

Pralelni računarski sistemi

Vektorski procesori

Vektorski procesori

- * Uvodjenjem protočnosti u izvršenju instrukcija značajno se može povećati brzina obrade.
- Kod protočne obrade javljaju se mnogi problemi koji mogu umanjiti vršne performsne
 1. Brzina protočne obrade može se povećati smanjenjem klok ciklusa.
 - Da bi se klok ciklus smanjio, potrebno je izvršiti podelu osnovne funkcije na veći broj podfunkcija, a to zahteva dublje protočne sisteme.
 - Što je dublji protočni sistem, to su šanse za nastupanje hazarda veće (kod neporotočnih sistema ne postoje hazardi)
 2. Maxmalna brzina rada protočne mašine u idealnom slučaju je $1\text{istr}/\text{clk}$
 - U realnim situacijama, br. istr/clock je manji zbog nastupanja hazarda
 - Superskalarnom i VLIW tehnikom je moguće povećati brzinu do tri puta
 - Pribavljanje i izdavanje većeg broja instrukcija je jako komplikovano.

Vektorski procesori (nast.)

3. Brza protočna mašina obično koristi keš memoriju da bi se izbegla velika latentnost kod instrukcija koje se obraćaju memoriji

- Obimni naučno-tehnički problemi često imaju aktivne skupove podataka kojima se pristupa sa niskom lokalnošću, što dovodi do loših performansi zbog memorijske hijerarhije
- Problem se može rešiti tako što se ovi podaci neće smeštati u keš
- Ako su oblici pristupa podacima poznati, adekvatnom organizacijom memorije se problemi mogu rešiti

*** Pbrojani problemi mogu uspešno biti rešeni korišćenjem vektorskih računara**

- Vektorski procesor je procesor specijalno projektovan da obavlja vektorska izračunavanja
 - Vektorske instrukcije kao perande imaju linearna polja (vektore) (npr. jednom vektorskom instrukcijom je poguće sabrati dva vektora)

Osobine vektorskih instrukcija

- * U vektorskoj instrukciji izračunavanje svakog elementa je nezavisno od prethodnog rezultata, što dozvoljava korišćenje veoma dubokih protočnih sistema bez šansi za nastupanje hazarda po podacima
 - Odsustvo hazarda je određeno od strane kompajlera ili programera, kada je odlučeno da se može iskoristiti vektorska instrukcija
- * Jedana vektorska instrukcija specificira veliki posao
 - ekvivalentan aje izvršenju cele petlje u koju se u svakoj iteraciji izračunava jedan element vektora, ažuriraju indeksi i vrši grananje na početak petlje
- * Vektorske instrukcije koje pristupaju memoriji imaju poznate oblike pristupa
 - adekvatnim načinom smeštanja podataka može se kompenzovati velika latentnost pristupa memoriji.
 - visoka latentnost pristupa glavnoj memoriji u odnosu na keš je amortizovana jer se jedan pristup inicira za ceo vektor, a ne za jednu reč (latentnost je vidljiva samo jednom za ceo vektor, a ne za svaki element vektora)

Osobine vektorskih instrukcija (nast.)

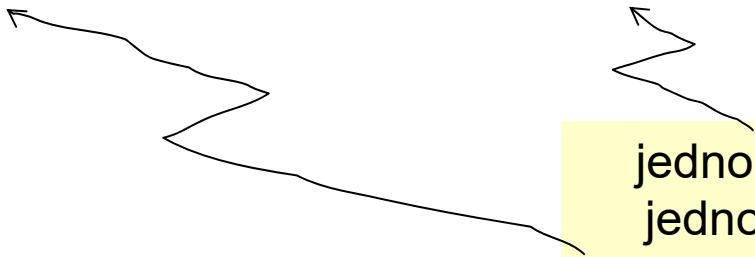
- * Pošto je cela petlja zamenjena vektorskom instrukcijom, kontrolni hazardi koji bi normalno nastupili zbog grananja na početak petlje, ne postoje!
- * Dobro vektorizovanim programskim kodom moguće je postići 10 do 20 puta veće brzine obrade od ekvivalentnog niza skalarnih instrukcija
- * PRIMER:

for i:=1 to n

A(I):=B(I)+C(I);

⇒

A(1:N)=B(1:N)+C(1:N)



jedno pribavljanje i
jedno dekodiranje
naspram n pribavljanja
i n dekodiranja

* Vektorske mašine maksimalno koriste protočnost

- u obradi instrukcija
- kod izvršenja ALU operacija
- pri izračunavanju efektivne adrese
- kod pristupa memoriji

* Većina vektorskih računara dozvoljava da se više vektorskih operacija obavlja jednovremeno, ako ne koriste iste funkcionalne jedinice

(uvodeći na taj način paralelizam u obradi)

