

Paralelni sistemi

Paralelizam na nivou instrukcija
Superskalarni i VLIW procesori

Da se podsetimo....

* Protočno izvršenje instrukcija:

1. Instruction fetch cycle (IF) – pribavljanje instrukcije
2. Dekodiranje instrukcije i pribavljanje operandi (ID)
3. Izvršenje / izračunavanje efektivne adrese (EXE)
4. Obraćanje memoriji /okončanje grananja (MEM)
5. Upis rezultata u registarski fajl (WB)

* Protočno izvršenje instrukcija

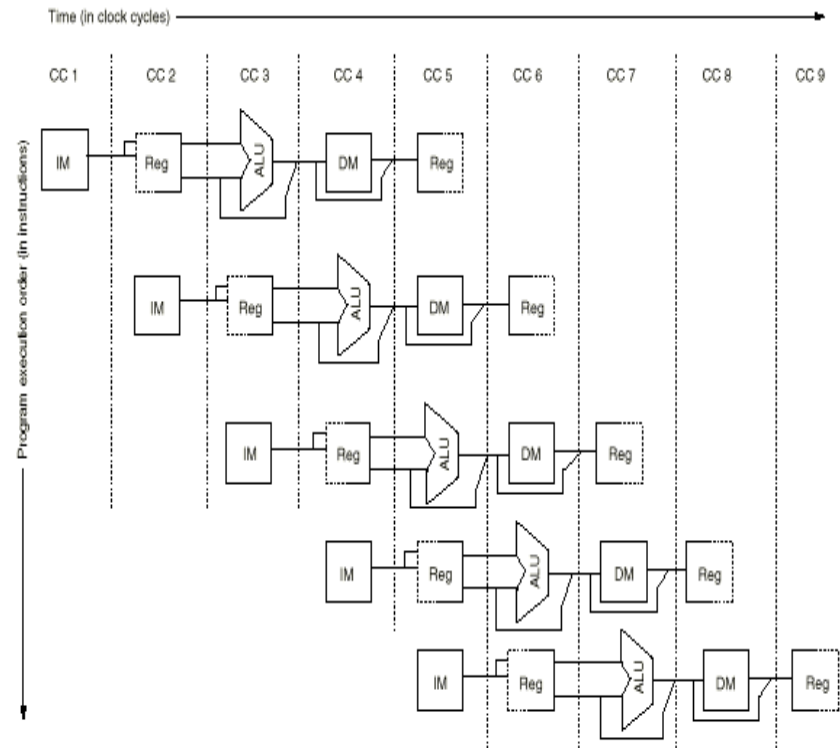


FIGURE 3.3 The pipeline can be thought of as a series of datapaths shifted in time.

Primeri instrukcija

- * LW R1, 30 (R2)
- * SW 500(R4), R3
- * ADDI R1,R2,#3
- * BEQZ R1, ime
- ADD R1,R2,R3

dejstvo $R1 \leftarrow \text{Mem}[30 + [R2]]$
 dejstvo $\text{Mem}[500 + [R4]] \leftarrow [R3]$
 dejstvo $R1 \leftarrow [R2] + 3$
 dejstvo if $R1 = 0$ then $PC \leftarrow PC + \text{ime}$
 dejstvo $R1 \leftarrow [R2] + [R3]$

Instruction number	Clock number								
	1	2	3	4	5	6	7	8	9
Instruction i	IF	ID	EX	MEM	WB				
Instruction $i + 1$		IF	ID	EX	MEM	WB			
Instruction $i + 2$			IF	ID	EX	MEM	WB		
Instruction $i + 3$				IF	ID	EX	MEM	WB	
Instruction $i + 4$					IF	ID	EX	MEM	WB

FIGURE 3.2 Simple DLX pipeline. On each clock cycle, another instruction is fetched and begins its five-cycle execution. If an instruction is started every clock cycle, the performance will be up to five times that of a machine that is not pipelined.

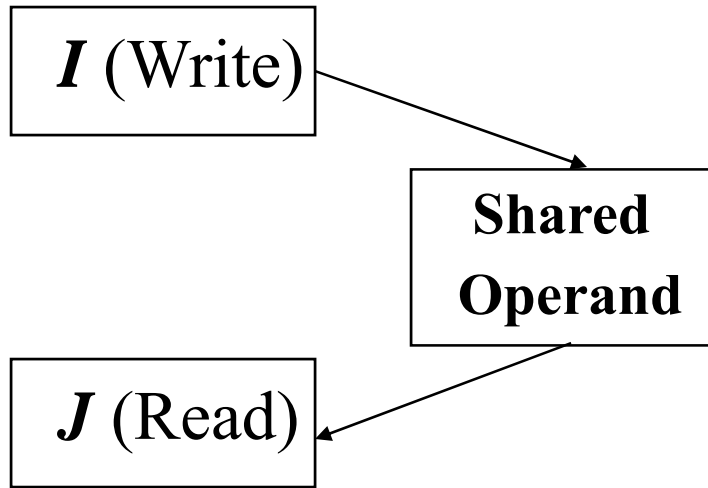
Problemi – hazardi protočnih sistema

- * Hazardi su situacije koje sprečavaju da izvršenje instrukcije otpočne u predviđenom klok ciklusu.
- * Hazardi redukuju idealne performanse protočnog sistema (izvršenje jedne instrukcije po klok ciklusu).
- * Hazardi se mogu klasifikovati u tri grupe:
 - Strukturni hazardi – nastaju zbog jednovremenih zahteva za korišćenjem istog hardverskog resursa
 - Hazardi po podacima – nastupaju zato što je redosled pristupa operandima izmenjen uvodjenjem protočnosti u odnosu na sekvencijalno izvršenje instrukcija
 - Kontrolni hazardi – nastupaju zbog zavisnosti u redosledu izvršenja instrukcija (izazivaju ih instrukcije koje mogu promeniti sadržaj PC)
- * Hazardi mogu izazvati zaustavljanje protočnog sistema (nekim instrukcijama se dozvoljava da produže sa izvršenjem)

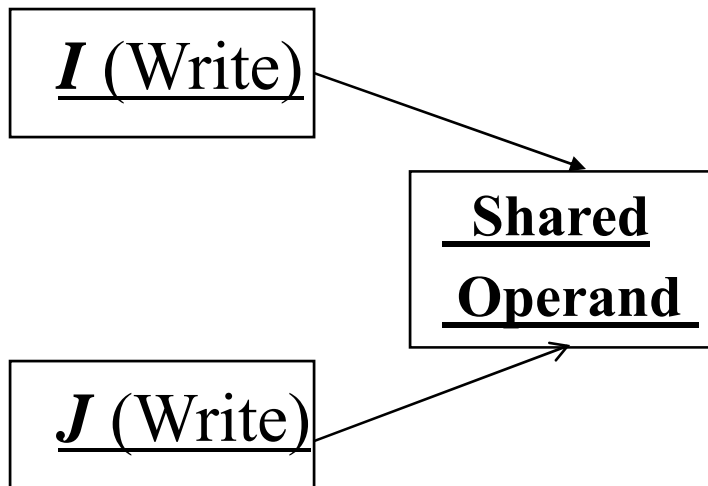
Klasifikacija hazarda po podacima

- * Ako su I i J dve instrukcije, pri čemu I prethodi J, hazardi po podacima se u zavisnosti od redosleda upisa i čitanja mogu klasifikovati u tri grupe:
 - **RAW** (Read After Write) – nastupa kada instrukcija J pokušava da pročita operand pre nego što je instrukcija I obavila upis (najčešći tip hazarda)
 - **WAR** (Write After Read) – nastupa kada instrukcija J pokušava da upiše novu vrednost pre nego što je instrukcija I obavila čitanje
 - **WAW** (Write After Write) – nastupa kada instrukcija J pokušava da upiše vrednost pre instrukcije I
 - **RAR** nije hazard.
- * WAR i WAW su hazardi imenovanja (name dependencies) a RAW su pravi hazardi (true dependencies)

