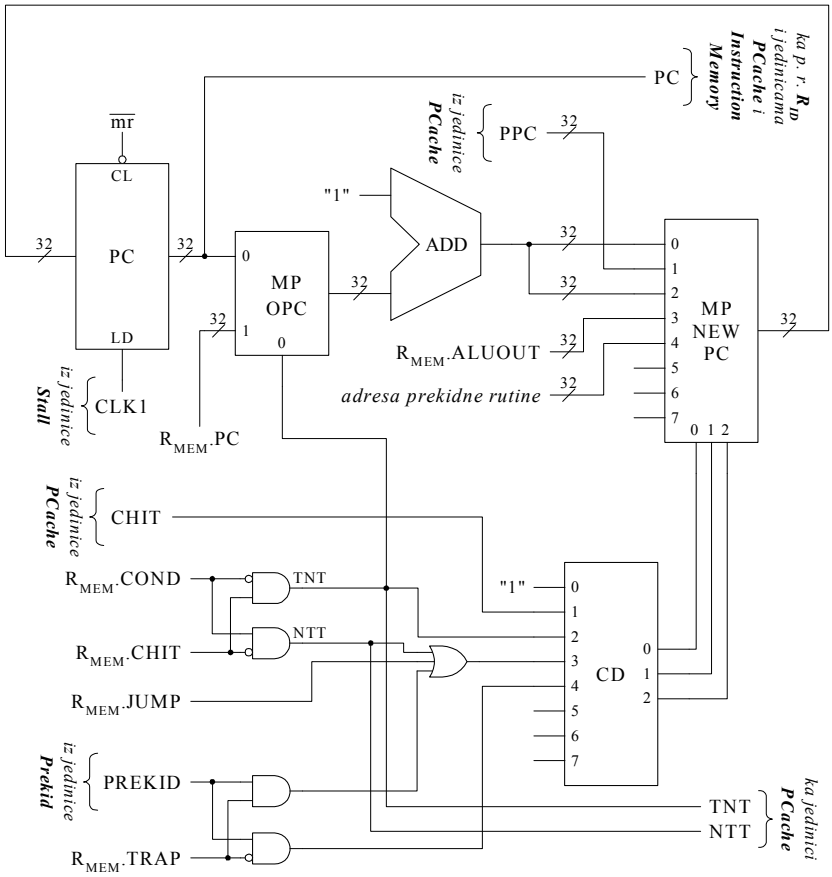
U većini situacija, nakon povratka iz prekidne rutine, izvršavanje programa treba nastaviti od sledeće instrukcije u odnosu na instrukciju koja se nalazila u stepenu MEM u taktu kada je utvrđen prekid i pokrenuto opsluživanje prekida. Kako se tada u registar R31 upisuje vrednost polja R_{MEM}.PC, u kome se nalazila adresa instrukcije u stepenu MEM, a ne sledeće instrukcije, bit INC registra ISCR se tada postavlja na aktivnu vrednost. Zbog toga prekidna rutina treba da inkrementira sadržaj registra R31. Međutim, u situaciji kada je u stepenu MEM instrukcija bezuslovnog skoka ili uslovnog skoka za koji je uslov ispunjen, izvršavanje programa, nakon povratka iz prekidne rutine, treba nastaviti od instrukcije na koju bi se skočilo da nije bilo prekida. Kako se tada u registar R31 upisuje vrednost polja R_{MEM}.ALUOUT, u kome se nalazi odredišna adresa skoka, bit INC registra ISCR se tada postavlja na neaktivnu vrednost. Zbog toga prekidna rutina ne treba da inkrementira sadržaj registra R31.