Osnovne komponente vektorskog računara

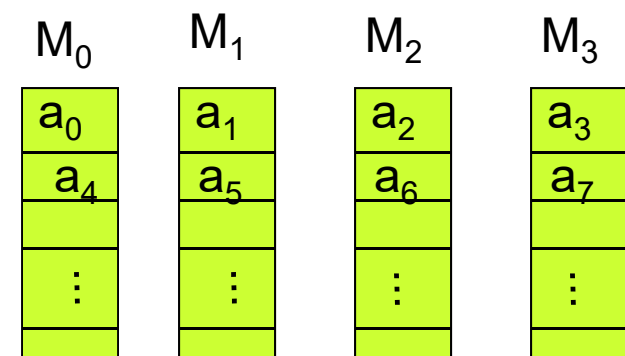
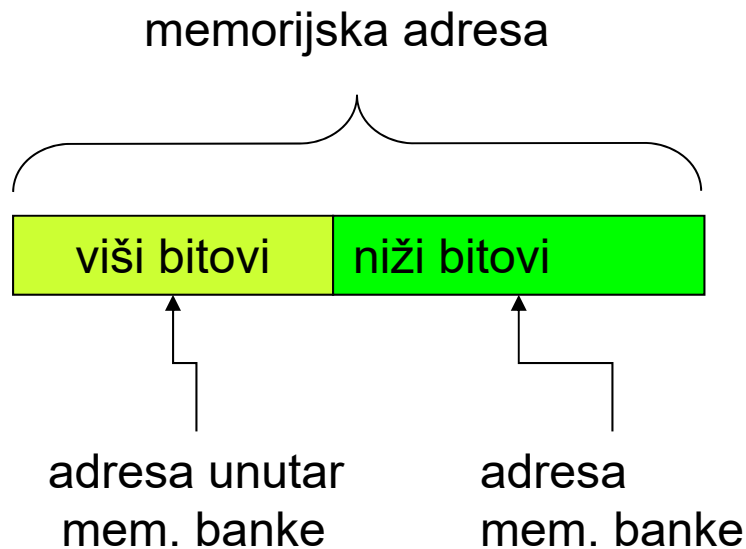
- * Memorija
- * Skalarni posistem
- * Vektorski podsistem

Memorija vektorskih računara

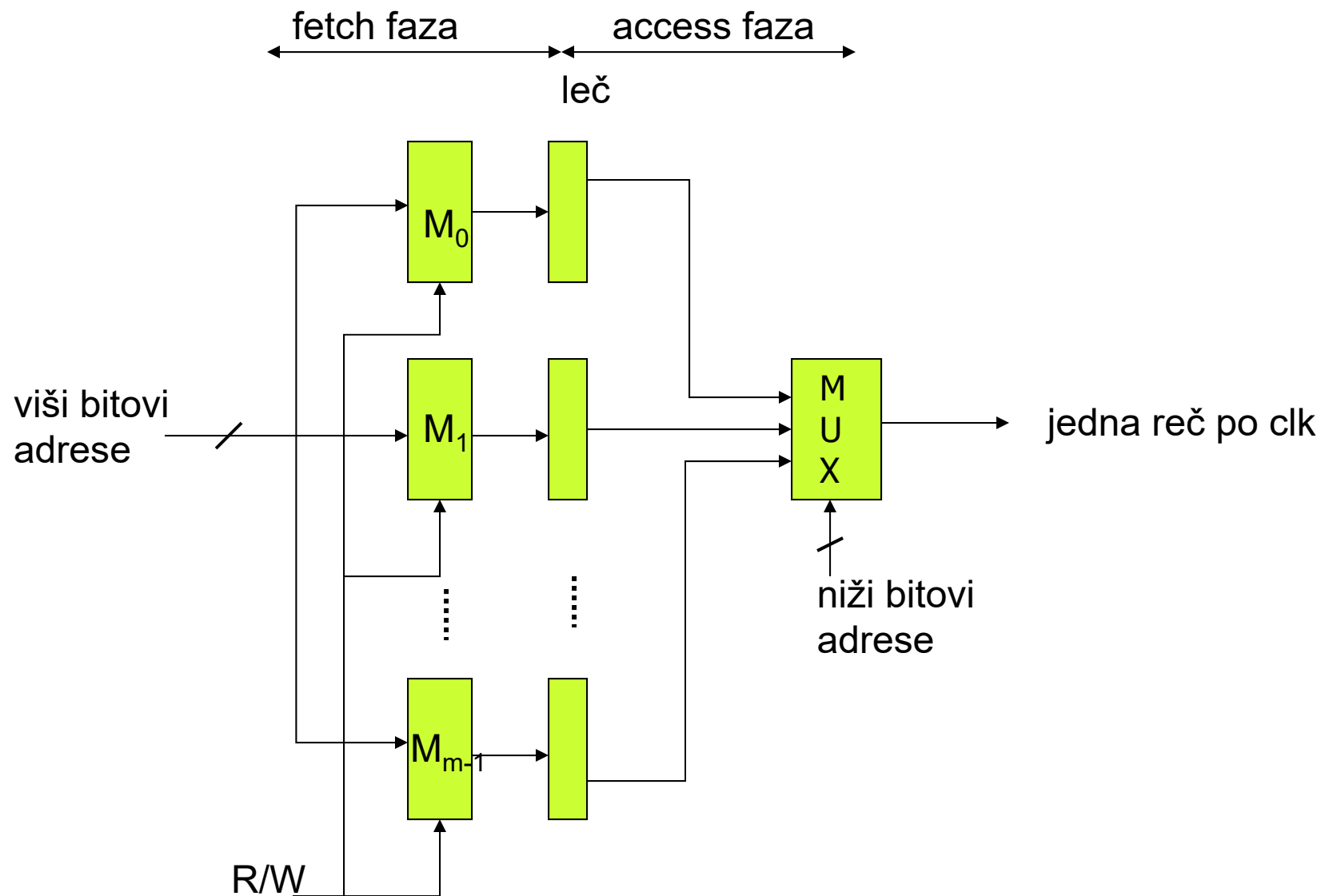
- Organizacija memorije direktno utiče na performanse celog sistema
- F-ja: pamti program i podatke u skalarnom i vektorskom obliku
- Podeljena je na niz memorijski banaka kojima se može jednovremeno pristupati