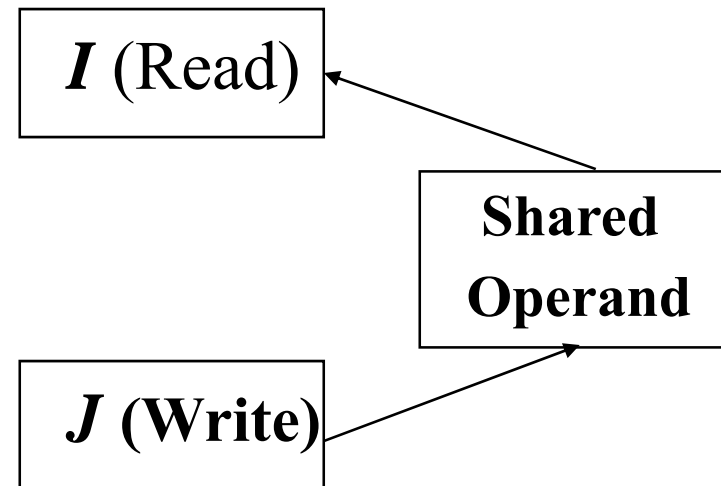
Klasifikacija hazarda



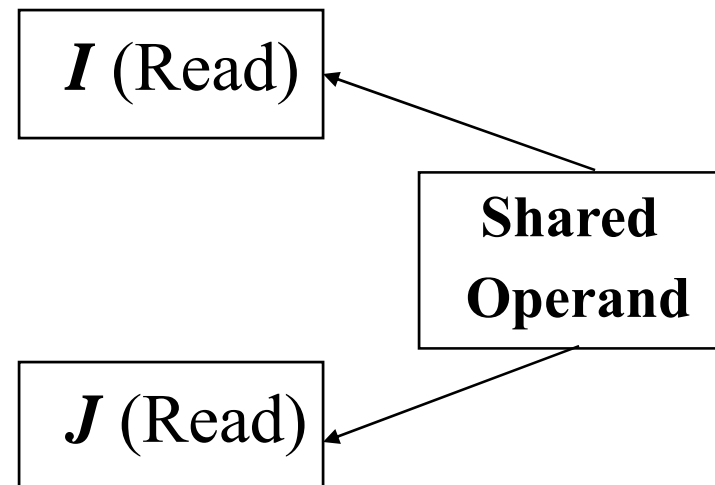
Read after Write (RAW)



Write after Write (WAW)



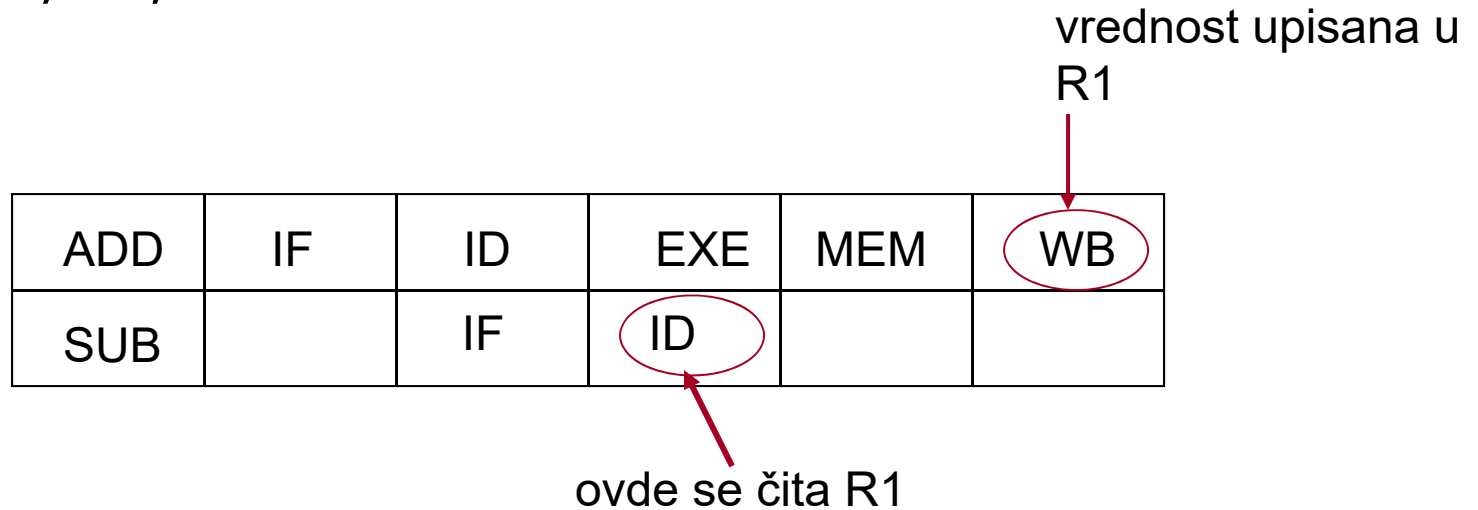
Write after Read (WAR)



Read after Read (RAR) nije hazard

RAW Hazard

ADD R1, R2, R3
SUB R4, R1, R5



Zaustavljanje protočnog sistema uzrokovano RAW hazardima se može eliminisati pribavljanjem unapred (bypassing, forwarding)

kontrolni hazardi

- * Mogu uzrokovati veći gubitak performansi nego hazardi po podacima.
- * Nastupaju zbog instrukcija koje mogu promeniti sadržaj PC (branch, jump, call, return).
- * Primer branch instrukcije: novi sadržaj PC poznat tek u MEM fazi, posle izračunavanja adrese i testiranja uslova.
- * Neophodno zaustaviti protočni sistem dok se ne dozna novi sadržaj PC.

branch	IF	ID	EX	MEM	WB
i+1		IF	--	--	IF

zaustavljanje protočnog sistema nije moguće odmah nakon pribavljanja branch jer nije završeno dekodiranje. Pribavljena instr. se briše (IF/ID registar)
Nakon Mem faze vrši se novo pribavljanje

