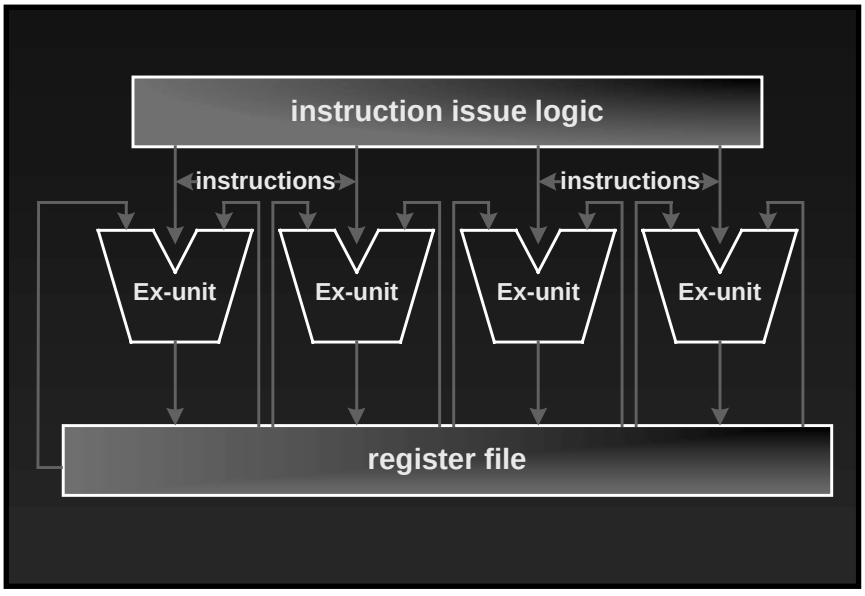
SADRŽAJ:

1. Paralelizam na nivou instrukcija-ILP

Već smo uočili da je protočnost važna tehnika za povećanje performansi računara. Ova tehnika se bazira na preklapanje izivršenja većeg broja instrukcija. Kao efekat ostvaruje se povećanje brzine sa kojom se instrukcije izvršavaju. No treba odmah naglasiti da postoje ograničenja koja prate protočnu obradu. Ograničenja u vezi su sa hazardima i dovode do pojave zastoja u radu protočnog sistema. Da bi uspešno rešili i ove probleme arhitekti računara se sve više odlučuju na uvodjenje paralelizma u obradi. U principu paralelni računari se mogu izvesti u sledeća dva oblika: a) **multiprocesori** -kod ovih sistema zadaci relativno velikog obima, kakve su procedure ili iterativne petlje, se izvršavaju paralelno; b) **paralelni procesori na nivou instrukcija** – u konkretnom slučaju individualne instrukcije se izvršavaju paralelno.

Procesori koji koriste paralelizam na nivou instrukcija (*Instruction Level Parallelism processors*– ***ILP procesori***) su daleko uspešniji u odnosu na multiprocesore, posebno na polju širokog tržišta radnih stanica i PC mašina, jer obezbedjuju značajno povećanje performansi kod izvršenja konvencionalnih programa. Naime, superskalarni procesori, kao tipični predstavnici ILP-procesora, ostvaruju relativno veliko ubrzanje u toku izvršenja programa, koji su kompajlirani za izvršenje na sekvencijalnim procesorima, bez da se izvrši njihova rekompilacija, tj. prevodjenje programa iznova.

Blok dijagram procesora koji koristi paralelizam na nivou instrukcija (*Instruction-Level Parallel Processor*- **ILP-P**) prikazan je na Slici 1. *ILP*-procesor poseduje veći broj funkcionalnih jedinica za izvršenje instrukcija, pri čemu svaka od njih čita svoje operande i upisuje svoje rezultate u jedinstveno, centralizovano *RF* polje(*Register File*). Kada operacija upiše rezultat u *RF* polje, taj rezultat, u narednom ciklusu, postoje vidljiv svim izvršnim jedinicama. Ovim se obezbedjuje da se operacije izvršavaju na različitim funkcionalnim jedinicama. Ovakvi procesori veoma često imaju ugradjeno složen **hardver za premošćavanje** (*hardware for results bypass*), koji sa ciljem da smanji kašnjenje izmedju zavisnih instrukcija, prosledjuje rezultate svake instrukcije svim izvršnim (funkcionalnim) jedinicama odmah nakon izračunavanja rezultata te instrukcije, tj odmah nakon faze *EX*.



Slika 1 *ILP-P*, Procesor koji koristi paralelizam na nivou instrukcija

Programskim instrukcijama manipuliše logika za iniciranje izvršenja instrukcija (*Instruction Issue Logic*-***III***), koja u jednom taktnom ciklusu može prema izvršnim jedinicama (*EX-U*) paralelno da inicira izvršenje većeg broja instrukcija. Na ovaj način obezbedjeno je da se promene toka upravljanja programom, kakve su recimo *Branch*, jave istovremeno u svim *EX-U*-jedinicama čime se olakšava kreiranje programa za *ILP*-procosore. Na Slici 1 sve *EX-U*-jedinice su prikazane kao identični moduli. U suštini, kod najvećeg broja *ILP*-procoesora, neke ili sve *EX-U*-jedinice su u stanju da izvršavaju podskup instrukcija procesora. Najčešća podela je na *integer* i *floating-point* jedinice, a razlog ovakvom pristupu je taj što se u suštini za izvršenje ovih operacija zahteva ugradnja raznorodnog hardvera. Implementacijom ova dva hardverska skupa kao posebne *EX-U*-jedinice povećava se broj instrukcija koje *ILP*-procesor može istovremeno da izvršava, ali bez da se pri tome značajno poveća obim hardver sistema. Kod drugih procesora postoji veći broj *integer EX-U*-jedinica koje su specijaliizovane za izvršenje samo neke od najčešće korišćenih operacija procesora kakve su *EX-U-MUL*, *EX-U-DIV*, *EX-U-ALU*, i druge.

1.1 Šta je to paralelizam na nivou instrukcija

ILP-procesori koriste činjenicu da najveći broj instrukcija u sekvencijalnom programu ne zavisi od instrukcija koje im u programskoj sekvenci neposredno prethode. Ilustracije radi analizirajmo program prikazan na levoj strani slike 2.

```
1. Ldd    r1, (r2)
2. Add    r5, r6, r7      ciklus 1: Ld, r1, (r2)       Add  r5, r6, r7
3. Sub    r4, r1, r4  ⇒   ciklus 2: Sub r4, r1, r4     Mul  r8, r9,r10
4. Mul    r8, r9, r10     ciklus 3: St  (r11), r4
5. St     r11), r4
```

Slika 2 Primer paralelizma na nivou instrukcija

Instrukcije 1, 3 i 5 su medjusobno zavisne. Instrukcija 1 generiše rezultat koji koristi instrukcija 3, a instrukcija 3 generiše rezultat za instrukciju 5. Instrukcije 2 i 4 ne koriste rezultate bilo koje instrukcije u datoj programskoj sekvenci, a takodje i rezultate koje one generišu ne koristi nijedna od instrukcija u sekvenci. Uočene zavisnosti izmedju instrukcija nalažu da se instrukcije 1, 3 i 5, sa ciljem da se generiše korektni rezultat, moraju da izvrše po redosledu, dok se instrukcije 2 i 4 mogu izvršavati pre, posle, ili paralelno sa bilo kojom drugom instrukcijom, bez opasnosti da se promeni rezultat programske sekvence.

Ako usvojimo da je procesor u stanju da izvršava po dve instrukcije istovremeno i da je latencija svake od instrukcija jedan ciklus programska sekvenca sa slike 2 levo izvršiće se za tri ciklusa (slika 2 desno). S obzirom da su instrukcije 1,3 i 5 medjusobno zavisne nije dalje moguće redukovati vreme izvršenja programske sekvence putem povećanja broja instrukcija koje procesor može paralelno, tj istovrmeno, da izvršava.

Navedeni primer ilustruje kako prednosti tako i nedostatke *ILP*-a. Naime, *ILP*-procesori mogu ostvariti značajno ubrzanje za veliki broj programa na taj način što će instrukcije izvršavati paralelno, ali su njihova maksimalna performansna poboljšanja ograničena zavisnošću izmedju instrukcija. U principu, ugradnjom novih *EX-U*-jedinica povećavaju se performanse, no kada se njihov broj poveća na 4, 8 ili više, najveći broj *EX-U*-jedinica postaje pasivan jer kompilatori nisu u stanju, prvenstveno zbog osobine programa, da izvlače značajan paralelizam u radu programa.