Adresiranje i pristup memoriji

- Koristi se "low order interleaving" za adresiranje memorijskih banaka
 - podaci koji se nalaze na susednim adresama smešteni su u različite banke



Pristup memoriji



Memorija vek. računara

* Prema tome odakle se pribavljaju operandi, vektorski računari se dele na

- vektorsko-registarske (sve vektorske instrukcije, izuzev LOAD i STORE, pribavljaju operande iz vektorskih registara i smeštaju rezultat u registre)
 - većina vektorskih računara je ovog tipa (Cray)
- memorijsko-memorijske (memory-to-memory) – operandi se pribavljaju iz memorije i smeštaju u memoriju
 - Cyber 205

Skalarni podsistem

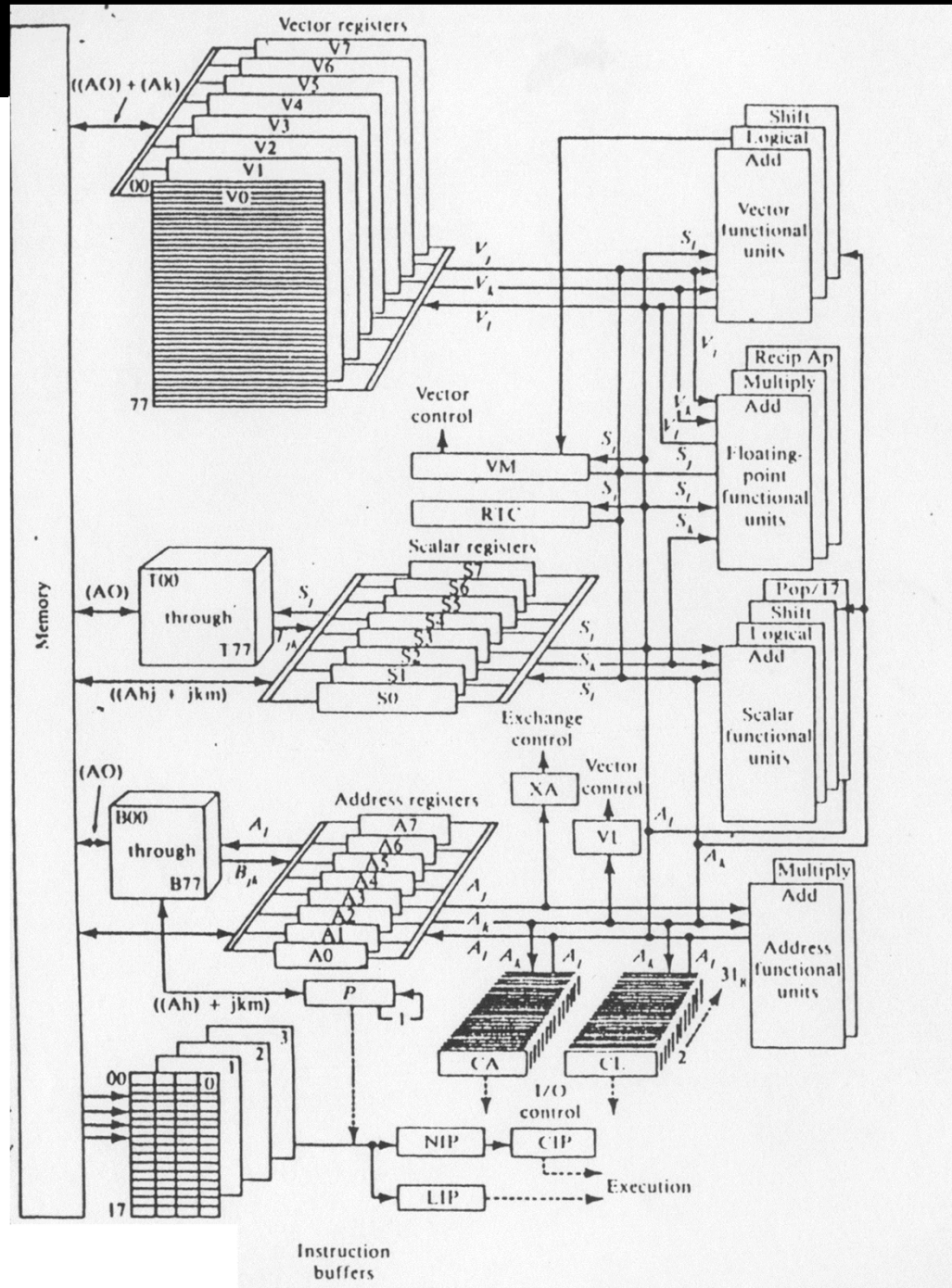
- Namenjen izvršenju skalarnih instrukcija
- Ne razlikuje se od protočne organizacije procesora
- skalarni registri i protočne funkcionalne jedinice

Vektorski podsistem

- Sadrži protočno organizovane funkcionalne jedinice za izvršenje vektorskih operacija
- veliki broj vektorskih registara