Zakašnjeno grananje

- * Ideja je da se iza naredbe grananja postavi instrukcija koja će se izvršiti bez obzira da li će se grananje obaviti ili ne (dok se ne odredi uslov i novi sadržaj PC).
- * Naziv potiče od činjenice da se efekat naredbe grananja odlaže.

conditional branch instruction

sequential successor₁

sequential successor₂

.....

sequential successor_n

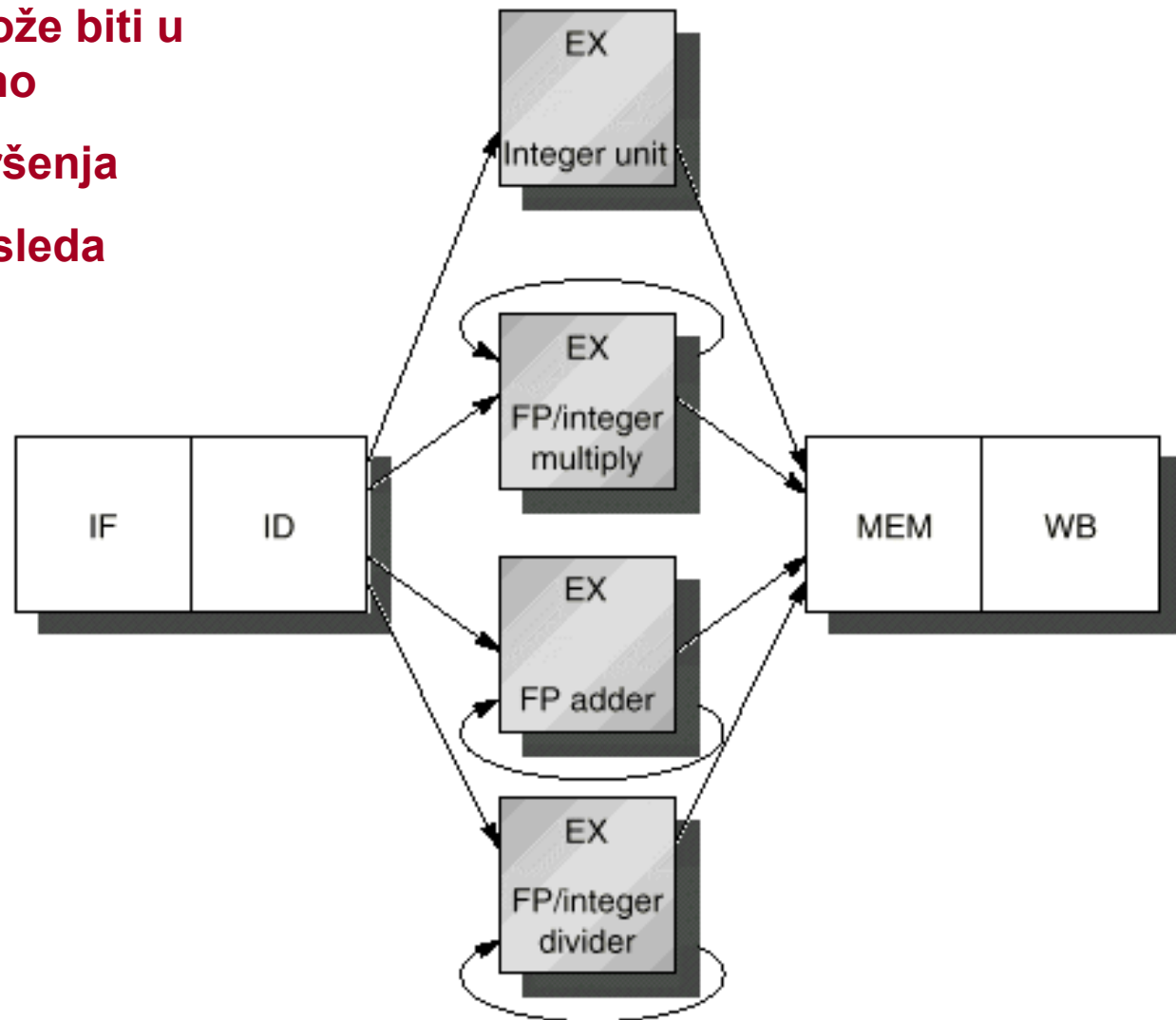
branch target if taken

slot zakašnjelog grananja

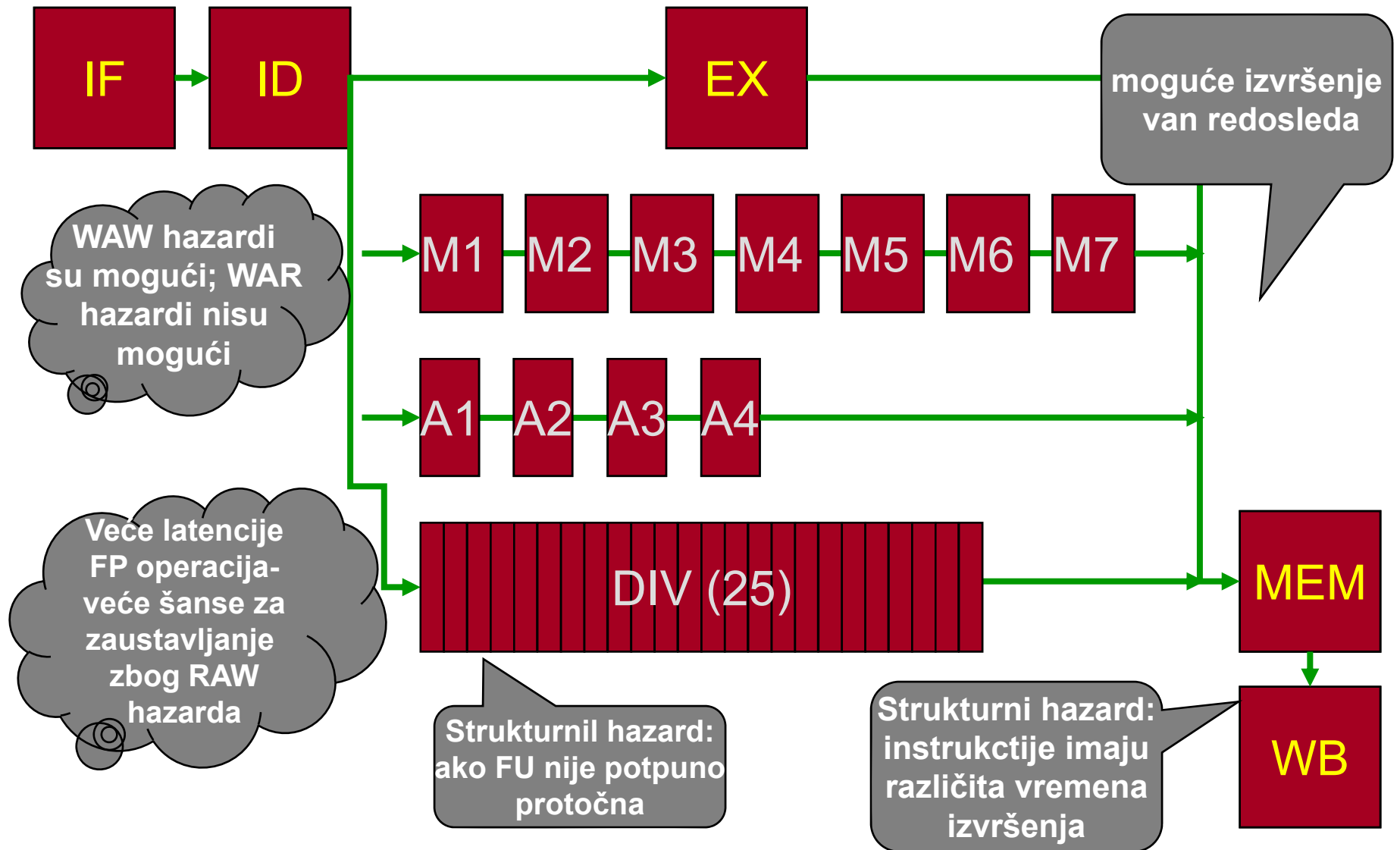
- * Za instrukcije koje slede iza naredbe grananja se kaže da se nalaze u slotu (prozoru) zakašnjelog grananja. Ove instrukcije se izvršavaju bez obzira da li dolazi do grananja ili ne.
- * U praksi je veličina prozora najčešće 1.
- * Zadatak kompajlera je da u ovaj prozor postavi važeće i korisne instrukcije

