7
PARALELIZEM NA NIVOJU UKAZOV

Z doslej obravnavanim načinom gradnje CPE je težko doseči CPI < 4

- zaporedno izvrševanje

Število ukazov na sekundo je

$$IPS = f_{CPE} / CPI$$

IPS ... Instructions Per Second

f_{CPE} ... frekvenca ure

CPI ... Clocks Per Instruction

IPS lahko povečamo:

- s povečanjem f_{CPE} (hitrejši logični elementi)
- z drugačno zgradbo CPE, ki bi zmanjšala CPI (več logičnih elementov)

V 20 letih se hitrost elementov poveča ~ 10x, št. elementov na čipu pa ~ 1000x

- zato je druga varianta bolj perspektivna

Z uporabo večjega števila elementov skušamo zmanjšati *CPI* (in s tem povečati *IPS*)

- Skušamo doseči čimvečjo **paralelnost** (istočasnost) operacij

Ena možnost je paralelno programiranje

- programer določi, kaj naj se izvaja paralelno
- dokaj kompliciranо: potrebujemo izvorno kodo in znanje, kako to narediti
- večina uporabnikov se s tem ne želi ukvarjati

Enostavneje je izkoristiti **paralelizem na nivoju ukazov** (instruction-level parallelism, ILP)

Najpogostejši način je **cevovod** (pipeline)

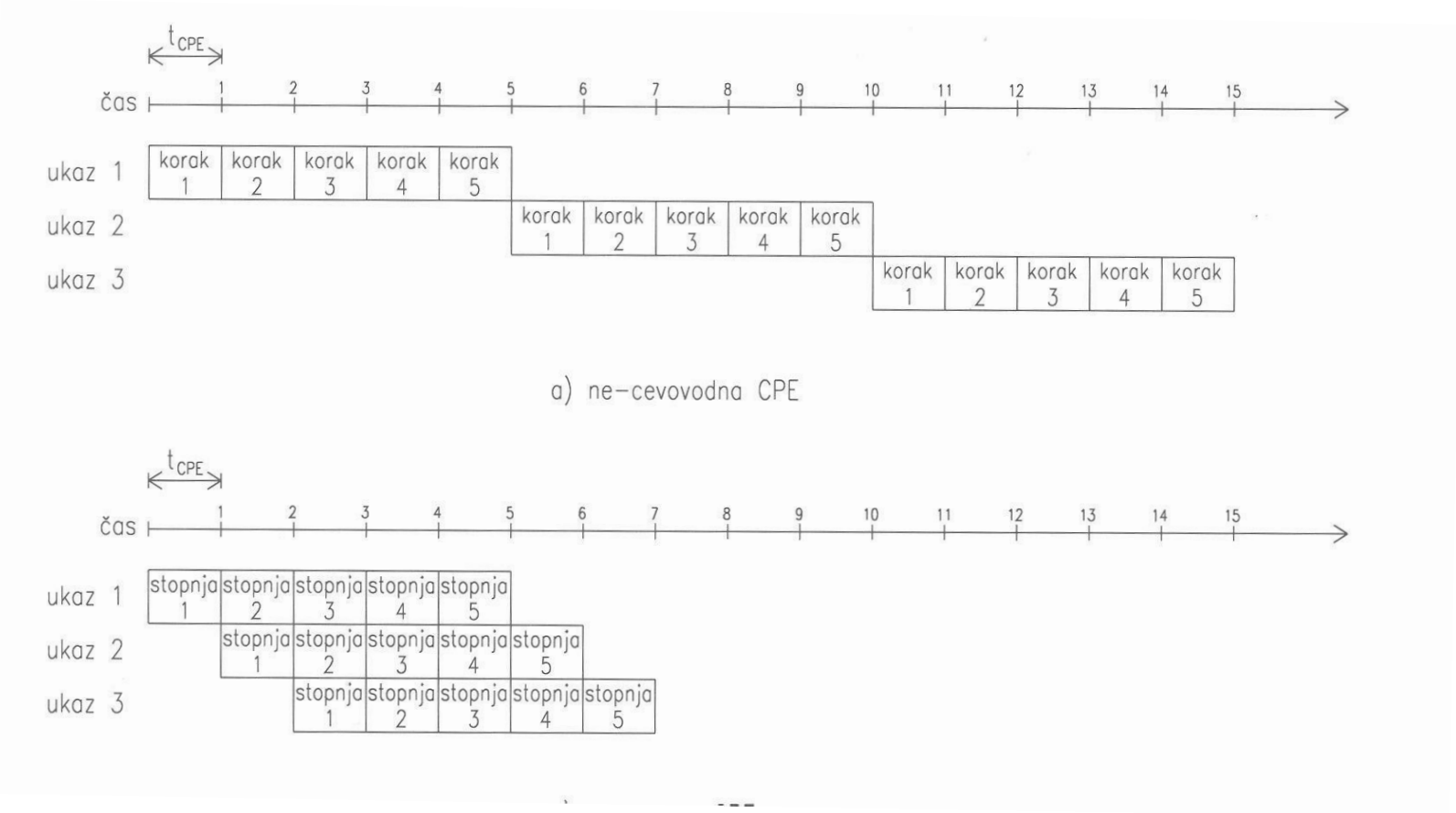
Cevovod - splošno

Cevovod (pipeline)

- Pri cevovodu se naenkrat izvršuje več ukazov tako, da se posamezni koraki izvrševanja prekrivajo
  - Podobno tekočemu traku pri proizvodnji
- Vsako podoperacijo opravi določen del cevovoda
  - **stopnja cevovoda** (pipeline stage) ali **segment cevovoda**
- Stopnje tvorijo nekakšno cev
  - ukazi vstopajo v cev, potujejo skoznjo in izstopajo na koncu cevi

Primer: ne-cevovodna CPE in cevovodna CPE

- v drugem primeru se izvrši 5x več ukazov (če zanemarimo začetno zakasnitev)



Čas med dvema pomikoma je enak urini periodi t_{CPE}

Uravnoteženost:

- Perioda ne more biti krajše od časa, ki ga za izvršitev svoje podoperacije potrebuje najpočasnejša izmed stopenj cevovoda
- zato je dobro, če so podoperacije časovno uravnotežene

Pri idealno uravnoteženi cevovodni CPE z *N* stopnjami je zmogljivost *N*-krat večja kot pri ne-cevovodni CPE

- hkrati se obdeluje *N* ukazov
- na izhodu iz cevovoda jih je zato *N*-krat več
- CPI *N*-krat manjši
- resnični cevovodi pa nikoli niso idealno uravnoteženi, zato zmogljivost ni *N*-kratna

Število izvršenih ukazov v danem času se poveča zaradi 2 vzrokov:

1. manjši CPI
   - čeprav je trajanje ukaza (**latenca**) enako ($N*t_{\text{CPE}}$)
   - $CPI = \frac{I+(N-1)}{I}$
     - pri velikem številu ukazov približno 1
2. krajša t_{CPE}
   - če uspemo narediti enostavne podoperacije oz. korake
   - $t_{\text{CPE}} = t_{\text{podoperacija}} + t_{\text{shranjevanje}}$

     $t_{\text{shranjevanje}}$ ... čas shranjevanja rezultata podoperacije v registre

   - z več stopnjami lahko zmanjšamo $t_{\text{podoperacija}}$, $t_{\text{shranjevanje}}$ pa ne
     - torej obstaja neka 'režija', ki se ji ne da izogniti

Supercevovodni računalniki

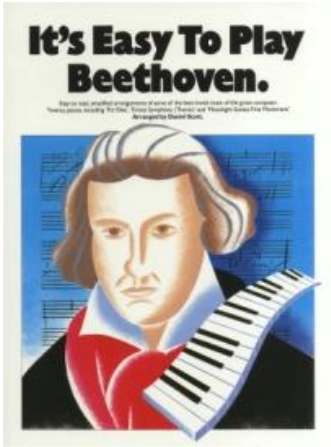
- Intel Pentium 4
  - 20-stopenjski, kasneje 31-stopenjski cevovod
    - želeli so doseči f_{CPE} 10GHz, a je bila poraba prevelika (problemi s hlajenjem, tj. odvajanjem toplote)
    - kasnejši CPU (npr. Core) imajo (le) 14-stopenjski cevovod

Mimogrede: najvišja dosežena frekvenca je okrog 9 GHz

- s pomočjo navijanja frekvence (overclocking), pa tudi polivanja čipa s tekočim dušikom (ali helijem)

Zakaj se ne uporabljajo poljubno dolgi cevovodi?