Machine	godina	Clock	Reg.	Elemen.	FUs
Cray 1	1976	80 MHz	8	64	6
Cray XMP	1983	120 MHz	8	64	8
Cray YMP	1988	166 MHz	8	64	8
Cray C- 90	1991	240 MHz	8	128	8
Cray T- 90	1996	455 MHz	8	128	8
Conv. C- 1	1984	10 MHz	8	128	4
Conv. C- 4	1994	133 MHz	16	128	3
Fuj. VP200 3	1982	133 MHz	8- 256	32- 1024	
Fuj. VP300 3	1996	100 MHz	8- 256	32- 1024	
NEC SX/ 2	1984	160 MHz	8+ 8K	256+ var	16
NEC SX/ 3	1995	400 MHz	8+ 8K	256+ var	16

Cray-1



Cray - 1

- * Najpoznatiji vektorski procesor, 1976.
- * Najbrži superračunar krajem 70s.
- * Sastoji se iz tri dela:
 - Glavne memorije
 - Skalarnog podsistema
 - Vektorskog podsistema

Cray-1: Glavna memorija

- * 16 banaka, svak aod po 64K, 64-bitnih reči.
- * Klok perioda 50 nSec, što iznosi 4 procesorska ciklusa.
- * Može preneti 1-4 reči po clock periodi, u zavisnosti da li vrši prenos u registre ili bafere.
- * 4 reči po clk ciklusu za instrukcioni bafer.

Cray-1: Skalarni podsistem

* Sastoji se od

- Instrukcionih bafera
- 2 skupa skalarnih registara (8 S reg. i 64 T reg.)
- 2 skupa adresnih registara (A i B)
- Skalarnih funkcionalnih jedinica (za integer instrukcije)
- Funkcionalne jedinice za operacije u pokretnom zarezu su zajedničke za skalarni i vektorski podsistem.

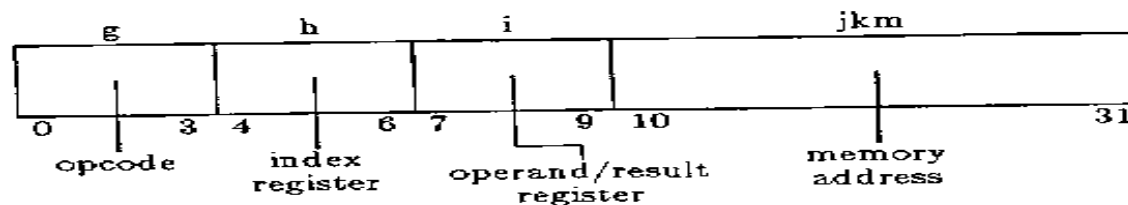
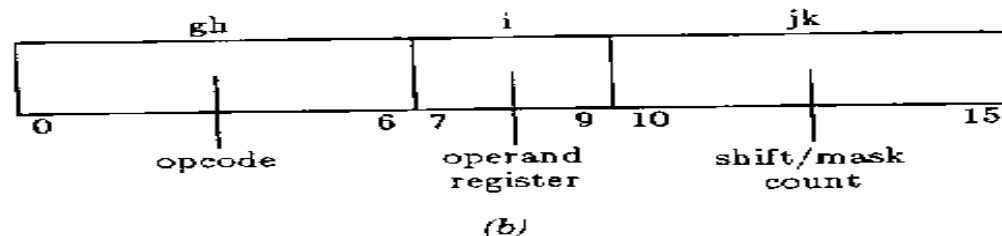
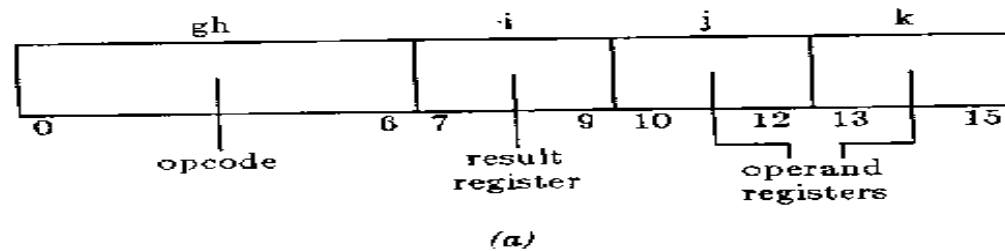
Cray-1: vektorski podsistem

* Sastoji se od

- 8 vektorskih registara (svaki dužine 64 64-bitne reči)
- 3 vektorske funkcionalne jedinice (integer sabiranje, logičke i shift)
- 3 zajedničke funkcionalne jedinice za operacije u pokretnom zarezu (sabiranje, množenje, izračunavanje recipročne vrednosti)
- Vektorske funkcionalne jedinice mogu pribavljati podatke iz skalarnih i vektorskih registara (i smeštati)
 - VL registar – pamti dužinu vektora koji se obradjuje
 - VM – vektor maske, dužine 64 bita. Svaki bit odgovara jednom elementu vektorskog registra)

Cray-1: Format instrukcija

- * Binarne aritmetičke i logičke instrukcije, 16-bitne (a)
- * Unarne shift i mask instrukcije, 16-bitne (b)
- * Instrukcije obraćanja memoriji, 32-bitne (c)



