SADRŽAJ

1. MIKROPROCESOR Y86

U današnjem stepenu razvoja tehnologije savremeni mikroprocesori sigurno spadaju medju najkompleksije sisteme izradjene od strane čoveka. Jedan silicijumski čip, verovatno ne veći od našeg palca, može da sadrži jedan visoko-performansni procesor, keš memoriju velikog kapaciteta, i logiku koja je neophodna za korektnu spregu sa spoljnim uredjajima. Sa aspekta performansi, procesori koji se implementiraju na jedinstvenom čipu, a čija cena nije veća od dve do tri stotine dolara, mogu da se uporede sa super-računarima koji su pre dvadesetak godina zauzimali prostor od nekoliko prostorija a su koštali oko deset miliona dolara. Šta više, i *embedded* procesori koji se danas ugradjuju u mobilnim telefonima, telekomunikacionim-, medicinskim-, procesnim-, kao i u drugim uredjajima, su mnogo moćnije mašine od ranijih računara namenjenih za ugradnju u opremu ovakve vrste. No jedno su želje, a drugo je realnost. Naime, šanse da kao inženjer ikad projektujete savremeni procesor su zaista male. Obično ovakav zadatak se dodeljuje ekspertima. Na svetskom nivou, broj eksperata koji se bavi projektovanjem procesora je mali iz razloga što je broj kompanija koje proizvode mikroprocesore ograničen na stotinak. Osnovno pitanje koje se sada postavlja čitaocu je sledeće: Zbog čega treba znati i učiti detalje koji se odnose na dizajn procesora? Razlozi su brojni, ali ključni su sledeći:

1. Sa aspekta intelektualnih saznanja rad procesora je veoma interesantan – svaki stručnjak iz oblasti arhitekture računara bi trebalo da zna kako savremeni procesori rade. Zbog ovoga je neophodno studioznije proučiti "interni" rad procesora čije funkcionisanje, za veliki broj inženjera i eksperata iz oblasti računarske tehnike, i dalje predstavlja misterija. Dizajn procesora ima ugradjeno u sebi veliki broj principa i dobrih praktičnih rešenja. Uobičajena je praksa da se, sa jedne strane, u strukturu procesora zahteva ugradnja što je moguće više jednostavnih kola, a sa druge strane, tom hardverskom strukturom obavljaju što je moguće složeniji zadaci. Pri ovome treba imati u vidu da projektovanje ovako složenih struktura, za svakog inženjera, predstavlja veliki izazov. To znači da su saznanja iz ovog domena projektovanja veoma interesantna za svakog stručnjaka iz oblasti arhitekture računara.

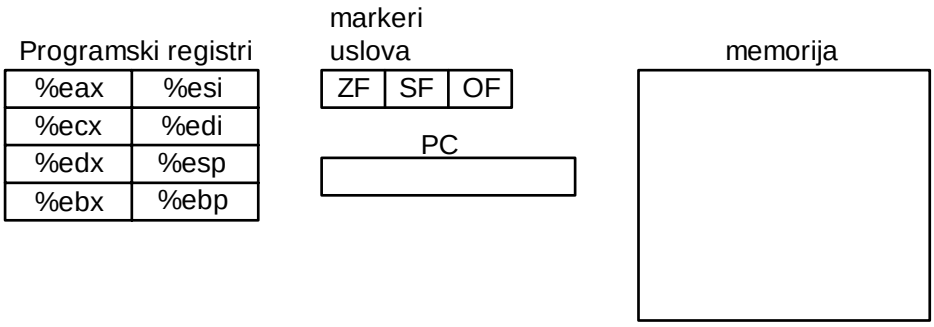
2 Način rada procesora nam pomaže da bolje razumemo kako ukupni računarski sistem radi – u ovom slučaju veoma je važno znati detalje koji se odnose na to kako treba ostvariti korektnu spregu procesor-memorija i procesor-spoljni svet.

3. I pored toga što je broj stručnjaka koji projektuju procesore mali, broj eksperata koji projektuju sisteme u kojima se ugradjuje procesor je veliki – više od 98% od svih procesora koji se danas proizvode ugradjuju se u *embedded* sisteme. Projektanti embedded sistema treba dobro da razumeju kako procesor radi, jer je za projektovanje i programiranje ovih sistema potrebno znati mnogo više detalja na nižem nivou apstrakcije nego što je to slučaj sa PC ili *desktop* mašinama.

4. Može da se desi da i Vi budete uključeni u rad koji se odnosi na projektovanje procesora – i pored toga što je broj kompanija koje proizvode mikroprocesore mali, projektanski timovi koji rade na razvoju novih procesora iz dana u dan postaju sve veći. Obično jedan tim čine oko 800-1000 eksperata koji rade na projektovanju različitih aspekata dizajna procesora. Uključenje u rad jednog ovakvog tima, s obzirom na globalizaciju sveta, postaje realnost.

1.1. ISA mašine Y86

Stanje mašine Y86 koje vidi programer (*programmer visible state*) prikazano je na slici 1. U konkretnom slučaju, za programera smatramo da je onaj ko piše programe na asemblerskom kodu, ili je to kompajler koji generiše kôd na mašinskom nivou.



Slika 1 Stanje mašine vidljivo od strane programera

Postoji osam programskih registara*:* `%eax, %ecx, %edx, %ebx, %esi, %edi, %esp i %ebp`. Svaki od njih je obima 32 bita i može da čuva podatak tipa reč. Registar `%esp` je pokazivač magacina i koristi se od strane instrukcija `Push, Pop, Call i Ret`. Ostali registri nemaju fiksno značenje ili vrednosti. Postoje tri jedno-bitna markera uslova: `ZF, SF i OF` koji pamte informaciju o efektu izvršenja najskorije aritmetičke ili logičke operacije. Programski brojač (`PC`) čuva adresu instrukcije koja se tekuće izvršava. Memorija, na koncepcijskom nivou, je uredjena kao veliko polje bajtova i koristi se za čuvanje programa i podataka. Programi mašine Y86 obraćaju se memorijskim lokacijama koristeći virtuelne adrese. Kombinacija hardvera i operativno sistemskog softvera prevodi ove adrese u stvarne fizičke adrese ukazujući na to gde se te vrednosti u memoriji stvarno čuvaju.
Na slici 2 prikazan je koncizan opis individualnih instrukcija ISA Y86. Nekih od detalja koje treba uočiti su sledeći:

a) Instrukcija `Movl` je podeljena na četiri različite instrukcije`: Irmovl, Rrmovl, Mrmovl, i Rmmovl`, koje eksplicitno ukazuju na formu izvorišta i odredišta. Izvršni operand može biti neposredni (*i*), registarski (*r*) ili memorijski (*m*), i predstavlja prvi karakter mnemonika. Odredišni operand može biti registatrski (r) ili memorijski (*m*) i predstavlja drugi karakter mnemonika.

b)Postoje četiri *integer* operacije koje su na slici 2 označene sa *OP1*, a to su `Addl, Subl, Andl i Xorl`. Ove operacije manipulišu nad operandima koji se čuvaju u registrima, a kao rezultat njihovog izvršenja postavljaju se markeri uslova `ZF, SF i OF`.

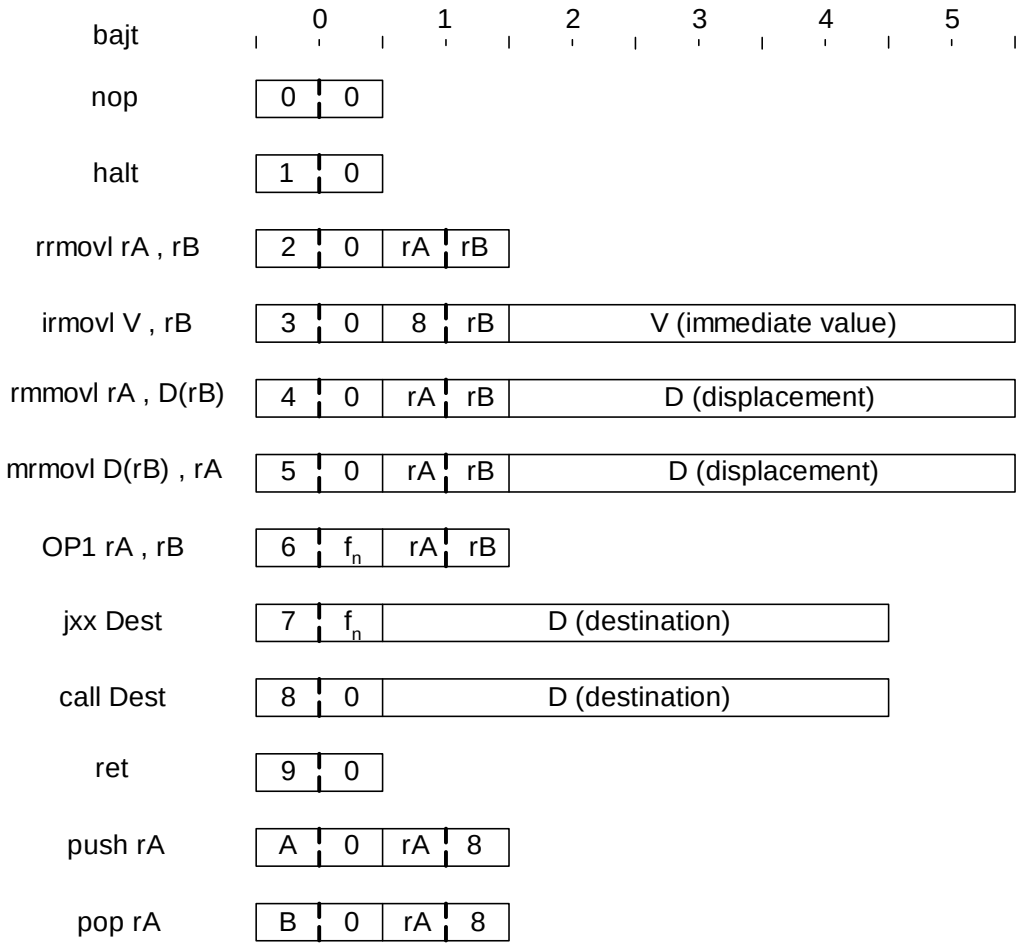
c)Postoji sedam instrukcija grananja, na slici 2 označenih sa `Jxx`, a to su: `Jmp,Jle,Jl,Je,Jne,Jge iJg`. Grananja se vrše u saglasnosti sa tipom grananja i postavljenosti markera uslova.

d)Instrukcija `Call` smešta povratnu adresu u magacin i skaće na odredišnu adresu. Instrukcijom *ret* obavlja se povratak iz potprograma.

e)Instrukcije `Push` i `Pop` koriste se za smeštanje i izbavljanje podataka u/iz magacina, respektivno.

f)Instrukcija `Halt` zaustavlja dalje izvršenje instrijcija, tj programa.

Kodiranje instrukcija na bajt nivou prikazano je na slici 2. Za kodiranje instrukcije, u zavisnosti od njenog tipa, neophodno je od jedan do šest bajtova. Svaka instrukcija ima inicijalni bajt kojim se identifikuje tip instrukcije.



Slika 2 Skup instrukcija mikroprocesora Y86

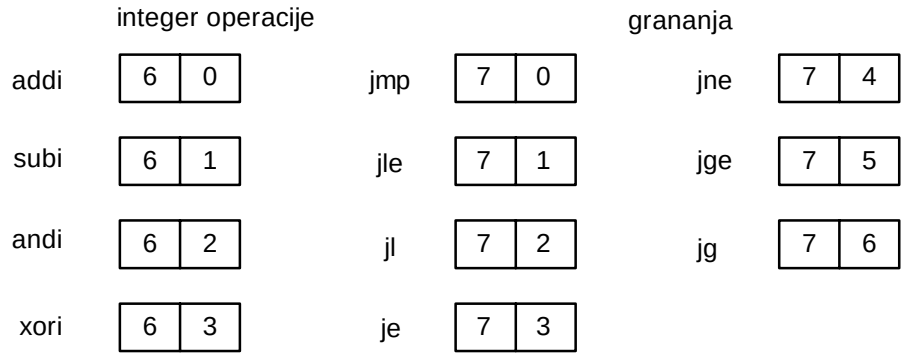
***Napomena**: obim instrukcije može biti od 1 do 6 bajtova; instrukcija sadrži jedan instrukcioni bajt, i u zavisnosti od tipa instrukcije jedan bajt za specifikaciju registra i četiri bajta kao konstanta; polje fn specificira neku od integer operacija ili odredjeni uslov grananja (Jxx).*

Instrukcioni bajt se deli na dva četvoro-bitna dela: *MS* deo, ili kodni deo, i *LS* deo, ili funkcijski deo. Kodne vrednosti se menjaju u opsegu od 0 do *B*. Funkcionalne vrednosti su od važnosti kada grupa srodnih instrukcija ima isti kod. Na slici 3 prikazana su specifična kodiranja koja se odnose na *integer* operacije, kao i instrukcije tipa `Branch`.

Kao što se vidi sa slike 4 svakom od osam programskih registara pridružen je registarski identifikator (*ID*) koji se nalazi u opsegu od 0 do 7. Programski registri pripadaju *RF* polju CPU-a.

Neke od instrukcija su obima jedan bajt, ali one druge koje specificiraju operande su većeg obima. Takodje, može da postoji bajt za specifikaciju registara, koji specificira jedan ili dva registara. Polja u okviru bajta za specifikaciju registara (vidi sliku 2) označena su sa `rA` i *`rB`*. Ova polja, u zavisnosti od tipa instrukcije, mogu da specificiraju registre koji se koriste za čuvanje izvorišnih i odredišnih podataka, kao i za specifikaciju baznog registra koji se koristi kod adresnih izračunavanja. Instrukcije koje ne koriste registarske operande, kakve su `Call` i *`Jxx`*, nemaju bajt za specifikaciju registara. Kod onih instrukcija koje koriste samo jedan

registarski operand, kakve su `Irmovl, Pushl i Popl`, drugo polje bajta za specifikaciju registara je postavljeno na vrednost 8. Neke od instrukcija zahtevaju dodatnu četvoro-bajtnu konstantnu reč. Ova reč se koristi kao neposredni podatak za instrukciju `Irmovl`, kao razmeštaj za adresne specifikatore instrukcije tipa `Rmmovl` ili `Mrmovl`, kao i kao odredište za instrukcije tipa `Jxx` i `Call.` Treba pri ovome uočiti da se odredišta za instrukcije `Jxx` i `Call` zadaju kao absolutne, a ne kao PC relativne adrese.



Slika 3 Funkcionalni kodovi skupa instrukcija procesora Y86

***Napomena**: Kodovi specificiraju pojedine integer operacije ili uslove grananja. Ove instrukcije su prikazane na slici 2 kao OP1 i Jxx.*

broj	ime registra
0	%eax
1	%ecx
2	%edx
3	%ebx
4	%esp
5	%ebp
6	%esi
7	%edi
8	nema registra

Slika 4 Programsko registarski identifikator mikroprocesora Y86

***Napomena**: Svakom od registara je dodeljen odgovarajući registarski identifkator koji se nalazi u opsegu od 0 do 7. ID = 8 ne specificira registar, tj. ukazuje na odsustvo registarskog operanda.*

Važna osobina skupa instrukcija je ta da kodiranje na bajt nivou mora da ima jedinstvenu interpretaciju. Pri ovome je neophodno je znati položaj prvog bajta u sekvenci, jer ako se to nezna nemoguće je pouzdano odrediti na koji način podeliti sekvencu na individualne instrukcije.

Primer 1

Za sledeću sekvencu instrukcija mikroprocesora Y86 odrediti kodiranje na nivou bajtova. Linija"`.pos 0x100`" ukazuje da je početna adresa objektnog koda `0x100.`

```
   .pos   0x100                # početak generisanja koda je na adresi 0x100
    Irmovl     $15,%ebx
    Rrmovl     %ebx,%ecx
 loop:
    Rmmovl     %ecx,-3(%ebx)
    Addl       %ebx,%ecx
    Jmp        loop
```

Odgovor

1	0x100:	`.pos 0x100`	#početna adresa generisanja koda je na adresi 0x100
2	0x100: 30830f000000	`irmovl $15,%ebx`	
3	0x106: 2031	`rrmovl %ebx,%ecx`	
4	0x108:	`loop`	
5	0x108: 4013fdffffff	`rmmovl %ecx,-3(%ebx)`	
6	0x10e: 6031	`addl %ebx,%ecx`	
7	0x110: 7008010000	`jmp loop`	

Primer 2

Za svaku od sledećih bajt sekvenci odrediti koju sekvencu instrukcija mikroprocesora Y86 ona kodira. Za slučaj da se javi neki nevažeći bajt u sekvenci, prikazati sekvencu instrukcija do tačke gde se javlja nevažeća vrednost. Za svaku sekvencu prikazati početnu adresu, kolonu, i nakon toga bajt sekvencu.

a) `0x100: 3083 fcffffff 40630008000010`

b) `0x200: a06880080200001030830a00000090`

c) `0x300: 50540700000000f0b018`

d) `0x400: 6113730004000010`

e) `0x500: 6362a080`

Odgovor

a) Operacije sa neposrednim vrednostima i adresnim razmeštajem

`0x100: 3083 fcffffff`	`Irmovl $-4,%ebx`
`0x106: 40630080000`	`Rmmovl %esi,0x800(%ebx)`

```
0x10c: 10                 | Halt
```

b) kod koji sadrži funkcijski `Call`

```
0x200: a068               | Pushl %esi
0x202: 8008020000         | Call proc
0x207: 10                 | Halt
0x208:                    | proc:
0x208: 30830a000000       | Irovl $10,%ebx
0x20e: 90                 | Ret
```

c) kod koji sadrži ilegalnu instrukciju, tj. bajt specifikator 0xf0

```
0x300: 505407000000         |  Mrmovl 7(%esp),%ebp
0x306: 00                   |  Nop
0x307: f0                   |  .byte 0xf0 # nevalidni opkod
0x308: b018                 |  Popl %ecx
```

d)kod koji sadrži instrukciju `jump`

```
0x400:                  |  loop:
0x400: 6113             |  Subl %ecx,%ebx
0x402: 7300040000       |  Je loop
0x407: 10               | Halt
```

d) kod koji sadrži nevažeći drugi bajt u instrukciji `pushl`

```
0x600: 6362     | Xorl %esi,%edx
0x502: a0       | .byte 0xa0  #opkod za pushl
0x503: 80       | .byte 0x80 #nevažeći registar bajt
```

1.2. Sekvencijalna implementacija mikroprocesora Y86

Analiziraćemo sada koje su komponente neophodne za implementaciju procesora Y86. Opisaćemo prvo procesor nazvan SEQ, tzv. skraćenica za sekvencijalni procesor. U svakom taktnom intervalu procesor SEQ obavlja sve neophodne korake potrebni za kompletno procesiranje instrukcije. U opštem slučaju, procesiranje instrukcije čini veći broj operacija. Organizaćemo operacije kao niz stepeni. Pri tome, sve instrukcije slede uniformnu sekvencu, nezavisno od toga da li se aktivnosti instrukcija medjusobno značajno razlikuju. Detalji procesiranja svake instrukcije zavise od tipa instrukcije koja se izvršava. Neformalni opis stepena i operacije koje instrukcije obavljaju su sledeće:

Fetch (pribavljanje) – čita iz memorije bajtove koji pripadaju instrukciji, koristeći, pri tome, PC kao memorijsku adresu. Iz instrukcionog bajta izdvajaju se dva četvorobitna polja nazvana `icode`, tzv kod instrukcije, i `ifun`, nazvano polje instrukcije. U zavisnosti od tipa instrukcije pribavlja se bajt koji specificira registre, kojim se biraju registarski operandi instrukcije *rA* i *rB*. Takodje, u zavisnosti od tipa instrukcije može da se pribavi i četvoro-bajtna konstantna reč *ValC*. Izračunava se *ValP* kao adresa instrukcije koja sledi tekuću instrukciju u instrukcionom redosledu. To znači da je vrednost *ValP* jednaka vrednosti PC-a plus obim pribavljene instrukcije.