- pač povečujemo N in višamo hitrost CPU …



Cevovodne nevarnosti

Razlog so **cevovodne nevarnosti** (pipeline hazards), zaradi katerih se mora cevovod ustaviti in počakati, da nevarno mine

3 vrste cevovodnih nevarnosti:

1. **Strukturne nevarnosti**
   - kadar več stopenj cevovoda v neki urini periodi potrebuje isto enoto
2. **Podatkovne nevarnosti**
   - kadar ukaz potrebuje kot vhodni operand rezultat prejšnjega, še ne dokončanega ukaza
3. **Kontrolne nevarnosti**
   - možne pri skokih, klicih in drugih kontrolnih ukazih, ki spreminjajo vsebino PC

Zato zmogljivost z večanjem števila stopenj nekaj časa narašča, nato pa začne padati!

Prednost cevovoda je, da ga je mogoče narediti tako, da je za programerja neviden

- arhitektura računalnika in programiranje ostane enako tudi z razvojem računalnika
  - starejši programi tečejo tudi na novejših računalnikih
- pri drugih vrstah paralelnega procesiranja to pogosto ne velja
- zadnji Intelov necevovodni procesor je bil 80386 (iz leta 1985)

Cevovodna podatkovna enota

Cevovodna realizacija je pri računalnikih RISC enostavnejša kot pri CISC

- preprostejši ukazi

Cevovodno CPE si bomo zato ogledali na primeru računalnika RISC

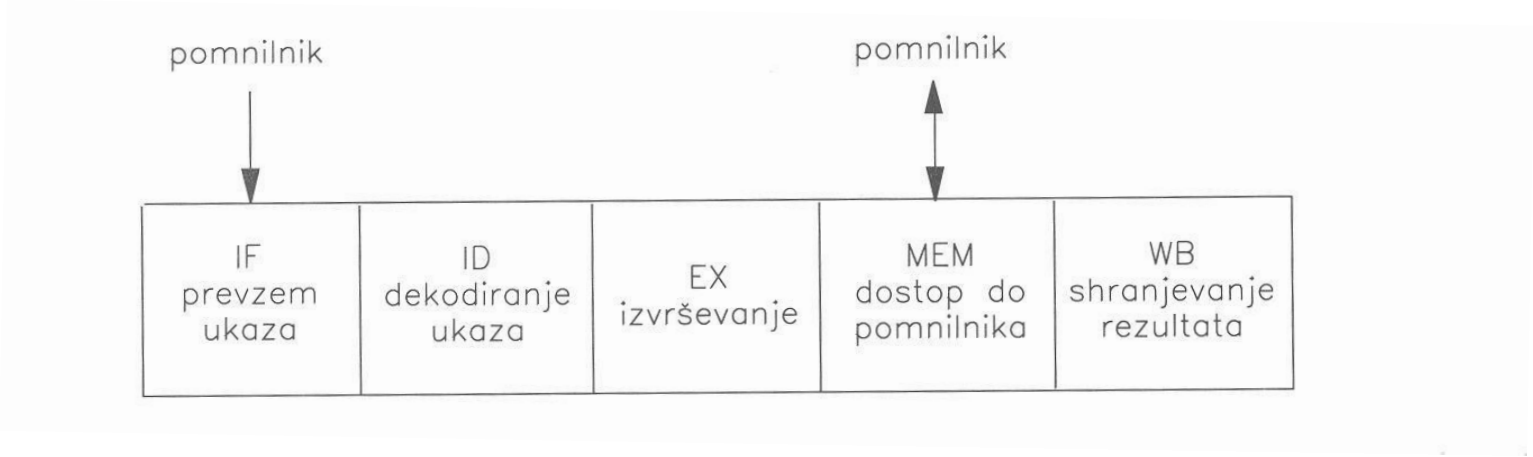
5 korakov izvrševanja ukazov: 5 stopenj cevovoda

1. IF: prevzem ukaza in sprememba PC
2. ID: dekodiranje ukaza in dostop do registrov
3. EX: izvrševanje operacije
4. MEM: dostop do pomnilnika
5. WB: shranjevanje rezultata

Vsaka stopnja mora opraviti svoje delo v eni urini periodi

- prej so nekateri koraki potrebovali 2 periodi

Klasična petstopenjska cevovodna podatkovna enota:

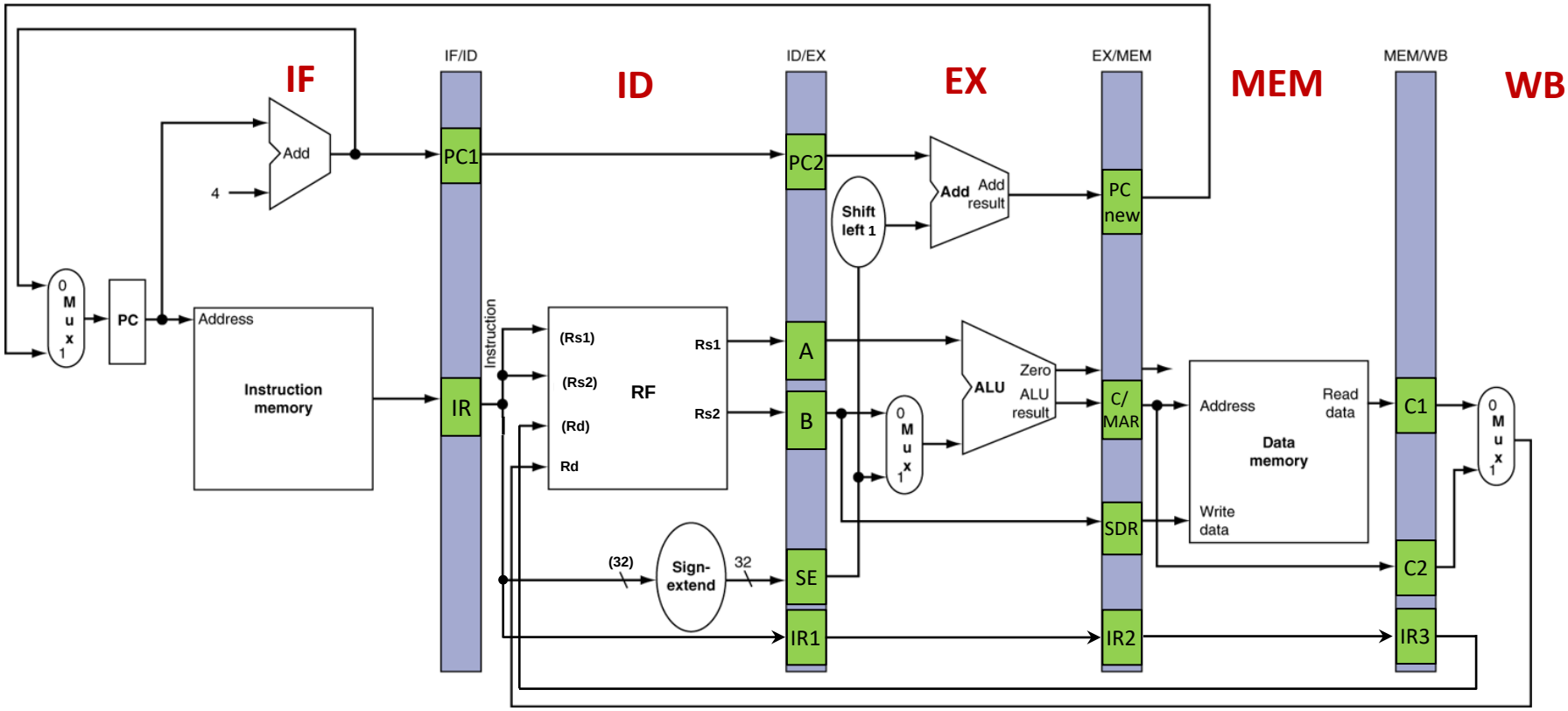


Včasih sta potrebna dva hkratna dostopa do pomnilnika

Cevovodni procesorji imajo zato 2 predpomnilnika:

- ukazni
- operandni

Cevovodna podatkovna enota



1. Stopnja IF

IR $\leftarrow_{32}$ IM[PC] IM ... instr. memory

PC $\leftarrow$ PC + 4 ali newPC

PC1 $\leftarrow$ PC+4

2. Stopnja ID (za load/store)

PC2 $\leftarrow$ PC1

A $\leftarrow$ Rs1

B $\leftarrow$ Rs2

SE $\leftarrow$ se(imm)

3. Stopnja EX

prehod v to stopnjo se imenuje **izstavitev ukaza** (instruction issue)

- tukaj se ukaza več ne da na preprost način izničiti (v IF in ID ni problema)

Delovanje stopnje EX je odvisno od vrste ukaza:

Ukazi za prenos podatkov

MAR $\leftarrow$ A + se(imm)

SDR $\leftarrow$ B

ALE ukazi

SDR $\leftarrow$ B

C $\leftarrow$ A op B (ali se(imm))

Skoki

NewPC $\leftarrow$ PC2 + 4 + se(imm)

4. Stopnja MEM

load

IR3 ← IR2

C1 ← M[MAR] branje iz operandnega PP

store

IR3 ← IR2

M[MAR] ← SDR SDR se shrani v operandni PP

Ostali ukazi

IR3 ← IR2

C1 ← C (zaradi WB)

5. Stopnja WB

ALE ukazi, load:

Rd ← C1

Idealen potek izvrševanja ukazov v cevovodu:

	Urina perioda										
Št. ukaza	1	2	3	4	5	6	7	8	9	10	11
ukaz i	IF	ID	EX	ME	WB						
ukaz i+1		IF	ID	EX	ME	WB					
ukaz i+2			IF	ID	EX	ME	WB				
ukaz i+3				IF	ID	EX	ME	WB			
ukaz i+4					IF	ID	EX	ME	WB		
ukaz i+5						IF	ID	EX	ME	WB	
ukaz i+6							IF	ID	EX	ME	WB

CEVOVODNE NEVARNOSTI

Ob pojavu nevarnosti se mora cevovod ustaviti in počakati, da nevarnost mine

Programsko odpravljanje cevovodnih nevarnosti

- pri prvih RISC (80. leta)
- vstavljanje ukazov NOP za ukaze, ki lahko povzročijo nevarnost
  - NOP ne spremeni stanja registrov
  - ekvivalentno čakanju eno urino periodo
  - pri višjih programskih jezikih jih vstavlja kar prevajalnik
- 2 slabosti:
  - potrebno je drugačno programiranje
  - upočasnjeno delovanje

Strukturne nevarnosti

- SN: kadar več stopenj cevovoda v neki urini periodi potrebuje isto enoto (reg., ALE, GP, PP)
- SN tudi na računalnikih, kjer nekateri ukazi trajajo več urinih period
  - Množenje, deljenje, FP operacije
- Izguba zaradi SN običajno bistveno manjša kot zaradi drugih nevarnosti
- Popolno odpravljanje SN je drago (večje število enot)
  - Na manjših računalnikih se pač dovoli, da do njih občasno pride
  - Če PP ne bi bil razdeljen, bi lahko prihajalo (load, store), sicer pa do SN ne prihaja

Podatkovne nevarnosti

- PN (data hazard): kadar ukaz potrebuje kot vhodni operand rezultat še ne dokončanega ukaza (taki nevarnosti rečemo tudi Read-After-Write oz. RAW)
  - medsebojna odvisnost ukazov, ki so blizu skupaj
- Npr.

```
addi   x20, x0, 0      # x20 ← 0
sub    x3, x4, x5      # x3 ← x4 - x5
add    x1, x3, x6      # x1 ← x3 + x6
and    x2, x3, x7      # x2 ← x3 & x7
xor    x8, x3, x9      # x8 ← x3 ∇ x9
or     x10, x3, x12    # x10 ← x3 ∨ x12
```

	Urina perioda										
Ukaz	1	2	3	4	5	6	7	8	9	10	11
addi x20, x0, 0	IF	ID	EX	ME	WB						
sub x3, x4, x5		IF	ID	EX	ME	WB					
add x1, x3, x6			IF	ID	EX	ME	WB				
and x2, x3, x7				IF	ID	EX	ME	WB			
xor x8, x3, x9					IF	ID	EX	ME	WB		
or x10, x3, x12						IF	ID	EX	ME	WB	

- Register x3 se zapiše v stopnji WB ukaza sub, ukaza add in add pa ga želita prebrati prej – rezultat bo (zelo verjetno) napačen!
- Ukaza xor in or nimata težave: or ga prebere kasneje, xor pa sicer v isti periodi, toda ...
  - Pri RISC-V bomo predpostavili, da ima implementacija na voljo dvofazni urin signal in da s tem lahko v prvi polovici urine periode zapiše vrednost v register (znotraj RF), v drugi polovici periode pa to vrednost lahko prebere
    - na ta način lahko v isti periodi ure isto vrednost zapiše in prebere (WB in ID)
    - brez dvofazne ure to ne bi bilo mogoče (v takem primeru bi bilo možno prebrati vrednost šele v naslednji periodi)

Rešitev 1:
Cevovodna zaklenitev (pipeline interlock)

- CPE ugotovi podatkovno nevarnost tako, da posebna *enota za ugotavljanje PN* (*hazard detection unit* – recimo kar HD) primerja *rd ukaza z rs1 in rs2 naslednjih 2 ukazov:*

1. Nevarnost v stopnji EX:

   EX/MEM.rd = ID/EX.rs1

   EX/MEM.rd = ID/EX.rs2

2. Nevarnost v stopnji MEM:

   MEM/WB.rd = ID/EX.rs1

   MEM/WB.rd = ID/EX.rs2

- V primeru, da ta enota odkrije enakost, se mora stopnja ID ustaviti za 2 periodi (zato se mora tudi IF, ker bi se sicer izgubil ukaz, ki je v njej).
  - V stopnji IF je ukaz add, ki ne gre naprej v ID
  - EX,MEM in WB morajo delovati naprej (sicer se vzrok za nevarnost ne bo odstranil).
- Logika za cevovodno zaklenitev ukaz zamenja z ukazom NOP
  - mehurček (bubble) je strojni ekvivalent operacije NOP
    - NOP je običajno addi x0, x0, 0 (pri RV32I 0x00000013)
  - stopnja EX “izvede” ukaz NOP
  - mehurček potuje od stopnje EX naprej

	Urina perioda												
Ukaz	1	2	3	4	5	6	7	8	9	10	11	12	13
addi x20, x0, 0	IF	ID	EX	MEM	WB								
sub x3, x4, x5		IF	ID	EX	MEM	WB							
add x1, x3, x6			IF	○ (ID)	○ (ID)	ID	EX	MEM	WB				
and x2, x3, x7				(IF)	(IF)	IF	ID	EX	MEM	WB			
xor x8, x3, x9							IF	ID	EX	MEM	WB		
or x10, x3, x12								IF	ID	EX	MEM	WB	