1.2 Ograničenja ILP-a

Performanse *ILP*-procesora su ograničene iznosom *ILP*-a koje kompilator i hardver mogu da lociraju u programu. *ILP* ograničavaju sledeća nekoliko faktora: zavisnosti po podacima (*RAW* hazardi), zavisnosti po imenu (*WAR* i *WAW* hazardi), i grananja. Pored toga sposobnost procesora da koristi *ILP* može biti ograničena kako brojem tako i tipom *EX*-*U*-jedinica ugradjenih u procesor, kao i restrikcijama programa koje se odnose na lociranje (identifikaciju) operacija koje se mogu izvršavati paralelno.

RAW zavisnosti ograničavaju performanse na taj način što zahtevaju da se instrukcije izvršavaju sekvencijalno, kako bi se generisali korektni rezultati. U suštini *RAW* zavisnosti predstavljaju fundamentalno ograničenje koje se odnosi na izvlačenje maksimalnog iznosa *ILP*-a u dostupnim programima.

Instrukcije izmedju kojih postoje *WAW* zavisnosti moraju se, u principu, takodje inicirati sekvencijalno, kako bi se na kraju programske sekvence ipak obezbedilo da u odredišni registar bude upisana korektna vrednost.

Instrukcije izmedju kojih postoje *WAR* zavisnosti mogu da se iniciraju u istom ciklusu, ali ne van redosleda, jer instrukcije čitaju svoje izvorišne oprande iz *RF* poja pre nego što se vrši njihovo iniciranje izvršenja. To znači da instrukcija koja čita sadržaj izvorišnog registra može da se inicira radi izvršenja u istom ciklusu kao i instrukcija koja upisuje rezultat svog izračunavanja u odredišni registar ali se javlja kasnije programu. Naime, instrukcija koja obavlja čitanje pročitaće ulazni registar (izvorišni operand) pre nego instrukcija koja vrši upis generiše novu vrednost i smesti je u odredišni registar. Nešto kasnije ukazaćemo na tehniku preimenovanja registara (*register renaming*), koja je u suštini hardverska tehnika, a obezbedjuje nam da se instrukcije izmedju kojih postoje *WAR* i *WAW* zavisnosti mogu izvršavati van redosleda bez opasnosti da dodje do promene rezultata u programu.

Instrukcije tipa *Branch* takodje ograničavaju *ILP* jer procesor ne zna koje će se instrukcije izvršiti nakon instrukcije *Branch* sve dok se dileme oko *Branch*-a ne razreše. Ovo zahteva od procesora da čeka sve dok instrukcija *Branch* ne završi sa izvršenjem, kako bi bio u stanju da izvršava instrukciju koja sledi nakon instrukcije *Branch.* Veliki broj procesora ima ugradjeno hardver za predikciju grananja sa ciljem da smanji uticaj instrukcije grananja na vreme izvršenja. Naime ova logika vrši predikciju odredišne adrese pe nego što se instrukcija *Branch* izvrši.

Primer1

a) Analizirajmo sledeći programski segment

```
Add    r1, r2, r3
Ld     r4, (r5)
Sub    r7, r1, r9
Mul    r5, r4, r4
Sub    r1, r12, r10
St     (r13), r14
Or     r15, r14, r12
```

Koliko dugo traje iniciranje izvršenja instrukcija na procesoru koji može da izvršava po dve instrukcije istovremeno?

b) Kakva je situacija sa procesorom koji može da izvršava po četiri instrukcije istovremeno?

Napomena: Pretpostavimo da procesor može da izvršava instrukcije u bilo kom redosledu, a da pri tome ne naruši zavisnosti po podacima, da sve instrukcije imaju latentnost od jednog ciklusa, i da sve *EX_U* jedinice procesora mogu da izvrše bilo koju od instrukcija u programskom segmentu.

Odgovor

Na procesoru koji omogućava da se po dve instrukcije izvršavaju istovremeno, iniciranje izvršenje programa trajaće četiri ciklusa. Jedna od sekvenci, napomenimo da ih ima veći broj, je oblika:

```
Ciklus 1: Add  r1, r2, r3     Ld   r4, (r5)
Ciklus 2: Sub  r7, r1, r9     Mul  r5, r4, r4
Ciklus 3: Sub  r1, r12, r10   St   (r13), r14
Ciklus 4: Or   r15, r14, r12
```

U slučaju da procesor može da izvršava po četiri instrukcije istovremeno, iniciranje izvršenja programa se može obaviti u dva ciklusa na sledeći način:

```
Ciklus 1: Add  r1,r2,r3  Ld r4,(r5)   St (r13),r14    Or r15,r14,r12
Ciklus 2: Sub  r7,r1,r9  Mul r5,r4,r4 Sub r1,r12,r10
```

Naglasimo da nezavisno od broja instrukcija koje procesor može istovremeno da izvršava, nije moguće inicirati izvršenje ove programske sekvence u jednom ciklusu, iz prostog razloga što postoje *RAW* zavisnosti izmedju instrukcija `Add r1,r2,r3` i `Sub r7,r1,r9`, kao i izmedju inststrukcija `Ld r4,(r5)` i `Mul r5,r4,r4`. Takodje ukažimo da izmedju instrukcija `Sub r7,r1,r9` i `Sub r1,r12,r10` postoji *WAR* zavisnost, ali se obe instrukcije iniciraju radi izvršenja u istom ciklusu

1.3 Superskalarni procesori

Za izvlačenje *ILP*-a iz sekvencijalnih programa superskalarni procesori koriste hardver. U toku svakog ciklusa, logika superskalarnog procesora za iniciranje izvršenja instrukcija ispituje instrukcije sekvencijalnog programa kako bi odredila za koju instrukciju iniciranje izvrešenja u tom ciklusu može da počne. Ako u okviru programa postoji dovoljan *ILP*, superskalarni procesor može da izvršava po jednu instrukciju u ciklusu u svakoj od *EX_U* jedinica, čak i u slučaju da je program bio kompajliran za izvršenje na procesoru koji može da izvršava samo jednu instrukciju po ciklusu.

Ova mogućnost predstavlja suštinska prednost superskalarnog procesora i tu treba tražiti razlog zbog čega su skoro svi CPU-ovi radnih stanica kao i PC mašine izvedeni sa superskalarnim procesorima. Superskalarni procesori mogu da izvršavaju programe koji su prvensteno kompajlirani za sekvencijalne procesore, ali pri tome mogu postići bolje performanse u toku izvršenja ovih programa od onih procesora koji nisu u stanju da iz programa izvlače *ILP*. Na ovaj način, korisnici koji kupuju sisteme sa ugradjenim superskalrnim CPU-ima mogu da instaliraju na ovakvim sistemima svoje stare programe i da u toku njihovog izvršenja ostvare bolje performanse u odnosu na stare sisteme.

Mogućnost superskalarnih procesora da eksploatišu *ILP* sekvencijalnih programa ne ukazuje automatski da su kompajleri ti koji su jedino irelevantni za postizanje boljih performansi. Organizacija hardvera mašine je isto tako važna. No treba ipak istaći da su dobri kompilatori najkritičniji sa aspekta performansi superskalarnih procesora, u odnosu na procesore sa strogim sekvencijalnim izvršenjem.

Superskalani procesori, u datom trenutku izvršenja programa ispituju samo jedan mali deo prozora instrukcija sa ciljem da odrede koje od instrukcija mogu da se izvršavaju paralelno. Ako je kompilator u

stanju da u okviru prozora isplanira izvršenje velikog broja instrukcija tada superskalarni procesor može da postigne dobre performanse. U slučaju kada, u datom trenutku u okviru prozora izmedju instrukcija postoje medjusobne zavisnosti, superskalarni procesor neće biti u stanju da izvrši program znatno brže u odnosu na sekvencijalni procesor. Nešto kasnije, ukazaćemo na tehnike koje kompilator koristi radi poboljšanja performansi programa kod superskalarnih procesora.