Protočni sistem sa FP funkcionalnim jedinicama

- više instrukcija može biti u EX fazi jednovremeno
- različito vreme izvršenja
- izvršenje van redosleda pribavljanja



FP Operacije – posledice



- Paralelizam na nivou instrukcija
- Odmotavanje petlje

Performanse protočnog sistema u prisustvu hazarda

- * Osnovni cilj uvođenja protočnosti u izvršenju instrukcija je postizanje CPI (Clocks per Instruction) od 1instr/clock
- * Zbog postojanja hazarda, realni CPI je veći

$$\text{Pipeline CPI} = \text{Ideal Pipeline CPI} + \text{Structural Stalls} + \text{RAW Stalls} + \text{WAR Stalls} + \text{WAW Stalls} + \text{Control Stalls}$$

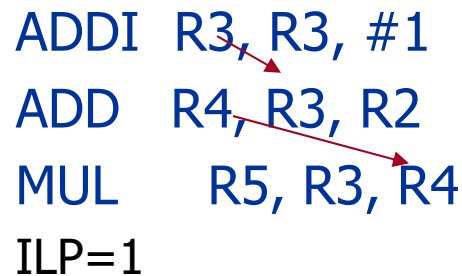
- * Da bi se postigle bolje performanse potrebno je iskoristiti paralelizam koji postoji na nivou instrukcija (ILP – Instruction Level Parallelism)

- LW R1, 30(R2)
- ADDI R3, R3, #1
- ADD R4, R4, R5

➤ ILP=3

ADDI R3, R3, #1
ADD R4, R3, R2
MUL R5, R3, R4

ILP=1



Kako povećati ILP?

- * Količina raspoloživog ILP u okviru tzv. osnovnog bloka (pravolinijski kod koji ne sadrži naredbe grananja izuzev na početku i na kraju) je veoma mala
- * Dinamika pojavljivanja naredbi grananja u programima je oko 15%, što znači da se 6-7 instrukcija izvršava između dve naredbe grananja
 - instrukcije unutar osnovnog bloka su verovatno zavisne jedna od druge, pa je ILP manji od 6.
- * Da bi se postiglo značajnije poboljšanje performansi, mora se eksploatirati ILP van osnovnog bloka.
- * Najjednostavniji način za povećanje ILP je da se iskoristi paralelizam između različitih iteracija petlje
 - paralelizam koji postoji između različitih iteracija petlje se zove LLP – Loop Level Parallelism
 - Primer:
 - for i:=1 to 100
 - x(i):=x(i)+y(i)

Ova petlja je potpuno paralelna. Svaka iteracija petlje može se preklapati u izvršenju sa bilo kojom drugom iteracijom

kako povećati ILP?

- * Postoji puno tehnika za konverziju LLP u ILP. Tehnike rade tako što vrše odmotavanje petlje, statički uz pomoć kompajlera, ili dinamički pomoću hw, čime se povećava veličina osnovnog bloka.
- * Da bi se izbeglo zaustavljanje protočnog sistema zbog nastupanja hazarda, instrukcije koje zavise jedna od druge moraju se razdvojiti drugim instrukcijama.
 - broj instrukcija kojima se moraju razdvojiti medjusobno zavisne instrukcije tako da se izbegne zaustavljanje protočnog sistema zavisi od latentnosti funkcionalne jedinice koja generiše rezultat.
 - da li će kompajler moći da izvrši preuredjenje koda tako da ne nastupi zastoje, zavisi od količine raspoloživog ILP.

Usvojene latencije funkcionalnih jedinica

* Sve funkcionalne jedinice su potpuno protočne
(period inicijacije = 1)

- latencije FP funkcionalnih jedinica:

Instrukcija koja generiše rezultat	Instrukcija koja koristi rezultat	Latentnost u Clock Cycles
FP ALU Op	druga FP ALU Op	3
FP ALU Op	Store Double	2
Load Double	FP ALU Op	1
Load Double	Store Double	0

- latentnost ALU integer operacija je 0, latentnost za LOAD i Branch 1 clk

Odmotavanje petlje – Primer

```
for (i=1000; i>0; i=i-1)  
    x[i] = x[i] + s;
```

- Petlja je potpuno paralelna, jer je svaka iteracija petlje nezavisna.
- Prvi korak u odmotavanju je prevodjenje na assembler.
 - usvajamo:
 - registar R1 inicijalno sadrži adresu poslednjeg elementa polja
 - Registar F2 sadrži skalarnu vrednost S
 - element x[1] se nalazi na adresi 0.

Loop: LD F0, 0(R1) ;F0= element vektora
ADDD F4, F0, F2 ;saberiti sa skalarom iz F2
SD 0(R1), F4 ;zapamti rezultat
SUBI R1, R1, 8 ;dekrementirati pointer za 8B (DW)
BNEZ R1, Loop ;skok na Loop ako je R1 ≠ 0

Kako izgleda protočno izvršenje instrukcija sa usvojenim latencijama:

1	Loop: LD	F0, 0(R1)	;F0=vector element
2		zastoj	
3		ADDD F4, F0, F2	;add scalar in F2
4		zastoj	
5		zastoj	
6		SD 0(R1), F4	;store result
7		SUBI R1, R1, 8	;decrement pointer 8B (DW)
8		BNEZ R1, Loop	;branch ako je R1≠ 0
9		zastoj	;delayed branch slot

9 clk/iteraciji: može li se kod preurediti da se minimiziraju zastoji?

Preuredjeni kod sa minimiziranim zastojsima

```
1 Loop: LD      F0, 0(R1)
2          zastoj
3          ADDD   F4, F0, F2
4          SUBI   R1, R1, 8
5          BNEZ   R1, Loop      ;delayed branch
6          SD     8(R1), F4      ;instrukcija postavljena u slot zak. grananja
```

Preuredjenjem instrukcija vreme izvršenja je smanjeno sa 9 na 6 clk/iteraciji.