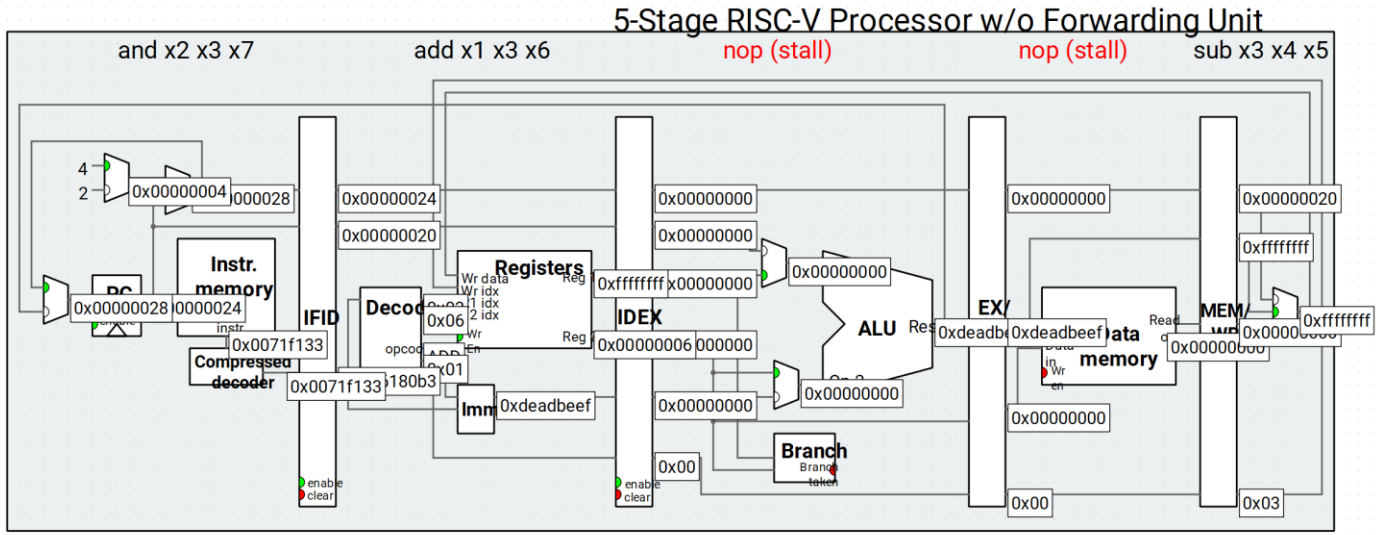
urina perioda	IF (IM)	ID (IR)	EX (IR1)	MEM (IR2)	WB (IR3)
1	addi				
2	sub	addi			
3	add	sub	addi		
4	and	add	sub	addi	
5	and	add	nop (stall)	sub	addi
6	and	add	nop (stall)	nop (stall)	sub
7	xor	and	add	nop (stall)	nop (stall)
78	or	xor	and	add	nop (stall)

6. perioda

add in and **čakata** v stopnjah ID in IF 2 dodatni periodi

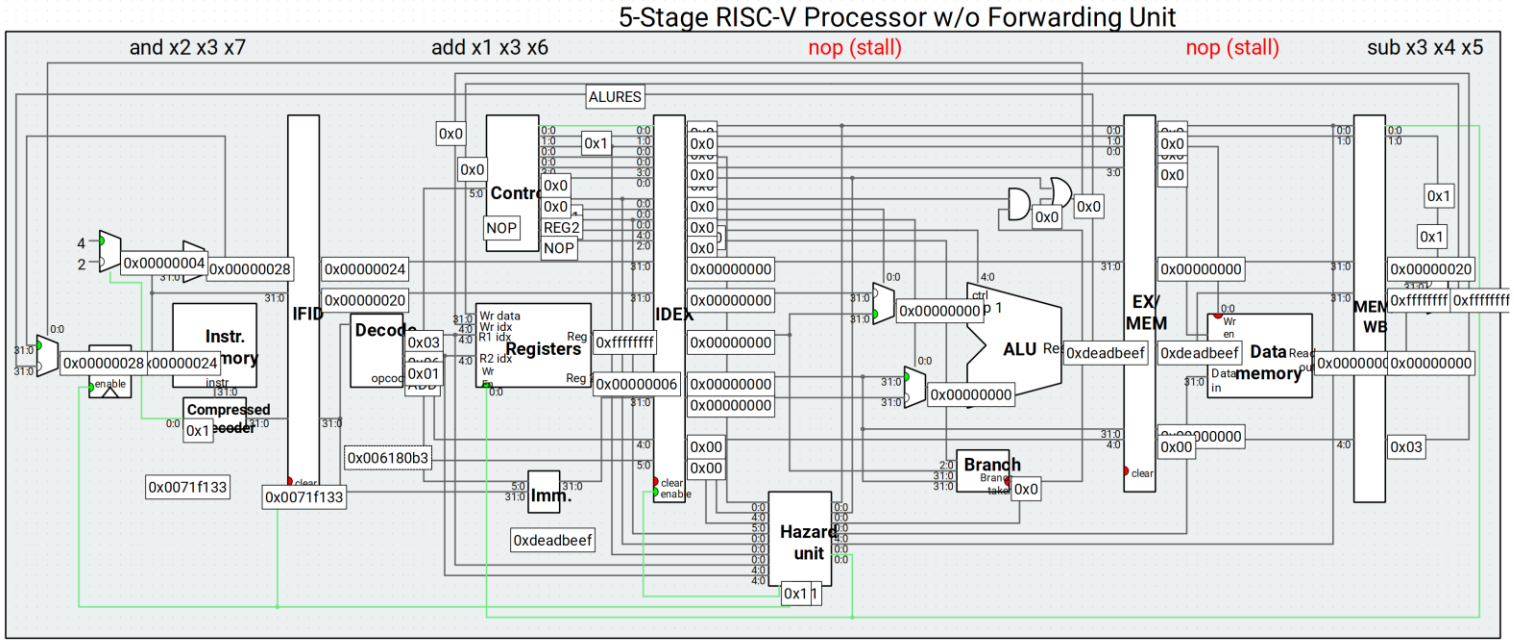
Simulator Ripes:

Select Processor:
5-Stage RISC-V z logiko za odpravljanje PN (hazard detection - HD), a brez premoščanja