1.3.1 Redosledno u odnosu na van-redosledno izvršenja

Jedan od najvažnijih kompromisa koji se tiče odnosa **složenost / performansi**, a treba da se donese i reši u toku faze projektovanja superskalarnog procesora odnosi se na sledeću činjenicu: Da li se od procesora zahteva da izvršava instrukcije po redosledu kako se one pojavljuju u programu (*in-order execution*), ili se zahteva da procesor može da izvršava instrukcije u bilo kom redosledu (tj. van-redosleda), ali da pri tome ne promeni kranji rezultat programa (ovakav redosled se naziva *out-of-order execution*). U principu van-redoslednim izvršenjem, u odnosu na izvršenje-po-redosledu, mogu se ostvariti znatno bolje performanse, ali po ceni povećanja kompleksnosti implementiranog hardvera.

1.3.2 Predikcija izvršenja instrukcija kod *in-order* procesora

Vreme izvršenja programa kod klasičnih protočnih procesora (tzv. skalarni procesori) se odredjuju na osnovu sledeće relacije:

Vreme_izvršenja (u ciklusima) =
= protočna_latentnost + vreme_iniciranja_izvršenja_instrukcija – 1

tj.

$$T_{EX} = T_{PL} + T_{IS} - 1$$

gde je: T_{EX} - vreme izvršenja; T_{PL} - protočna latentnost; T_{IS} - vreme iniciranja izvršenja instrukcija instrukcija

Kod protočnih *ILP* procesora vreme T_{EX} se odredjuje na isti način kao i kod klasičnih protočnih procesora, sa tom razlikom što izračunavanje T_{IS} postaje znatno složenije jer procesor može da inicira izvršenje većeg broja instrukcija u jednom ciklusu. Sa druge strane, vreme T_{PL} ,od jednog programa do drugog se ne menja značajno pa zbog toga celokupnu našu dalju pažnju usmerićemo ka odredjivanju T_{IS} kod *ILP* procesrora. Kod *in-order* superskalarnih procesora vreme T_{IS} se može odrediti sekvenciranjem programskog kada na principu korak-po-korak i odredjivanjem trenutka kada svaka instrukcija može da započne sa izvršenjem, slično tehnici koja se koristi kod protočnih procesora koji izvršavaju jednu instrukciju po taktnom ciklusu, tj. skalarnih procesora. Ključna razlika izmedju *in-order* superskalarnih i skalarnih procesora je ta što superskalarni procesor može da inicira izvršenje većeg broja instrukcija po taktnom ciklusu, pod uslovom da zavisnosti po podacima to dozvoljavaju i da broj instrukcija ne premaši broj raspoloživih *EX-U* jedinica.

Primer 2

In-order procesor ima dve *EX-U* jedinice pri čemu svaka može da izvši bilo koju instrukciju. Odrediti vreme izvršenja sledeće sekvence u ciklusima

```
Ld      r1, (r2)
Add     r3, r1, r4
Sub     r5, r6, r7
Mul     r8, r9, r10
```

Neka vreme latencije instrukcije `Ld` iznosi dva ciklusa, a latentnost svih ostalih operacija je jedan ciklus. Usvojiti da je protočni sistem dubine pet, i da ga čine sledeći stepeni, *FI*, *DI*, *RR*, *EX* i *WB*.

Odgovor

T_{LP} ovog procesora iznosi $5T_{CLK}$, gde je T_{CLK} vreme procesorskog ciklusa.

Usvojimo da se iniciranje izvršenja instrukcije `Ld` vrši u ciklusu *n*, i da se izvršenje instrukcije `Add` ne može izvršiti pre ciklusa *n+2*, jer `Add` zavisi od `Ld`. Iinstrukcija `Sub` ne zavisi od `Add` i `Ld` pa se njeno izvršenje može vršiti u ciklusu *n+2*. (Izvršenje instrukcije `Sub` se ne može vršiti u ciklusu *n* ili *n+1* jer procesor mora da inicira izvršenje instrukcija po redosledu). Instrukcija `Mul` ne zavisi od prethodnih instrukcija, ali na iniciranje izvršenja mora da čeka do ciklusa *n+3*, jer procesor može u jednom taktnom ciklusu da inicira izvršenje samo po dve instrukcije. Zbog čega, potrebna su četiri ciklusa za iniciranje izvršenja svih instrukcija u programu, tako da je ukupno vreme izvrešenja

$T_{EX} = 5 + 4 - 1$
$= 8$ ciklusa

1.3.3 Predikcija vremena izvršenja kod *out-of-order* procesora

Odredjivanje T_{IS}, za sekvencu instrukcija, kod *out-of-order* procesora je znatno teže od odredjivanja T_{IS} kod *in-order* procesora jer postoji veći broj potencijalnih redosleda po kojima se instrukcije mogu izvršavati. U opštem slučaju, najbolji pristup je onaj koji počinje sa analizom sekvence instrukcija i lociranjem zavisnosti koje postoje izmedju njih. Nakon što su zavisnosti izmedju instrukcija identifikovane instrukcije se dodeljuju ciklusima za iniciranje izvršenja sa ciljem da se minimizira kašnjenje izmedju prve i zadnje instrukcije u sekvenci.

Napori koji se čine da se odredi najbolji mogući redosled skupa instrukcija esponencijalno se povećava sa brojem instrukcija u skupu, jer je potrebno razmatrati sve moguće potencijalne reoslede izvršenja. Sa ciljem da se smanji složenost logike za iniciranje izvršenja instrukcije postavljaju se neka ograničenja. Ograničenja se pre svega odnose na to da logika inicira izvršenje instrukcija po redosledu. Pretpostavku koju činimo je sledeća: Procesor inicira izvršenje instrukcije odmah (sa prvim ciklusom) kada zavisnosti u okviru programa to dozvole. Ako je u okviru ciklusa moguće inicirati izvršenje većeg broja instrukcija i procesor poseduje dovoljan broj *EX-U* jedinica, procesor koristi gramžljiv (*greedy*) pristup i inicira izvršenje onih instrukcija koje se u programu javljaju ranije. U suštini kompilator razmatra (analizira) sve moguće redoslede izvršenja instrukcija i odredjuje redosled čije je vreme izvršenja najkraće (kompilator je u stanju da posveti mnogo više pažnje i napora kako bi obavio analizu planiranja izvršenja instrukcija u odnosu na logiku za iniciranje izvršenja instrukcija). Koristeći gramžljivi (pohlepan) pristup u postupku iniciranju izvršenja sekvence instrukcija kod *out-of-order* procesora mnogo je lakše odrediti vrreme T_{IS} .

Naime, počinje se prvom instrukcijom u sekvenci, zatim se ide na naredne, pri čemu se svaka instrukcija ako ima dostupne sve svoje operande dodeljuje na izvršenje odmah u narednom ciklusu. Broj instrukcija koje se u tekućem ciklusu dodeljuju radi izvršenja treba: (a) da je manji ili jednak broju instrukcija čije izvršenje procesor može istovremeno da inicira; (b) broj instrukcija koje se iniciraju ne sme da premaši broj dostupnih *EX-U* jedinica procesora. Ponavljajući ovaj proces za sve instrukcije u sekvenci moguće je odrediti T_{IS}.

Primer 3

Odrediti T_{IS} sledeće programske sekvence kod *out-of-order* procesora koji ima dve *EX-U* jedinice. Svaka *EX-U* jedinica može da izvrši bilo koju od operacija. Latencija operacije `Ld` je dva ciklusa, a ostalih operacija jedan ciklus.

```
Ld     r1, (r2)
Add    r3, r1, r4
Sub    r5, r6, r7
Mul    r8, r9, r10
```

Odgovor

Sekvenca je identična kao ona u Primeru 2 koji se odnosi na procesor koji ima *in-order* iniciranje izvršenja instrukcija.