Da bi se sačuvala adresa povratka iz prekidne rutine, potrebno je vrednost polja R_{MEM}.PC ili R_{MEM}.ALUOUT upisati u registar opšte namene R31. Međutim, direktan upis vrednosti iz stepena MEM u registar R31 registarskog fajla u stepenu ID nije moguć, jer bi se stvorio strukturalni hazard u situaciji kada se u stepenu WB nalazi instrukcija koja u istom taktu upisuje rezultat u neki od registara opšte namene registarskog fajla. Osim toga, u nekim slučajevima u tom taktu, u kome je u stepenu MEM utvrđen prekid, u stepenu MEM može se naći neka od instrukcija koja svoj rezultat treba da upiše u neki od registara opšte namene registarskog fajla, pa njoj treba dozvoliti prelazak iz stepena MEM u stepen WB i upis u registarski fajl. Zbog toga se u taktu, u kome se u stepenu MEM utvrdi postojanje prekida, normalno iz stepena WB vrši upis u registarski fajl. Na prvi sledeći signal takta, instrukcija iz stepena MEM se prebacuje u stepen WB radi eventualnog upisa njenog rezultata u registarski fajl, stepeni IF, ID i EX *pipeline*–a zaustavljaju i u *pipeline* registar R_{MEM} stepena MEM upisuju signali koji su potrebni da bi se adresa povratka upisala u registarski fajl kada se na sledeći takt sadržaj *pipeline* registra R_{MEM} prebaci u *pipeline* registar R_{WB}. Na signal takta na koji se sadržaj *pipeline* registra R_{MEM} prebacuje u *pipeline* registar R_{WB}, čime se omogućuje upis adrese povratka u registar R31 registarskog fajla, u programski brojač PC stepena IF se upisuje adresa prekidne rutine i sadržaji *pipeline* registara R_{ID}, R_{EX} i R_{MEM} brišu. Pošto se sadržaj *pipeline* registara menja iz takta u takt, i pošto postoji potreba da prekidna rutina analizira vrednosti bitova TRAP, UOE i ARE *pipeline* registra R_{MEM} koje su bile aktuelne u trenutku utvrđivanja prekida, potrebno je, prilikom pokretanja opsluživanja prekida, prepisati te bitove u odgovarajuće bitove registra ISCR.

Specijalan slučaj zaustavljanja se javlja prilikom pokretanja opsluživanja prekida. Da bi se pojednostavilo čuvanje stanja procesora, deo 1 *pipeline* registra R_{MEM} se, prilikom prekida, zaustavlja za jedan takt i podaci iz tog dela ponovo prolaze kroz stepen MEM. Za taj deo se koristi signal takta **CLK2** generisan u ovoj jedinici. Ako je signal **PREKID** aktivan, **CLK2** je neaktivan, a u suprotnom se **CLK2** menja na isti način kao **CLK**.