Layout: Standard

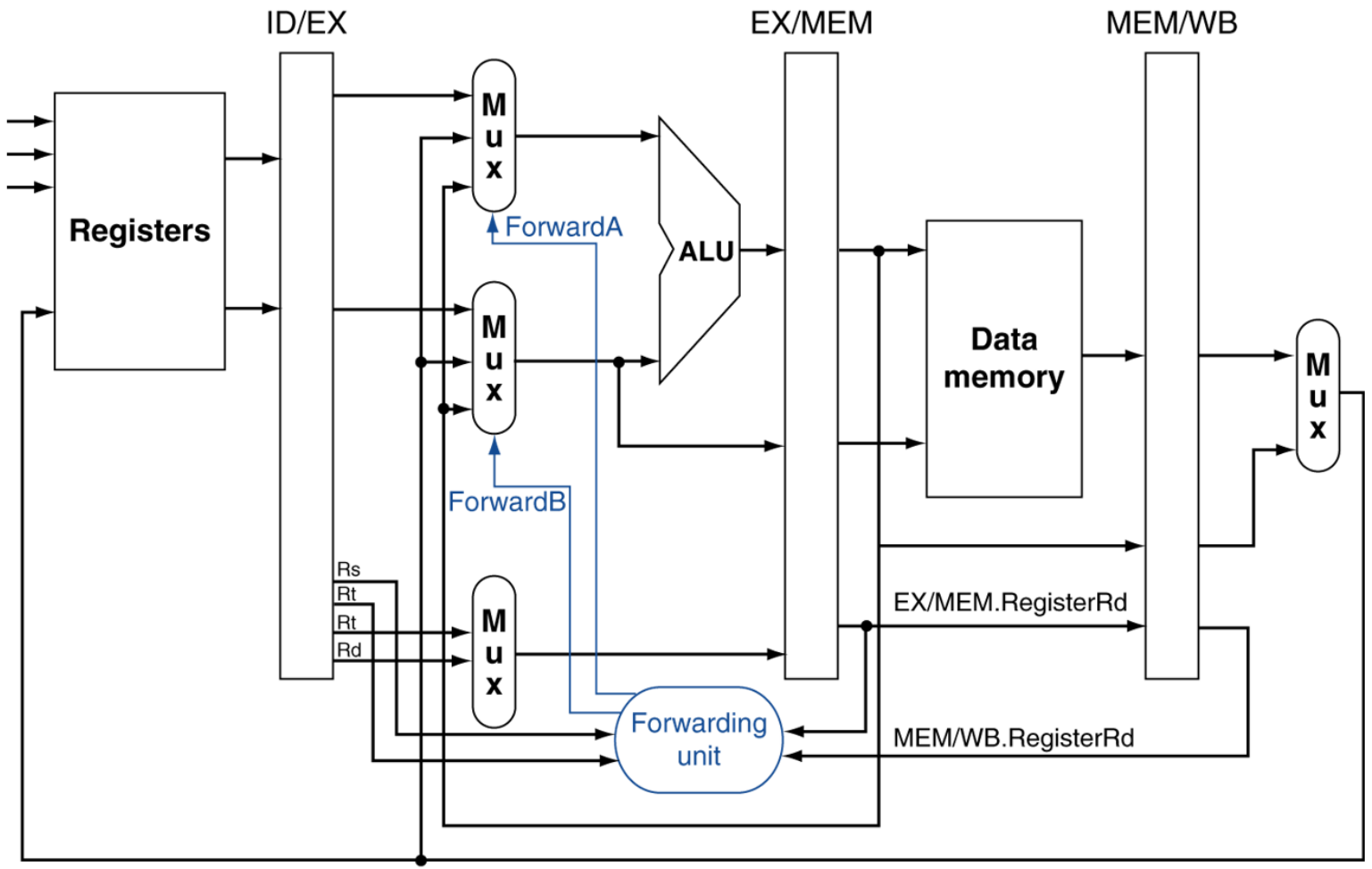
Layout: Extended



Rešitev 2:
Premoščanje (bypassing, data forwarding)

- Rezultat ukaza iz registra C (rezultat ALE) iz stopnje MEM prenesemo v EX in ustavljanje ni potrebno
- *Logika za premoščanje* (*forwarding unit* – recimo ji FW) mora omogočati tudi prenos iz stopenj MEM in WB (poleg stopnje ID) v stopnjo EX
- PN se ugotavljajo s primerjavo Rs1 in Rs2 v stopnji ID z Rd, ki ga uporabljajo ukazi v stopnjah EX in MEM (vendar le, če gre za ukaze, ki sploh pišejo v Rd!)
  - V ta namen se uporabi logika za ugotavljanje PN (hazard detection unit - HD)

Cevovodna PE s premoščanjem (forwarding unit)



Naslovni vhodi mux	Izvor	Operacija
ForwardA = 00	ID/EX	prvi ALE operand pride iz RF
ForwardA = 10	EX/MEM	prvi ALE operand pride iz rezultata ALE
ForwardA = 01	MEM/WB	prvi ALE operand pride iz DM ali iz od ALE prej
ForwardB = 00	ID/EX	prvi ALE operand pride iz RF
ForwardB = 10	EX/MEM	prvi ALE operand pride iz rezultata ALE
ForwardB = 01	MEM/WB	prvi ALE operand pride iz DM ali iz od ALE prej

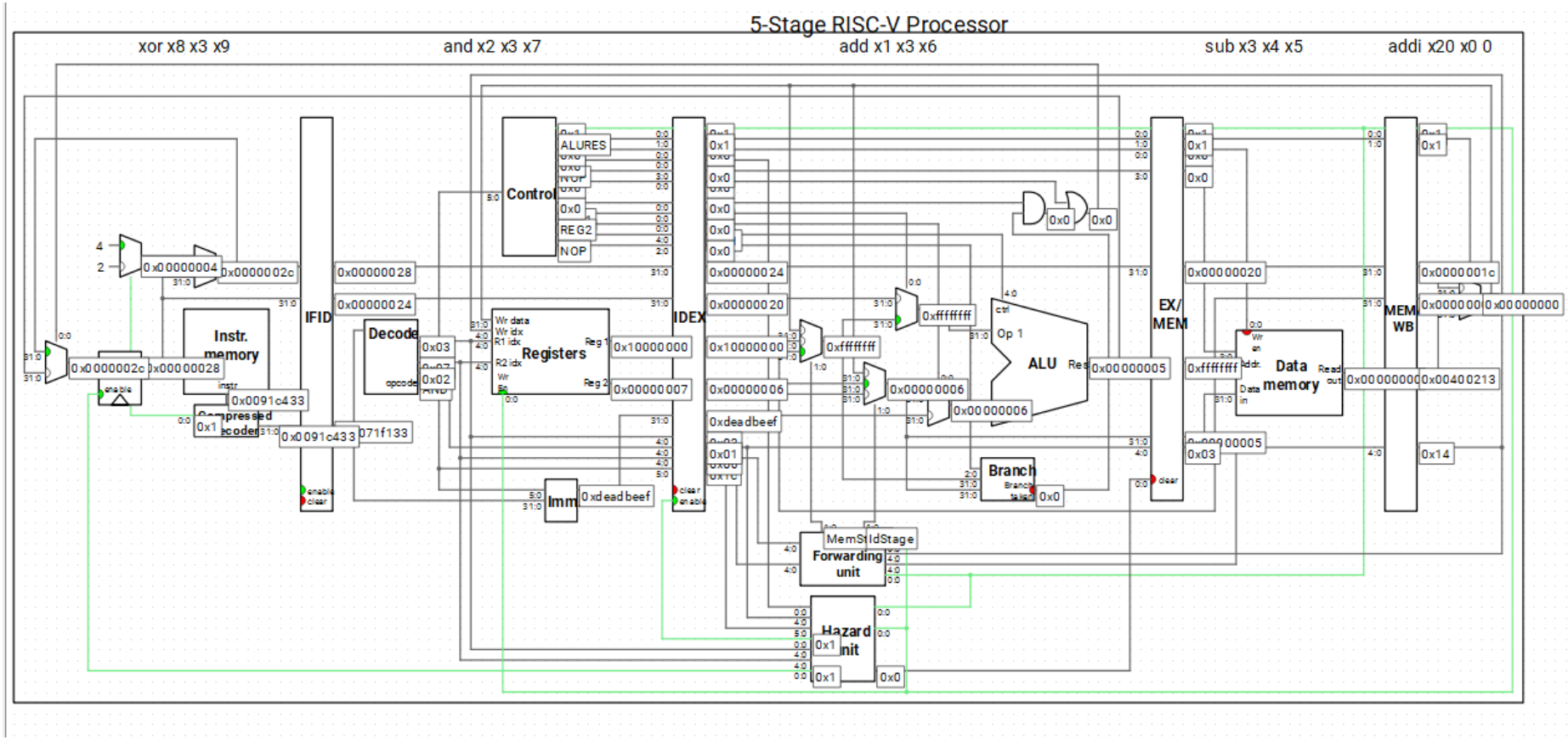
1. Nevarnost v stopnji EX:

if (EX/MEM.RegWrite and EX/MEM.rd ≠ 0 and EX/MEM.rd = ID/EX.rs1)
    ForwardA = 10
if (EX/MEM.RegWrite and EX/MEM.rd ≠ 0 and EX/MEM.rd = ID/EX.rs2)
    ForwardB = 10

2. Nevarnost v stopnji MEM:

if (MEM/WB.RegWrite and MEM/WB.rd ≠ 0 and MEM/WB.rd = ID/EX.rs1)
    ForwardA = 01
if (MEM/WB.RegWrite and MEM/WB.rd ≠ 0 and MEM/WB.rd = ID/EX.rs2)
    ForwardB = 01

Cevovodna PE z logiko za ugotavljanje podatkovnih nevarnosti (hazard detection) in premoščanjem (forwarding unit) v simulatorju Ripes