Jedina zavisnost u ovoj sekvenci (*RAW* zavisnost) postoji izmedju instrukcija `Ld` i `Add`. Zbog ove zavisnosti instrukcija `Add` mora da se inicira radi izvršenja najmanje dva ciklusa nakon `Ld`. Obe instrukcije, `Sub` i `Mul`, moguće je inicirati u istom ciklusu kao i `Ld`. Koristeći gramžljiv (pohlepan) pristup, u ciklusu *n* se iniciraju `Sub` i `Ld`, u ciklusu *n+1* se inicira `Mul`, dok se `Add` inicira u ciklusu *n+2*. To znači da je ukupno potrebno vreme od tri ciklusa za iniciranje, dok je vreme izvršenja

$$\begin{aligned} T_{EX} &= T_{LP} + T_{IS} - 1 \\ &= 5+3-1 \\ &= 7 \text{ ciklusa} \end{aligned}$$

1.3.4 Implementacija iniciranja izvršenja kod *out-of-order* procesora

Kod ***in-order* procesora**, instrukcioni prozor (broj instrukcija koje procesor ispituje kako bi selektovao koju od instrukcija da inicira radi izvršenja u svakom ciklusu) može biti relativno mali, jer se procesoru ne dozvoljava da inicira izvršenje instrukcije sve dok se ne iniciraju sve prethodne koje se javljaju ranije u programu. Za procesor sa *n EX-U* jedinica, samo *n* narednih programskih instrukcija je moguće inicirati u narednom ciklusu, tako da je instrukcioni prozor dužine *n*, u opštem slučaju, dovoljan.

***Out-of-order* procesori** zahtevaju znatno veći instrukcioni prozor u odnosu na *in-order* procesore. Veći instrukcioni prozor pruža znatno veću mogućnost kod odredjivanja koju će od instrukcija inicirati u datom ciklusu. Ipak treba naglasiti da se logika prozora usložnjava sa kvadratom broja instrukcija koje se čuvaju u instrukcionom prozoru. Naime, svaka instrukcija u prozoru mora da se uporedjuje sa svim ostalim

instrukcijama kako bi se odredile zavisnosti izmedju njih. Ovo čini da implementacija instrukcijonog prozora, sa aspekta iznosa ugradjenog hardvera, bude skupa. Proceduru na koju smo već ukazali, a odnosi se na odredjivanje vremena izvršenja instrukcione sekvence kod *out-of-order* procesora, predpostavlja da je instrukcioni prozor procesora dovoljno veliki i da obezbedjuje procesoru mogućnost da istovremeno ispita sve instrukcije u sekvenci. Ako ovo nije slučaj, predikcija vremena izvršenja postaje veoma teška, jer postaje neophodno potrebno čuvati trag o tome koje se od instrukcija u datom ciklusu nalaze u instrukcionom prozoru, a selektovati radi izvršenja samo one instrukcije koje pripadaju tom skupu.

Obrada prekida i programskih izuzetaka unosi dodatnu implementacionu teškoću kod *out-of-order* procesora. Ako se instrukcije mogu izvršavati van redosleda, veoma je teško tačno odrediti koju od instrukcija treba izvršiti kada se u toku izvršenja instrukcije javi izuzetak ili prekid. Šta više ova situacija predstavlja teškoću čak i za programera kada on treba da odredi uzrok generisanja izuzetka. Takodje ovo predstavlja i problem sistemu jer je sada potrebno da se sistem ponovo vrati na izvršenje osnovnog (prekinutog) programa nakon završetka rutine za obradu prekida.

Da bi izašli na kraj sa ovim problemom, skoro svi *out-of-order* procesori koriste tehniku nazvanu povlačenje-po-redosledu (*in-order retirement*). Kada instrukcija generiše rezultat, rezultat će se upisati u *RF* polje samo ako su u programu završene sve prethodne instrukcije. Inače, rezultat se pamti (čuva) sve dok sve instrukcije koje prethode ne završe, pa se tek nakon toga rezultati upisuju u *RF* polje. S obzirom da se rezultati u *RF* polje upisuju po redosledu, hardver može na veoma jednostavan način da izbaci (obriše) sve rezultate koji čekaju na upis u *RF* polje, u slučaju kada se javi izuzetak ili generiše prekid. Na ovaj način se stiče iluzija da se instrukcije izvršavaju po redosledu, a to omogućava programeru na relativno lak način da otkloni greške u programu, i obezbedi nastavak programa od naredne instrukcije u slučaju kada završi rutina za obradu prekida. Procesori koji koriste ovu tehniku, u opštem slučaju, imaju ugradjenu logiku za premošćavanje (*bypassing logic*), ili nekih drugih tehnika za prosledjivanje rezultata instrukcije unapred (*forwarding the result of instruction*) zavisnim instrukcijama pre nego što se rezultat upiše u *RF* polje. Na ovaj način zavisnim instrukcijama je obezbedjeno iniciranje izvršenja odmah nakon što instrukcija generiše rezultat, bez da se čeka da se rezultat instrukcije upiše u *RF* polje u protočnoj fazi obrade *Write-Back* (*WB*).

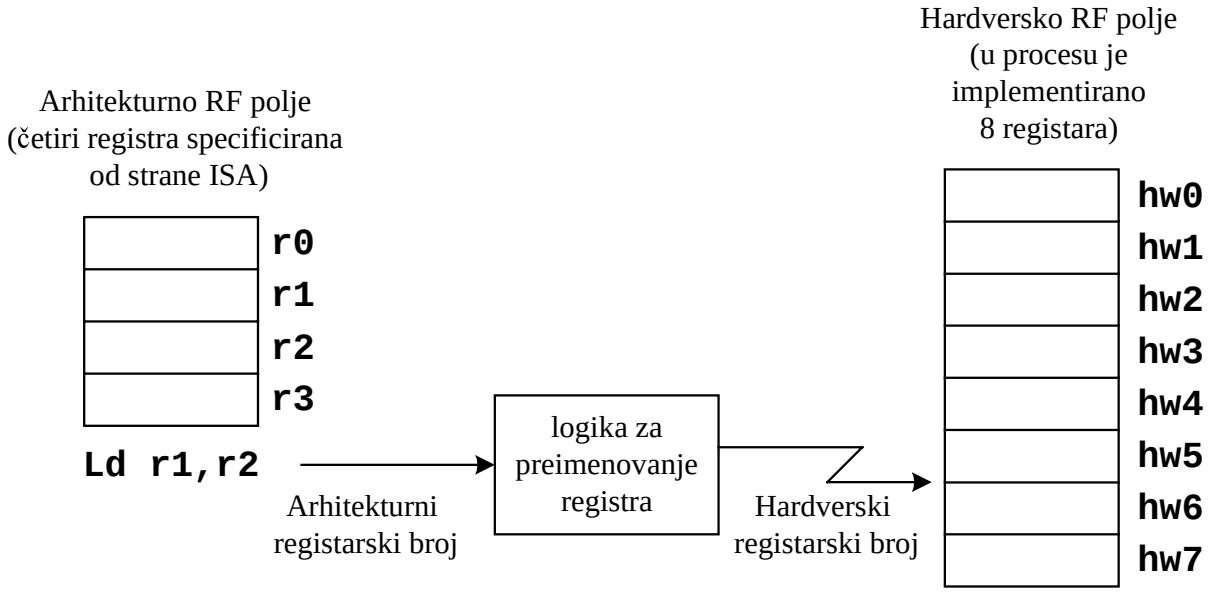
1.3.5 Preimenovanje registara

WAR i *WAW* zavisnosti često se alternativno nazivaju *name dependencies*. Ove zavisnosti rezultat su činjenice da su programi veoma često prisiljeni da intenzivno koriste registre. Imajući u vidu da je broj registara u *RF* polju ograničen, često se za jedan kratak period jedan te isti registar koristi za više namena. WAR i WAW zavisnosti mogu kod superskalarnih procesora da ograniče *ILP*, jer je potrebno obezbediti da sve instrukcije koje čitaju sadržaj registra završe sa operacijom čitanja (aktivnost u protočnom stepenu RR) pre nego što bilo koja druga instrukcija upiše novu (*overwrites*) vrednost u taj registar.

Preimenovanje registara (*register renaming*) je tehnika kojom se smanjuje uticaj *WAR* i *WAW* zavisnosti na ostvarivanje paralelizma u toku rada, putem dinamičke dodele svake programski generisane vrednosti novom registru, otklanjajući na taj način uticaj *WAR* i *WAW* zavisnosti. Tehnika preimenovanja registra je prikazana na slici 3. Svaki skup instrukcija poseduje svoje arhitekturno *RF* polje, koje predstavlja skup registara koje taj skup instrukcija koristi. Sve instrukcije specificiraju svoje ulaze i izlaze van arhitekturnog *RF* polja. Kada je u pitanju procesor, obimnije *RF* polje, poznato kao hardversko *RF* polje, implemenntira se umesto arhitkturnog *RF* polja. Logika za primenovanje prati preslikavanje izmedju registara arhitekturnog i registara hardverskog *RF* polja.