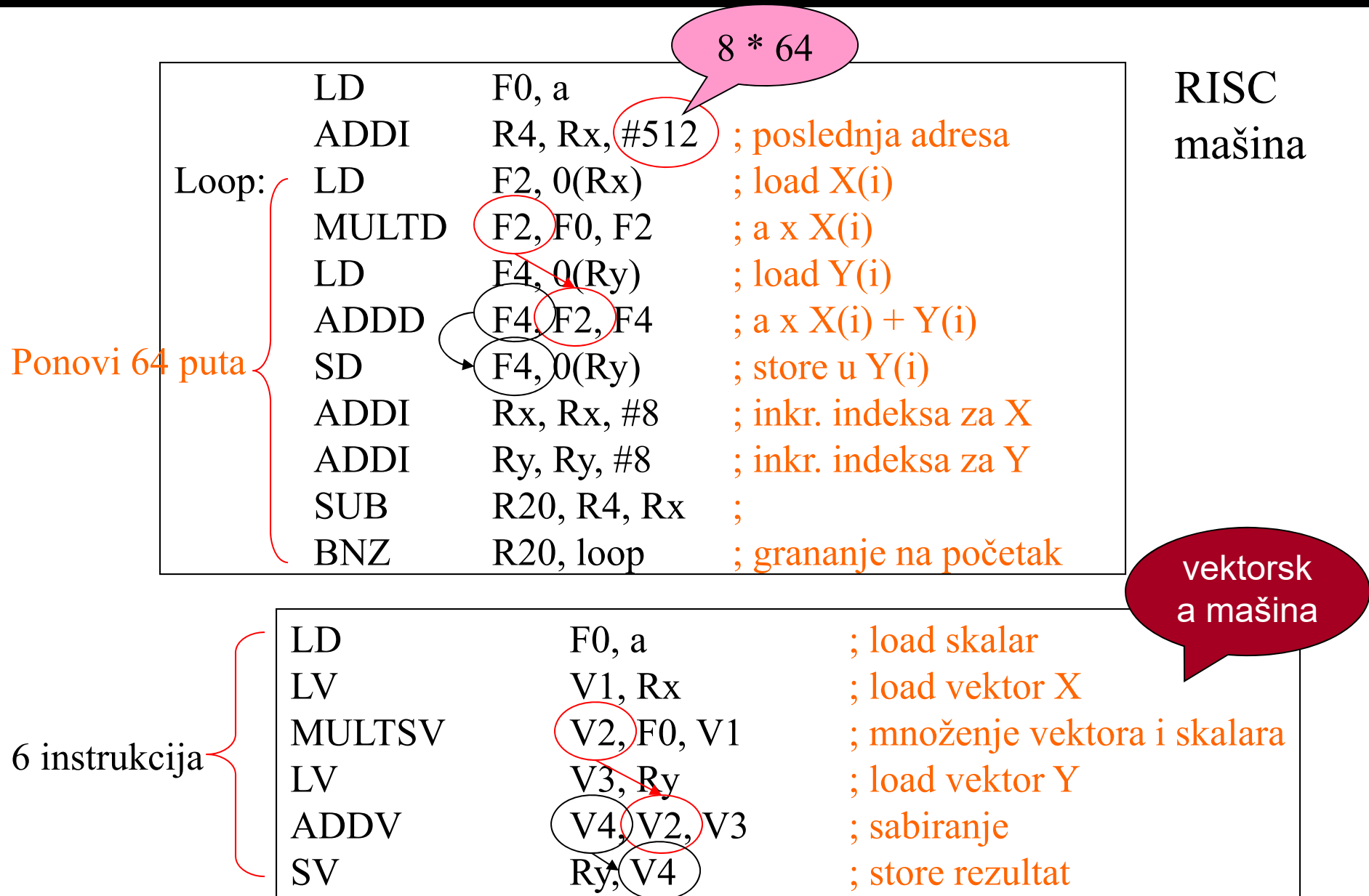
Cray-1 set instrukcija

Instruction	Operands	Function
ADDV	V1, V2, V3	Add elements of V2 and V3, then put each result in V1.
ADDSV	V1, F0, V2	Add F0 to each element of V2, then put each result in V1.
SUBV	V1, V2, V3	Subtract elements of V3 from V2, then put each result in V1.
SUBVS	V1, V2, F0	Subtract F0 from elements of V2, then put each result in V1.
SUBSV	V1, F0, V2	Subtract elements of V2 from F0, then put each result in V1.
MULTV	V1, V2, V3	Multiply elements of V2 and V3, then put each result in V1.
MULTSV	V1, F0, V2	Multiply F0 by each element of V2, then put each result in V1.
DIVV	V1, V2, V3	Divide elements of V2 by V3, then put each result in V1.
DIVVS	V1, V2, F0	Divide elements of V2 by F0, then put each result in V1.
DIVSV	V1, F0, V2	Divide F0 by elements of V2, then put each result in V1.
LV	V1, R1	Load vector register V1 from memory starting at address R1.
SV	R1, V1	Store vector register V1 into memory starting at address R1.
LVWS	V1, (R1, R2)	Load V1 from address at R1 with stride in R2, i.e., $R1+i \times R2$.
SVWS	(R1, R2), V1	Store V1 from address at R1 with stride in R2, i.e., $R1+i \times R2$.
LVI	V1, (R1+V2)	Load V1 with vector whose elements are at $R1+V2(i)$, i.e., V2 is an index.
SVI	(R1+V2), V1	Store V1 to vector whose elements are at $R1+V2(i)$, i.e., V2 is an index.
CVI	V1, R1	Create an index vector by storing the values $0, 1 \times R1, 2 \times R1, \dots, 63 \times R1$ into V1.
S--V	V1, V2	Compare the elements (EQ, NE, GT, LT, GE, LE) in V1 and V2. If condition is true, put a 1 in the corresponding bit vector; otherwise put 0. Put resulting bit vector in vector-mask register (VM). The instruction S--SV performs the same compare but using a scalar value as one operand.
S--SV	F0, V1	
POP	R1, VM	Count the 1s in the vector-mask register and store count in R1.
CVM		Set the vector-mask register to all 1s.
MOVI2S	VLR, R1	Move contents of R1 to the vector-length register.
MOVS2I	R1, VLR	Move the contents of the vector-length register to R1.
MOVF2S	VM, F0	Move contents of F0 to the vector-mask register.
MOVS2F	F0, VM	Move contents of vector-mask register to F0.