- Jedna iteracija petlje se obavi za 6 clk, ali korisni posao nad elementima polja zahteva samo 3 clk ciklusa (LD, ADD i SD). Preostala 3 clk ciklusa potiču od instrukcija za obradu petlje (SUBI i BNEZ) i zastoja zbog RAW hazarda uzrokovanog LD instrukcijom (50% gubitaka)

Kako dalje smanjiti broj clk/iteraciji?

- * Potrebno je da postoji više “korisnih” instrukcija unutar petlje u odnosu na ukupan broj instrukcija.
- * Jednostavan način za povećanje broja “korisnih” instrukcija je odmotavanje petlje.
- * Odmotavanje se obavlja repliciranjem tela petlje više puta i podešavanjem dela koda za okončanje petlje
- * Da bi se izbegli zastoji, različite iteracije tela petlje koriste različite registre

Petlja odmotana četiri puta

```
1 Loop: LD      F0, 0(R1)
2      ADDD     F4, F0, F2
3      SD       0(R1), F4      ; uklonjene SUBI & BNEZ
4      LD       F6, -8(R1)
5      ADDD     F8, F6, F2
6      SD       -8(R1), F8     ; uklonjene SUBI & BNEZ
7      LD       F10, -16(R1)
8      ADDD     F12, F10, F2
9      SD       -16(R1), F12   ; uklonjene SUBI & BNEZ
10     LD       F14, -24(R1)
11     ADDD     F16, F14, F2
12     SD       -24(R1), F16
13     SUBI     R1, R1, #32     ; sve SUBI su objedinjene u jednu (4*8=32)
14     BNEZ     R1, LOOP
```

pretpostavka je da je R1 umnožak od 32

Protočno izvršenje odmotane petlje

```
1 Loop: LD      F0, 0(R1)
2      zastoj
3      ADDD     F4, F0, F2
4,5    zastoj, zastoj
6      SD       0(R1), F4
7      LD       F6, -8(R1)
8      zastoj
9      ADDD     F8, F6, F2
10,11  zastoj, zastoj
12     SD       -8(R1), F8
13     LD       F10, -16(R1)
14     zastoj
15     ADDD     F12, F10, F2
16,17  zastoj, zastoj
18     SD       -16(R1), F12
19     LD       F14, -24(R1)
20     zastoj
21     ADDD     F16, F14, F2
22,23  zastoj, zastoj
24     SD       -24(R1), F16
25     SUBI     R1, R1, #32
26     BNEZ     R1, LOOP
27     zastoj
```

27/4= 6.8 clk/iteraciji

```
1 Loop: LD      F0, 0(R1)
2      LD       F6, -8(R1)
3      LD       F10, -16(R1)
4      LD       F14, -24(R1)
5      ADDD     F4, F0, F2
6      ADDD     F8, F6, F2
7      ADDD     F12, F10, F2
8      ADDD     F16, F14, F2
9      SD       0(R1), F4
10     SD       -8(R1), F8
11     SD       -16(R1), F12
12     SUBI     R1, R1, #32
13     BNEZ     R1, LOOP
14     SD       8(R1), F16; 8-32 = -24
```

14 clk ciklusa, ili 3.5 clk po iteraciji

Rezime-odmotavanja petlje

* U prethodnom primeru ključna su sledeća zapažanja:

- Otkriti da je legalno premestiti SD posle SUBI i BNEZ; pronaći offset za SD.
- Utvrditi da su iteracije petlje nezavisne i da će odmotavanje dovesti do poboljšanja.
- Korišćenje različitih registara za različite iteracije da bi se izbegli **WAR** i **WAW** hazardi.
- Eliminisanje ekstra testiranja i grananja i podešavanjem koda za okončanje petlje.
- Load i store iz različitih iteracija mogu zameniti mesta
- Preuredjenje koda tako da se sačuvaju sve zavisnosti koje obezbeđuju korektno izvršenje programa.

Superskalarni i VLIW procesori

- * Tehnika odmotavanja petlje ima za cilj da poveća količinu raspoloživog ILP-a
- * Scoreboard i Tomasulo tehnike imaju za cilj da postignu idealni CPI od 1inst/clk
- * CPI ne može biti <1 ako se pribavlja i izdaje jedna instrukcija u clk ciklusu
- * Da bi se CPI dalje redukovao potrebno je pribaviti i izdati više od jedne instrukcije u jednom clk ciklusu.
 - Rad superskalarnih (SS) i VLIW (Very Long Instruction Word) procesora se zasniva na ovoj ideji

SS i VLIW

* SS procesori mogu da izdaju različit broj instrukcija po clk ciklusu

- Kod tipičnog SS procesora hw može da izda od 1 do 8 instrukcija u jednom clk ciklusu (u zavisnosti od raspoloživog ILP-a)
- SS procesori mogu da koriste statičko planiranje izvršenja instrukcija (uz pomoć kompajlera) ili dinamičko zasnovano na Sc tehnici i Tomasulovom algoritmu

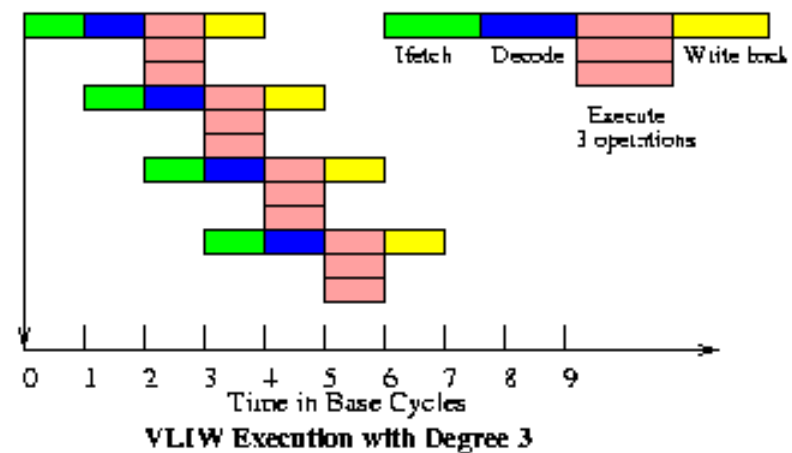
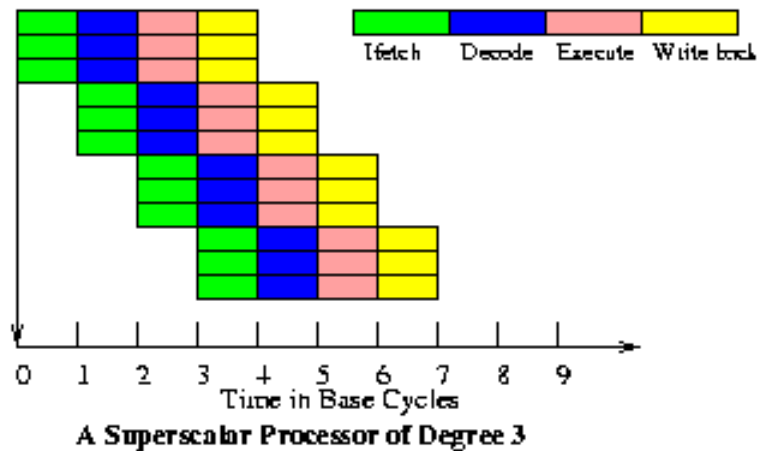
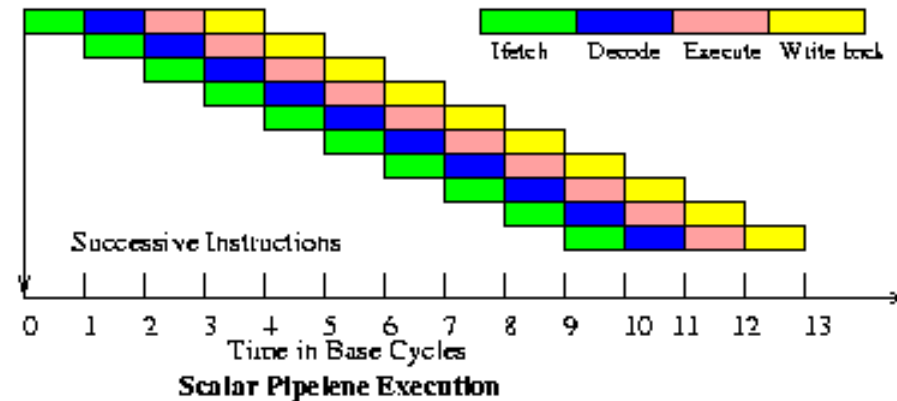
➤ IBM PowerPC, Sun UltraSparc, DEC Alpha, HP 8000, MIPS 10000, AMD K5