Uvek kada instrukcija čita registar arhitekturnog *RF* polja, identifikator (*ID*) registra se predaje preko logike za preimenovanje kako bi se odredilo kom se od registara hardverskog *RF* polja pristupa. Kada instrukcija upisuje u arhitekturno *RF* polje, logika za preimenovanje kreira novo preslikavanje izmedju arhitekturnog registra u kome se upisuje i registra koji pripada hardverskom *RF* polju. Instrukcije koje kao

naredne slede u programu i čitaju arhitekturni registar pristupaju novom hardverskom registru i vide rezultat te instrukcije.



Slika 3 Preimenovanje registara

Na slici 4 prikazano je na koji način tehnika za preimenovanje registara može da poboljša performanse. U početnom (pre preimenovanja) programu, postoji *WAR* zavisnost izmedju instrukcija `Ld r7,(r3)` i `Sub r3,r12,r11`. Kombinacija od *RAW* i *WAR* zavisnosti koje postoje u programu prisiljavaju program sa slike 4 da potroši najmanje tri ciklusa na iniciranje izvršenja instrukcija, jer `Ld` mora da se inicira nakon `Add`, dok se `Sub` ne može inicirati pre `Ld`, a `St` se ne može inicirati sve dok ne završi `Sub`.

pre preimenovanja	**nakon preimenovanja**
`Add r3, r4, r5`	`Add hw3, hw4, hw5`
`Ld r7, (r3)`	`Ld hw7, (hw3)`
`Sub r3, r12, r11`	`Sub hw20, hw12, hw11`
`St (r15), r3`	`St hw15, hw10`

Slika 4 Primer primenovanja registara

Preimenovanjem registara, prvi upis u `r3` se preslikava u hardverski registar `hw3`, dok se drugi upis preslikava u `hw20`. Ovakvim preslikavanjem konvertuje se početno zadati četvoro-instrukcijski lanac (leva strana slike 4) u dvo-instrukcijski lanac (desna strana slike 4), koji se može izvršavati paralelno ako procesor dozvoljava *out-of-order* izvršenje. U opštem slučaju, preimenovanje registra kao tehnika je od veće koristi za *out-of-order* nego za *in-order* procesore, jer procesoru tipa *out-of-order* mogu da preurede redosled izvršenja instrukcija nakon što se tehnikom preimeovanja registara razbiju *name dependencies* (*WAR* i *WAW* zavisnosti).

Primer 4

Za *out-of-order* superskalarni procesor sa 8 *EX-U* jedinica odrediti vreme izvršenja sledeće programske sekvence sa i bez preimenovanja registara, ako bilo koja od *EX-U* jedinica može da izvrši bilo koju instrukciju, a latentnost svih instrukcija je jedan ciklus. Usvojiti da hardversko *RF* polje sadrži dovoljan broj registara za preslikavanje svakog odredišnog registra u različiti hardverski registar, i da je protočni sistem dubine

```
Ld   r7, (r8)
Mul  r1, r7, r2
Sub  r7, r4, r5
Add  r9, r7, r8
Ld   r8, (r12)
Div  r10, r8, r10
```

Odgovor

U konkretnom slučaju, *WAR* zavisnosti predstavljaju obziljno ograničenje koje se odnosi na paralelizam, prisiljavajući instrukciju `Div` da se inicira radi izvršenja tri ciklusa nakon prve `Ld`, tako da je ukupno vreme izvršenja 8 ciklusa (instrukcije `Mul` i `Sub` mogu da se izvršavaju paralelno, kakav je slučaj sa `Add` i drugom `Ld`). Nakon preimenovanja registara, oblik programa je sedeći:

```
Ld   hw7, (hw8)
Mul  hw1, hw7 hw2
Sub  hw17, hw4, hw5
Add  hw9, hw17, hw8
Ld   hw18, (hw12)
Div  hw10, hw18, hw10
```

Korišćenjem tehnike preimenovanja registra, program se razbija na tri skupa sa po dve zavisne instrukcije (`Ld` i `Mul`, `Sub` i `Add`, i `Ld` i `Div`). Instrukcija `Sub` i druga `Ld` mogu sada da se iniciraju radi izvršenja u istom ciklusu kao i prva `Ld`. Instrukcije `Mul`, `Add` i `Div` mogu da se iniciraju radi izvršenja u narednom ciklusu, što rezultuje ukupnom vremenu izvršenja od 6 ciklusa.

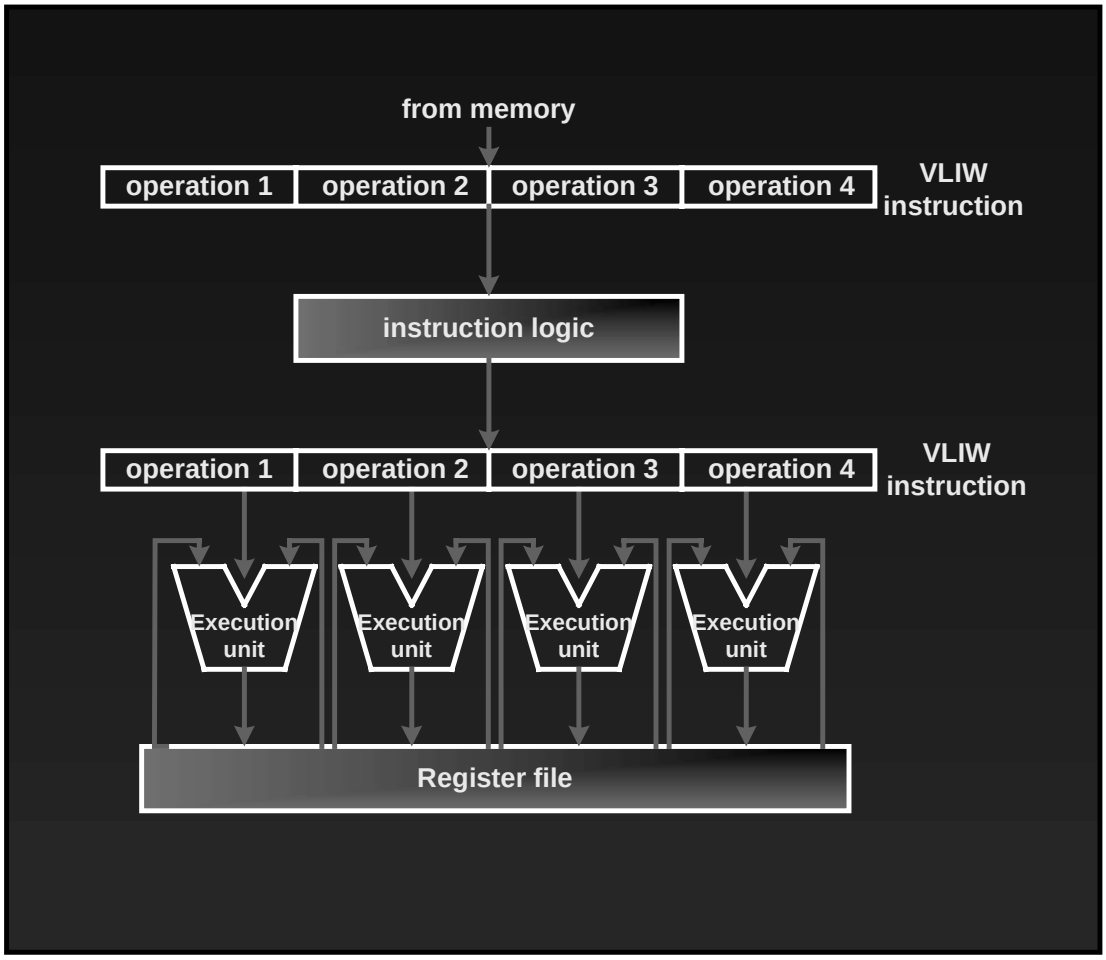
Implementacijom tehnike preimenovanja registara u principu se ostvaruju manja poboljšanja u odnosu na slučaj kada bi se promenila *ISA* i učinilo da novi registri budu sastavni deo arhitekturnih registara, iz prostog razloga što kompilator ne može da korosti nove registre za memorisanje privremenih vrednosti u programu (hardverski registri jesu sastavni deo procesora ali nisu vidljivi kompilatoru, tj. kompilatoru su vidljivi samo arhitekutrni a ne i hardverski registri). Ipak, tehnika preimenovanje registara omogućava novim procesorima da sačuvaju kompatibilnost sa programima kompajliranim za starije verzije procesora jer ova thenika ne zahteva promenu *ISA*. Pri ovome treba odmah naglasiti da se sa povećanjem broja arhitekturnih registara u procesoru povećava i broj bitova potreban svakoj instrukciji za kodiranje registra, tj. potreban je sada veći broj bitova za kodiranje kako izvorišnih operanada tako i odredišnog registra.