Decode (dekodiranje) - čita do dva operanda iz RF polja. Pročitani sadržaji registara dodeljuju se poljima *ValA* i *ValB*. Obično se čitaju sadržaji registara specificirani poljima instrukcije *rA* i *rB*, ali kod nekih instrukcija čita se sadržaj registra *%esp*.

Execute (izvršenje) – u ovom stepenu ALU može da: a) izvrši operaciju specificiranu od strane instrukcije (u saglasnosti sa vrednošću `ifun`); b) izračuna efektivnu adresu memorijske lokacije kojoj se obraćamo, ili c) inkrementira ili dekrementira pokazivač magacina. Rezultantna vrednost se naziva *ValE*. Postavljaju se takodje i makreri uslova. U toku izvršenja instrukcije `jump` testiraju se markeri uslova kao i uslov grananja (definisan poljem `ifun`) da bi se procenilo da li će do grananja doći ili neće doći.

Memory (memorija) – memorijski stepen može upisivati podatke u memoriji, ili može čitati podatke iz memorije. Vrednost koja se iz memorije čita naziva se *ValM*.

Writeback (upis rezultata) ovaj stepen upisuje do dva rezultata u RF polje.

PC updates (ažuriranje PC-a) – PC se postavlja na adresu naredne instrukcije.

Aktivnosti pojedinih instrukcija

Aktivnosti koje se obavljaju izvršenjem pojedinih instrukcija od strane procesora SEQ objasnićemo preko procesiranja tih instrukcija.
Na slilci 5 prikazano je procesiranje koje obavljaju instrukcije tipa `OP1` (*integer* i logičke operacije), `Rrmovl` (kopiranje registar-u-registar), i `Irmovl` (kopiranje neposredne vrednosti u registar).
Na slici 6 prikazano je izvršenje instrukcija `rmmovl` i `mrmovl` na sekvencijalno implementiranoj mašini Y86. Ovim instrukcijama vrši se čitanje/upis iz/u memoriju.

stepen	OP1 rA , rB	rrmovl rA , rB	irmovl V , rB
Fetch	icode : ifun$\leftarrow M_1[PC]$ rA : rB $\leftarrow M_1[PC+1]$ Val P $\leftarrow$ PC + 2	icode : ifun$\leftarrow M_1[PC]$ rA : rB $\leftarrow M_1[PC+1]$	icode : ifun$\leftarrow M_1[PC]$ rA : rB $\leftarrow M_1[PC+1]$ Val C $\leftarrow M_4[PC+2]$ Val P $\leftarrow$ PC + 6
Decode	val A $\leftarrow$ R (rA) val B $\leftarrow$ R (rB)	val A $\leftarrow$ R (rA)	
Execute	val E$\leftarrow$val B OP val A	val E $\leftarrow$ 0 + val A	val E$\leftarrow$ 0 + val C
Memory			
Write-back	R(rB) $\leftarrow$ val E	R(rB) $\leftarrow$ val E	R(rB) $\leftarrow$ val E
PC update	PC $\leftarrow$ val P	PC $\leftarrow$ val P	PC $\leftarrow$ val P

Slika 5 Izračunavanje instrukcija *0P1, Rrmovl* i *Irmovl* kod sekvencijalne implementacije procesora Y86

***Napomena**: Notacija `icode:ifun` ukazuje na dve komponente instrukcionog bajta: polja `rA:rB` se odnose na dve komponente bajta za specifikaciju registara; Notacija `M₁[x]` ukazuje da se pristupa (radi čitanja ili upisa) jedinstvenom bajtu na memorijskoj lokaciji `x`, dok `M₄[x]` ukazuje da se pristupa podatku obima četiri bajta.*

stepen	rmmovl rA , D(rB)	mrmovl D(rB) , rA
Fetch	icode : ifun$\leftarrow M_1[PC]$ rA : rB $\leftarrow M_1[PC+1]$ val C $\leftarrow M_4[PC+2]$ val P $\leftarrow$ PC + 6	icode : ifun$\leftarrow M_1[PC]$ rA : rB $\leftarrow M_1[PC+1]$ val C $\leftarrow M_4[PC+2]$ val P $\leftarrow$ PC + 6
Decode	val A $\leftarrow$ R[rA] val B $\leftarrow$ R[rB]	val B $\leftarrow$ R[rB]
Execute	val E $\leftarrow$ val B + val C	val E $\leftarrow$ val B + val C
Memory	M_4[val E] $\leftarrow$ val A	val M $\leftarrow M_4$[val E]
Write-back		R[rA] $\leftarrow$ val M
PC update	PC $\leftarrow$ val P	PC $\leftarrow$ val P

Slika 6 Aktivnosti kod instrukcija *Rmmovl* i *Mrmovl* kod sekvencijalne implementacije mikroprocesora Y86

stepen	pushl rA	popl rA
Fetch	icode : ifun ← M_1[PC] rA : rB ← M_1[PC + 1] val P ← PC + 2	icode : ifun ← M_1[PC] rA : rB ← M_1[PC + 1] val P ← PC + 2
Decode	val A ← R[rA] val B ← R[%esp]	val A ← R[%esp] val B ← R[%esp]
Execute	val E ← val B + (- 4)	val E ← val B + 4
Memory	M_4[val E] ← val A	val M ← M_4[val A]
Write-back	R[%esp] ← val E	R[%esp] ← val E R[rA] ← val M
PC update	PC ← val P	PC ← val P

Slika 7 Izvršenje instrukcija *Pushl* i *Popl* na sekvencijalne implementacije procesoru Y86

***Napomena**: Obe instrukcije imaju za efekat prenos podataka ka/iz magacina*

stepen	jxx Dest	call Dest	ret
Fetch	icode : ifun ← M_1[PC] val C ← M_4[PC + 1] val P ← PC + 5	icode : ifun ← M_1[PC] val C ← M_4[PC + 1] val P ← PC + 5	icode : ifun ← M_1[PC] val P ← PC + 1
Decode		val B ← R[%esp]	val A ← R[%esp] val B ← R[%esp]
Execute	Bch ← Cond (CC , ifun)	val E ← val B + (- 4)	val E ← val B + 4
Memory		M_4[val E] ← val P	val M ← M_4[val A]
Write-back		R[%esp] ← val E	R[%esp] ← val E
PC update	PC ← Bch ? val C : val P	PC ← val C	PC ← val M

Slika 8 Izvršenje instrukcija *Jxx, Call* i *Ret* na sekvencijalne implementacije procesora Y86

***Napomena**:Sve tri instrukcije imaju za efekat prenos upravljanja*

Izvršenje instrukcija `Pushl` i `Popl` prikazano je na Slici 7, a način izvršenja instrukcija `Jxx`, `Call` i `Ret` na slici 8.

1.3. Hardverska struktura procesora SEQ

Izračunavanje koje je potrebno da se implementiraju sve instrukcije procesora Y86 se može organizovati kao niz od sledećih šest stepeni: *Fetch, Decode, Execute, Memory, Write-back*, i *PC-update*. Na slici 9 prikazan je apstraktni pogled hardverske strukture koja može da obavi ova izračunavanja.
Hardverske jedinice koje su pridružene razlličitim procesnim stepenima su:

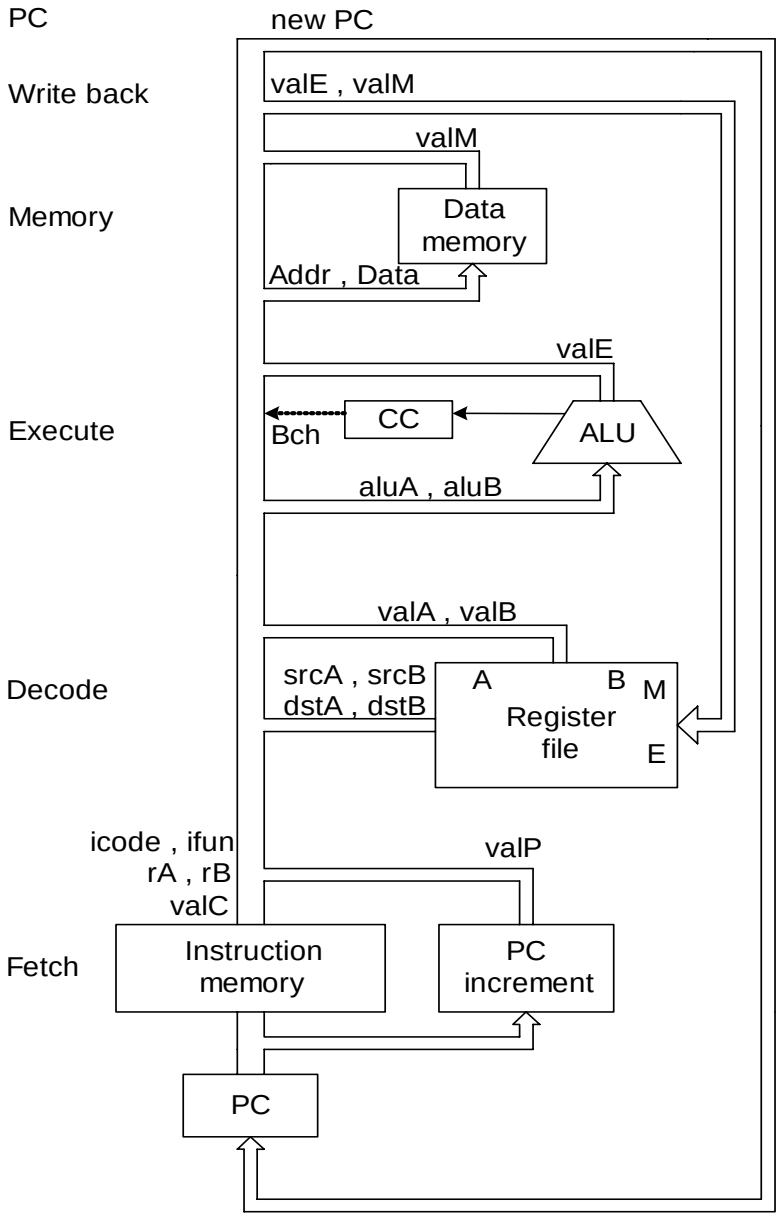
Fetch: koristeći sadržaj PC regisra kao adresu, iz instrukcione memorije se pribavljaju bajtovi instrukcije. Blok *PC-increment* izračuvana *valP*, inkrementiranu vrednost PC-a.

Decode: RF polje ima dva porta za čitanje, A i B, preko kojih se registarske vrednosti *valA* i *valB* čitaju istovremeno.

Execute: Ovaj stepen, u zavisnosti od tipa instrukcije, koristi ALU za različite potrebe. Za *integer* operacije ALU obavlja specificiranu operaciju. Kod ostalih instrukcija ALU se koristi kao sabirač da bi se inkrementirao ili dekrementirao pokazivač magacina, zatim za izračunavanje adrese, ili da se prenese jedan od ulaza na izlaz sabirajući ga sa 0. Registar CC čuva tri bita markera uslova. Nove vrednosti, za markere uslova, izračunavaju se od strane ALU-a. Kada se izvršava instrukcija `Jump`, signal grananja *Bch* se odredjuje na osnovu stanja markera uslova i tipa `Jump`-a.

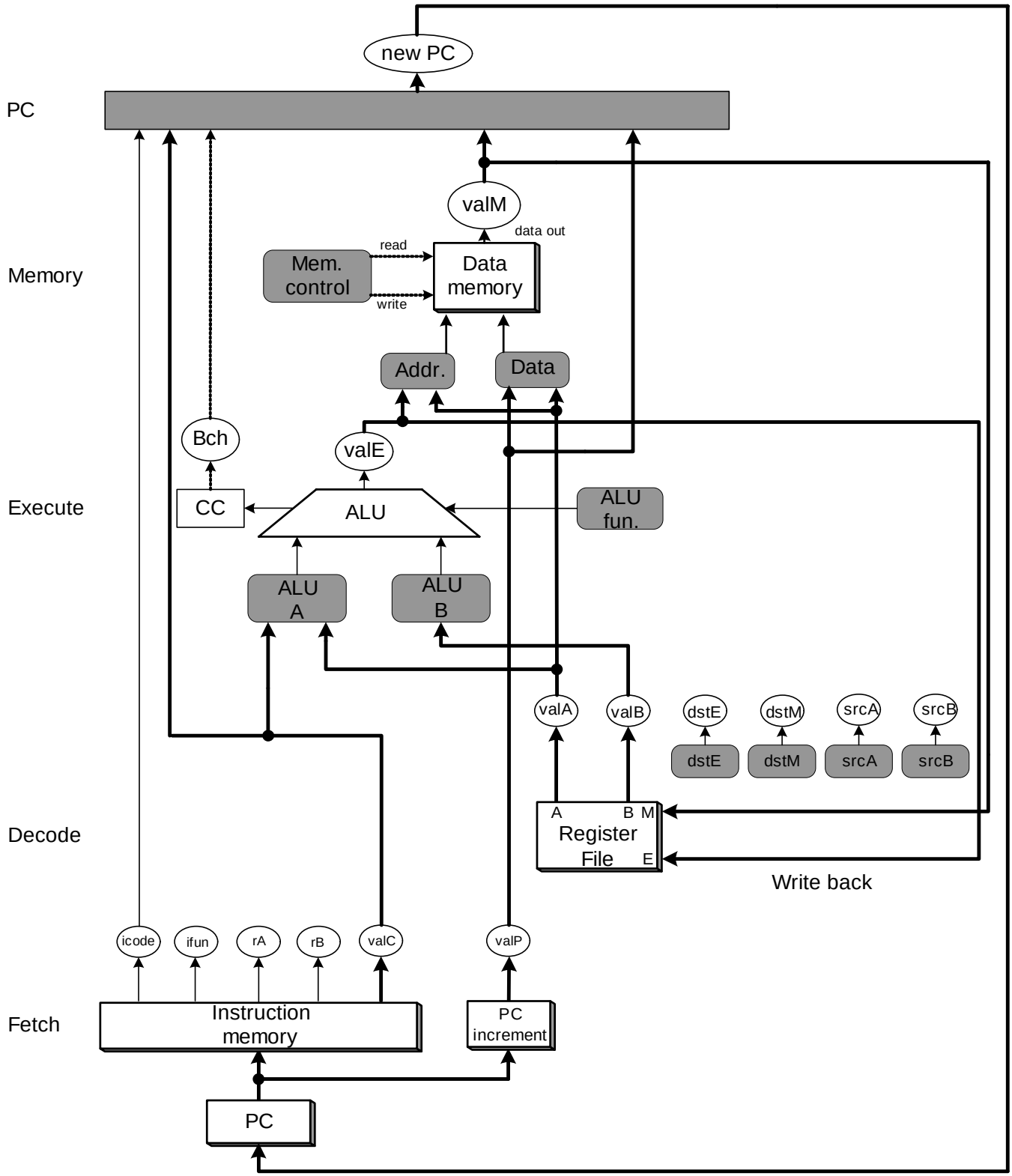
Memory: u toku izvrešenja instrukcije koje koriste memoriju podataka iz ove memorije se čitaju ili se u njoj upisuju podaci.

Write-back: RF polje ima dva porta za upis. *Port E* se koristi za upis vrednosti koje se izračunavaju od strane ALU-a, dok se *port M* koristi za upis vrednosti koje se čitaju iz memorije podataka



Slika 9 Apstraktni pogled sekvencijalne mašine SEQ

Na slici 10 prikazan je detaljniji pogled hardvera koji je neophodan za implementaciju mašine SEQ. Za prikaz slike korišćene su sledeće konvencije:



Slika 10 Hardverska struktura procesora SEQ

Prikaz na slici 10 koristi sledeću konvenciju:

Hardverske jedinice, kakve su Data Memory, ALU, CC, Register File, Instructionl Memory i *PC increment*, su date kao neosenčani kvadrati. Tretiraćemo ove jedinice kao crne-kutije i njihovu strukturu nećemo dalje izučavati.

- Blokovi upravljačke logike, kakvi su *New PC*, *Mem Control*, *ALU fun*, *ALU A*, *ALU B*, dstE, *dstM*, *srcA*, *srcB* su prikazani kao osenčani kvadrati. Ovi blokovi se koriste za selekciju jedan od izvora signala, ili za izračunavanje neke *Boole*-ove funkcije. Srukturu ovih blokova izučavaćemo detaljnije.
- Imena žica (veza) označena su slovima u belim krugovima.
- Veze po kojima se prenose podaci tipa reč (32-bitni podaci) predstavljene su debljim linijama. Ove veze se koriste za paralelni prenos podataka od jedne hardverske jedinice do druge.
- Jedno-bitne veze su prikazane tačkastim linijama.

1.3.1. Implementacija stepena procesora SEQ

U ovom delu poglavlja opisaćemo upravljačku logiku blokova koji su neophodni za implementaciju procesora SEQ. Vrednosti koje se odnose na kodiranje instrukcija, ID-ovi registara i ALU operacije prikazani su na slici 11.

Ime	Vrednost (Hex)	Značenje
INOP	**0**	Kod za `Nop` instrukciju
IHALT	**1**	Kod za `Halt` instrukciju
IRRMOVL	**2**	Kod za `Rrmovl` instrukciju
IIRMOVL	**3**	Kod za `Irmovl` instrukciju
IRMMOVL	**4**	Kod za `Rmmovl` instrukciju
IMRMOVL	**5**	Kod za `Mrmovl` instrukciju
IOPL	**6**	Kod za instrukcije operacija tipa *integer*
IJXX	**7**	Kod za `Jump` instrukcije
ICALL	**8**	Kod za `Call` instrukciju
IRET	**9**	Kod za `Ret` instrukciju
IPUSHL	**a**	Kod za `Pushl` instrukciju
IPOPL	**b**	Kod za `Popl` instrukciju
RESP	**6**	ID Registar za %esp
RNONE	**8**	Indicira nepristupanje registar fajlu
ALUADD	**0**	Funkcija za operaciju sabiranja

Slika 11 Vrednosti koje se ondose na način kodiranja opkoda instrukcije, ID-ova registara i ALU operacije

Analizirajući sliku 11 uočavamo da su uvedene dve nove instrukcije, `Halt` i `Nop`. Ove instrukcije, u principu, prolaze kroz stepen SEQ ne obavljajući nikakvo procesiranje sa izuzetkom što inkrementiraju sadržaj PC-a za 1. Nećemo prikazati detalje koje se odnose na instrukciju `Halt` koja u suštini stopira rad procesora. Jednostavno smatraćemo da se procesor zaustavlja kada pribavi opkod čija je vrednost 1.

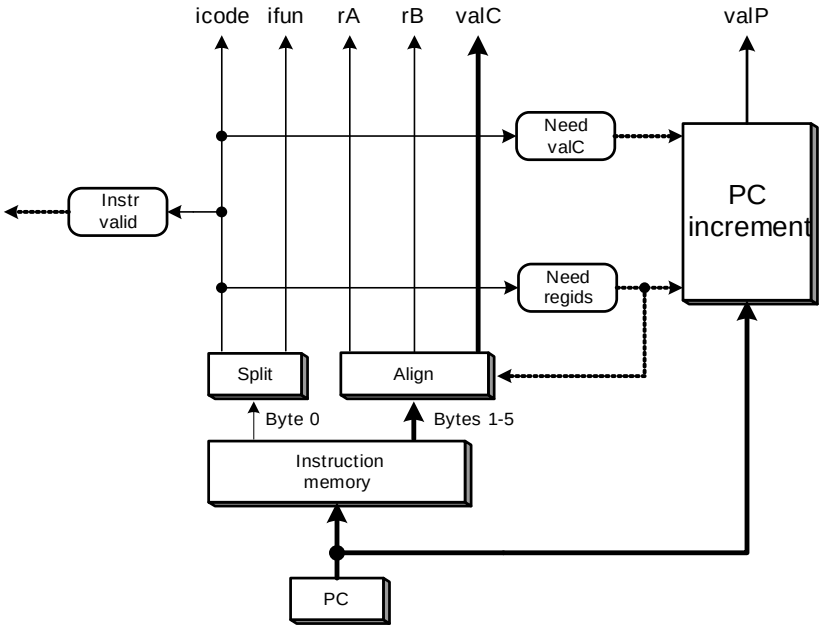
1.3.2. Stepen pribavljanja

Kao što se vidi na slici 12 ovaj stepen pribavlja instrukciju iz memorije-za- instrukcije. Istovremeno ova jedinica čita po šest bajtova koristeći PC adresu radi pristup prvom bajtu instrukcije (bajt 0). Ovaj bajt se interpretira kao instrukcioni-bajt i razdvaja se od strane bloka označen kao *Split* na dve četvorobitne vrednosti `icode` i ifun. U zavisnosti od vrednosti `icode` generišu se tri-jednobitna signala (prikazana tačkastim linijama) čija je funkcija sledeća:

. `intr_valid` : ovaj signal ukazuje da li bajt 0 pripada (odgovara) legalnom skupu instrukcija procesora Y86. To znači da se ovaj signal koristi za detekciju nelegalne instrukcije.