* VLIW (zovu se još i EPIC – Explicitly Parallel Instruction Computer) izdaje fiksni broj instrukcija koje su formatirane kao jedna velika instrukcija ili kao fiksni instrukcioni paket. (paralelne instrukcije su grupisane u blokove)

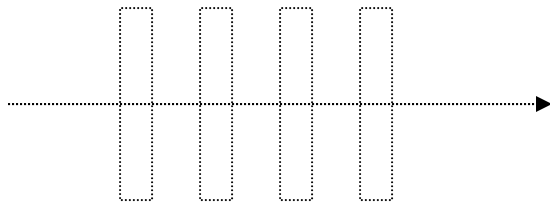
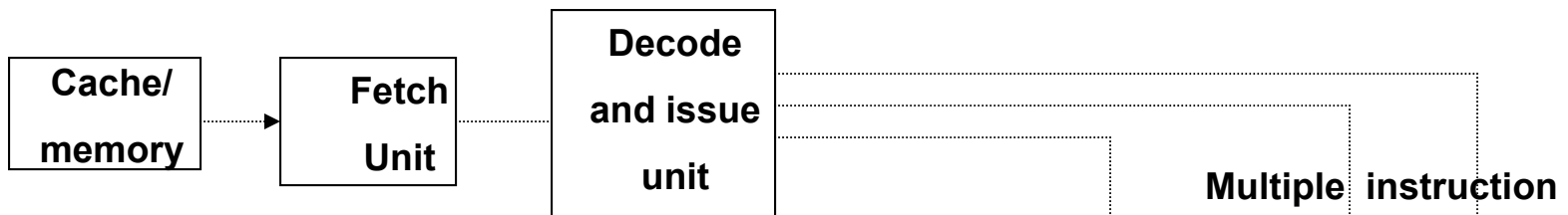
- VLIW su po definiciji sa statičkim planiranjem izvršenja instrukcija

➤ Intel Itanium

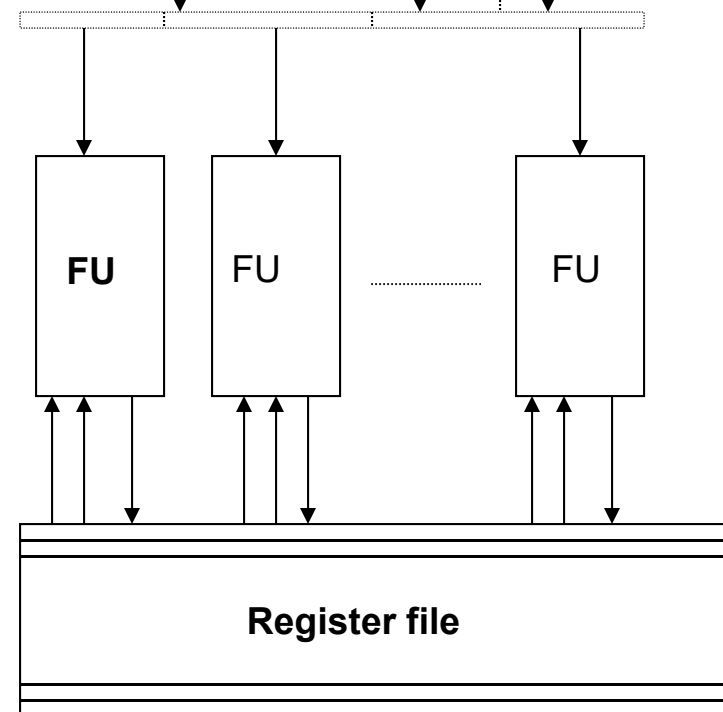
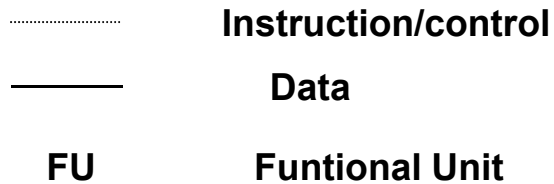
Superskalarni i VLIW procesori