FIGURE B.3 The DLXV vector instructions. Only the double-precision FP operations are shown. In addition to the vector registers, there are two special registers, VLR (discussed in section B.3) and VM (discussed in section B.5). The operations with stride are explained in section B.3, and the use of the index creation and indexed load-store operations are explained in section B.5.

Primer

$$Y = a * X + Y$$



Vektorski procesori

Vektorizacija petlji

Vektorizacija petlji

- * Da bi se uspešno koristila vektorska mašina potrebni su ili efikasni viši programski jezici ili vektorizirajući kompajleri .
 - U prvom slučaju deo programa koji se može izvršiti u vektorskom obliku eksplicitno specificira programer korišćenjem odgovarajućih naredbi višeg programskog jezika.
 - U drugom slučaju program je napisan na nekom standardnom sekvencijalnom programskom jeziku, a kompajler određuje koji deo programa se može vektorizovati.
- * Jedini kandidati za vektorizaciju su petlje.
 - Uvek se vektorizuje najdublje ugnježdjena petlja.
- * Kompajler mora biti u stanju da prepozna da li se petlja ili deo petlje može vektorizovati i da shodno tome generiše odgovarajući vektorski kod.
 - Da bi to učinio kompajler mora utvrditi koje zavisnosti postoje izmedju promenljivih u telu petlje
 - Samo RAW hazardi mogu sprečiti vektorizaciju.

Primer

```
        for i=1 to n  
S1      A(i+1)=A(i)+B(i)  
S2      B(i+1)=B(i)*A(i+1)  
        endfor
```

* U petlji je moguće uočiti sledeće zavisnosti:

- Naredba S koristi vrednost koju je S izračunala u prethodnoj iteraciji.
 - Ovakva zavisnost postoji u ovom primeru, jer S1 u i+1-iteraciji koristi vrednost A(i) koja je izračunata u i-toj iteraciji kao A(i+1). Isto važi i za naredbu S2 kada su u pitanju B(i) i B(i+1).
- S1 koristi vrednost koju računa S2 u prethodnoj iteraciji.
 - I ovakav tip zavisnosti postoji u ovom primeru: S1 koristi vrednost B(i) u i+1-iteraciji koja je izračunata kao B(i+1) u i-toj iteraciji
- S2 koristi vrednost izračunatu u S1 u istoj iteraciji,
 - tj. A(i+1)

Zašto do ovih zavisnost dolazi?

- * Pošto su vektorske operacije protočne i latentnost može biti veoma duga, prethodna iteracija može da se ne završi a da sledeća već otpočne.
 - Tj. $i+1$ -iteracija može otpočeti a da se i -ta iteracije ne završi.
 - Tako se može desiti da se rezultat i -te iteracije ne upiše, a da $i+1$ -iteracija krene.
 - Shodno tome, ako postoje situacije tipa 1 i 2, vektorizacija petlje će dovesti do RAW hazarda (hazarda koji vektorska mašina ne proverava).
 - Zbog toga se ovakve petlje ne mogu vektorizovati

Analiza zavisnosti

- * Zavisnost se može uočiti ako se izvrši razvijanje petlje po indeksnoj promenljivoj:

for i=1 to 100

$A(i+1)=A(i)+B(i)$

$B(i+1)=B(i)*A(i+1)$

endfor

- za i=1

$A(2)=A(1)+B(1)$

$B(2)=B(1)*A(2)$

-
- i=2

$A(3)=A(2)+B(2)$

$B(3)=B(2)*A(3)$

- Crvene strelice prikazuju zavisnosti izmedju iteracija.

klasičan
RAW

- * U situaciji 3 normalni hardver za detekciju zavisnosti može otkriti ovaj hazard i zaustaviti izdavanje vektorske instrukcije.

- Zbog toga petlje koje sadrže samo ovakve zavisnosti mogu biti vektorizovane.
- *Zavisnosti koje su posledica korišćenja vrednosti izračunatih u nekoj od prethodnih iteracija zovu se **loop-carry** (prenos iz petlje) zavisnosti*
- Prvi zadatak kompajlera je da utvrdi da li u petlji postoje loop-carry zavisnosti.
- Kompajler to ostvaruje pomoću algoritma za analizu zavisnosti
- analiza može biti veoma kompleksna.

Analiza zavisnosti

- * Najjednostavniji slučaj nastupa kada se ime polja nalazi samo sa jedne strane naredbe dodeljivanja

```
for i=1 to 100  
  A(i)=B(i)+C(i)  
  D(i)=E(i)*F(i)  
endfor
```

- * U ovom slučaju ne može doći do loop-carry zavisnosti.

```
for i=1 to 100  
  A(i)=B(i)+C(i)  
  D(i)=A(i)*E(i)  
endfor
```

- * Ni u ovom slučaju ne može doći do loop-carry zavisnosti.

- Treći tip zavisnosti postoji, ali ovaj tip zavisnosti procesor može detektovati i zaustaviti izdavanje naredbe za množenje, mada bi $+$ i $*$ mogle paralelno da se izvršavaju jer ne koriste iste funkcionalne jedinice.