. `need_regids` : vrednost ovog signala ukazuje da li se ovom instrukcijom specificira bajt za specifikaciju-registra (recimo instrukcija `Nop` nema ovaj bajt, dok instrukcija `Add` ga ima)

. `need_valc` : ukazuje da li instrukcija sadrži konstantnu 32-bitnu reč (recimo `Rrmovl` nema, dok instrukcija `Irmovl` ima)



Slika 12 Stepen *Fetch* kod procesora SEQ

***Napomena**: Po šest bajtova se čitaju iz memorije-za-instrukcije pri čemu sadržaj PC-a pokazuje na početnu adresu. Na osnovu instrukcionih bajtova generišu se različita polja u okviru instrukcije.*

PC-increment blok generiše signal `valP`. Izlaz bloka *need-regids* odredjuje da li instrukcija sadrži bajt kojim se specificiraju registri. HCL opis jedno-bitnog signala na izlazu *nedd-regids* je oblika:

```
Bool need_regids =

        icode in {IRRMOVL,IOPL,IPUSHL,IPOPL,IIRMOVL,IRMMOVL,IMRMOVL} ;
```

HCL kod za jedno-bitni signal koji se generiše na izlazu bloka *need-Valc* je oblika

`Bool need_vals = icode in { IIRMOVL,IRMMOVL,IMRMOVL,IJXX,ICALL} ;`

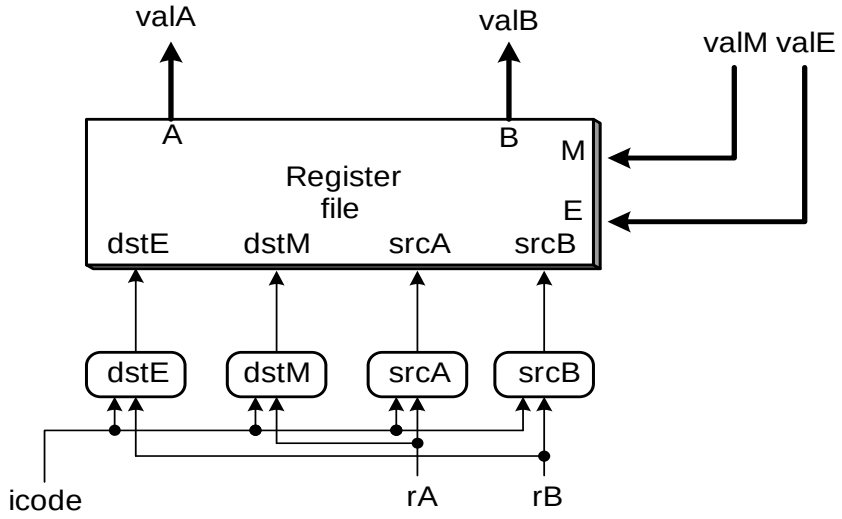
Kao što se vidi sa slike 12 ostala pet bajta koja se čitaju iz memorije-za- instrukcije koriste se za kodiranje bajta-za-specifikaciju-registra i četvoro-bajtne-konstante. Ovi bajtovi se procesiraju od strane bloka `Align`. Kada je upravljački signal *neded_regids* = 1 tada se bajt 1 deli na polja za specifikaciju registara `rA` i `rB`. Inače su ova dva polja postavljena na vrednost 8 (`RNONE`), što znači da se ovom instrukcijom ne specificiraju registri. Ako se nekom instrukcijom specificira samo jedan registar (kao na primer kod `Pushl rA` ili `Popl rA`), tada drugo polje `rB`, bajta za specifikaciju-registra, se postavlja na vrednost (`RNONE`).

Blok `Align` takodje generiše 32-bitnu konstantnu reč `valc`. To mogu biti bajtovi od 1 do 4 (kod instrukcije (`Jxx Dest` i `Call Dest`) ili bajtovi 2 do 5 (kod instrukcija `Irmovl, Rmmovl, i Mrmovl`).. Upravljanje radom bloka `Align` obavlja se od strane kontrolnih signala koji se generišu na izlazu bloka *need-regids*. Hardverski blok *PC-incrementer* na osnovu tekuće vrednosti PC-a i stanja dvaju upravljačkih signala `need-regids` i `need-valc` generiše signal `valP`. Ako je tekuća vrednost PC-a jednaka *p*, vrednost `need-regids` iznosi *r*, a `need-valC` je *i*, tada blok PC-incrementer generiše vrednost:

`p +r + 4i`

1.3.3. Stepeni za dekodiranje i upis-rezultata

Na slici 13 prikazana je logika koja implementira stepene za dekodiranje *Decode* i upis-rezultata *Write-Back.*



Slika 13 Stepeni *Decode* i *Write-Back* mikroprocesora Y86

***Napomena**: Polja instrukcije se dekodiraju i generišu četiri RF adresnih identifikatora (dva radi čitanja, a dva radi upisa). Vrednosti koje se iz RF polja čitaju su `valA` i `valB`, dok vrednosti koje se u RF polje upisuju su `valM` i `valE`.*

RF polje čine četiri porta. Polje podržava do dva simultana čitanja(portovi `A` i `B`) i dva simultana upisa (portovi `E` i `M`). Adresni ulazi portova za čitanje su `srcA` i `srcB`, a portova za upis su `dstE` i `dstM`. Na ulaz portova za upis podatakal `M` i `E` dovode se 32-bitni podaci `valM` i `valE`, respektivno. Ako se na ulaz bilo kog adresnog porta dovede adresni identifikator 8 (`RNONE`) tada se ne pristupa nijednom od registara RF polja.

HCL opisi adresnih ulaza `srcA` i i `srcB` su oblika

```
Int srcA = [
      icode in {IRRMOVL,IRMMOVL,IMRMOVL,IOPL,IPUSHL} ; rA;
     icode in {IPOPL, IRET};RESP;
     1 : RNONE  # registar nije potreban
];
```

Napomena: `RESP` se odnosi na ID registra `%esp`.

```
Int srcB = [
icode in {IRRMOVL,IRMMOVL,IMRMOVL,IOPL}: rB;
icode in {IPUSHL,IPOPL,ICALL IRET} : RESP;
1 : RNONE;
];
```

Polje `dstE` se koristi za identifikaciju registra kome se preko `porta E` pristupa radi upisa, a vrednost koja se upisuje je `valE`.

HCL opis dstE-a je oblika

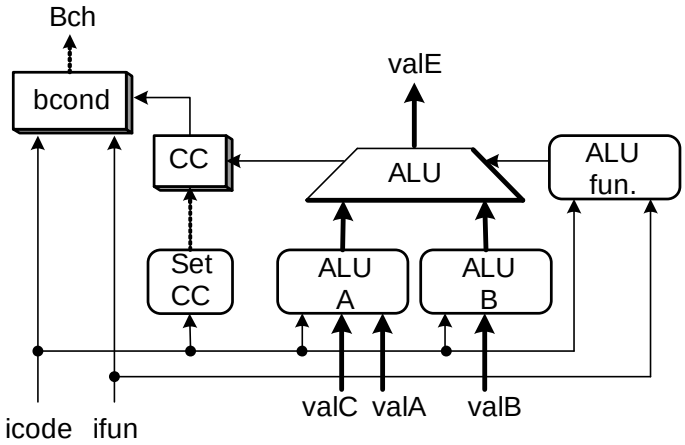
```
Int dstE  = [
 icode in (IRRMOVL,IIRMOVL,IOPL); rB;
 icode in (IPUSHL,IPOPL,ICALL,IRET); RESP;
 1 : RNONE       # registar nije potreban
];
```

Sa druge strane polje `dstM` ukazuje na adresu odredišnog registra u kome se preko `porta M` pristupa radi upisa, a vrednost koja se upisuje je `dstM`. HCL opis `dstM`-a je oblika:

```
Int dstM = [
 icode in (IRMMOVL,IOPL); rA;
 1 : RNONE       # registar nije potreban
];
```

1.3.4. Stepen Execute

Stepen *Execute* sadrži ALU. Ova jedinica u zavisnosti od vrednosti kontrolnog ulaza `alufun` obavlja nad ulazima `aluA` i `aluB` operaciju tipa `ADD`, SUBTRACT, `AND`, `OR`, ili `EXCLUSIVE-OR`. Na slici 14 prikazana je stuktura stepena *Execute*.



Slika 14 Struktura stepena *Execute*

***Napomena**: ALU obavlja aritmetičke ili logičke operacije nad podacima tipa integer. Registar `CCR` se postavlja u saglasnosti sa rezultatom `ValE` koji se generiše na izlazu ALU-a. Markeri uslova se od strane uslovnih instrukcija grananja testiraju sa ciljem da se proveri da li će do grananja doći ili ne.*

Pri operand kod ALU operacije je `aluB` a drugi `aluA`, što znači da se kod instrukcije subl vrednost `valA` oduzima od vrednosti `valB`. Kao što se vidi sa slike 14 vrednost za `valA` može, u zavisnosti od tipa instrukcije, biti vrednost `valA`, `valC`, -4 ili +4.
Ponašanje upravljačkog bloka koji generiše vrednost `aluA` se može opisati sledećim HCL kodom.

```
Int aluA = [
  icode in (IRRMOVL,IOPL); valA;
  icode in (IIRMOVL,IRMMOVL,IMRMOVL); valC;
  icode in (ICALL,IPUSHL): -4;
  icode in (IREST,IPOPL): +4;
  # ostale instrukcije ne koriste ALU
  ];
```

HCL opis signala `aluB` sledećeg je oblika:

```
Int aluB = [
 icode in (IRMMOVL,IMRMOVL,IOPL,ICALL,IPUSHL,IRET,IPOPL); valB;
 icode in (IRRMOVL,IIRMOVL):0;
 # ostale instrukcije ne koriste ALU
  ];
```

Koju će operaciju obaviti ALU definiše izlaz bloka `alufun`. HCL opis ALU kontrole je slecdećeg oblika

```
Int alufun =  [
   icode ==IOPL:ifun
   1:ALUADD;
  ];
```

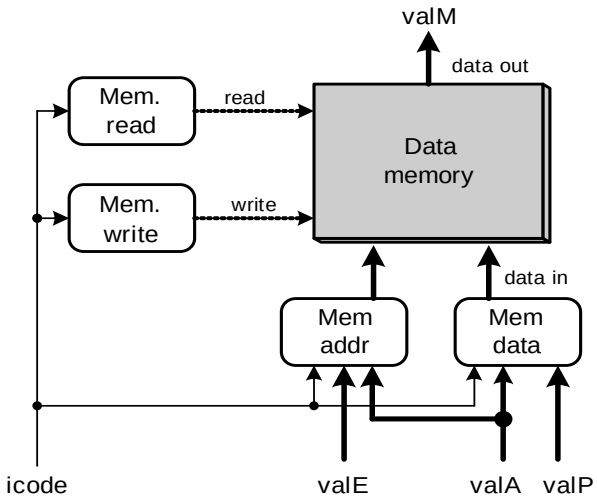
Sastavni deo stepena Execute je i registar CCR. U konkretnom slučaju na izlazu ALU-a generišu se tri markera uslova: *zero*, *sign* i *overflow*. Markeri ulsova se mogu unapred postaviti generisanjem signala `set-cc` koji kontroliše da li će se registar CCR ažurirati illi ne. Odgovarajući HCL kod signala `set_cc` je oblika

```
Bool set_cc = icode in (IOPL);
```

Hardverski blok `bcond` odredjuje da li će, kod uslovne instrukcije grananja, do grananja doći ili ne.

1.3.5. Memorijski stepen

Memorijski stepen ima zadatak da vrši upis i čitanje programskih podataka. Kao što je prikazano na slici 15 blok `mem_adr` generiše adresu pristupa memoriji za podatke, blok `mem_data` u toku operacije upis postavlja važeće podatke na ulazu memorije-podataka (`data in`), dok blokovi `mem_read` i `mem_write` generišu signal čitanja (`read`) i upisa (`write`), respektivno. Kada se obavlja operacija čitanja memorije na izlazu (`data_out`) se generiše vrednost valM.



Slika 15 Memorijski stepen sekvencijalnog procesora SEQ tipa Y86.

***Napomena**: iz/u memorije podataka se može čitati ili upisivati. Vrednost koja se čita iz memorije je `valM`. Adresa za upis u memoriji za podatke može biti `valA` ili `valE`, respektivno.*

HCL opis adresnog ulaza `mem_adr` oblika je

```
int mem_adr = [
 icode in "IRRMOVL,IPUSHL,ICALL,IMRMOVL":valE;
 icode in "IPOPL,IRET":valA;
];
```

HCL kod signala `mem_data` ima formu

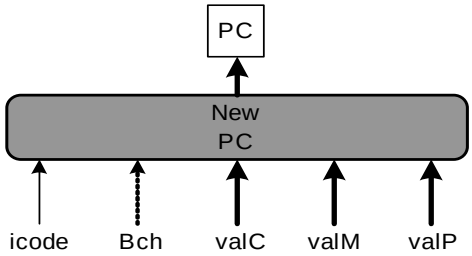
```
int mem_data = [
  icode in "IRMMOVL,IPUSHL":valA;   # vrednost iz registra
  icode == ICALL :valP;             #povratna vrednost za PC
                                    # po definiciji ne upisuje se ništa
];
```

HCL kodovi za upravljačke signale `mem_read`, i `mem_write` su oblika

```
bool mem_read = icode in "IMRMOVL,IPOPL,IRET";
bool mem_write = icode in "IRMMOVL,IPUSHL,ICALL";
```

1.3.6. Stepen za ažuriranje sadržaja PC-a

Struktura ovog stepena prikazana je na slici 16



Slika 16 Stepen za ažuriranje sadržaja PC-a kod sekvencijalnog procesora SEQ tipa

***Napomena**: Naredna vrednost koja se bira izmedju signala `valC`, `valM`, i `valP` zavisi od `icode` i markera `Bch`.*

Selekcija vrednosti `new_pc` kojom se ažurira stanje PC-a opisano je sledećim HCL kodom.

```
Int new_pc = [
# call – koristi konstantu koja je sastavni deo instrukcije
icode == ICALL:valC;
# do grananja je došlo, koristi se konstanta definisana instrukcijom
icode == IJXX && Bch:valC;
#završetak instrukcije RET, pribavlja se vrednost iz magacina
icode == IRET:valM;
# po  definiciji inkrementira se sadržaj PC-a
1:valP;
];
```

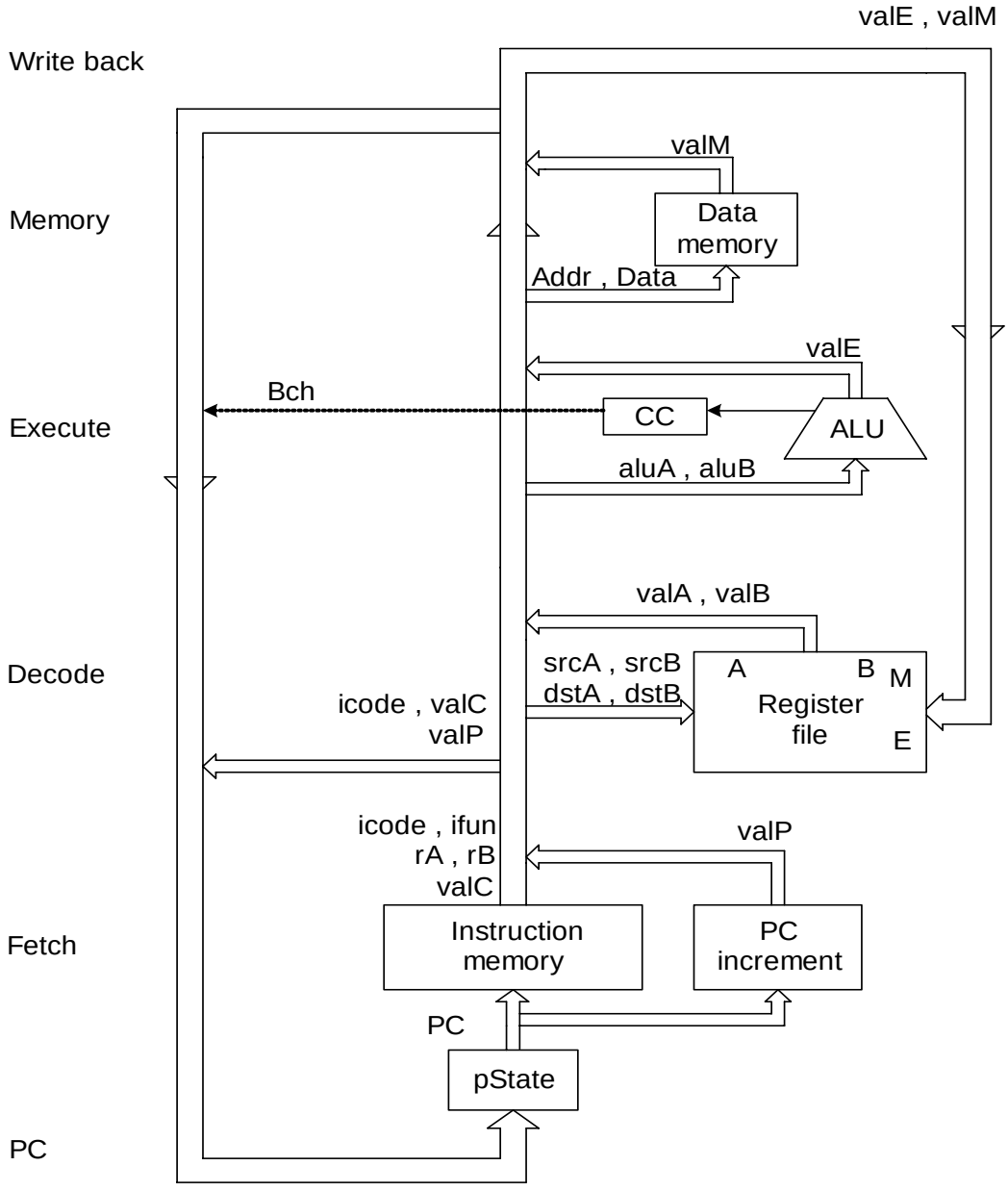
1.3.7. Razmatranja u vezi sekvencijalnog procesora SEQ tipa Y86

Glavni nedostatak procesora SEQ je taj što je ova mašina suviše spora. Taktna perioda mora biti dovoljno duga i da obezbedi propagaciju signala kroz sve stepene u toku jedne periode. Kao ilustraciju posmatrajmo procesiranje instrukcije `Ret`. Na početku taktnog intervala počinje se sa ažuriranjem vrednosti PC-a, zatim se čita instrukcija iz memorije za instrukcije, nakon toga se čita SP iz RF polja, da bi posle toga ALU dekrementirao SP, i na kraju adresa podatka se mora pročitati iz memorije sa ciljem da se odredi naredna vrednost PC-a. Sve ove aktivnosti se moraju završiti do kraja taktnog intervala.
Ovakav stil implementacije nije dobar za korišćenje od strane hardverskih jedinica jer je u datom trenutku aktivan samo deo hardvera. Sagledajmo sada načine kako se mogu poboljšati performanse uvodjenjem protočne obrade.