	Urina perioda										
Ukaz	1	2	3	4	5	6	7	8	9	10	11
`addi x20,x0,0`	IF	ID	EX	MEM	WB						
`sub **x3**,x4,x5`		IF	ID	**EX**	MEM	WB					
`add x1,**x3**,x6`			IF	ID	EX	MEM	WB				
`and x2,**x3**,x7`				IF	ID	EX	MEM	WB			
`xor x8,x3,x9`					IF	ID	EX	MEM	WB		
`or x10,x3,x12`						IF	ID	EX	MEM	WB	

Rezultat odštevanja iz ALE (za x3) se vpiše v register C (tj. C/MAR) in se nato iz stopnje MEM prenese nazaj (bypass, forwarding) v stopnjo EX za ukaz add, ki rezultat x3 rabi kot vhodni operand

- Poleg tega se isti rezultat eno periodo kasneje prenese v stopnjo WB – tega pa uporabi ukaz and
- Ukaz xor tudi potrebuje x3, a ga dobi že po ‘normalni’ poti iz RF (x3 je tedaj že vpisan v prvi polovici periode, v drugi polovici pa se prebere v register A)

Tako se bistveno zmanjša izguba zaradi PN, ni pa popolnoma odpravljena

- Včasih premoščanje ni možno, ker operanda ni v cevovodu
- Npr.

```
lw   x3, 56(x4)     x3 ←32 M[56 + x4]
sub  x1, x3, x6     x1 ← x3 – x6
add  x2, x3, x7     x2 ← x3 + x7
xor  x8, x3, x9     x8 ← x3 ∇ x9
```

- `lw` dobi operand šele v stopnji MEM
- Premoščanje v ID ni možno, ker operanda ni v CPE
- Čakanje je nujno (se pa da zmanjšati za eno periodo)
- Pri add pa čakanje ni potrebno (zaradi premoščanja iz WB)
- Ukazi load so edini, pri katerih je pri premoščanju potreben 1 mehurček
  - pri ostalih ni čakanja

Potek izvajanja ukazov pri premoščanju

	Urina perioda								
Ukaz	1	2	3	4	5	6	7	8	9
`lw x3,56(x4)`	IF	ID	EX	MEM	WB				
`sub x1,x3,x6`		IF	○ (ID)	ID	EX	MEM	WB		
`add x2,x3,x7`			(IF)	IF	ID	EX	MEM	WB	
`xor x8,x3,x9`					IF	ID	EX	MEM	WB

Rešitev 3: Cevovodno razvrščanje (pipeline scheduling)

- Prevajalnik lahko s spreminjanjem vrstnega reda ukazov pogosto odpravi nevarnost
- Npr.:

a = b + c (naslovi naj bodo v registrih Ra, ..., Rf)
d = e – f

lw	x2, 0(Rb)	x2 ← b	
lw	x3, 0(Rc)	x3 ← c	
lw	x4, 0(Re)	x4 ← e	ukaz prestavljen naprej
add	x5, x2, x3	x5 ← b + c	
lw	x6, 0(Rf)	x6 ← f	ukaz prestavljen naprej
sw	x5, 0(Ra)	a ← b + c	
sub	x7, x4, x6	x7 ← e – f	
sw	x7, 0(Rd)	d ← e – f	

Večina prevajalnikov danes uporablja cevovodno razvrščanje

- čakanja pa se vedno ne da odpraviti
- delež ukazov load, kjer se kljub temu pojavi PN:
  - 4 – 40% (odvisno od programa), povprečno pa nekje 19%
  - ukazov load je v povp. 24%
  - 0,19*0,24=0,0456
  - CPI = (1 – 0,0456)*1 + 0,0456*2 = 1,0456
  - zaradi PN pri load je cevovod za 4,6% počasnejši

Povzetek odpravljanja podatkovnih nevarnosti

Če povzamemo, imamo glede PN naslednje rešitve:

0. Vstavljanje ukazov NOP s strani programerja oz. prevajalnika
1. Zaklepanje cevovoda
   - procesor ima enoto HD in sam vstavlja mehurčke (NOP)
2. Premoščanje
   - procesor ima enoti HD in FW in po potrebi pelje rezultat neposredno iz stopenj MEM ali WB v EX (tako odpravi vse zaklenitve, razen 1 u.p. pri ukazih load)
3. Cevovodno razvrščanje
   - s strani prevajalnika

Kontrolne nevarnosti

Kontrolne nevarnosti (KN) se pojavijo pri ukazih, ki spremenijo PC drugače kot PC ← PC + 4

- to so kontrolni ukazi oz. skoki:
  - brezpogojni skoki, pogojni skoki, klici (procedur), vrnitve
  - JAL, JALR, BEQ, BNE, BLT, BGE, BLTU, BGEU
- Skočni naslov se prenese v PC običajno v stopnji EX
- KN: Kadar se v stopnji EX spremeni PC, je vsebina stopenj IF in ID neveljavna!
  - v njiju sta ukaza, ki sledita skočnemu ukazu
  - ne smeta se izvršiti
  - najenostavnejša rešitev je vstavljanje mehurčka v IF in ID

Odpravljanje kontrolnih nevarnosti

Prvi korak je zmanjšanje skočne zakasnitve:

1. *Preverjanje pogoja za skok* naj se izvaja čim bližje prvi stopnji cevovoda
   - V primeru preprostih komparatorjev (za beq in bne) je to enostavneje
2. *Izračun skočnega naslova* (branch target address) naj se izvaja čim bližje prvi stopnji cevovoda

- RV: preverjanje skočnega pogoja za BEQ izvaja ALE z odštevanjem
  - računanje skočnega naslova je možno v že stopnji ID
    - BEQ in BNE uporabljata PC-relativno naslavljanje, vrednost PC pa je že v reg. PC1
  - preverjanje pogoja šele v stopnji EX
    - glej komparator Branch v simulatorju Ripes (ta vpliva na mux pred registrom PC, ki izbere za vpis v PC rezultat iz ALE)

Vstavljanje mehurčkov

- Najenostavnejša rešitev je vstavljanje mehurčka v IF in ID
- V tem primeru se v primeru skoka razveljavi ukaza v IF in ID (z mehurčki), če pa se skok ne izvede, pa deluje normalno naprej brez ustavljanja
  - **skočna zakasnitev** (branch delay), čakanje 2 periodi
- Vsak skočni ukaz povzroči 2 čakalni periodi
- Tem mehurčkom se tudi reče *flush*, ker se morajo 'izplakniti'
  - mehurčka potujeta proti izhodu cevovoda

Št. ukaza	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Skočni ukaz	IF	ID	EX	ME	WB			
Skočni ukaz + 1		IF	ID	O (EX)	O (MEM)			
Skočni ukaz + 2			IF	O (ID)	O (EX)			
. . .								
Ukaz na skočnem naslovu				IF	ID	EX	MEM	WB

Primer

Ukaz	u.p.:	1	2	3	4	5	6	7	8	9
	`bne x1, x2, L1`	IF	ID	EX	ME	WB				
	`add x3, x4, x5`		IF	ID	O (EX)	O (MEM)	O (WB)			
	`sub x6, x7, x8`			IF	O (ID)	O (EX)	O (MEM)	O (WB)		
`. . .`										
`L1:`	`xori x3, x3, 1`				IF	ID	EX	MEM	WB	
	`and x6, x7, x8`					IF	ID	EX	MEM	
	`lb  x5, ABC(x0)`						IF	ID	EX	

u.p.	IF	ID	EX	MEM	WB
1	bne				
2	add	bne			
3	sub	add	bne		
4	xori	nop (flush)	nop (flush)	bne	
5	and	xori	nop (flush)	nop (flush)	bne
6	lb	and	xori	nop (flush)	nop (flush)
7		lb	and	xori	nop (flush)
8			lb	and	xori

- če pogoj za skok ni izpolnjen, ni čakanja
- cevovod predpostavi, da skoka ne bo

- V povprečju:
  - pogojnih skokov 12,5%
    - pogoj izpolnjen pri ~ 2/3 primerov
  - brezpogojnih skokov 2,5%
- Sprememba PC v stopnji EX:
  - 0,125*2/3 + 0,025 = 0,109
  - pri 10,9% ukazov je CPI = 3, sicer 1 (če FW in brez upoštevanja 1u.p. pri load)
  - CPI = 3*(0,109) + 1*(1 - 0,109) = 1,218
    - tj. več kot 20% izguba (daljši čas računanja)
  - Pri rač. z dolgimi cevovodi in pri CISC so izgube še večje