Analiza zavisnosti

- * Često se isto ime pojavljuje i kako izvor i kao odredište unutar petlje:

```
for 10 i=1 to 100  
  y(i)=ax(i)+y(i)  
endfor
```

- * U ovom slučaju ne postoji loop-carry zavisnost jer rezultat izračunavanja y ne zavisi od prethodnog izračunavanja.
 - Ovaj problem se može rešiti preimenovanjem promenljivih

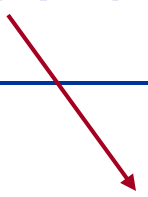
Analiza zavisnosti

- * Sledeća petlja, koja se često zove rekurentna petlja, sadrži loop-carry zavisnost:

```
for i=2 to 100  
  y(i)=y(i-1)+y(i)  
endfor
```

- * Zavisnost se može uočiti razvijanjem petlje:

za i=2	$y(2)=y(1)+y(2)$
<hr/>	
za i=3	$y(3)=y(2)+y(3)$



Kako kompajler detektuje loop carry zavisnosti generalno?

for $i_1=L_1, U_1$
 for $i_2=L_2, U_2$

 for $i_n=L_n, U_n$

S1: $A(f(i_1, i_2, \dots, i_n)) = \dots$

S2: $\dots = A(g(i_1, i_2, \dots, i_n))$

- * iterativni vector, I je celobrojni vector $I=(i_1, i_2, \dots, i_n)$, pri čemu i_k predstavlja vrednost iterativnog indeksa i_k na nivou k
- * LC zavisnost postoji ako postoje dva iterativna vektora I i J , $I < J$, takva da važi da je $f(I)=g(J)$
- * problem se svodi na rešavanje sistema celobrojnih jednačina (Diofantove jednačine)
- * Problem je np kompleksan
 - sprovede se jednostavniji testovi koji samo daju dovoljne uslove (ali ne i potrebne) za postojanje zavisnosti

- * Da bi bilo koji test mogli da primenimo petlja mora da bude normalizovana, tj. da indeksna promenljiva kreće od 0 (ili 1) i da se menja sa korakom 1.

```
for (i = 2; i <= 100; i = i+3)  
    Z[i] = 0;
```

- * Normalizovana petlja:

```
for (in = 0; in <= 32; in++)  
    Z[2 + 3* in] = 0;
```

- * U opštem slučaju $i^n = (i - \text{lowerBound}) / i_{\text{step}}$
- * Gornja granica: $(\text{upperBound} - \text{lowerBound}) / i_{\text{step}}$

Primer2: ugnježdene petlje

```
for (i = 3; i <= 7; i++)  
    for (j = 6; j >= 2; j = j - 2 )  
        Z[i, j] = Z[i, j+2] + 1
```

* Normalizovani oblik

```
for (in = 0; in <= 4; i++)  
    for (jn = 0; jn <= 2; j++)  
        Z[3 + in, 6 - jn*2] = Z[3 + in, 6 - jn*2+2] + 1
```

NZD (GCD) test za jednostruke petlje

- * Pretpostavimo da se u petlji vrši upis u element polja sa indeksom $a*i+b$, a da se pristupa elementu sa indeksom $c*i+d$, gde je i indeks petlje koji se kreće u granicama od m do n .
- * Loop-carry zavisnost postoji ako važi sledeće:
 - Postoje dva iterativna indeksa j i k ($j < k$), oba unutar granica petlje, tako da se u petlji pamti rezultat u element sa indeksom $a*j+b$, a zatim se pribavlja (čita) isti element kada je indeksiran sa $c*k+d$, tj. kada je $a*j+b = c*k+d$.
- * Jednostavan i dovoljan test koji se koristi za detekciju zavisnosti je NZD (najveći zajednički delilac) test.
(GCD)

NZD test

- * Ako loop-carry zavisnost postoji tada $\text{NZD}(c, a)$ mora deliti $(d - b)$ bez ostatka (baš $(d - b)$ jer je $j < k$).
- * NZD test je dovoljan da garantuje da nema nikakve zavisnosti.
- * Može se desiti da NZD test da odgovor da zavisnost postoji a da nema nikakve zavisnosti.
 - Ovo može da nastupi zato što NZD test ne uzima u obzir granice petlje.

Primer1

- * Korišćenjem NZD testa utvrditi da li postoji loop-cary zavisnost u sledećoj petlji:

for i=1 to 100

x(2*i+3)=x(2*i)+k

endfor

- * **Rešenje** $a = 2, \quad b = 3, \quad c = 2, \quad d = 0$

$$NZD(c, a) = NZD(2, 2) = 2$$

$$d - b = 0 - 3 = -3$$

- Pošto 2 ne deli -3 bez ostatka to nema loop-cary zavisnosti.
- Ako razvijemo petlju videćemo da zavisnosti zaista nema:

$$i=1 \quad x(5)=x(2)+k$$

$$i=2 \quad x(7)=x(4)+k$$

$$i=3 \quad x(9)=x(6)+k$$

Primer2

- * Korišćenjem NZD testa utvrditi da li postoji loop-carry zavisnost u sledećoj petlji:

```
for i=2, 100, 2  
  x(50*i+1)=x(i-1)+k  
endfor
```

- * REŠENJE:

- Da bi NZD test mogao da se primeni prvo treba normalizovati petlju, pošto se u primeru indeks petlje uvećava sa korakom 2.
- Normalizovana petlja ima sledeći izgled

```
for i=1, 50  
  x(100*i+1)=x(2*i-1)+k  
endfor
```

$$a = 100, \quad b = 1, \quad c = 2, \quad d = -1$$

$$NZD(c, a) = NZD(2, 100) = 2$$

$$d - b = -1 - 1 = -2$$

NZD test prolazi, mada nema loop carry zavisnosti

Primer2 – nast.