1.4. SEQ+: Procesor sa preuredjenim stepenima

Pre nego što se upustimo u analizu protočnog dizajna, kao medjukorak u projektovanju procesora uvešćemo jednu modifikaciju. Modifikacija se sastoji u tome što ćemo obradu PC stepena sa kraja prebaciti na početak taktnog intervala. Ovakav dizajn nazvaćemo SEQ+.

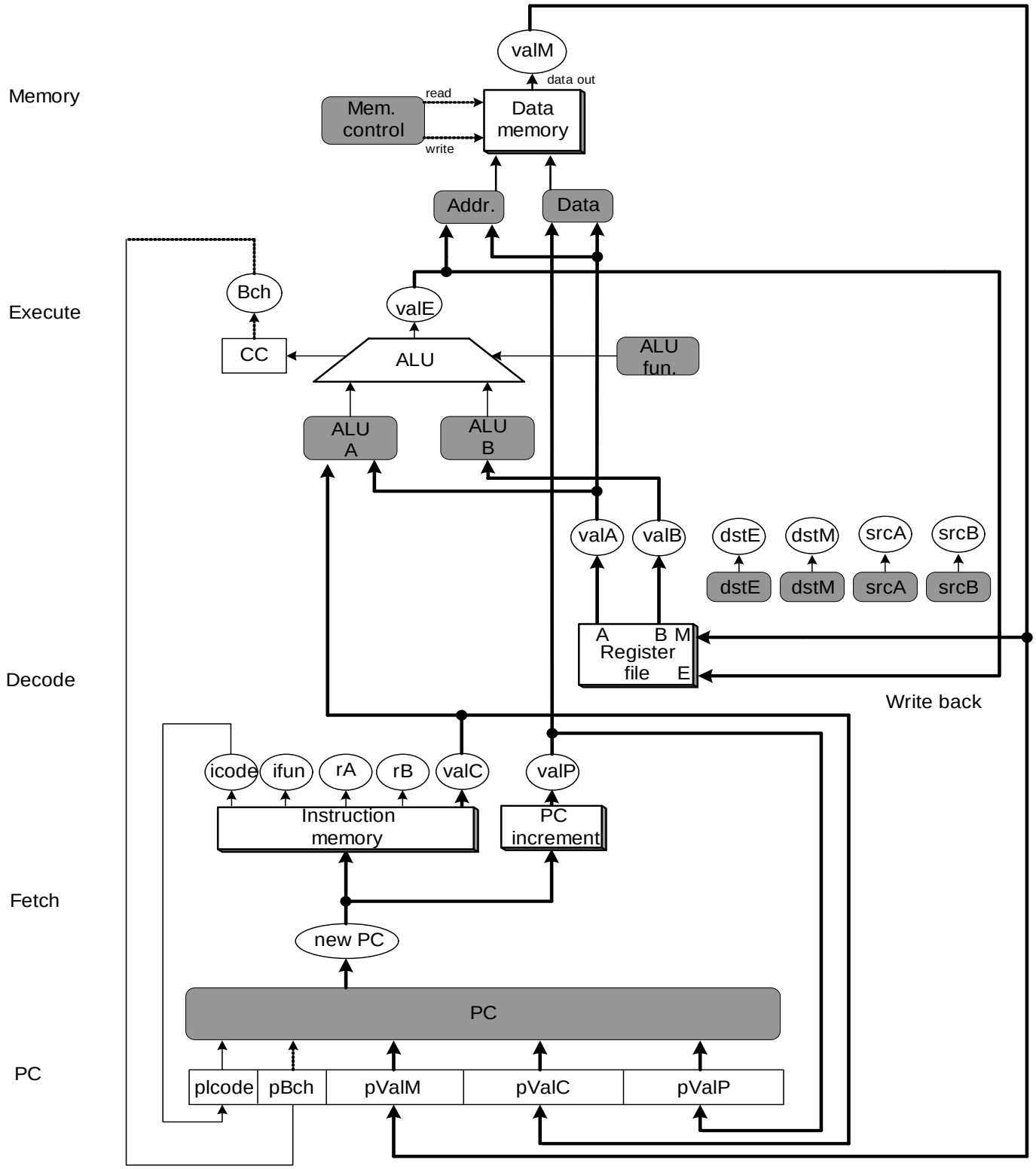
Na slici 17 prikazano je kako je izvršeno premeštanje PC-a tako da je njegova logika aktivna na početku taktnog intervala. Izračunata vrednost PC-a koja se odnosi na adresu tekuće instrukcije predaje se stepenu FETCH. Dalje procesiranje instrukcije se odvija na isti način kao i kod procesora SEQ. Kombinaciona logika na kraju taktnog intervala generiše sve neophodne signale za odredjivanje novog sadržaja PC-a. Ove vrednosti se pamte u skup registara nazvan pState. Zadatak PC stepena se sastoji u selekciji PC vrednosti za tekuću instrukciju, a ne u izračunavanju i ažuriranju PC-a za narednu instrukciju.



Slika 17 Abstraktni pogled na procesor SEQ+

***Napomena**: Kod ove verzije selekcija programskog brojača za tekuću instrukciju se vrši na početku ciklusa na osnovu informacije koja se u prethodnom ciklusu pamti u registru `pState`. Ovakva struktura omogućava lakšu implementaciju protočne obrade*

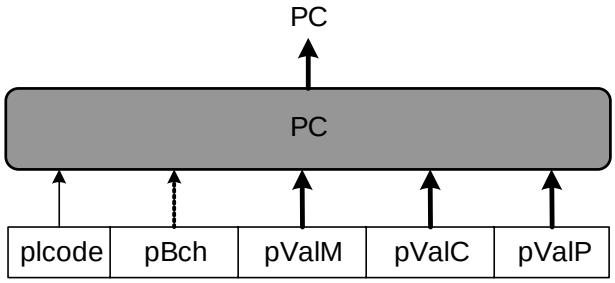
Na slici 18 prikazan je detaljniji pogled na hardver procesora SEQ+. Kao što se može videti on sadrži iste hardverske jedinice i upravljačke blokove koji su bili prisutni kod procesora SEQ (slika 10), ali je logika PC-a premeštena na dnu. Naime, rezultati prethodne instrukcije se čuvaju u registrima prikazanih na dnu slike 18, pri čemu ispred ovih vrednosti stoji prefiks slovo `p` koje asocira da su to prethodne (*previous*) vrednosti.



Slika 18 Struktura hardvera procesora SEQ+

Jedinstvena izmena u upravljačkoj logici se sastoji u redefinisanju PC izračunavanja tako da on koristi vrednosti iz prethodnog stanja. Princip odredjivanja nove vrednosti PC-a kod procesora SEQ i SEQ+ je prikazan na slici 19.

Jedina razlika izmedju ova dva bloka se sastoji u premeštanju registara, koji čuvaju stanje procesora, tj. kod SEQ procesora blokovi koji su bili locirani nakon PC izračunavanja premešteni su kod SEQ+ da budu pre PC izračunavanja. Izmene u tajmiranju kod procesora SEQ+ imaju efekat samo na promenu prezentacije stanja, ali ne i na logičko ponašanje procesora SEQ+.



Slika 19 Princip odredjivanja nove vrednosti PC-a kod procesora SEQ i SEQ+

HCL opis kojim se izračunava PC vrednost kod procesora SEQ+ je oblika

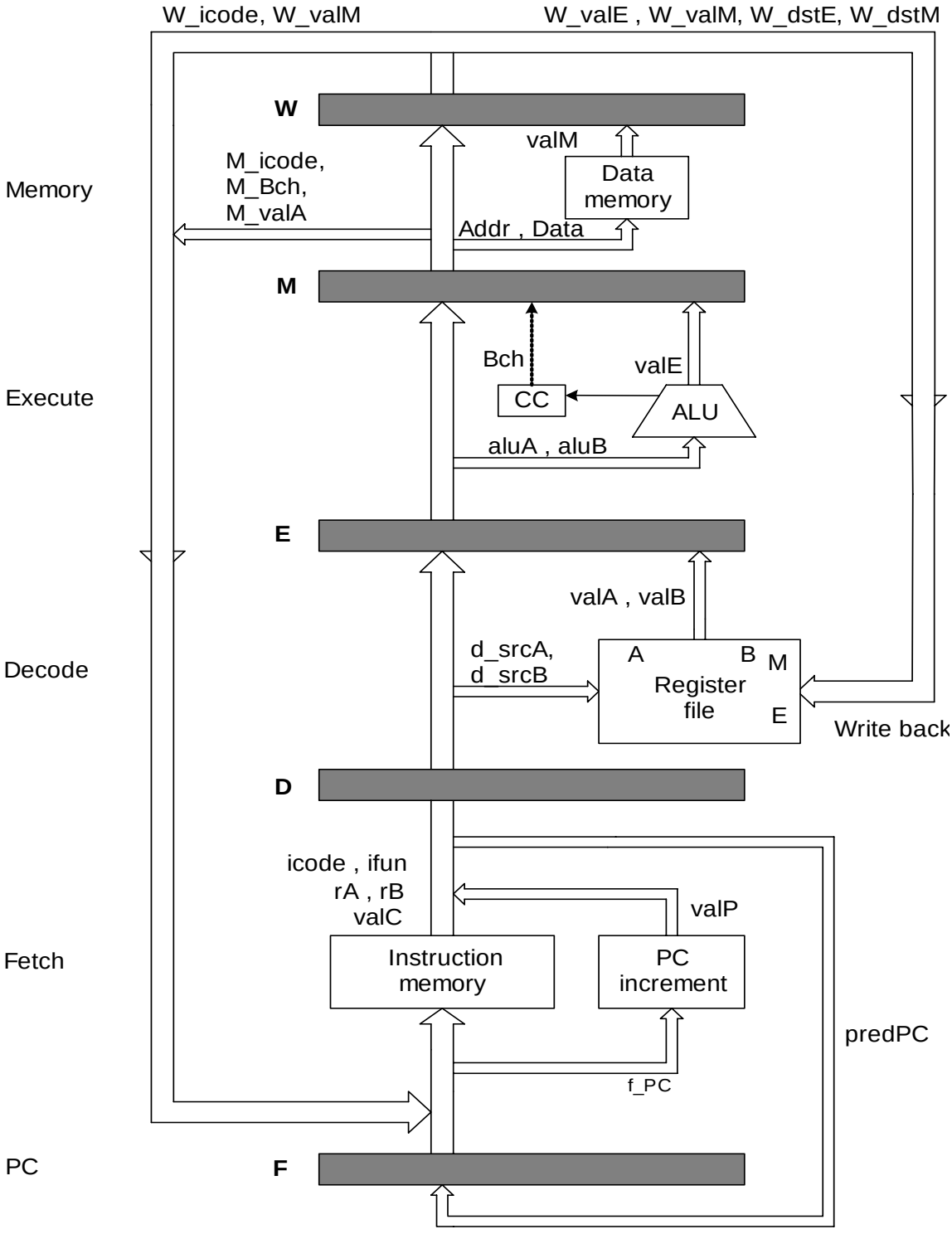
```
int pc = [
# call – koristi se 32-bitna konstanta specificirana instrukcijom
pIcode == ICALL:pValC;
#do grananja dolazi – koristi se 32-bitna konstana specificirana instrukcijom
 pIcode == IJXX && pBch:pValC;
# završetak instrukcije RET - koristi se vrednost koja se pribavlja iz magacina
 pIcode == IRET:pValM;
# po definiciji - koristi se inkrementiranje sadržaja PC-a
1:pValP;
];
```

1.5. Protočna imiplementacija procesora Y86

Početna osnova za protočnu implementaciju procesora Y86 biće procesor SEQ+. Modifikacija se sastoji u: a) ubacivanju protočnih registara izmedju stepeni; i b) preuredjenju nekih od signala. Novi procesor nazvaćemo PIPE-, gde simbol `"-"` označava da ovaj procesor ima lošije performanse u odnosu na ciljni. Abstraktna struktura procesora PIPE- je prikazana na slici 20. Svaki od protočnih registara čuva veći broj bajtova i reči. Treba pri ovome naglasiti da PIPE- koristi takoreći iste hardverske jedinice kao i SEQ+.

Protočni registri su označeni kao:

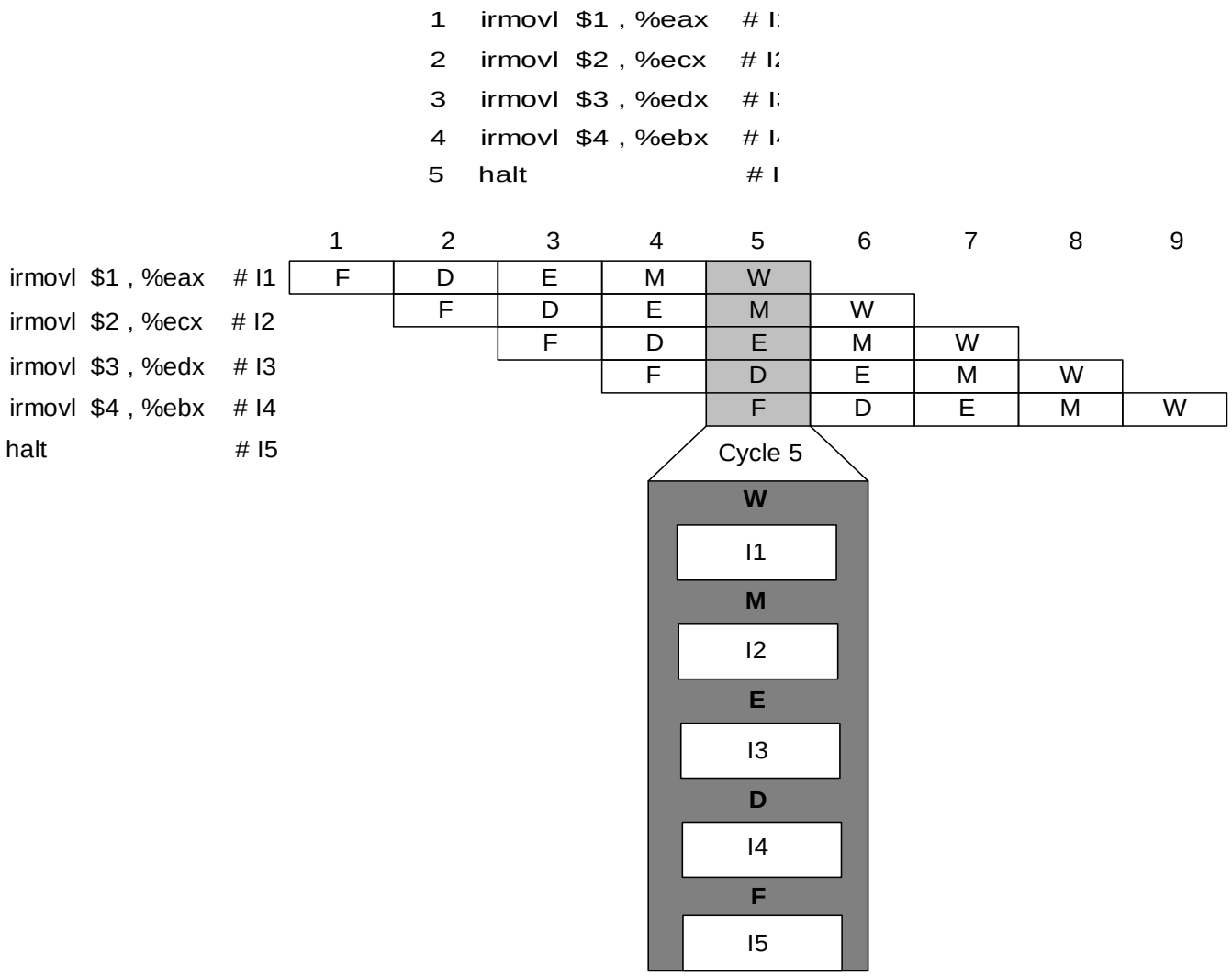
F-čuva prediktovanu vrednost PC-a
`D`-čuva informaciju o najskorije pribavljenoj instrukciji radi procesiranja od strane stepena *Decode*
`E`-pamti informaciju o najskorije dekodiranoj instrukciji i vrednosti koje se čitaju iz RF polja radi procesiranja od strane stepena *Execute*.
`M`-čuva rezultate najskorije izvršene instrukcije radi procesiranja od strane stepena *Memory*. Takodje čuva informaciju o uslovima grananja i ciljnoj adresi grananja radiprocesiranja od strane uslovnih instrukcija grananja.
`W`-predaje izračunate rezultate RF polju radi upisivanja u odredišne registre, kao i povratnu adresu logici za selekciju vrednosti kojom se ažurira stanje PC-a kada se izvršava instrukcija `Ret`.



Slika 20 Abstraktni pogled na procesor PIPE-

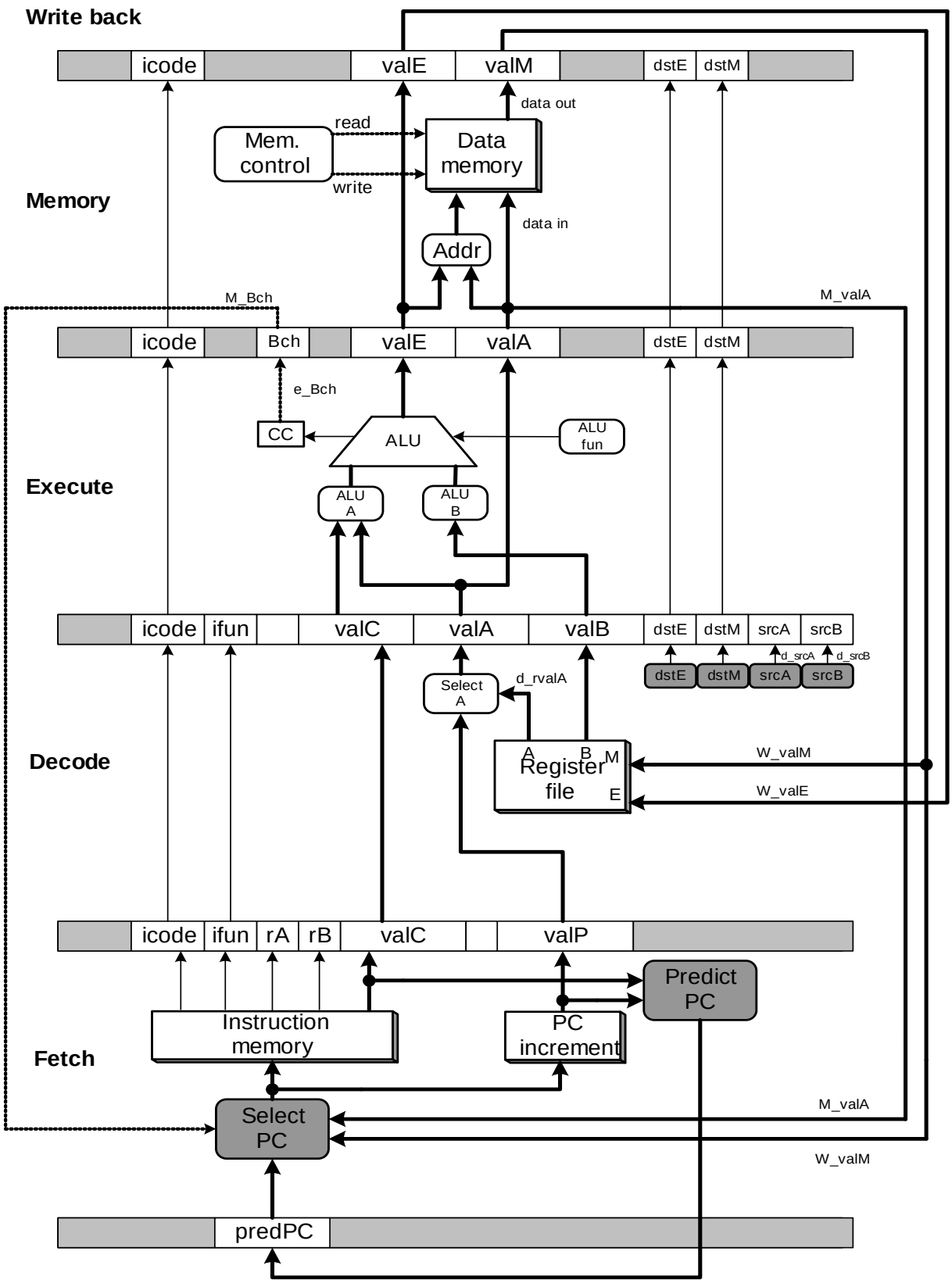
***Napomena**: Ubacivanjem protočnih registara u procesor SEQ+ mi kreiramo peto-stepeni protočni sistem*

Na slici21 prikazan je tok izvršenja jedne proizvoljne sekvence instrukcija kroz PIPE-.