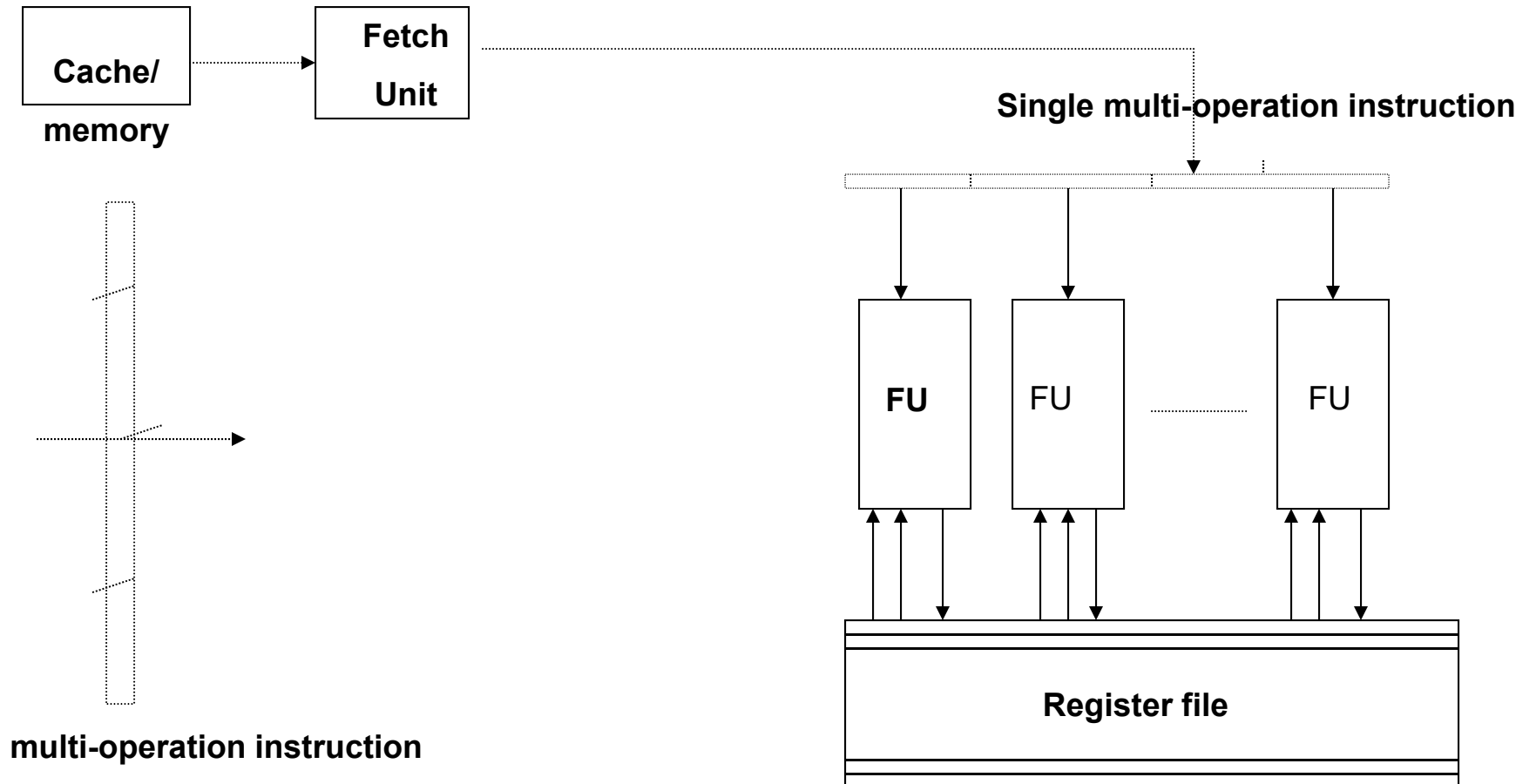
Superskalarni procesori



Sequential stream of instructions



VLIW procesori



SS primer

```
Loop:  LD    F0 , 0 ( R1 )
        ADDD  F4 , F0 , F2
        SD    0 ( R1 ) , F4
        SUBI  R1 , R1 , #8
        BNEZ  R1 , Loop
```

Instrukcija koja generiše rezultat	Instrukcija koja koristi rezultat	Latentnost u Clk
FP ALU operacija	druga FP ALU operacija	3
FP ALU operacija	Store double (SD)	2
Load double (LD)	FP ALU operacija	1
Load double (LD)	Store double (SD)	0

Latencije FP operacija

SS primer

- * Instrukcije koje se mogu jednovremeno izdavati moraju biti nezavisne
- * U toku jednog clk ciklusa samo jedno obraćanje memoriji je moguće
 - Ako neka instrukcija u nizu zavisi od neke prethodne ili ne zadovoljava kriterijum za izdavanje, samo instrukcije koje joj prethode biće izdate
- * **PRIMER: izdavanje dve instrukcije**
 - **Prva:** Jedna load/store/branch/integer-ALU op.
 - **Druga:** Jedna floating-point op.
 - Uvek je prva instrukcija u paru Integer instrukcija
 - Druga instrukcija se može izdati samo ako se prva može izdati
 - Izdavanje Integer instrukcije paralelno sa FP operacijom je mnogo manje zahtevno od izdavanja dve proizvoljne instrukcije – **koriste različite funkcionalne jedinice i različite skupove registara** (problem može da nastupi ako su u pitanju load i store u FP registe)

SS primer

- * Latentnost load je 1 clk, pa se kod SS procesora rezultat load ne može koristiti u istom i narednom clk ciklusu (tj. u naredne 3 instrukcije)
- * Kašnjenje za branch takodje može biti 3 instrukcije, jer branch mora biti prva u paru
- * Za odabrani primer potrebno je 5 puta odmotati petlju da bi se izbeglo zaustavljanje protočnog sistema kod SS procesora (kod skalarnog - 4 puta)

Instruction type		Pipe stages						
Integer instruction	IF	ID	EX	MEM	WB			
FP instruction	IF	ID	EX	MEM	WB			
Integer instruction		IF	ID	EX	MEM	WB		
FP instruction		IF	ID	EX	MEM	WB		
Integer instruction			IF	ID	EX	MEM	WB	
FP instruction			IF	ID	EX	MEM	WB	
Integer instruction				IF	ID	EX	MEM	WB
FP instruction				IF	ID	EX	MEM	WB

FIGURE 4.26 Superscalar pipeline in operation.

Odmotavanje petlje koje minimizira zastoje kod skalarnog procesora

1	Loop:	LD	F0, 0(R1)	LD to ADDD: 1 Cycle
2		LD	F6, -8(R1)	ADDD to SD: 2 Cycles
3		LD	F10, -16(R1)	
4		LD	F14, -24(R1)	
5		ADDD	F4, F0, F2	
6		ADDD	F8, F6, F2	
7		ADDD	F12, F10, F2	
8		ADDD	F16, F14, F2	
9		SD	0(R1), F4	
10		SD	-8(R1), F8	
12		SUBI	R1, R1, #32	
11		SD	16(R1), F12	
13		BNEZ	R1, LOOP	
14		SD	8(R1), F16	

14 clk ciklusa, ili 3.5 po iteraciji

Superskalarno izvršenje

	<i>Integer instruction</i>	<i>FP instruction</i>	<i>Clock cycle</i>
Loop:	LD F0,0(R1)		1
	LD F6,-8(R1)		2
	LD F10,-16(R1)	ADDD F4,F0,F2	3
	LD F14,-24(R1)	ADDD F8,F6,F2	4
	LD F18,-32(R1)	ADDD F12,F10,F2	5
	SD 0(R1),F4	ADDD F16,F14,F2	6
	SD -8(R1),F8	ADDD F20,F18,F2	7
	SD -16(R1),F12		8
	SUBI R1,R1,#40		10
	SD 16(R1),F16		9
	BNEZ R1,LOOP		11
	SD 8(R1),F20		12

* 12 clk, ili 2.4 clk po iteraciji

Primeri nekih SS procesora

* PowerPC 604

- six independent execution units:
 - Branch execution unit
 - Load/Store unit
 - 3 Integer units
 - Floating-point unit
- in-order issue
- register renaming