Primer:

```
        .data
A:      .byte 1, 2, 3, 4, 5
B:      .byte 0, 0, 0, 0, 0

        .text
        addi x1, x0, 5
        addi x3, x0, 0
        addi x4, x0, A
ZANKA:  lb x2, 0(x4)
        add x3, x3, x2
        sb x3, 5(x4)
        addi x4, x4, 1
        addi x1, x1, -1
        bne x1, x0, ZANKA
        xori x6, x3, -1
        ori x7, x3, 7
```

a. Koliko mehurčkov (stall oz. RAW) in kje mora procesor vstaviti zaradi podatkovnih nevarnosti, če ne uporablja premoščanja?
b. Kaj pa s premoščanjem?
c. Koliko mehurčkov (flush) in kje mora procesor vstaviti zaradi kontrolnih nevarnosti?
d. Koliko urinih period bi trajal program, če ne bi bilo nobenih cevovodnih nevarnosti?
e. Koliko urinih period traja program, če se uporablja premoščanje, in koliko brez njega?
f. Kako bi s spremembo vrstnega reda ukazov odpravili čimveč čakalnih period, če procesor ne uporablja premoščanja?

Rešitev:

a. PN: Št. čakalnih urinih period = 2(lb) + 5*(2(add) + 2(sb) + 2(bne)) = 32 u.p.
b. 5*1(lb) = 5 u.p.
c. 4*2(bne) = 8 u.p. Pri 4 obhodih zanke je pogoj izpolnjen in je treba ukaza xori in or razveljaviti.
d. 3 + 5*6 + 2 + 4 (zaradi latence) = 39 u.p. (ukazov je 35). Ker ukazi trajajo N period (=št. stopenj cevovoda), je treba prišteti še začetno latenco (N-1), saj prvih N-1 period še ni končan noben ukaz, potem pa vsako u.p. po eden
e. Št. u.p. s premoščanjem (FW) = 39 + 5 + 8 = 52, CPI = 52/35 = 1,49 (ukazov je 35).
   Št. u.p. brez premoščanja: 39 + 32 + 8 = 79, CPI = 79/35 = 2,26
f. Če smemo spremeniti tudi odmike: 39 + 5*2 + 8 = 57, CPI = 57/35 = 1,63
   Samo sprememba vrstnega reda: 39 + 5*3 + 8 = 57, CPI = 62/35 = 1,77

Če smemo spremeniti tudi odmike:

```
        .text
        addi x4, x0, A  (za 2 nazaj)
        addi x1, x0, 5
        addi x3, x0, 0
ZANKA:  lb x2, 0(x4)
        addi x4, x4, 1  (za 2 nazaj)
        addi x1, x1, -1 (za 2 nazaj)
        add x3, x3, x2
        sb x3, 4(x4)       2 RAW
        bne x1, x0, ZANKA
        xori x6, x3, -1
        ori x7, x3, 7
```

Samo sprememba vrstnega reda:

```
        .text
        addi x4, x0, A  (za 2 nazaj)
        addi x1, x0, 5
        addi x3, x0, 0
ZANKA:  lb x2, 0(x4)
        addi x1, x1, -1 (za 3 nazaj)
        add x3, x3, x2   1 RAW
        sb x3, 5(x4)     2 RAW
        addi x4, x4, 1
        bne x1, x0, ZANKA
        xori x6, x3, -1
        ori x7, x3, 7
```

Drug način je napoved (predikcija) izpolnitve skočnega pogoja in napoved skočnega naslova (če se skok izvede)

- vezje, ki napoveduje (ne)izpolnjenost pogoja, se imenuje *branch predictor*

2 skupini:

- s statično predikcijo
- z dinamično predikcijo

Statična predikcija

Prevajalnik skuša napovedati bolj verjeten rezultat preverjanja skočnega pogoja

- med izvrševanjem programa se zato ne spreminja → statična predikcija
- tudi že omenjeni primer (ki predpostavi neizpolnjenost pogoja) je preprost primer statične predikcije

Statična predikcija

- prednost: večino dela opravi prevajalnik
- hiba: večino dela opravi prevajalnik
  - zahteva drugačno programiranje → problemi s kompatibilnostjo za nazaj

Statična predikcija z zakasnjenimi skoki

Ta metoda je bila priljubljena pri starejših RISC

- en ukaz (ali dva), ki je v programu pred skokom, se prestavi v t.i. *skočno režo* (branch slot)
- Pri uporabi zakasnjenih skokov se (ne glede na izpolnjenost pogoja) izvršijo vsi prevzeti ukazi
- ukaz (oz. ukaza) v skočnih režah se ne nadomesti z mehurčki
- ker se vedno izvrši (izvršita), izgleda kakor da se skok izvede kasneje

Pri pogojnih skokih je težje:

- ukaza, ki vpliva na pogoj, ne smemo dati v režo
- namesto njega damo ukaz NOP (to dela prevajalnik)

Dokler je bil cevovod, razmeroma preprost, je to delovalo

- Težave pa so se pojavile, ko so začeli spreminjati vrstni red ukazov, superskalarne procesorje, daljše cevovode, itd.
- Skočne kazni so postale večje in tu ena perioda ne pomeni dosti
- Program postane težaven za razumevanje
- Pri sedanji tehnologiji so zakasnjeni skoki postali nepotrebna komplikacija z malo koristi

Dinamična predikcija skokov

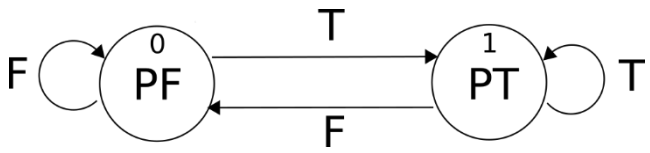
- Danes se bolj kot statična predikcija uporabljajo strojni načini za dinamično predikcijo skokov
- Dinamična predikcija se prilagaja dogajanju v programu

Več vrst dinamične predikcije:

1. **1-bitna prediktorska tabela**
   - *prediktorska tabela* (branch prediction table, branch history table)
   - to je majhen pomnilnik, iz katerega se v stopnji IF bere vrednost (1 bit pri 1-bitni tabeli)
   - naslov določajo spodnji biti naslova ukaza
   - če je pogoj izpolnjen, se vpiše 1, sicer 0
     - v stopnji EX, ko je to znano

- služi kot napoved izpolnjenosti pogoja
- če je napovedan izpolnjen pogoj, potrebujemo še skočni naslov
  - ta je običajno dostopen šele v stopnji ID
  - zato privarčujemo le en urino periodo
  - če je bila napoved napačna (izvemo v EX), je treba v IF in ID vstaviti mehurčke
- metoda ni posebno zanesljiva
  - npr. pri vgnezdenih zankah bo napoved tipično napačna dvakrat

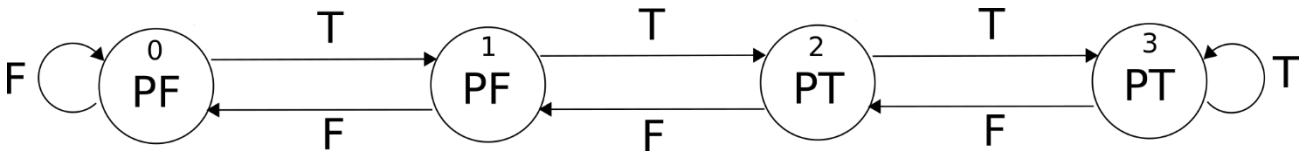
Stanje	Skočni pogoj ni izpolnjen (F)	Skočni pogoj izpolnjen (T)
0 (skočni pogoj ne bo izpolnjen – PF (predict false))	napoved pravilna -> stanje ostane 0 (PF)	napoved napačna -> stanje 1 (PT)
1 (skočni pogoj bo izpolnjen – PT (predict true))	napoved napačna -> stanje 0 (PF)	napoved pravilna -> stanje ostane 1 (PT)