Slika 21 Primer toka izvršenja jedne proizvoljne sekvence instrukcije kroz PIPE-

Prošireni pogled na ciklus 5 sa slike 21 prikazuje protočne stepene sa stepenom `Fetch` na dnu, a `Write-back` na vrhu dijagrama protočno organizovanog hardvera sa slike 20 i slike 22. Ako se pogleda redosled instrukcija u protočnim stepenima (slika 21 b)) videćemo da se one pojavljuju u istom redosledu kao što je to navedeno u listingu programa (slika 21 a)).



Slika 22 Hardverska struktura procesora PIPE-

Na slici 22 prikazan je detaljniji pogled na hardversku strukturu procesora PIPE-. Kao što se vidi svaki protočni registar sadrži veći broj polja. Polja odgovaraju signalima koji su pridruženi različitim instrukcijama koje se obradjuju od strane protočnog sistema.
Ukazaćemo u daljem tekstu na osnovne razlike koje postoje izmedju procesora SEQ+ i PIPE-.

1.5.1. Preuredjenje i preimenovanje signala

Kao što smo već ukazali procesor SEQ+ procesira po jednu instrukciju po taktnom intervalu. Shodno tome postoje jedinstvene vrednosti signala kakve su `valC`, `srcA`, i `valE`. Sa druge strane, kod protočnog dizajna, u toku jednog taktnog intervala, postoji veći broj verzija ovih vrednosti koje prate tok izvršenja različitih instrukcija kroz protočni sistem. Tako na primer, detaljnom analizom strukture procesora PIPE- (slika 22) uočava se da postoje četiri polja označena sa `"icode"` koja čuvaju `icode` signale za četiri različite instrukcije. Da bi se izbegle konfuzije u njihovoj interpretaciji i korišćenju usvojićemo šemu imenovanja signala kod koje zapamćeni signal u protočnom registru se identifikuje na jedinstveni način pomoću prefiksa koji stoji ispred njegovog imena. Tako na primer, četiri kopije `icode` signala se imenuju kao `D_icode`, `E_icode`, `M_icode`, i `W_icode`. Slični primeri su `d_src`, `e_Bch` i dr.

Dekoderski stepeni procesora SEQ+ i PIPE- generišu signale dstE i dstM koji pokazuju na odredišne registre za vrednosti valE i valM, respektivno. Kod procesora SEQ+ ovi signali se direktno povezuju na adresne ulaze portova za upis u RF polju. Kod procesora PIPE- ovi signali se prenose kroz protočni sistem preko stepena *Execute* i Memory, i dovode se RF polju tek nakon stepena *Write-back*.

Opšti princip rada protočnog sistema je sledeći: Sva ifnormacija o pojedinoj instrukciji se čuva u okviru jedinstvenog protočnog stepena i kao takva se prenosi od stepena do stepena. Blok označen kao Select_A je implementiran kod PIPE-, a ne egzistira kod SEQ+. Ovaj blok generiše verednost `valA` koja se pamti u protočni registar `E` selekcijom vrednosti `valP` koja se dovodi od protočnog registra `D` ili vrednosti koja se čita sa `porta A` iz RF polja. Ovaj blok smanjuje broj stanja koji se direktno prosledjuje izmedju protočnih registara `E` i `M`. Od svih instrukcija samo instrukcija `Call` zahteva vrednost `valP` u stepenu *Memory*, a samo instrukcije `Jump` zahtevaju vrednost `valP` u stepenu *Execute* (u slučaju kada do grananja ne dolazi). Nijedna od ovih instrukcija ne zahteva vrednost koja se čita iz RF polja. Zbog ovoga moguće je redukovati obim protočnog registra kombinovanjem ova dva signala (putem selekcije) i prenosom ka daljoj obradi jedinstvenog signala `valA.` Ovo eleiminiše ugradnju bloka označen kao `Data` u procesoru SEQ i SEQ+.

1.5.2. Predikcija naredne vrednosti programskog brojača

Cilj protočnog dizajna je da inicira izvršenje (*issue*) od pojedine nove instrukcije po svakom taktnom intervalu. To znači da će svakog taktnog intervala po jedna instrukcija da ulazi u stepen *Execute* i po jedna da ga napušta. Postizanjem ovog cilja ostvaruje se propusnost od jedne instrukcije po taktnom intervalu. Da bi uradili ovo neophodno je odrediti lokaciju naredne instrukcije odmah nakon što smo pribavili tekuću instrukciju. Na nesreću ako je pribavljena instrukcija tipa uslovno grananje mi nećemo saznati da li će do grananja doći ili ne za nekoliko ciklusa kasnije, tj. sve dok instrukcija ne prodje kroz stepen *Execute*. Na slikan način, ako je pribavljene instrukcija tipa *ret*, nemoguće je odrediti adresu povratka sve dok instrukcija ne prodje kroz stepne *Memory*.

Sa izuzetkom instrukcija uslovnog grananja i instrukcije *ret*, mi mižemo odrediti adresu naredne instrukcije na osnovu informacije koja se izračunava u toku stepena *Fetch*. Za instrukcije *Call* i *Jmp* (bezuslovno grananje) to će biti vrednost *valC*, 32-bitna konstanta koja je specificirana kao neposredna vrednost u okviru instrukcije, dok će za sve ostale instrukcije to biti vrednost *valP* to biti adresa naredne

instrukcije. Na osnovu prethodne diskusije je jasno da u najvećem broju slučajeva možemo ostvariti cilj od iniciranja izvršenje od jedne instrukcije po taktnom intervalu ako korektno predvidjamo narednu vrednost programskog brojača. Za najveći broj tipova instrukcija, predikcija biće u potpunosti pouzdana. Za uslovne instrukcije grananja moguće je birati izmedju sledeće dve varijante: a) do grananja dolazi-što znači da nova vrednost programskog brojača biti `valC`; ili b) do grananja ne dolazi – nova vrednost programskog brojača biti `valP`. U oba slučaja moraju se preduzeti odgovarajuće korektivne akcije ako predikcija nija bila korektna iz razloga što su već pribavljene i delimično izvršene pogrešne instrukcije.

Tehnika nagadjanja smera grananja a nakon toga iniciranja pribavljanja instrukcija u saglasnosti sa nagadjanjem je poznata kao predikcija grananja (*branch prediction*). Ona se u odredjenoj formi koristi kod ovih procesora. Na ovom polju vršena su brojna istraživanja. Kod nekih od sistema ugradjuje se obiman hardver da se reši ovaj problem. Mi ćemo u daljem tekstu koristiti strategiju da do grananja uvek dolazi (*always taken branch prediction strategy*), što ukazuje da uvek predvidjamo da će nova vrednost kojom se ažurira stanje programskog brojača biti `valC`. Istraživanja koja su obavljena u vezi ove strategije pokazuju da je stopa uspešnosti nagadjanja 60%. To znači da kod strategije nagadjanja da do grananja nikad ne dolazi (*never taken (NT) branch strategy*) stopa uspešnosti pogadjanja biti 40%. Jedna sofisticiranija strategija, poznata kao do grananja prema nižim adresama dolazi a prema višim ne dolazi (*backward taken, forward not taken – BTFNT*) se takodje veoma često implementira i ona ima stopu uspešnosti pogadjanja oko 65%. Ovako relativno visoka stopa uspešnosti pogadjanja posledica je činjenice da programske sekvence tipa petlja (*loop*) se uvek zatvaraju prema nižim adresama, a što je još važnije petlje se, u opštem slučaju, izvršavaju više puta (respetitivno). Grananja prema višim adresama (*forward branches*) se obično koriste kod uslovnih operacija, ali pri tome manje je verovatnoća da će do grananja doći.

Nazvisno od izabrane strategije za predikciju grananja problem nagadjanja koji se odnosi na instrukciju `Ret` ostaje i dalje veoma aktuelan.Naime, nasuprot uslovnih instrukcija grananja kod instrukcije `Ret` imamo takoreći neograničen skup mogućih rezultata, iz razloga što povratna adresa može biti bilo ka reč sa vrha magacina. U našem rešenju mi nećemo ni pokušavati da nagadjamo bilo kakvu vrednost za povratnu adresu. Umesto toga zaustavićemo na dalje procesiranje novih instrukcija sve dok instrukcija `Ret` ne prodje kroz stepen *Write-back.*

Kod najvećeg broja programa veoma je lako tačno predvideti povratnu adresu iz razloga što se pozivi procedure (*call procedures*) i odgovarajući povratci (*returns*) javljaju kao parovi. Kod najvećeg broja slučajeva instrukcija tipa poziv-procedure povratak se vrši na instrukciju koja, u pozivnom programu, neposredno sledi iza instrukcije poziv-procedure. Ova osobina je iskorišćena kod visoko-performansnih procesora na taj način što se u okviru stepena *Fetch* ugradjuje hardverski magacin u kome se čuva povratna adresa zapamćena u toku izvršenja instrukcije poziv-procedure. Svaki put kada se izvrši instrukcija poziv procedure (*call procedure*) povratna adresa se smešta u magacin. Sa druge strane, kada se izvršava instrukcija `Ret` pribavlja se vrednost sa vrha magacina kao prediktovana povratna adresa. Na sličan način kao i kod predikcije grananja mora da postoji mehanizam za oporavak u slučaju da je predikcija pogrešna, iz razloga što postoje situacije kada izmedju para instrukcija `Call` i `Ret` ne postoji uparenost. No ipak treba naglasiti da je stopa nagadjanja adrese povratka veoma visoka. Stepen *Fetch* procesora PIPE- je odgovoran za: a) predvijdanje naredne vrednosti programskog brojača; i b) selekciju aktuelne vrednosti programskog brojača radi pribavljanja instrukcija. Gradivni blok `Predict_PC` može da selektuje: 1) vrednost `valP`, izračunatu od strane `PC incrementer` jedinice; ili 2) vrednost `valC` koja predstvlja 32-bitnu neposrednu vrednost specificiranu instrukcijom.Selektovana vrednost sa izlaza bloka `Predict_PC` se pamti u protočni registar `F` kao prediktovana vrednost za programski brojač. Blok označen kao `Select_PC` sličan je bloku označen kao kod SEQ+ `PC_selekcionog-stepena`. Blok `Select_PC` bira jednu od sledećih triju vrednosti koje se koristi za adresiranje instrukcione memorije: i) prediktovana vrednost programskog brojača; ii) vrednost `valP` za instrukciju tipa do grananja-ne-dolalzi (*not-taken branch instruction*) vrednost `M_valA` koja se vraća iz registra `M`; i iii) vrednost adrese povratka kada instrukcija ret pristigne do protočnog registra `W` (vrednost `W_valM`).

1.5.3. Protočni hazardi

Povratne sprege su tipične za rad protočnih procesora. Na žalost ovakve sprege mogu izazvati probleme u radu sistema posebno kada izmedju sukcesivnih instrukcija postoje zavisnosti. Ovi aspekti se moraju uspešno rešiti pre nego što se kompletira dizajn procesora. Pri ovome se mogu identifikovati sledeće tri forme zavisnosti:

1) zavisnosti po podacima (*data dependencies*) – rezultati koji su izračunati od strane jedne instrukcije koriste se kao podaci za narednu instrukciju,

2) upravljačke zavisnosti (*control dependencies*) – jedna instrukcija odredjuje lokaciju naredne instrukcije, kakav je to slučaj kod instrukcija `Jump`, `Call`, i `Return`.

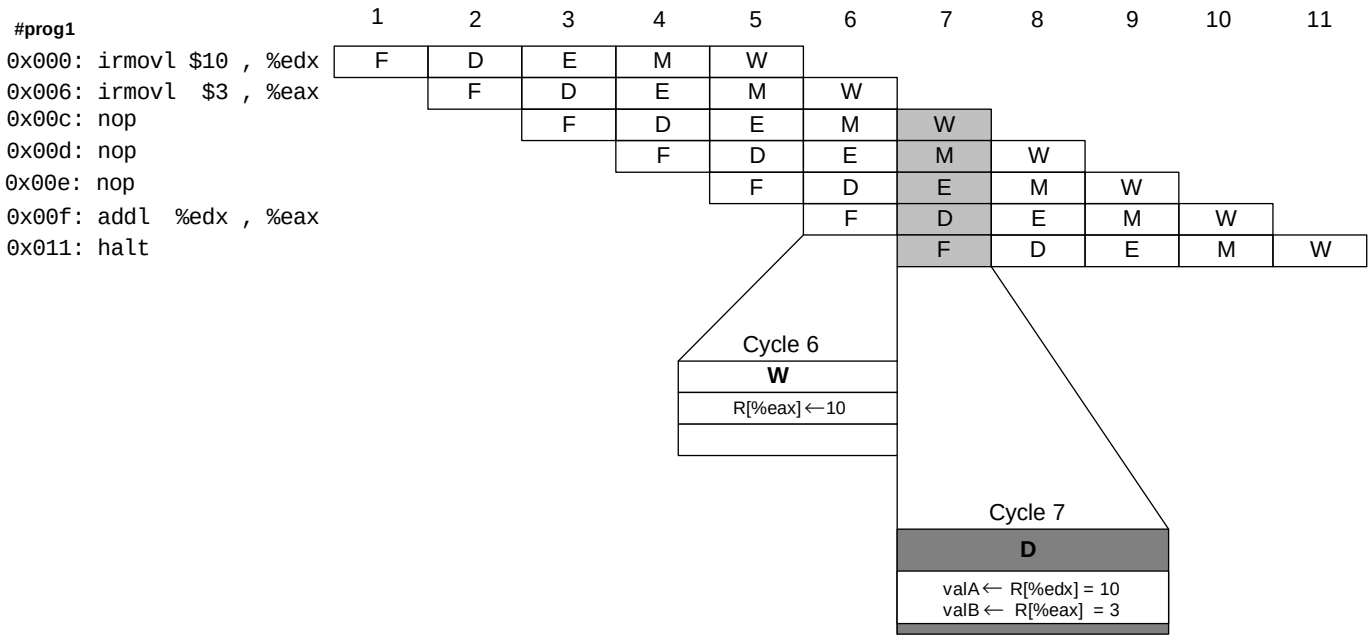
3) zavisnosti-po-resursima (*resource dependencies*) – javljaju se kada dve uzastopne instrukcije koriste jedan isti resurs, kao na primer dve instrukcije množenja koriste jedinstveni hardverski množač implementiran u procesoru. U konkretnom slučaju, zbog fizičkog ograničenja, sve dok prvo množenje ne završi drugo ne može da započne. Imajući u vidu da su ova ograničenja pre svega posledica fizičkih, a ne logičkih (programskih) ograničenja, u daljem našem razmatranju usvojićemo da je broj raspoloživih hardverskih resursa implementiran u procesoru dovoljno veliki, pa shodno tome smatraćemo da ova ograničenja ne egzistiraju.

Ako zavisnosti pod (1) i (2) imaju potencijal da uzrokuju pogrešno izračunavanje u protočnom sistemu, tada te zavisnosti nazivamo **hazarde**. Na sličan način kao i zavisnosti, i hazarde možemo klasifikovati kao:

hazarde-po-podacima (*data hazards*), i

hazarde-po-upravljanju (*control hazards*)

Na slici 23 prikazano je kako se na procesoru PIPE- vrši procesiranje sekvence instrukcija koju ćemo nazvati `prog1.`

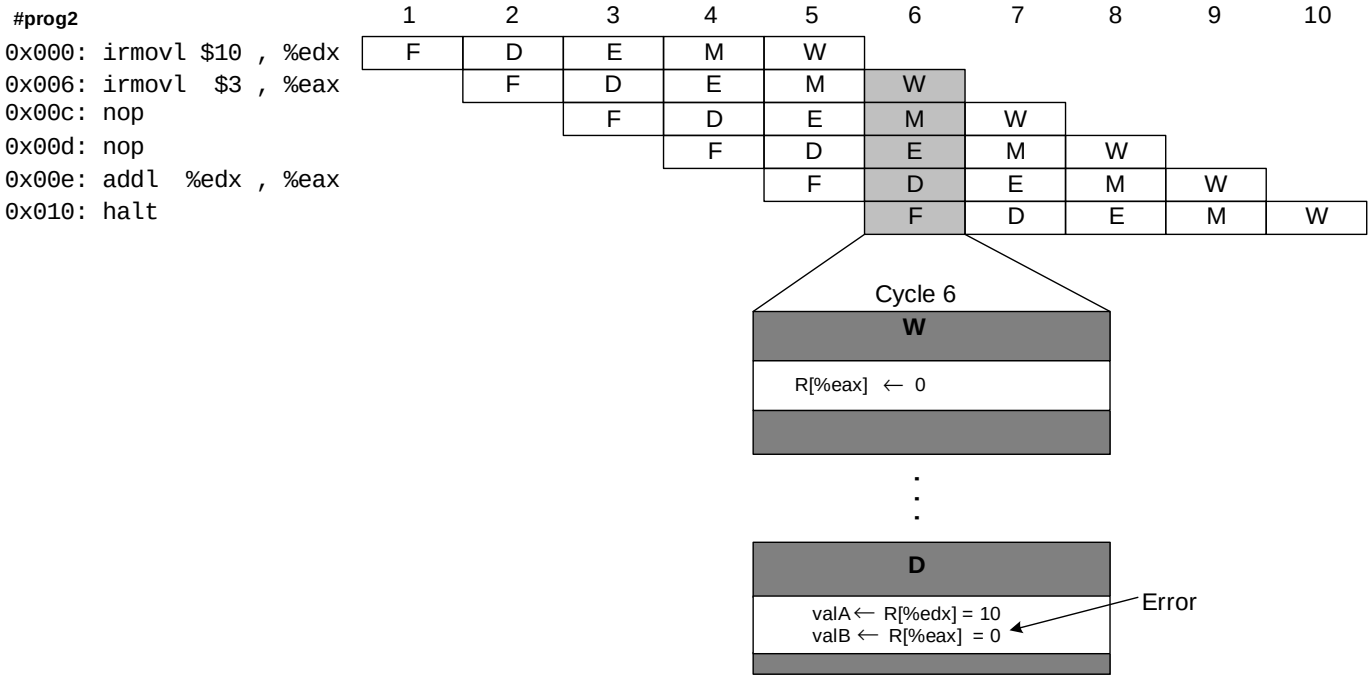


Slika 23 Protočno izvršenje *prog1*

***Napomena**: U ciklusu 6 druga `Irmovl` upisuje rezultat u registar `%eax`. Instrukcija `Addl` čita svoje izvorne operande u ciklusu 7, tako da upisuje korektne vrednosti u oba registra `%edx` i `%eax`, respektivno.*

Program sa slike 23 puni vrednosti 10 i 3 u registre `%edx` i `%eax`, izvršava tri instrukcije `Nop`, a zatim sabira sadržaje registara `%edx` i `%eax`. Fokusiraćemo sada našu pažnju na potencijal pojavljivanja hazarda-po-podacima koji rezultiraju zbog zavisnosti-po-podacima izmedju obe `Irmovl` instrukcije i `Addl` instrukcije. Na desnoj strani slike, prikazan je protočni dijagram za instrukcionu sekvencu sa leve strane slike. Na slici 24 detaljno su prikazane *Write-back* aktivnosti u ciklusu 7, i *Decode* aktivnosti u ciklusu 7. Na početku ciklusa 7 obe `Irmovl` instrukcije su već obavile procesiranje u stepenu *Wtite-back,* tako da su u RF polju registri `%edx` i `%eax` korektno ažurirani. To znači da zavisnosti-po-podacima izmedju dve `Irmovl` instrukcije i `addl` instrukcije nisu, u ovom slučaju, uzrokovale hazarde-po-podacima.