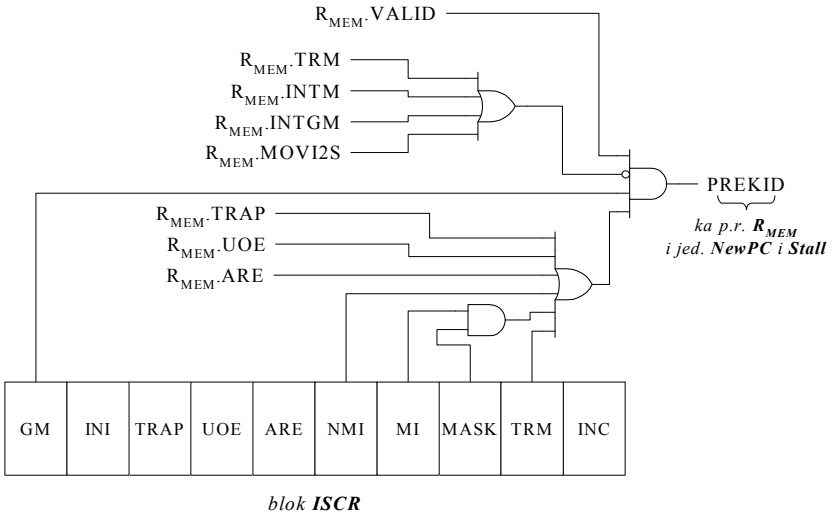
Ukoliko je signal **PREKID** aktivan, a signal $\mathbf{R_{MEM}}$**.TRAP** neaktivan, što je indikacija da se pokreće opsluživanje prekida koje nije izazvano instrukcijom **`trap`**, na izlaz multipleksera MPNEWPC se propušta vrednost sa ulaza 4, koja predstavlja fiksnu adresu prekidne rutine. Ukoliko su ili signali **PREKID** i $\mathbf{R_{MEM}}$**.TRAP** aktivni, što je indikacija da se pokreće opsluživanje prekida izazvano instrukcijom **`trap`**, ili je signal $\mathbf{R_{MEM}}$**.JUMP** aktivan, što je indikacija da se u stepenu MEM nalazi instrukcija bezuslovnog skoka, ili je signal **NTT** aktivan, što je indikacija da se u stepenu MEM nalazi instrukcija uslovnog skoka za koju je predikcija bila *not taken*, pa je signal $\mathbf{R_{MEM}}$**.CHIT** neaktivan, a ishod je *taken*, pa je signal $\mathbf{R_{MEM}}$**.CHIT** aktivan, na izlaz multipleksera MPNEWPC se propušta vrednost R_{MEM}.ALUOUT sa ulaza 3, koja predstavlja zadatu adresu prekidne rutine, u slučaju instrukcije **`trap`**, ili sračunatu adresu skoka, u slučaju instrukcija bezuslovnog i uslovnog skoka. Ukoliko je signal **TNT** aktivan, što je indikacija da se u stepenu MEM nalazi instrukcija uslovnog skoka za koju je predikcija bila *taken*, pa je signal $\mathbf{R_{MEM}}$**.CHIT** aktivan, a ishod je *not taken*, pa je signal $\mathbf{R_{MEM}}$**.CHIT** neaktivan, na izlaz multipleksera MPNEWPC se propušta vrednost sa ulaza 2, koja predstavlja adresu instrukcije iz stepena MEM R_{MEM}.PC uvećanu za 1. Ako je signal **CHIT** aktivan, što je indikacija da u kešu za predikciju postoji ulaz za instrukciju određenu vrednošću registra PC, na izlaz multipleksera MPNEWPC se propušta vrednost PPC sa ulaza 1, koja predstavlja prognoziranu odredišnu adresu skoka pročitanu iz keša za predikciju. Ako su svi signali na ulazima 4 do 1 kodera prioriteta CD neaktivni, što je indikacija da treba produžiti sa sekvencijalnim izvršavanjem programa, na izlaz multipleksera MPNEWPC se propušta vrednost sa ulaza 0, koja, zbog neaktivne

vrednosti signala **TNT** i selekcije vrednosti PC kroz multiplekser MPOPC, predstavlja vrednosti PC uvećanu za 1.



Slika 2 Programski brojač PC i jedinica ***NewPC***

Jedinica ***Prekid*** (slika 3) se sastoji od registra ISCR i kombinacionih mreža za generisanje signala **PREKID**.



Slika 3 Jedinica ***Prekid***

Svi signali zahteva za prekid su povezani na 6-ulazno ILI kolo, i to signali **TRAP**, **UOE** i **ARE** iz *pipeline* registra R_{MEM} i signali **NMI**, **MI**, koji je uslovljen aktivnom vrednošću bita MASK, i **TRM** iz registra ISCR. Međutim, aktivna vrednost signala na izlazu ovog kola daje aktivnu vrednost signala **PREKID**, čime se pokreće opsluživanje prekida, ukoliko su

ispunjena još tri uslova. Prvi je da stepen MEM nije prazan i da se u njemu nalazi neka instrukcija, na šta ukazuje aktivna vrednost signala $\mathbf{R_{MEM}}$.**VALID**. Drugi je da se u stepenu MEM ne nalazi neka od instrukcija koje menjaju sadržaj registra ISCR, jer je u tom slučaju pokretanje opsluživanja prekida zabranjeno, na šta ukazuju neaktivne vrednosti signala $\mathbf{R_{MEM}}$.**TRM**, $\mathbf{R_{MEM}}$.**INTM**, $\mathbf{R_{MEM}}$.**INTGM** i $\mathbf{R_{MEM}}$.**MOVI2S**. Registar ISCR je realizovan kao blok od 10 zasebnih flip-flopova sa pratećim logičkim elementima, da bi se omogućilo nezavisno postavljanje svakog od razreda registra ISCR posebnim instrukcijama i/ili spoljnim signalima. Međutim, pošto svih 10 flip-flopova ima isti signal takta **CLK** i isti signal za brisanje mr, i pošto postoje instrukcije za programsko čitanje/upis svih 10 flip-flopova odjednom, može se smatrati da tih 10 flip-flopova logički sačinjavaju registar ISCR.