1.4 VLIW procesori

Superskalarni procesori o kojima smo do sada diskutovali koriste hardver za eksploataciju *ILP*-a. Ovi procesori lociraju instrukcije koje se u sekvencijalnim programima mogu paralelno izvršavati. Mogućnost da ostvare performansna poboljšanja nad starim programima i očuvaju kompatibilnost izmedju različitih verzija procesora u okviru iste familije, učinila je superskalarne procesore, sa ekonomske tačke gledišta, veoma uspešne. Ipak, dalje povećanje performansi superskalarnih procesora zahteva značajno usložnjavanje hardvera. Sa druge strane ***VLIW*** (*Very Long Instrucition Word*) **procesori** eksploatišu *ILP* na jedan drugačiji način u odnosu na superskalarne procesore. Naime, ovi procesori se oslanjaju na pomoć kompilatora, koji odredjuje koja se od instrukcija može izvršavati paralelno i predajom te informacije hardveru.

Kod *VLIW* procesora, svaka instrukcija specificira nekoliko nezavisnih operacija koje se paralelno izvršavaju od strane hardvera, kako je to prikazano na slikama 5 i 6.

Slika 5 Jedan od formata *VLIW* instrukcije



Slika 6 Klasična arhitektura *VLIW* procesora

Svaka operacija kod *VLIW* instrukcije ekvivalentna je jednoj instrukciji kod superskalarnog ili procesora sa strogim sekvencijalnim izvršenjem instrukcija (skalarnog procesora). Broj operacija kod *VLIW* instrukcije jednak je broju izvršnih jedinica procesora. Svaka od operacija specificira instrukciju koja će se izvršiti na odgovarajućoj *EX-U* jedinici, u ciklusu u kome se inicira izvršenje *VLIW* instrukcije. Ne postoji potreba za ugradnjom hardvera koji ispituje koja se od instrukcija može izvršavati paralelno, jer je kompilator taj koji je zadužen da obezbedi da se sve operacije u instrukciji mogu izvršavati simultano. Imiajući u vidu ovu činjenicu, logika za iniciranje izvršenja instrukcija kod *VLIW* procesora je značajno jednostavnija u odnosu na logiku koja obavlja istu funkciju kod superskalarnog procesora sa istim brojem *EX-U* jedinica.

Najveći broj *VLIW* procesora u svojim *RF* poljima nema ugradjeno *scoreboard*-ove. Umesto toga, kompilator je taj koji je odgovoran i obezbedjuje da se iniciranje izvršenja instrukcije neće obaviti sve dok operandi instrukcije ne budu spremni. U toku svakog ciklusa, instrukciona logika pribavlja *VLIW* instrukciju iz memorije i inicira izvršenje aktiviranjem *EX-U* jedinica. U suštini, kompilator može da predvidi koliko tačno ciklusa je potrebno da prodje izmedju izvršenja dve operacije u programskoj sekvenci, putem brojanja broja *VLIW* instrukcija izmedju te dve operacije. Pored toga, kompilator može da planira van-redosledno izvršenje instrukcija izmedju kojih postoji *WAR* zavisnost onoliko dugo sve dok instrukcija koja obavlja čitanje registra završi pre instrukcije koja upisuje u registar, tj. preko stare vrednosti koja se čuva u registar se ne dopisuje nova vrednost sve dok se ne završi instrukcija koja prethodno treba da obavi operaciju upis. Tako na primer kod *VLIW* procesora kod koga je latencija usled punjenja (*load*) dva-ciklusa, sekvenca `Add r1,r2,r3` i `Ld r2,(r4)` biće planirana radi izvršenja tako da se operacija `Add` javi u instrukciji nakon operacije, punjenja (*load*), s obzirom da punjenje neće dopisati novu vrednost u `r2` sve dok ne protektnu dva ciklusa.

1.3.1 Za i protiv VLIW

Glavne prednosti *VLIW* arhitekture su sledeće:

a) jednostavnost instrukcione logike koja, u odnosu na superskalarne procesore, omogućava rad pri višim frekvencijama; i

b) b) kompilator je taj koji ima potpunu kontrolu nad izvršenjem operacija.

U opštem slučaju, kompilator ima širi i bolji pregled programa nad izvršenjem operacija u odnosu na pogled koji ima instrukciona logika kod superskalarnog procesora. Imajući u vidu ovu činjenicu efikasnost pronalaženja operacija koje se mogu izvršavati paralelno kod *VLIW* procesora je bolja. Jednostavnija instrukciona logika *VLIW* procesora doztvoljava da se na istoj površini silicijuma ugradi veći broj EX-U jedinica, u poredjenju sa superskalarnim procesorom.

Najznačajniji nedostatak *VLIW* procesora je taj da *VLIW* programi isključivo rade korektno samo kada se izvršavaju na procesoru sa istim brojem *EX-U* jedinica i istom instrukcionom latentnošću kao i na procesoru za koga je vršena kompilacija, što čini takoreći nemoguće da se očuva kompatibilnost izmedju generacija procesora u familiji. Kada se broj *EX-U* jedinica kod naredne generacije procesora iz iste familije poveća, novi proceosr će pokušati da kombinuje u istom ciklusu operacije od većeg broja instrukcija, potencijalno uzrokujući da se zavisne instrukcije izvršavaju u istom ciklusu. Razlike u instrukcionim latentnostima koje postoje izmedju generacija procesora iz iste familije uzrokuju da se operacije izvršavaju pre nego što su njihovi ulazi dostupni (spremni) ili se preko ulaznih vrednosti upisuju nove vrednosti (*overwrite*), što rezultira nekorektnom ponašanju programa. Pored ostalog, kada kompilator ne može da pronadje dovoljno

paralelnih operacija kako bi popunio slotove u instrukciji, on mora eksplicitno da popuni odgovarajuće slotove operacija operacijama tipa *Nop*. Ovo ima za posledicu da *VLIW* programi zauzimaju znatno veći prostor u memoriji u odnosu na ekvivalentne programe koji se izvršavaju na superskalarnim procesorima. Imajući u vidu nabrojane prednosti i nedostatke, *VLIW* procesori češće se koriste kod aplikacija tipa *DSP*, gde su visoke performanse i niska cena imperativ. Danas se, *VLIW* procesori sa znatno manje uspeha koriste kod procesora opšte namene kakve su radne stanice i PC mašine, jer za korisnika je od vitalne važnosti očuvanje kompatibilnost softvera izmedju generacija procesora.