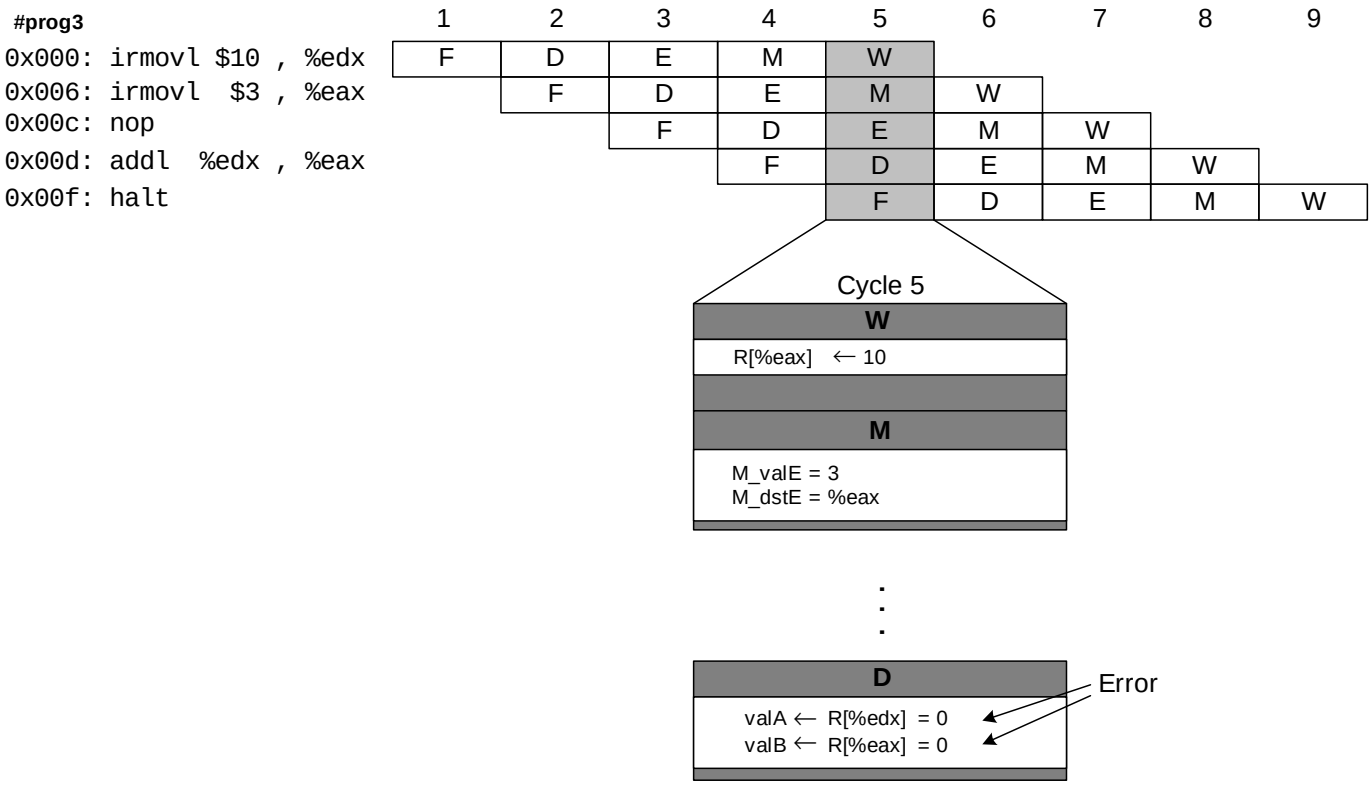
Analizirajmo sada šta će se desiti ako iz `prog1` izbacimo jednu `Nop` instrukciju. Na slici 24 prikazano je protočno izvršenje programa nazvan `prog2` koji izmedju druge `Irmovl` i `Addl` instrukcije sadrži dve `Nop` operacije. U ovom slučaju ključni korak predstavlja ciklus 6, tj. trenutak kada instrukcija `Addl` čita izvorne operande iz RF polja. Detaljan opis aktivnosti protočnih stepeni, u ovom ciklusu, je skiciran na slici 24. Kao što se vidi sa slike 24, u ciklusu 6, prva `Irmovl` instrukcija je već prošla kroz stepen *Write-back* tako da je sadržaj registra `%edx` ažuriran, no druga `Irmovl` instrukcija je u stepenu `Write-back`. Imajući u vidu da se upis u registar RF polja obavlja na kraju ciklusa 6 (registar `%eax` biće korektno ažuriran tek na početku ciklusa 7), to znači da se iz registra `%eax` čita nekorektni rezultat (usvajamo da su svi registri inicijalno postavljeni na 0). Na osnovu ovog primera zajljučujemo da je potencijalna zavisnost-po-podacima uzrokovala hazard-po-podacima. Cilj projektanta je da prilagodi rad protočnog sistema kako bi se na jedan korektan način elilminisao hazard-po-podacima.



Slika 24 Protočno izvršenje *prog 2*

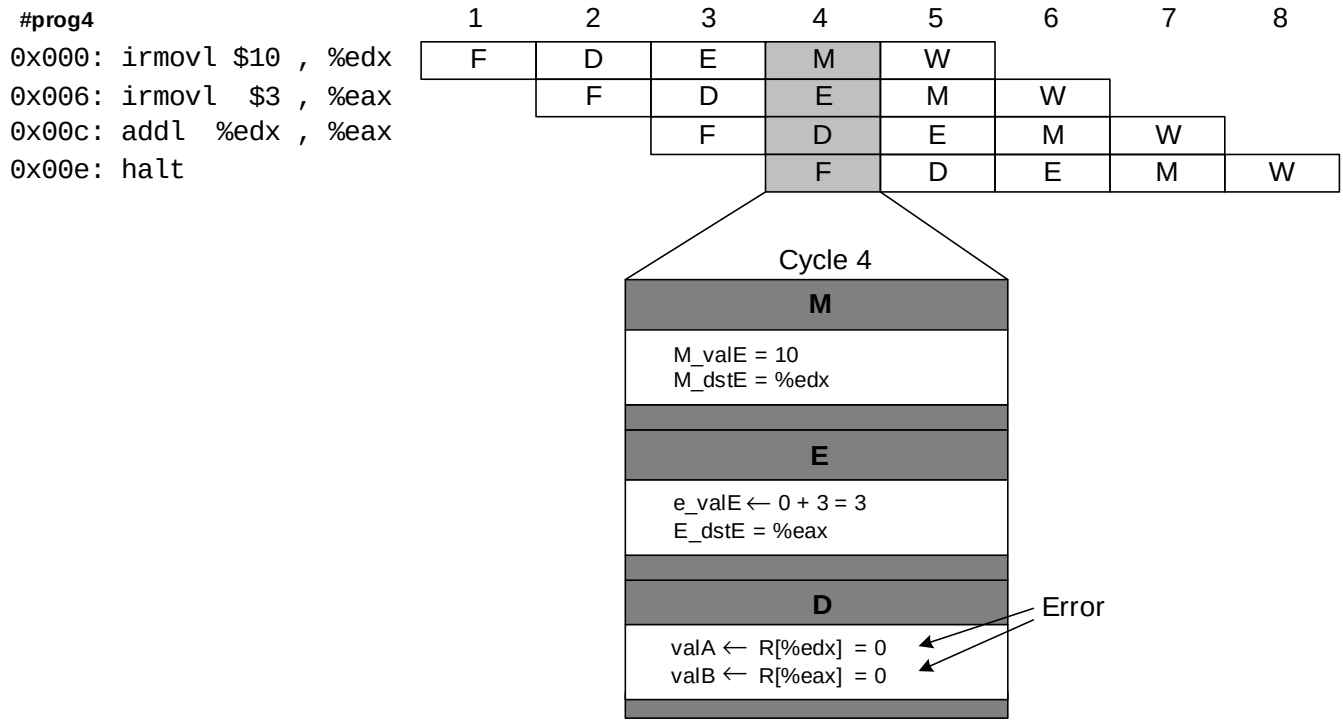
***Napomena**: Upis u registar `%eax` se dešava na kraju ciklusa 6, što uzrokuje da instrukcija `Addl` pribavi nekorektnu vrednost u stepenu Decode.*

Na slici 25 i slici 26 prikazani su efekti izvršenja sekvence instrukcija kada izmedju druge `Irmovl` i `Addl` postoji jedna `Nop`, i kada ne postoji Nop operacija, respektivno.



Slika 25 Protočno izvršenje *prog3*

***Napomena**: U ciklusu 5 instrukcija `Addl` čita izvorne operande iz RF polja, ali u tom trenutku prva `Irmovl` instrukcija je počela sa procesiranjem u stepenu Write-back tako da registar `%edx` još nije ažuriran, dok druga `Irmovl` instrukcija se procesira u stepenu `Memory` što znači da i registar `%eax` nije ažuriran. Kao rezultat, oba operanda `valA` i `valB` imaće nekorektne vrednosti.*



Slika 26 Protočno izvršenje *prog4*

***Napomena**: U ciklusu 4 instrukcija Addl čita izvorne operande iz RF polja. U tom trenutku prva Irmovl instrukcija se procesira u stepenu Memory, a druga Irmovl instrukcija u stepenu Execute. To znači da obe instrukcije još nisu stigle da ažuriraju registre %edx i %eax, pa će vrednosti valA i valB biti nekorektne.*

Primeri sa slika 24, 25 i 26 jasno ukazuju na hazarde-po-podcima koji se javljaju izmedju instrukcija. U principu hazardi-po-podacima se potencijalno javljaju kada jedna instrukcija ažurira deo programskog stanja koji će biti pročitan od strane neke instrukcije koja se kasnije izvršava. Programsko stanje čine programski registri, markeri uslova, memorija, i programski brojač. Sagledajmo sada na mogućnosti pojave hazarda za svaki oblik programskog stanja.

Registri vidljivi programeru: ove hazarde smo već identifikovali kroz primere sa slika 24, 25 i 26. Hazardi ovog tipa se pre svega javljaju zbog toga što se podatak iz RF polja čita u jednom stepenu, a upis vrši u drugom stepenu, što dovodi do neželjenih interakcija izmedju različitih instrukcija.

Markeri uslova: stanje markera uslova menja se operacijom upisa od strane neke *integer* instrukcije (`Add, Sub,Cmp,And,Or,` i dr), a čita od strane uslovnih instrukcija grananja (J_{zero},J_{carry}, i dr) u stepenu *Execute*. Za slučaj kada instrukcija uslovnog grananja `Jcc` prolazi kroz ovaj stepen, a neka od prethodnih *integer* operacija je već kompletirala svoje procesiranje u stepenu *Execute* do hazarda ne dolazi.

Programski brojač: Konflikti koji se javljaju usled ažuriranja i čitanja stanja programskog brojača uzrokuju upravljačke hazarde. Do hazarda ne dolazi kada logika *Fetch* stepena, pre nego što se pribavi naredna instrukcija, korektno predvidi novu vrednost. Pogrešno prediktovana grananja kao i instrukcija `Ret` zahtevaju poseban tip manipulisanja o kome ćemo kasnije govoriti.

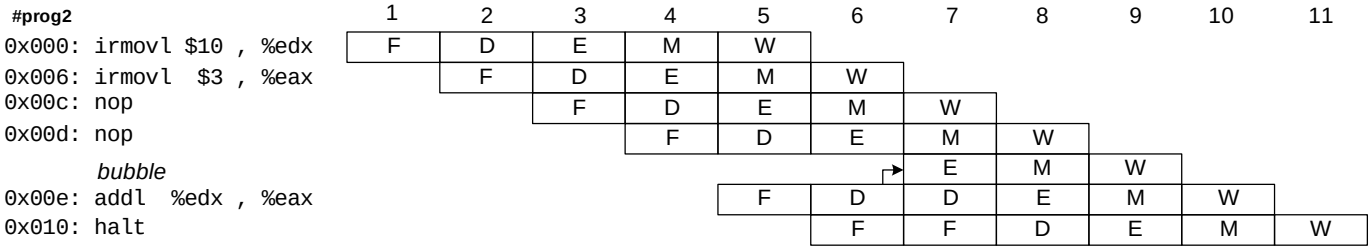
Memorija: Operacije upis i čitanje memorije podataka se dešavaju u stepenu *Memory*. U trenutku kada instrukcija koja čita ovu memoriju pristigne u ovaj stepen, neka od prethodnih instrukcija koja je obavljala upis u tu memoriju je već završena, pa opasnost od hazarda ne postoji. No sa druge strane, postoji interferencija izmedju instrukcija koje upisuju podatke u stepenu *Memory* i čitanje instrukcija u stepenu *Fetch*, za slučaj kada memorija za instrukcije i podatke dele isti adresni prostor. Ova situacija se javlja kod programa koji sadrže samo-modificirajući-kod, tj. instrukcije upisuju u deo memorije iz koga se kasnije

pribavljaju (*fetched*) instrukcije. Neki od sistema imaju implementirano veoma složene mehanizme za detekciju i premošćavanje (izbegavanje) ovih hazarda, dok drugi ne dozvoljavaju da se koristi samomodifikujući kod.

1.5.4. Izbegavanje hazarda po podacima pomoću zastoja

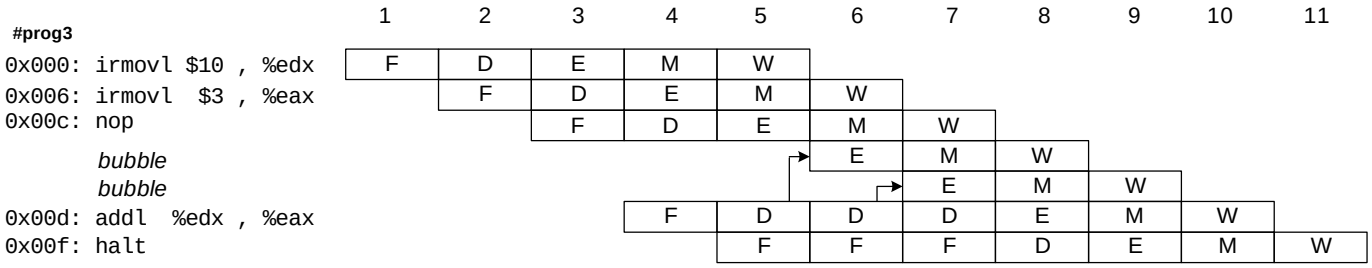
Jedna od standardnih tehnika za izbegavanje hazarda se zasniva na uvodjenju **zastoja** (*stalling*). Princip rada ove tehnike se sastoji u sledećem: Procesor prema početnim stepenima protočne obrade zaustavlja procesiranje jedne ili veći broj instrukcija sve dok se ne reši uzrok koji je doveo do hazarda. Tako na primer, procesor PIPE- izbegava hazarde-po-podacima zadržavajući izivršenje instrukcije u stepenu *Decode* onoliko dugo sve dok se njegovi operandi ne generišu od strane instrukcije koje se procesira u zadnjem stepenu protočne obrade *Write-back*. Ova tehnika je prikazana na slici 27 (`prog2`), 28 (`prog3`), i 29 (`prog4`).

Kada se instrukcija `Addl` nalazi u stepenu `Decode` upravljačka logika protočnog procesora detektuje da najmanje jedna od instrukcija koja se nalazi u stepenima *Execute*, *Memory*, ili *Write-back* treba da ažurira stanje registra `%edx` ili `%eax`. Umesto, pri tome, da dozvoli instrukciji `Addl` da prodje kroz stepen *Decode* sa pogrešno pribavljenim vrednostima izvornih operanada, procesor zaustavlja provesiranje instrukcije `Addl` u stepenu *Decode* za jedan taktni interval kod `prog2`, dva taktna intervala kod `prog3`, i tri kod `prog4`. Kod ova tri programa, instrukcija `Addl` konačno dobija korektne vrednosti za oba svoja izvorna operanda tek nakon ciklusa 7, posle čega produžava sa daljim procesiranjem. Kada se instrukcija `Addl` zaustavi u stepenu *Decode*, tada se i procesiranje instrukcije `Halt` zaustavlja u stepenu *Fetch*. Ovo se izvodi na taj način što se sadržaj programskog brojača fiksira, tj.ne inkrementira kako bi ukazao na narednu instrukciju, tako da se instrukcija `Halt` više puta pribavlja iz memorije-za-instrukcije, sve dok se ne reše uzroci koji su doveli do zastoja.



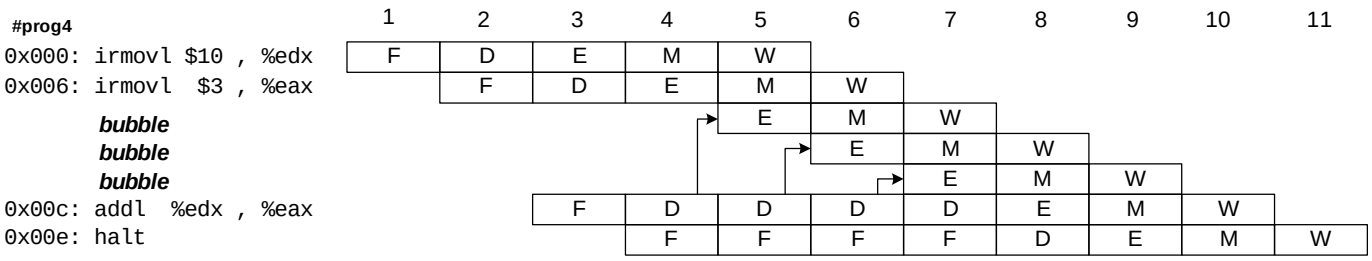
Slika 27 Protočno izvršenje *prog2* koristeći zastoje

***Napomena**: Nakon dekodiranja instrukcije `Addl` u ciklusu 6, upravljačka logika detektuje hazard-po-podacima zbog toga što registar `%eax` još nije ažuriran u stepenu Write-back. Zbog ovoga se ubacuje mehur (bubble) u stepenu Execute i ponavlja dekodiranje instrukcije Addl u ciklusu 7. Kao krajnji efekat imamo to da je mašina dinamički ubacila instrukciju `Nop` tako da je procesiranje instrukcije sada slično kao ono za `prog1` prikazano na slici 23.*



Slika 28 Protočno izvršenje *prog3* koristeći zastoje

***Napomena**: Nakon dekodiranja instrukcije `Addl` u ciklusu 5, upravljačka logika detektuje hazard-po-podacima zbog toga što oba registra `%edx` i `%eax` nisu još ažurirana (njihovo procesiranje nije prošlo kroz stepen Write-back). Kao posledica, ubacuje se mehur u stepenu Execute i ponavlja dekodiranje instrukcije `Addl` u ciklusu 6. Nakon toga se se ponovo detektuje hazard zbog registra `%eax`, ubacuje drugi mehur u stepenu Execute, i ponavlja dekodiranje instrukcije `Addl` u ciklusu 7. Kao efekat, mašina je dinamički insertovala dve instrukcije, sada slično kao ono za `prog1` prikazano na slici 23.*



Slika 29 Protočno izvršenje *prog4* koristeći zastoje

***Napomena**: Nakon dekodiranja instrukcije `Addl` u ciklusu 4, upravljačka logika detektuje da postoje hazardi-po-podacima zbog toga što oba izvorna registra nisu dostupna.. Kao posledica se ubacuje mehur u stepenu Execute i ponavlja dekodiranje instrukcije `Addl` u ciklusu 5. U ciklusu 6 ponovo se detektuju hazardi zbog toga što izvorni registri nisu i sada dostupni, u stepenu Execute se ubacuje mehur, a dekodiranje instrukcije `Addl` se ponavlja. U ciklusu 7 i dalje se detektuje hazard zbog toga registar `%eax` nije ažuriran pa se ubacuje mehur u stepenu Execute i ponavlja dekodiranje instrukcije `Addl`. Krajnji efekat je taj da je mašina dinamički ubacila tri instrukcije tipa `Nop`, a tok izvršenja je sličan kao onaj za `prog1` prikazan na slici 23.*

Zaustavljanje uzrokuje da se prema nazad (prvim stepenima u lancu protočne obrade) stopira procesiranje jedne grupe instrukcija u njihovim stepenima, dok druge instrukcije produžavaju procesiranje kroz protočni sistem. U konkretnom primeru, zadržava se za period od tri taktna intervala procesiranje instrukcije `Addl` u stepenu *Decode*, a instrukcije `Halt` ustepenu *Fetch*, dok procesiranje obe `Irmovl` instrukcije kao i *`Nop` instrukcije (slučajevi `prog2` i `prog3`) produžava, i one prolaze kroz stepene Execute, Memory i Write-back*. Zaustavljanje se vrši na taj način što svaki put kada treba da zadržimo procesiranje instrukcije u stepenu *Decode* ubacujemo mehurove u stepenu *Execute*. Efekat je takav da se mehur ponaša kao dinamički insertovana instrukcija `Nop`, tj. ona kao operacija bez efekta ne uzrokuje promenu stanja registara, memorije, ili markera uslova. Na slikama 27, 28 i 29 ubacivanje mehurova je označeno strelicom koja počinje od bloka `D` instrukcije `Addl` a usmereno je ka bloku `E`. Strelice ukazuju da je u stepenu *Execute* umesto instrukcije `Addl`, koja normalno treba da predje iz stepena `Decode` u stepen `Execute`, sada insertovan mehur.