* Power PC 620

- provides in addition to the 604 out-of-order issue

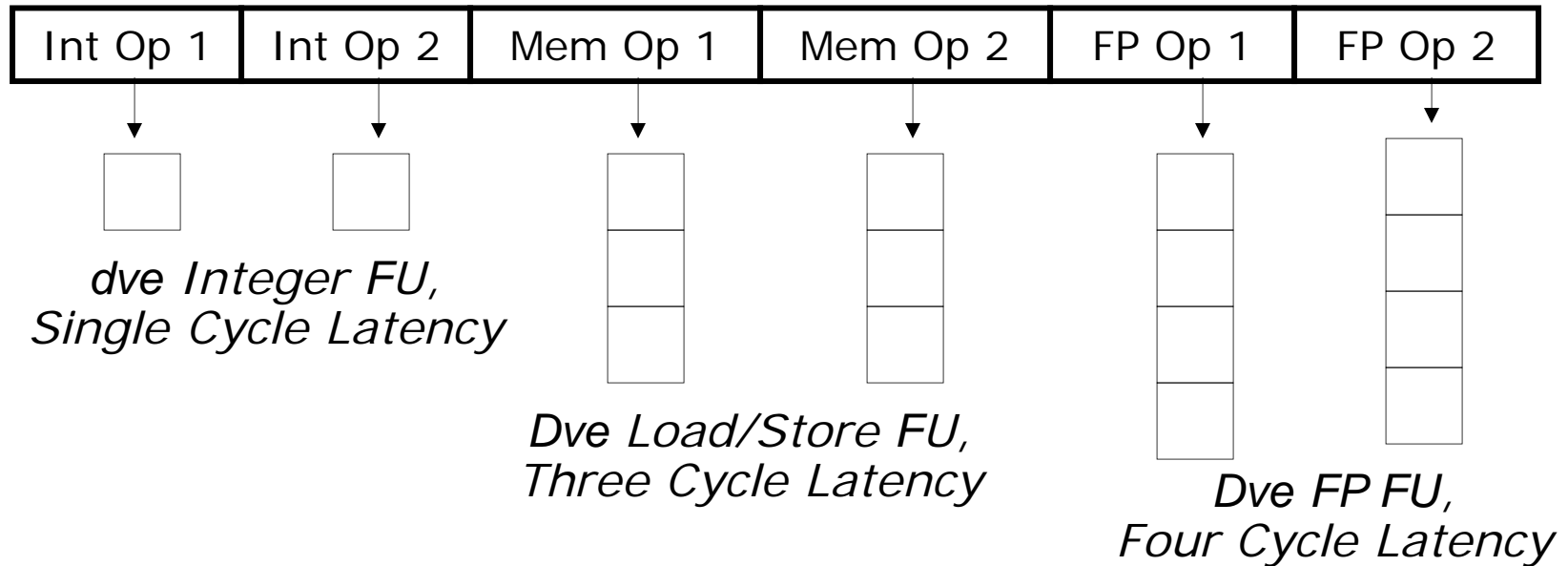
* Pentium

- three independent execution units:
 - 2 Integer units
 - Floating point unit
- in-order issue

VLIW

- * Upravljanje se ostvaruje veoma dugim instrukcijama
 - Instrukcija sadrži upravljačko (control) polje za svaku od funkcionalnih jedinica
- * Dužina instrukcije zavisi od
 - Broja funkcionalnih jedinica (5-30 FU)
 - Dužine polja control za svaku FU je 16-32 bita.
 - Dužina instrukcije od 256 do 1024 bita
- * Planiranje izvršenja instrukcija (tj. Formiranje duge instrukcije) se obavlja softverski (kompajler)
- * Manja hardverska kompleksnost se može iskoristiti
 - Da se poveća učestanost kojom se taktuje procesor (clock rate)
 - Poveća nivo paralelizma kroz ugradnju većeg broja FU
- * Mane:
 - u proseku samo jedan broj upravljačkih (control) polja se realno koristi (prazna polja se popunjavaju NOP operacijama).
 - Složeni kompajleri
 - Kompajler mora da vodi računa o hardverskim detaljima
 - Broj FU, latentnost FU, period iniciranja FU,..
 - Keš promašaji: kompajler mora da uzme u obzir najgori mogući slučaj
 - Zavisnost kompajlera od hardvera sprečava korišćenje istog kompajlera za familiju VLIW procesora

VLIW: Very Long Instruction Word



- * Više operacija u jednom instrukcionom paketu
- * Svako polje u instrukcionom paketu je za fiksnu FU
- * Latencije FU su konstantne
- * Između instrukcija koje se nalaze u jednom paketu nema nikakvih zavisnosti

VLIW primer

- * Neka instrukcioni paket ima 5 instrukcija
 - 2 FP, 2 Memory, 1 branch/integer
- * Kompajler detektuje sve hazrde
 - po definiciji sve instrukcije koje kompajler postavi u jednu veliku instrukciju (paket) su nezavisne, tj. mogu se paralelno izvršavati
- * Svi slotovi u instrukcionom paketu ne moraju biti uvek popunjeni

Odmotavanje petlje i VLIW

<i>Memory reference 1</i>	<i>Memory reference 2</i>	<i>FP operation 1</i>	<i>FP op. 2</i>	<i>Int. op/ branch</i>	<i>Clock</i>
LD F0, 0 (R1)	LD F6, -8 (R1)				1
LD F10, -16 (R1)	LD F14, -24 (R1)				2
LD F18, -32 (R1)	LD F22, -40 (R1)	ADDD F4, F0, F2	ADDD F8, F6, F2		3
LD F26, -48 (R1)		ADDD F12, F10, F2	ADDD F16, F14, F2		4
		ADDD F20, F18, F2	ADDD F24, F22, F2		5
SD 0 (R1), F4	SD -8 (R1), F8	ADDD F28, F26, F2			6
SD -16 (R1), F12	SD -24 (R1), F16				7
SD -32 (R1), F20	SD -40 (R1), F24			SUBI R1, R1, #48	8
SD 0 (R1), F28				BNEZ R1, LOOP	9

- petlja odmotana 7 puta da bi se izbegli zastoji
- 9 clk, ili 1.3 clk po iteraciji
- 2.5 operacije po clk, 50% efikasnosti

Napomena: Potrebno je više registara kod VLIW nego kod SS (15 naspram 6 kod SS)

Prednosti VLIW

Kompajler priprema fiksne instrukcione pakete koji sadrže više operacija, tj. pravi plan izvršenja

- Zavisnosti detektuje kompajler i koristi ih da planira izvršenje instrukcija
- Funkcionalne jedinice dodeljuje kompajler na osnovu pozicije u instrukcionom paketu
- Kompajler generiše kod koji nema nikakvih hazarda tako da nema potrebe za hardverom za detekciju zavisnosti ili planiranjem izvršenja

Nedostaci VLIW

Kompatibilnost koda

- VLIW code se ne može korektno izvršavati na mašini sa različitim brojem funkcionalnih jedinica i/ili različitim latencijama FU.

Gustina koda

- Niska iskorišćenost slotova unutar dugačke instrukcije (uglavnom nop-ovi)