Primer 5

Dati odgovor na sledeća pitanja:

a) *VLIW* je arhitekturna ideja kojom se efikasno eksploatiše kôd (program) kod koga je korektno od strane kompilatora obavljeno planiranje-izvršenja-instrukcija (*instruction scheduling*)?

b) Kod VLIW arhitekture postoji nekoliko izvršnih jedinica koje mogu raditi konkurentno?

c) Instrukcionu reč *VLIW* arhitekture čine operacije za svaku od izvršnih funkcionalnih jedinica pri čemu se operacije u okviru instrukcije mogu konkurentno pasporedjivati radi izvršenja (*dispatched*)?

d) kod *VLIW* arhitektura kompilator ima zadatak da garantuje da će se sve operacije u okviru jedinstvene instrukcije istovremeno izvršavati?

e) Kod *VLIW* arhitektura svi operandi, za sve operacije u okviru instrukcije u trenutku kada se instrukcije rasporedjuju radi izvršenja, treba da su dostupni u registrima.

f) Kod prave *VLIW* implementacije, kompilator planira izvršenje instrukcija kako bi ostvario zahteve definisane pod stavkom e), dok hardver treba da obavi sledeće zadatke: (1) odredi koje operacije se mogu rasporedjivati radi konkurentnog izvršenja; (2) zabrani rasporedjivanje izvršenja sve dok svi operandi nisu dostupni u registrima.

g) Jedan od problema na koje se nailazi kod *VLIW* mašina odnosi se na nekompatibilnost kôda (programa) kako naviše tako i naniže, za različite verzije *VLIW* arhitekture.

h) Tokom vremena *VLIW* arhitektura može da se usavršava. Tako na primer, da se ugradi veći broj funkcionalnih izvršnih jedinica čime se stvaraju mogućnosti da se paralelno izvršava veći broj instrukcija. Da li su objektni kôdovi stare mašine i nove identični?

i) Neka je kod *VLIW* arhitekture, za specificirani iznos latencije funkcionalnih jedinica kod odredjenih implementacija kôd (program) perfektno isplaniran za izvršenje. Da li je kod neke druge nove implementacije*VLIW* mašine zbog promene latencije funkcionalnih jedinica potrebno izvršiti rekompilaciju kôda (programa) stare *VLIW* mašine kako bi se isti korektno izvršio i na novoj mašini.

j) Kôd (program) *VLIW* arhitekture zavisi od broja funkcionalnih jedinica u implementaciji. Da li je kod napisan za *VLIW* sa četiri *integer* funkcionalne jedinice identičan kao i kod za *VLIW* arhitekturu sa osam *integer* funkcionalnih jedinica?

k) Kod *VLIW* arhitekture ne ugradjuje se složen hardver koji podržava van-redosledno izvršenje instruikcija koji je tipičan za superskalarne procesore, ali se nudi bilji potencijal za efikasniju podršku paralelizma na nivou izvršenja instrukcija?

l) Glavna teškoća koja se javlja kod korišćenja *VLIW* arhitekture je sledeća: Korišćenje jedinstvenog objektnog koda na nekoliko generacija iste arhitekture je nemoguće, a to čini da ova arhitektura ne bude tako atraktivna i široko prihvaćena?

Napomena: Odgovor na svako od postavljenih pitanja dati u formi tvrdnja je tačna, netačna ili delimično tačna.
Za slučaj da je odgovor netačan ili delimično tačan formulisati tačan.

Primer 6

Pokazati na koji način će kompilator planirati izvršenje sledeće sekvence operacija kada se ona izvršava na *VLIW* procesoru sa tri *EX-U* jedinice. Smatrati da sve operacije imaju latentnost od dva ciklusa, i da svaka *EX-U* jedinica može da izivrši bilo koju operaciju.

```
Add    r1, r2, r3
Sub    r16, r14, r7
Ld     r2, (r4)
Ld     r14, (r15)
Mul    r5, r1, r9
Add    r9, r10, r11
Sub    r12, r2, r14
```

Odgovor

Plan izvršenja operacija oblika je

Instrukcija 1	`Add`	`r1,r2,r3`		`Ld`	`r2,(r4)`	`Ld`	`r14,(r15)`
Instrukcija 2	`Sub`	`r16,r14,r7`		`Add`	`r9,r10,r11`	`Nop`	
Instrukcija 3	`Mul`	`r5,r1,r9`		`Sub`	`r12,r2,r14`	`Nop`	

Uočimo da se operacija `Ld r14,(r15)` planira za izvršenje u Instrukciji 1, tj. pre operacije `Sub r16,r14,r7` (pripada Instrukciji 2) uprkos činjenici da se `Sub r16,r14,r7` u početno zadatom programu javlja pre `Ld r14,(r15)`, i čita sadržaj odredišnog registra operacije `Ld r14,(r15)`. S obzirom da *VLIW* operacije ne upisuju vrednosti u registar sve dok ne završe, prethodna vrednost u `r14` je dostupna još dva ciklusa nakon iniciranja Instrukcije 1, a to omogućava operaciji `Sub` da pročita staru vrednost iz `r14` i generiše korektan rezultat. Van-redosledno planiranje izvršenja ovih operacija omogućava programu da isplanira izvršenje u nekoliko instrukcija (konkretno 3). Na sličan način operacija `Add r9,r10,r11` se planira radi izvršenja pre operacije `Mul r5,r1,r9`, i pored toga što se ove operacije mogu smestiti u istu instrukciju, bez da se poveća broj instrukcija u programu.

1.5 Kompilatorske tehnike za ILP

Kompilator koristi veliki izibor tehnika za povećavanje performansi kompajliranih programa, uključujući konstantnu propagaciju, eliminaciju nepotrebnog koda, i dodela registara. Opšta teorija o konstrukciji kompilatora izučava se u predmetu "Programski jezici i prevodioci" pa zbog toga se na izučavanje ove teorije nećemo zadržavati. Na ovom mestu ukazaćemo samo na sledeće dve značajne tehnike:

a) **odmotavanje petlje** – optimizacija koja značajno povećava *ILP*; i
b) **softverska protočnost**

1.3.1 Odmotavanje petlje

Individualne iteracije kod petlji poseduju relativno nizak nivo *ILP* jer:

i) u najvećem broju slučajeva čine ih lanci zavisnih instrukcija; i
ii) izmedju grananja postoji relativno mali broj instrukcija, tj. telo petlje čini mali broj instrukcija.

Odmotavanje petlje (*loop unrolling*), kao jedna od tehnika, sa odredjenim uspehom izlazi na kraj sa ovim ograničenjima, na taj način što transformiše petlju od *N* interacija na petlju sa *N/M* iteracija, gde svaka iteracija nove petlje obavlja posao od *M* iteracija stare petlje. Na ovaj način povećava se broj instrukcija izmedju grananja, a to obezbedjuje kompilatoru i hardveru bolju mogućšnost da poveća *ILP*. U slučaju kada su iteracije kod početno zadate petlje nezavisne ili sadrže samo mali broj zavisnih izračunavanja, odmotgavanje petlje može da kreira veći broj lanaca sa zavisnim instrukcijama. Treba pri tome imati u vidu da je pre odmotavanja postojao samo jedan lanac, a sada ih je više čime se stvaraju uslovi za efikasnije iskorišćenje *ILP*-a. Na slici 7 prikazano je kako se vrši odmotavanje petlje za slučaj da je kod napisan na višem programskom jeziku C. Kod prvobitne petlje u toku jedne iteracije manipuliše se sa po jednim elementom od svakog vektora, izračunavaju se sume odgovarajućih elemenata izvornih vektora i smešta rezultat u odredišni vektor. Kod odmotane petlje u toku svake iteracije manipuliše se sa po dva elementa od oba vektora, i obavlja se posao koji odgovara dveju iteracija prvobitne petlje. U konkretnom slučaju (slika 7 desna strana) petlja je odmotana dva puta. Prvobitna petlja se može odmotati četiri puta dodavanjem 4 vrednosti indeksa petlje i u toku svake iteracije i izvršavanjem posla od četiri iteracija prvobitno zadate petlje u svakoj iteraciji odmotane petlje.

Prvobitna petlja

```
for (i = 0; i < 100;  i ++){
       a[i] = b[i]  + c[i] ;
                            }
```

Odmotana petlja

```
for i = 0; i < 100; i += 2; {
     a[i] = b[i]  + c[i];
     a[i + 1] = b[i + 1]  + c[i + 1] ;
                              }
```

Slika 7. Primer odmotavanja petlje

Ovaj primer jasno ukazuje na jednu važnu prednost koja se nudi odmotavanjem petlje, a to je smanjenje režijskog vremena petlje (*loop overhead*). Odmotavanjem prvobitne petlje za dva puta, smanjuje se broj iteracija u petljii sa 100 na 50. Na ovaj način prepolovljuje se broj uslovnih instrukcija grananja koje treba izvršavati na kraju svake iteracije u petlji. Time se smanjuje ukupan broj instrukcija koje sistem treba da izvrši u toku ovih iteracija u petlji, a što je značajnije povećava se iznos dostupnog *ILP*-a. Treba istaći da

je odmotavanje petlje od koristi i kod strogo sekvencijalnih procesora, ali se ipak značajnija poboljšanja ostvaruju kod *ILP* procesora.