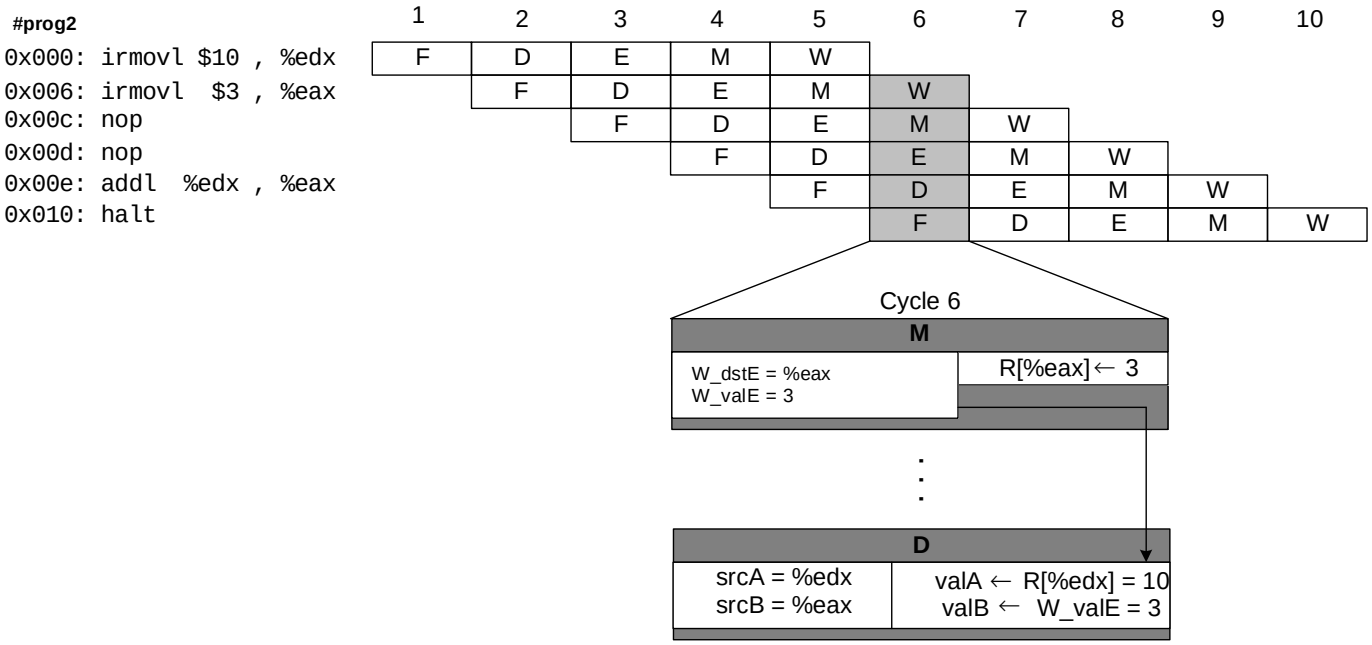
Ubacivanjem zastoja sa ciljem da se reše hazardi-po-podacima efektivno programi `prog2`, `prog3` i `prog4` se izvršavaju tako što se dinamički generiše protočni tok upravljanja kao onaj tipičan za `prog1` prikazan na

slici 24. Ubacivanje jedan mehur za `prog2`, dva za `prog3`, i tri za `prog4` ima isti efekat kao da postoje tri `Nop` instrukcije izmedju druge `Irmovl` instrukcije i instrukcije `Addl`.
Treba na kraju naglasiti da izbegavanje hazarda-po-podacima korišćenjem zastoja ima za efekat smanjenje propusnosti u radu sistema, tj. degradacije performansi.

1.5.5. Izbegavanje hazarda-po-podacima pomoću premošćavanja

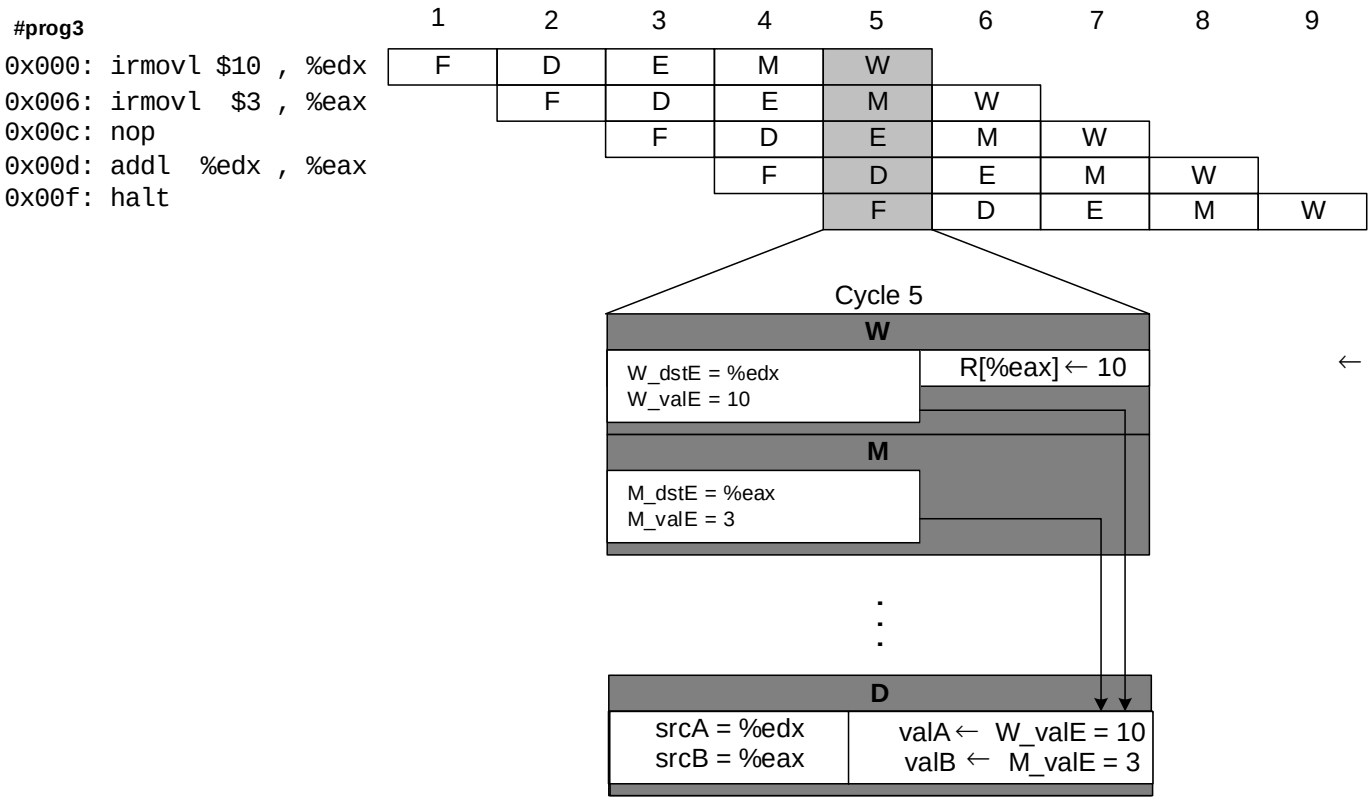
Hardver i povezivanje kod procesora PIPE- (slika 22) je tako izvedeno da se čitanje izvornih operanada iz RF polja obavlja u stepenu *Decode*, a upis u izvorne registre vrši se u stepenu *Write-back*. Pri ovome treba napomenuti da se vrednost koju treba upisati u izvorni registar izračunava u stepenu *Execute*, zatim ta vrednost se prenosi kroz stepen *Memory* bez da se pri tome nad njom obavlja neko procesiranje, pa se tek nakon toga ta vrednost upisuje u neki od izvornih registara RF polja. Svakom čitaocu odmah pada u oči da se nepotrebno gubi vreme od dva ciklusa, tj. od trenutka kada se ta vrednost izračuva do trenutka kada ona postane dostupna. Da bi se uspešno premostio ovaj problem projektanti procesora se odlučuju na rešenje da izračunata vrednost nakon stepena *Execute* postane odmah dostupna za dalje korišćenje narednim instrukcijama u programskoj sekvenci, a da se ona istovremeno prenosi ka narednim stepenima u protočnoj obradi (prvo prolazi kroz stepen *Memory* a nakon toga se vrši upis u stepen *Write-back*). Na slici 30 prikazana je strategija ovakvog načina izvršenja instrukcija pri čemu je detaljno prikazan ciklus 6 za `prog2`. Upravljačka logika stepena *Decode* detektuje da je registar `%eax` izvorni registar za operand `valB`, i da je potrebno obaviti upis u registar `%eax` putem operacije upis u `port-E` RF polja (slika 22). Na ovaj način moguće je izbeći zastoje koristeći podatak koji se dovodi na `port-E` RF polja (signal `W-valE`) kao vrednost za operand `valB`. Ova tehnika koja se sastoji u tome da se rezultat iz jednog protočnog stepena direktno predaje drugom protočnom stepenu koji mu u lancu obrade prethodi (vrećanje rezutlata unazad) naziva se prosledjivanje-podataka (*data forwarding*, ili često se koristi alternativni termin *bypassing*, tj premošćavanje). Ova tehnika omogućava da se instrukcije `prog2` procesiraju kroz protočni sistem bez zastoja.

Na slici 31 prikazano je kako se u toku izvršenja `prog3` koristi prosledjivanje podataka kada vrednost koju treba upisati u registar prolazi kroz stepen *Memory*. U ciklusu 5 logika stepena *Decode* detektuje potrebu da se u registar `%edx` preko `porta-E` u stepenu *Write-back* upiše podatak, ali takodje i da se u registar `%eax` upiše podatak koji se trenutno premešta preko stepena *Memory*.



Slika 30 Protočno izvršenje *prog2* koristeći prosledjivanje podataka

***Napomena**: U ciklusu 6 logika stepena Decode detektuje da je ustepenu Write-back potrebno obaviti operaciju upis uregistar `%eax`. Vrednost koju treba upisati interpretira se od strane logike kao izvorni operand `valB`, a ne kao vrednost koju treba čitati iz RF polja.*



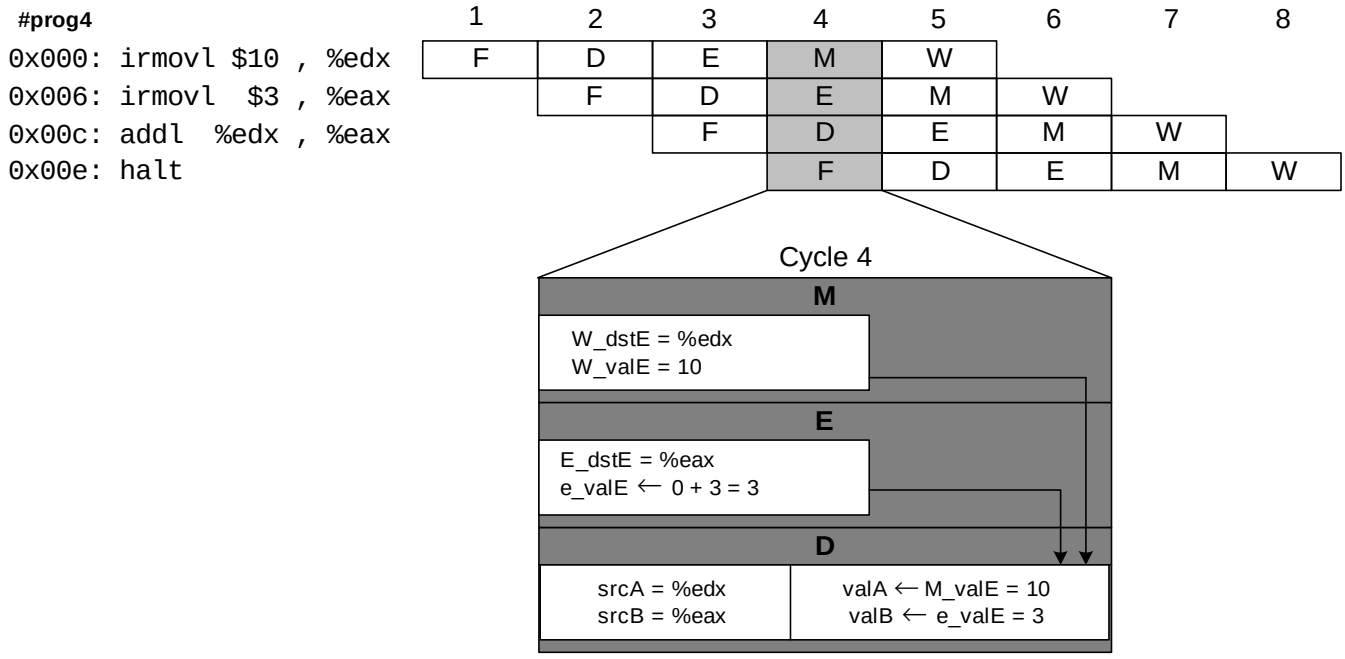
Slika 31 Protočno izvršenje prog3 koristeći prosledjivanje podataka

***Napomena**: U ciklusu 5 logika stepena Decode detektuje da vrednost koju treba upisati u registar `%edx` prolazi kroz stepen Write-back a vrednost koju treba upisati u registar `%eax` prolazi kroz stepen Memory. Vrednosti koje treba upisati u registre `%edx` i `%eax` a prolaze kroz stepene Write-back i Memory interpretiraju se od strane logike kao `valA` i `valB`, a ne kao vrednosti koje se čitaju iz RF polja.*

Umesto da se koristi zastoj u radu protočnog sistema koji bi trajao sve dok se operacije upis u registre `%edx` i `%eax` ne završe moguće je koristiti vrednost iz stepena *Write-back*, signal `W-valE,` kao operand `valA`, i vrednost koja prolazi kroz stepen *Memory* signal `M_valE`, kao operand `valB.`

Da bi se efekti ubrzanja, koje se pružaju prosledjivanjem-podataka, iskoristile u pravom smislu reči moguće je proslediti novo-izračunate vrednosti sa izlaza stepena *Execute* ka stepenu *Decode* čime se kako je to prikazano na slici 32 izbegava zastoj u `prog4`. Naime, u ciklusu 4 logika stepena *Decode* detektuje da sledi upis vrednosti u registar `%edx` ali ta vrednost trenutno prolazi kroz stepen *Memory*, a takodje detektuje da će vrednost koja se tekuće izračunava od strane ALU-a u stepenu *Execute* biti kasnije upisana u registar `%eax.` Pri ovome treba naglasiti da je moguće koristiti vrednost stepena *Memory* (signal `M_valE`) kao operand `val_A`. Takodje je moguće koristiti izlaz ALU-a (signal `e_valE`) kao operand `val_B`, pri čemu korišćenje ALU-ovog izlaza ne stvara probleme koji se odnose na sinhronizaciju. Za potrebe stepena *Decode* neophodno je samo na kraju taktnog intervala generisati signale `valA` i `valB` tako da se protočni registar `E` može puniti sa rezultatima iz stepena *Decode* sa početkom usponske ivice narednog taktnog intervala. Ono što je važno uočiti je to da je izlaz ALU-a važeći pre ovog trenutka.

Način korišćenja tehnike-premošćavanja ilustrovan je preko izvršenja programa `prog2` do `prog4`, pri čemu se u svim slučajevima implicitno podrazumeva da se prosledjuju vrednosti generisani od strane ALU-a, koji su namenjeni za upis u `port E` RF polja.