* Zašto je NZD dao odgovor da loop carry zavisnost postoji?

- zato što test ne uzima u obzir granice petlje!

i=1 $x(101)=x(1)+k$

.

.

.

i=50 $x(5001)=x(99)+k$

i=51 $x(5101)=x(101)+k$

zavisnost postoji
izmedju 1. i 51.
iteracije (ali su
granice 1,50 !)

GCD Test

- ❑ Može se primeniti i kada je indeks polja linearna funkcija indeksnih promenljivih

- ❑ Npr.

```
do i = li,hi
  do j = lj,hj
    A(a1*i + a2*j + a0) = ... A(b1*i + b2*j + b0) ...
  enddo
enddo
```

- ❑ Ako loop-carry zavisnost postoji tada
 - $a1*i1 - b1*i2 + a2*j1 - b2*j2 = b0 - a0$
 - Celobrojna rešenja postoje ako
 - $\text{gcd}(a1,a2,b1,b2)$ deli $b0 - a0$

Primer

Example

Code

```
do i = li, hi
  do j = lj, hj
    A(4*i + 2*j + 1) = ... A(6*i + 2*j + 4) ...
  enddo
enddo
```

$\text{gcd}(4, -6, 2, -2) = 2$

Does 2 divide 4-1?

Ograničenja NZD testa

* Nije uvek moguće NZD testom utvrditi postojanje zavisnosti:

- S obzirom da se analiza zavisnosti izvodi u fazi kompilacije, vrednosti a , b , c i d mogu biti nepoznate, što znači da se ovaj test može primeniti samo ako su a , b , c i d konstante.
- nije ga moguće uvek primeniti kod višedimenzionalnih polja

Eliminisanje loop carry zavisnosti

- Mnoge loop-carry zavisnosti mogu biti eliminisane preuredjenjem naredbi unutar petlje i razbijanjem petlje.
- **PRIMER.** U sedećoj petlji postoji loop-carry zavisnost izmedju naredbi S1 i S2 kada je u pitanju vektor **y**:

```
        for i=1, N
S1      x(i)= y(i-1) * z(i)
S2      y(i)= 2*y(i)
        endfor
```

- Preuredjenjem ove petlje i razbijanjem na dve mogu se eliminisati loop-carry zavisnosti:

```
        for i=1, N
          y(i)= 2*y(i)
➤      endfor
        for i=1,N
          x(i) = y(i-1)*z(i)
        endfor
```

- Ovo je moguće jer izračunavanje $y(i)$ ne zavisi od $x(i)$, tj. ne postoji kružna zavisnost izmedju naredi S1 i S2 u polaznoj petlji.

Eliminisanje loop carry zavisnosti

- U sedećoj petlji postoji loop-carry zavisnost izmedju naredbi S1 i S2 kada je u pitanju vektor y :

```
for i=1, N
  S1  x(i)= y(i-1) * z(i)
  S2  y(i)= 2*y(i)
endfor
```

* Razvijemo petlju po i

i=1 x(1)=y(0)*z(1)

i=2 y(1)=2*y(1)
 x(2)=y(1)*z(2)

i=3 y(2)=2*y(2)
 x(3)=y(2)*z(3)
 y(3)=2*y(3)

```
x(1)=y(0)*z(1)
for i=1, N-1
  S2:  y(i)= 2*y(i)
  S1   x(i+1)= y(i) * z(i+1)
endfor
y(N)=2*y(N)
```

- Ova tehnika se zove softverska protočnost (software pipelining)

Eliminacija zavisnosti

- U petlji mogu postojati i zavisnosti tipa WAR koje se još zovu i antizavisnosti, WAW koje se zovu izlazne zavisnosti.
- Medjutim, ove zavisnosti kod vektorski računara nisu parvi hazardi i mogu se eliminisati preimenovanjem registara u fazi kompilacije.
- Ovi hazardi se zovu još i *konflikti imenovanja*

```
do 10 i = 1, 100
➤ 1   y(i) = x(i)/S
➤ 2   x(i) = x(i) + S
➤ 3   z(i) = y(i) + S
➤ 4   y(i) = S - y(i)
10    continue
```

- Prave zavisnosti (RAW) postoje izmedju naredbi 1 i 3, i 1 i 4, zbog $y(i)$.
 - Ove zavisnosti nisu loop-carry i neće sprečiti vektorizaciju.
 - Ove zavisnosti usloviće da naredbe 3 i 4 čekaju dok se naredba 1 ne završi, mada ne koriste iste funkcionalne jedinice;
- WAR (antizavisnosti) postoje izmedju 1 i 2 zbog $x(i)$, i 3 i 4 zbog $y(i)$;
- WAW (izlazna zavisnost) postoji izmedju 1 i 4 zbog $y(i)$.
 - Sledećom verzijom petlje mogu se eliminisati ove pseudozavisnosti:

```
do 10 i = 1, 100
• 1   T(i) = x(i)/S
• 2   X1(i) = x(i) + S
• 3   Z(i) = T(i) + S
• 4   Y(i) = S - T(i)
10    continue
```