Na slici 8 prikazano je kako se na asemblerskom jeziku može implementirati petlja sa slike 7, i na koji način kompilator može da odmota petlju i planira izvršenje petlje na superskalarnom procesoru koji manipuliše sa 32-bitnim poodacima. Čak i u prvobitnoj petlji, treba uočiti da je kompilator isplanirao izvršenje tako što je potencirao što je moguće veći iznos *ILP*-a, tako taj što je postavio obe inicijalne `Ld` instrukcije ispred instrukcije *Add* koja inkrementira pokazivač i obavljanjem inkrementiranja pokazivača ispred `Add` koja implementira `a[i] = b[i] + c[i]` . Uredjenjem instrukcija na takav način da su nezavisne instrukcije u programu medjusobno blizu stvaraju se uslovi da hardver superskalarnog procesora lakše locira *ILP*, dok postavljanjem instrukcije za inkrementiranje pokazivača izmedju `Ld` i instrukcije za izračunavanje `a[i]` povećava se broj operacija koje se nalaze izmedju `Ld` i korišćenjem njihovih rezultata, a to čini više verovatno da instrukcije `Ld` završe pre nego što su im rezultati potrebni.

Odmotana petlja počinje sa tri instrukcije `Add` koje postavljaju pokazivače na `a[i+1]`, `b[i+1]` i `c[i+1]`. Izdvajanjem ovih pokazivača u posebne registre u odnosu na pokazivače na `a[i]`, `b[i]` i `c[i]` omogućava se da se operacije *Load* i *Store* koje se odnose na *i*-ti i (*i+1*)-vi elemenat svakog vektora izvode paralelno, umesto da se vrši inkrementiranje svakog od pokazivača izmedju operacija obraćanja memoriji.

Neodmotana petlja

```
      Mov   r31, #0;          /*inicijalizacija i*/
loop: Ld    r1, (r2)          /*r2 je pokazivač na b[i]*/
      Ld    r3, (r4)          /*r4 je pokazivač na c[i]*/
      Add   r2, #4, r2;       /*inkrement b[i+1]*/
      Add   r4, #4, r4;       /* inkrement c[i+1]*/
      Add   r6, r1, r3;
      St    (r7), r7;         /* r7 je pokazivač na a[i]*/
      Add   r7, #4, r7;       /* inkrement a[i+1]* /
      Add   r31, #1, r31;     /*inkrement i */
      Bne   loop, r31, #100;  /*go to naredna*/
                              /*iteracija*/
```

Odmotana petlja

```
      Mov   r31, #0
      Add   r8, #4, r2        ; /*r8 je pokazivač na b[i+1]*/
      Add   r10, #4, r4;      /*r10 je pokazivač na c [i+1]*/
      Add   r13, #4, r6;      /*r13 je pokazivač na a [i+1]*/
loop: Ld    r1, (r2)
      Ld    r9, (r8);
      Ld    r3, (r4);
      Ld    r11, (r10)
      Add   r2, #8, r2;       /*inkrement za 2*32 bita*/
      Add   r4, #8, r4;
      Add   r8, #8, r8;
      Add   r10, #8, r10;
      Add   r6, r1, r3;
      Add   r12, r9, r11;
      St    (r7), r6;
      St    (r13), r12;
      Add   r7, #8, r7;
      Add   r13, #8, r13;
      Add   r31, #2, r31;     /*r31 se inkrementira za 2*/
                              /*umesto za 1*/
      Bne   loop, r31, #100;
```

Slika 8 Primer odmotane petlje na asemblerskom jeziku

Inicijalni blok koji se odnosi na instrukcije za postavljanje (početno definisanje) vrednosti kod odmotane petlje naziva se **preambula petlje**. U okviru tela petlje, kompilator je smestio sva punjenja u jedan blok, nakon toga slede inkrementi pokazivača, zatim se izvode izračunavanja `a[i]` , `a[i + 1]`, i konačno smeštanje rezultata i grananje na početak petlje. Na ovaj način maksimizira se kako paralelizam tako i vreme za koje punjenje (*load*) treba da se završi.

U primeru koga smo razmatrali, predpostavili smo da je broj iteracija prvobitne petlje bez ostatka deljiv sa stepenom odmotavanja petlje, što čini odmotavanje petlje jednostavnim. Kod velikog broja slučajeva, na žalost, broj iteracija u petlji nije deljiv sa stepenom odmotavanja, ili pak nije poznat u trenutku kompajliranja, što otežava postupak odmotavanja. Tako na primer, neka kompilator želi da odmota petlju sa

slike 7 osam puta, ili neka broj iteracija bude ulazni parametar za proceduru koja sadrži petlju. Kod ovakvih situacija kompilator generiše dve petlje. Prva petlja predstavlja razmotanu verziju početne petlje, i izvršava se sve dok je broj preostalih iteracija manji od stepena odmotavanja petlje. Druga petlja zatim izvršava ostale iteracije. Kod petlji gde broj iteracija na početku petlje nije poznat, kakve su petlje kod kojih se vrši analiza niza u cilju nalaženja karaktera *end-of-string*, postaje teže izvršiti odmotavanje petlje pa se zbog toga koriste prefinjenije (sofisticiranije) tehnike.

Na slici 9 prikzano je kako se petlja sa slike 7 može odmotati 8 puta. Prva petlja prelazi kroz 8 iteracija odjednom sve dok se ne detektuje uslov `i + 8 >= 100`. Druga petlja počinje narednom iteracijom i prolazi kroz ostale iterativne korake. S obzirom da je *i integer* promenljiva, izračunavanje `i = ((100 / 8) * 8)` ne postavlja `i` na 100. *Integer* izračunavanje 100 / 8 generiše samo integer deo količnika (zanemarujući ostatak), pa množenjem rezultata sa 8 daje najveći multipl od 8 koji je manji od 100, a to je 96, što je i tačka gde druga petlja počinje sa svojim iteracijama.

Prvobitna petlja

```
for (i = 0; i < 100; i++){
a[ i ]= b[ i ]+ c[i] ; }
```

Odmotana petlja

```
For (i=0; i <100; i+=8){
 a[ i ] = b[ i ]+ c[ i ]  ;
 a[i+1] = b[i+1] + c[i+1] ;
 a[i+2] = b[i+2] + c[i+1] ;
 a[i+3] = b[i+3] + c[i+3] ;
 a[i+4] = b[i+4] + c[i+4] ;
 a[i+5] = b[i+5] + c[i+5] ;
 a[i+6] = b[i+6] + c[i+6] ;
 a[i+7] = b[i+7] + c[i+7] ;}
for i = ((100/8)*8); i< 100; i++){
 a [i ]= b[ i ]+ c [i] ;}
```

Slika 9. Odmotavanje petlje sa ostatkom iteracija

1.3.2 Softrverska protočnost

Tehnika odmotavanje petlje poboljšava performanse sistema putem povećanja broja nezavisnih operacija koje se nalaze u okviru iteracije petlje. Jedna druga optimizacija, nazvana **softverska protočnost** (*software pipelining*) poboljšava performanse sistema u toku izvršenja programa putem raspodele svake iteracije prvobitne petlje na veći broj iteracija protočne petlje, tako da svaka iteracija nove petlje obavlja neki posao koji pripada više- strukim iteracijama prvobitne petlje. Tako na primer, petlja koja pribavlja i `b[i]` i `c[i]` iz memorije, zatim vrši sabiranje elemenata vektora da bi se dobio elemenat `a[i],` i na kraju vrši upis elementa `a[i]` u memoriju, se može tako transformirati da svaka iteracija upisuje prvo u memoriju `a[i-1],` zatim izračunava `a[i]` na osnovu vrednosti `b[i]` i `c[i]` koje su pribavljene u toku zadnje iteracije, i konačno pribavlja iz memorije `b[i+1]` i `c[i+1]` da bi pripremila te vrednosti za narednu iteraciju. Na ovaj način posao izračunavanja elementa vektora a se distribuira kroz tri iteracije nove petlje. Preplitanjem (*interleaving*) delova različitih iteracija petlji povećava se *ILP*, na isti način kao i kod tehnike odmotavanja

petlji. Veliki broj kompilatora kombinuje tehnike odmotavanje petlji i softversku protočnost sa ciljem da poveća iznos *ILP*-a.