Slika 32 Protočno izvršenje prog4 korišćenjem premošćavanja- podataka

***Napomena**: U toku ciklusa 4 logika stepena Decode detektuje da sledi upis u registar `%edx` ali da podatak koga treba upisati prolazi kroz stepen Memory, a takodje da novo izračunatu vrednost sa izlaza stepena Execute treba upisati u registar `%eax`. Ove vrednosti umesto da se kasnije čitaju kao izvorni operandi iz RF polja, koriste se kao vrednosti `valA` i `valB`.*

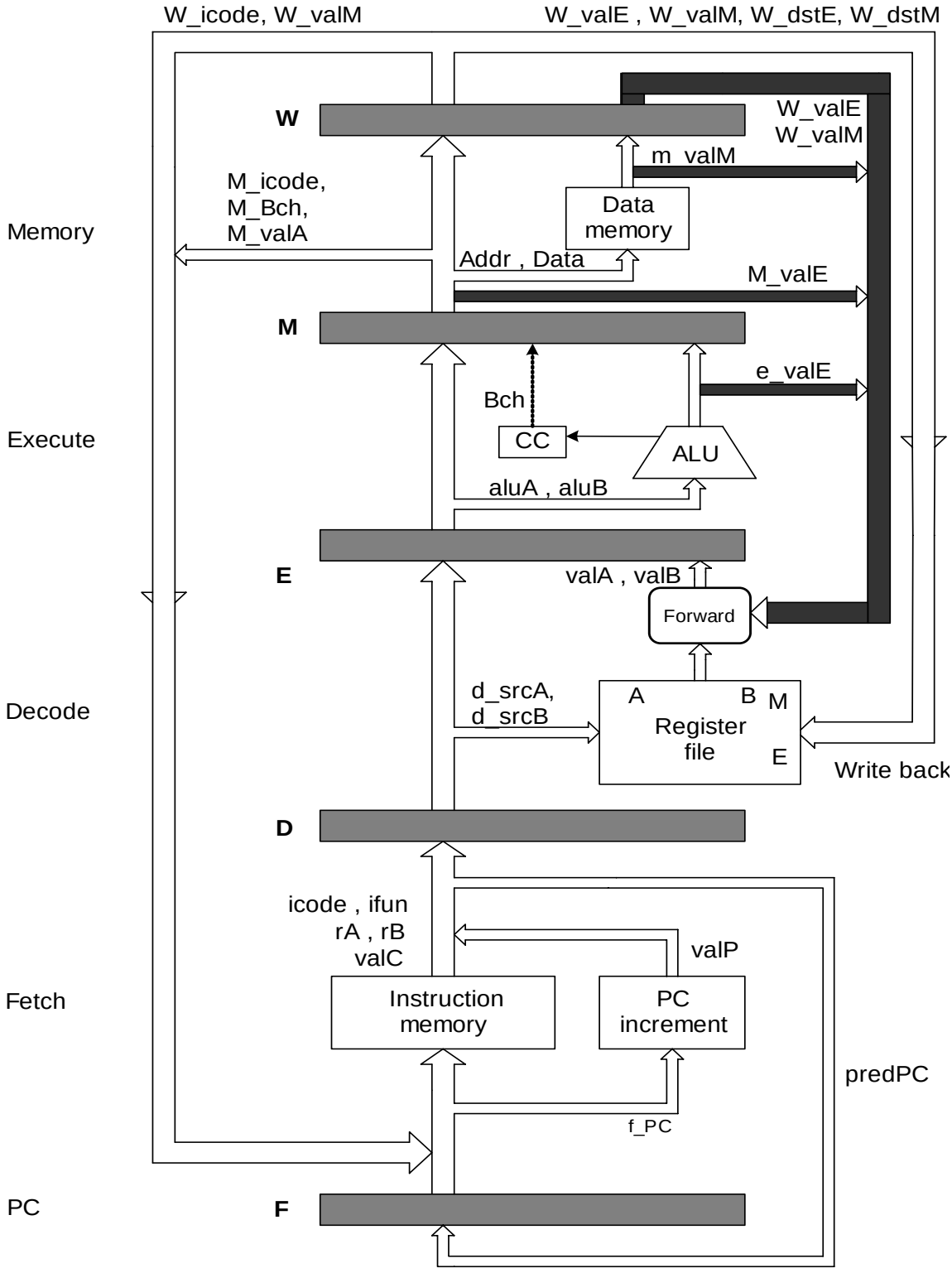
Prosledjivanje podataka se takodje može koristiti kada se čitaju vrednosti iz memorije za podatke koji se zatim dovode radi upisa na ulaz `porta E` RF polja. Sa izlaza stepena *Memory* moguće je takodje proslediti vrednost koja je upravo pročitana iz memorije-za-podatke (signal `m_valM`). Sa stepena *Write-back* možemo takodje proslediti podatak koga treba upisati preko `porta M` u RF polje (signal `W_valM`). Na osnovu prethodne diskusije evidentno je da je moguće proslediti podatke sa pet različitih izvorišta (`e_valM, m_valM,M_valE`, `W_valM, i W_valE`), na dva različita odredišta za prosledjivanje (`valA` i `valB`).

Na slikama 30, 31 i 32 prikazano je kako logika stepena *Decode* može da odredi da li treba da koristi vrednost koju treba da čita iz RF polja ili da koristi vrednost koja se prosledjuje.

Svakoj vrednosti koja se sa izlaza stepena *Write-back* upisuje u RF polje pridružuje se ID (identifikaciona adresa) odredišnog registra. Logika stepena *Decode* sa ciljem da detektuje da li treba da ostvari prosledjivanje uporedjuje ove ID vrednosti sa ID-ovima izvorišnih registara `srcA` i `srcB`. Moguće je da se desi situacija kada dolazi do uparivanja većeg broja ID-ova koji se odnose na odredišni registar sa jednim izvorišnim ID-om. U tom slučaju da bi se obavila korktna manipulacija sa podacima neophodno je uvesti prioritet u prosledjivanju-vrednosti sa različitih izvorišta.

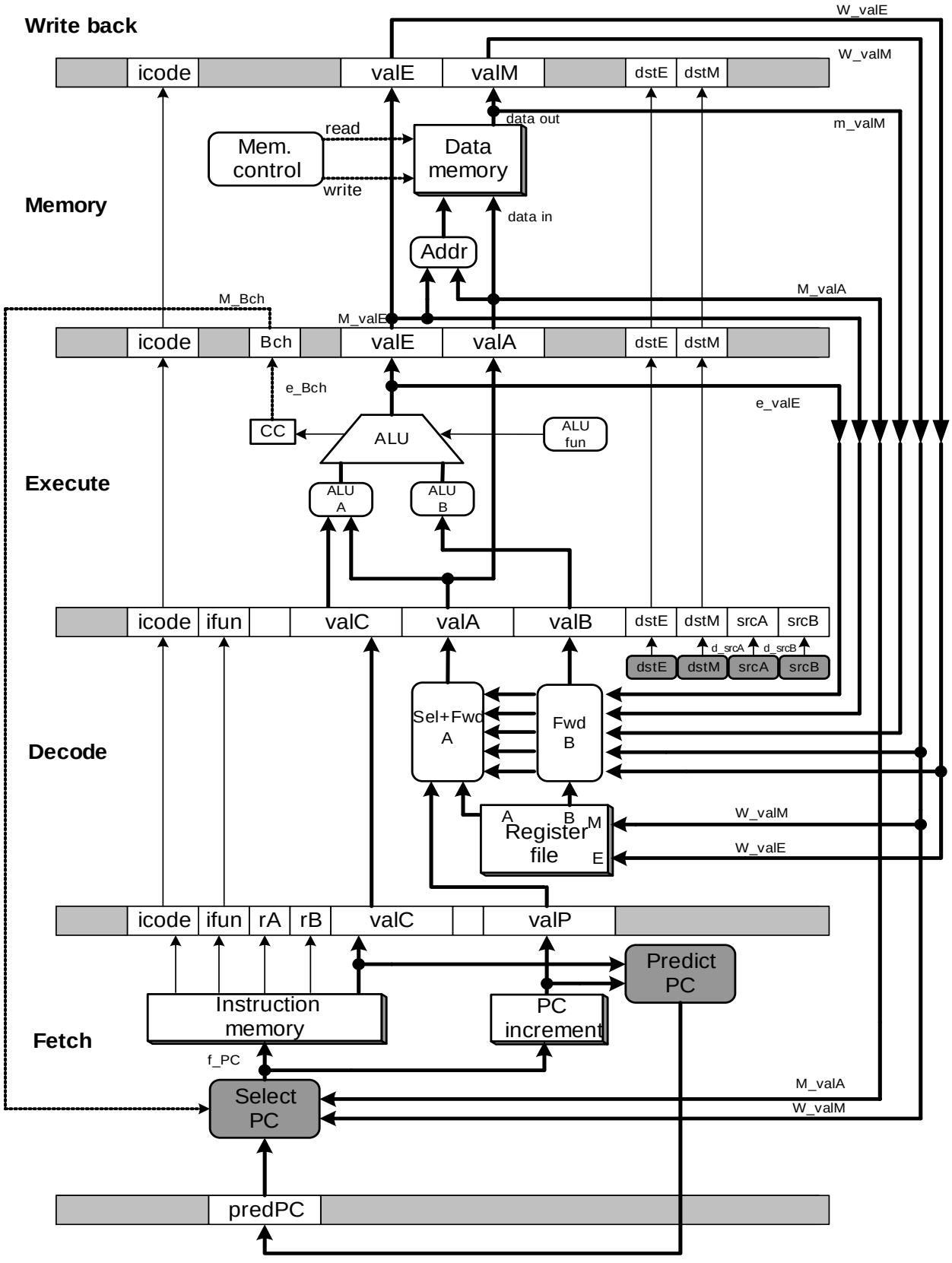
Na slici 33 prikazana je abstraktna struktura procesora PIPE, koji predstavlja modifikovanu verziju procesora PIPE-. Modifikacija se pre svega odnosi na uvodjenje logike za prosledjivanje podataka. U odnosu na strukturu procesora PIPE- kod procesora PIPE uvedeni su dodatni povratni putevi podataka. To su putevi sa pet rahzličitih izvorišta za premošćavanje usmereni ka stepenu *Decode*. Putevi za premošćavanje (*bypass path*) se dovode na blok označen kao *Forward* koji je pridružen stepenu *Decode*. Blok *Forward* generiše izvorne operande `valA` i `valB` na osnovu vrednosti koje se čitaju iz RF polja, ili vrednosti koje su povratno prosledjene.

Detaljan pogled na hardversku strukturu procesora PIPE- prikazan je na sllici 34. Uporedjivanjem struktura procesora PIPE- (slika 34) sa strukturom procesora PIPE uočavamo da se vrednosti sa pet izvorišta za prosledjivanje dovode na ulaz dva bloka koji su u bloku *Decode* označeni kao `Sel+FwdA` i `FwdB`. Blok `Sel+FwdA` spaja ulogu bloka `SelectA` koji postoji kod procesora PIPE- sa ulogom logike za prosledjivanje-podataka kod procesora PIPE. Blok `Sel+FwdA` obebedjuje da `valA` za protočni registar `M` predstavlja vrednost inkrementiranog programskog brojača `valP`, vrednost koja se čita sa `porta A` iz RF polja, ili jednu od prosledjenih vrednosti. Blok označen sa `FwdB` implementira logiku – premošćavanja za izvorni operand `valB`



Slika 33 Abstraktni pogled na procesor PIPE- konačna protočna implementacija

***Napomena**: Dodatno izvedeni putevi u odnosu na procesor PIPE- obezbedjuju premošćavanje-podataka time što omogućavaju da se vrši prosledjivanje rezultata od tri instrukcije koje su prethodno prošle procesiranje kroz stepen Execute. Premošćavanje nam obezbedjuje efikasno manipulisanje sa najvećim brojem formi hazarda-po-podacima bez da se pri tome zaustavlja rad protočnog sistema.*

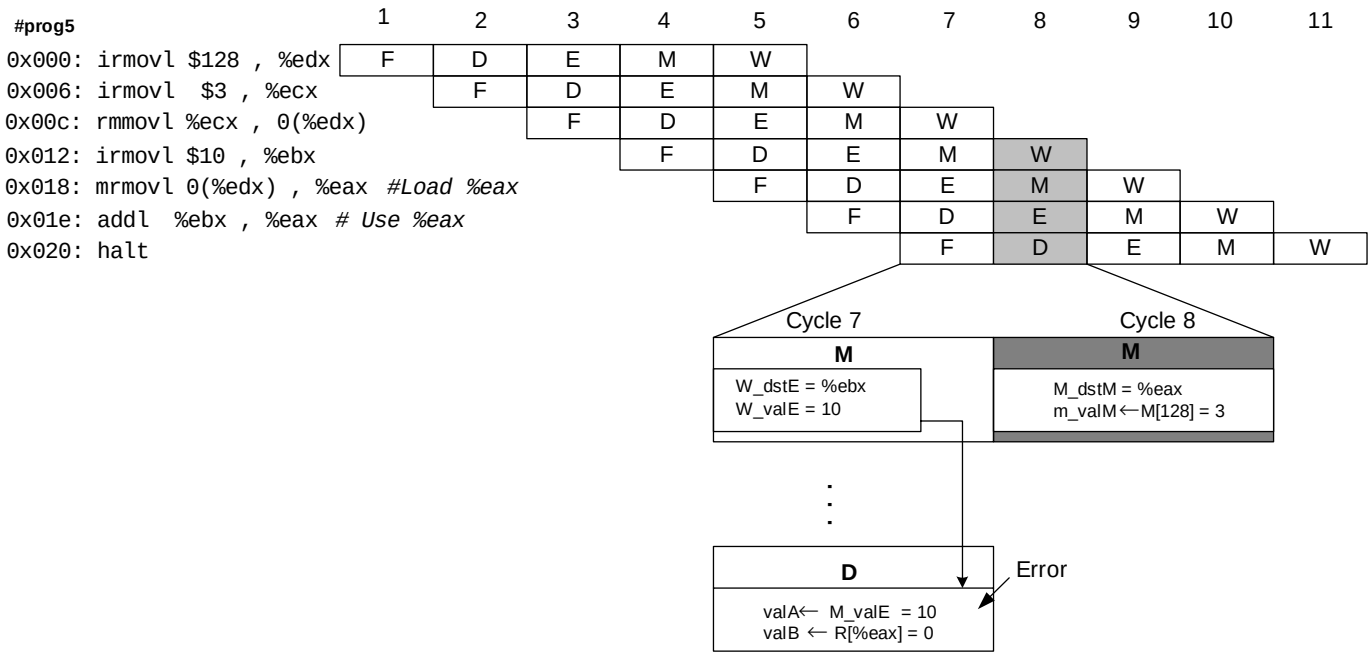


Slika 34 Hardverska struktura procesora PIPE- konačna implementacija

***Napomena**: Neka od povezivanja nisu prikazana*

1.5.6. Hazardi po podacima zbog korišćenja instrukcije Load

Jedna od klasa hazarda-po-podacima nazvana `Load-use` ne može se uspešno rešiti implementiranjem tehnike prosledjivanja-podataka. Na slici 35 prikazan je primer hazarda tipa `Load-use`, gde jedna instrukcija (konkretno je to `Mrmovl` koja se nalazi na adresi 0x018) čita vrednost iz memorije koju treba upisati u registar `%eax`, dok je narednoj instrukciji (to je `Addl` na adresi 0x01e) potrebna ova vrednost kao izvorni operand. Detaljan pogled koji se odnosi na cikluse 7 i 8 prikazan je na slici 35 (u donjem delu slike). Instrukciji `Addl` je potrebna vrednost registra u ciklusu 7, ali se ona ne može generisati od strane instrukcije `Mrmovl` sve dok ne završi ciklus 8. Sa ciljem da se obavi prosledjivanje-vrednosti od instrukcije `Mrmovl` do instrukcije `Addl`, logika-za-prosledjivanje treba posmatrano vremenski, na neki način unazad da generiše tu vrednost. S obzirom da je to nemoguće, mora se pronaći neki drugi efikasan mehanizam koji će obezbediti korektno manipulisanje sa ovim tipom hazarda-po-podacima.



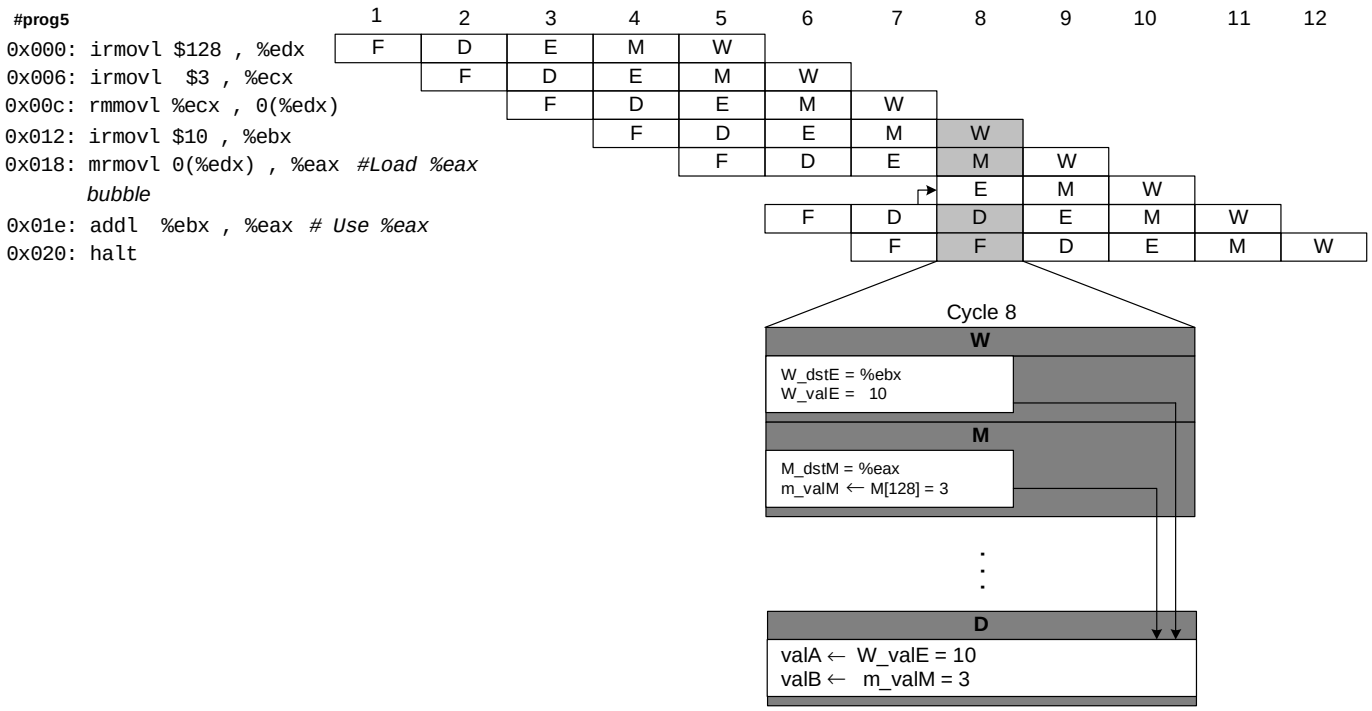
Slika 35 Primer hazarda po podacima tipa *Load-use*

***Napomena**: instrukciji `Addl` u stepenu Decode, u ciklusu 7, je potrebna vrednost registra `%eax`. Prethodna instrukcija `Mrmovl` čita novu vrednost iz registra `%eax` u stepenu Memory, a to je u ciklusu 8, što je mnogo kasno za instrukciju `Addl`.*

Uočimo da vrednost registra `%ebx` koja se generiše od strane instrukcije `Irmovl` na adresi 0x00C, može da se usmeri od stepena *Memory* ka instrukciji `Addl` u stepenu *Decode* u toku ciklusa 7. Na slici 36 prikazano je kako se može izbeći hazard-po-podacima tipa `Load-use` korišćenjem kombinacije zastoja i prosledjivanja. Nakon što instrukcija `Mrmovl` prodje kroz stepen *Execute*, upravljačka logika protočnog stepena detektuje da je instrukciji koja se nalazi u stepenu *Decode* potreban rezultat koji se čita iz memorije. Zbog toga zaustavlja se dalje procesiranje instrukcije u stepenu *Decode* za jedan ciklus, što dovodi do ubacivanja mehura u stepenu *Execute*. Ovaj detalj za ciklus 8, u ekspendovanoj formi, je prikazan na slici 36. Vrednost koja se čita iz memorije se može proslediti iz stepena *Memory* ka instrukciji `Addl` u stepenu *Decode*. Vrednost za registar `%edx` se takodje prosledjuje od stepena *Write-back* ka stepenu *Memory*. Kao što se vidi sa protočnog dijagrama na slici 37 usmerena strelica koja polazi sa bloka označen kao `D` u ciklusu

l7, a dolazi do bloka označen kao `E` u ciklusu 8, ukazuje da ubačeni mehur (operacija `Nop`) zamenjuje instrukciju `Addl` koja bi u normalnim okolnostima (za slučaj da ne postoji hazard) trebalo da produži sa procesiranjem kroz protočni sistem.

Korišćenje zastoja u cilju manipulisanja sa hazardima tipa `Load-use` naziva se *load-interlock*. Tehnika *load-interlock* u kombinaciji sa prosledjivanjem dovoljno uspešno rešava probleme koji se odnose na sve moguće forme zavisnosti-po-podacima. No pri ovome moramo biti svesni da se primernom tehnike *load-interlock* redukuje propusnost protočne obrade, tj ne može se postici iniciranje izvršenja od po jedne instrukcije po svakom taktnom intervalu.



Slika 36 Manipulisanje sa hazardom tipa *Load-use* korišćenjem zastoja

***Napomena**: Zaustavljanjem instrukcije `Addl` za jedan ciklus u stepenu Decode,vrednost `valB` se može proslediti od instrukcije `Mrmovl` koja se izvršava u stepenu Memory ka instrukciji `Addl` koja se izvršava od strane stepena Decode.*