2. 2-bitna prediktorska tabela

- 4 vrednosti (0..3)
- Povečanje ali zmanjšanje za 1
- 0 in 1: neizpolnjen pogoj, 2 in 3: izpolnjen
- Pri vgnezdenih zankah le 1 napačna napoved

Stanje	Skočni pogoj ni izpolnjen (F)	Skočni pogoj izpolnjen (T)
0 (PF00)	napoved pravilna -> stanje ostane 0 (PF)	napoved napačna -> stanje 1 (PT)
1 (PF01)	napoved pravilna -> stanje 0 (PF)	napoved napačna -> stanje 2 (PT)
2 (PT10)	napoved napačna -> stanje 1 (PF)	napoved pravilna -> stanje 3 (PT)
3 (PT11)	napoved napačna -> stanje 2 (PT)	napoved pravilna -> stanje ostane 3 (PT)



Možna tudi n-bitna prediktorska tabela, n > 2

- Vendar ni dosti boljša kot 2-bitna
- Tabele so velikosti največ 4096 (12 bitov naslova)

Primer:

```
                .data # 0x400
tab:            .byte  -7, 12, -3, 15, 8

                .text # 0
0x0             addi  x3, x0, 0
0x4             addi  x5, x0, 5
0x8   loop:     lb    x1, tab(x3)
0xC             slti  x2, x1, 0      # x2 <- (x1 < 0)?
0x10            beq   x2, x0, skok
0x14            add   x4, x4, x1
0x18  skok:     addi  x3, x3, 1
0x1C            bne   x3, x5, loop
```

Velikost predikcijske tabele je 4 polja, dostop do tabele pa je narejen z naslovnimi biti A3-A2. Na začetku so v tabeli ničle, kar pomeni, da se predvideva neizpolnjen pogoj. Določite, v katere vrstice tabele se preslika posamezen skočni ukaz in za vsak obhod zanke zapišite stanje predikcijskih bitov po izvršenem skočnem ukazu ter ali je bila posamezna napoved pravilna ali napačna. Izračunajte tudi odstotek pravilnih napovedi.

Rešitev:

`beq x2,x0,skok` A_{31-0} = 0x00000010 = 0b0..010000, A_{3-2} = 00

`bne x3,x5,loop` A_{31-0} = 0x0000001C = 0b0..011100, A_{3-2} = 11

Enobitna predikcijska (napovedna) tabela:

A3-A2	p	Ukaz
00	0 (PF)	beq
01	0 (PF)	
10	0 (PF)	
11	0 (PF)	bne

Obhod	x1	x2	beq	p(00)	x3	bne	p(11)
1	-7	1	ni skoka (F)	0 (PF) -> 0 (PF)	1	skok (T)	0 -> 1
2	12	0	skok (T)	0 (PF) -> 1 (PT)	2	skok	1 -> 1
3	-3	1	ni skoka (F)	1 (PT) -> 0 (PF)	3	skok	1 -> 1
4	15	0	skok (T)	0 (PF) -> 1 (PT)	4	skok	1 -> 1
5	8	0	skok (T)	1 (PT) -> 1 (PF)	5	ni skoka	1 -> 0

Pravilnih napovedi: 5/10 = 50%

2-bitna predikcijska tabela:

(začnemo npr. z enicami (stanje 1 oz. PF01))

A3-A2	p	Ukaz
00	01 (PF)	beq
01	01 (PF)	
10	01 (PF)	
11	01 (PF)	bne

Obhod	x1	x2	beq	p(00)	x3	bne	p(11)
1	-7	1	ni skoka (F)	PF01 -> PF00	1	skok (T)	PF01 -> PT10
2	12	0	skok (T)	PF00 -> PF01	2	skok	PT10 -> PT11
3	-3	1	ni skoka (F)	PF01 -> PF00	3	skok	PT11
4	15	0	skok (T)	PF00 -> PF01	4	skok	PT11
5	8	0	skok (T)	PF01 -> PT10	5	ni skoka	PT11 -> PT10

Pravilnih napovedi: 5/10 = 50%

Bolje pa se 2-bitna predikcijska tabela obnese pri vgnezdenih zankah, še posebej pri večjem številu obhodov

3. Korelacijski prediktor

Correlating branch prediction table

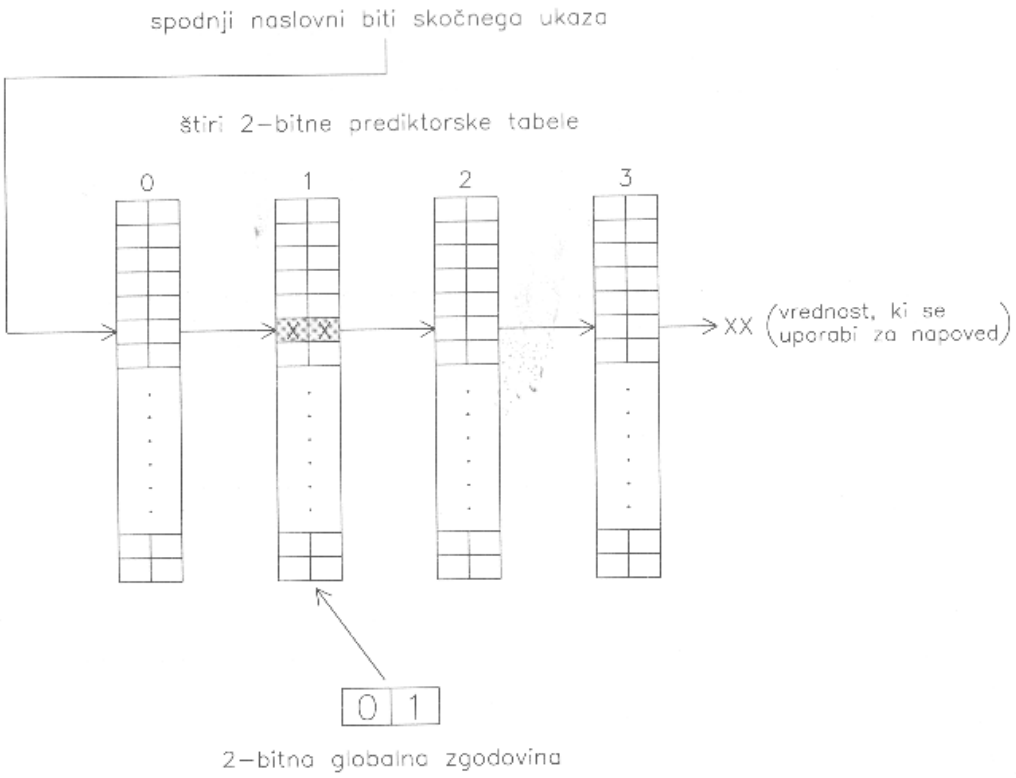
Primer:

```
          if ( a == 2)
             a = 0;
          if ( b == 3)
             b = 0;
          if ( a != b) {


          addi     x4, x0, 2
          bne      x1, x4, L1        # skok s1
          add      x1, x0, x0        # a ← 0
L1:       addi     x4, x0, 3
          bne      x2, x4, L2        # skok s2
          add      x2, x0, x0        # b ← 0
L2:       beq      x1, x2, L3        # skok s3
```

- Skok s3 odvisen od s1 in s2
- Običajna prediktorska tabela tega ne more zajeti

- Korelacijski prediktor (m,n) uporablja obnašanje prejšnjih m skokov (t.i. *globalna* zgodovina), da izbere eno od 2^m n-bitnih predliktorskih tabel
  - Navadna 2-bitna tabela bi bila k.p. (0,2)
    - Imenuje se tudi *lokalni* prediktor
- Primer: korelacijski prediktor (2,2)
  - Za globalno zgodovino lahko uporablja 2-bitni pomikalni register (pomika v levo)



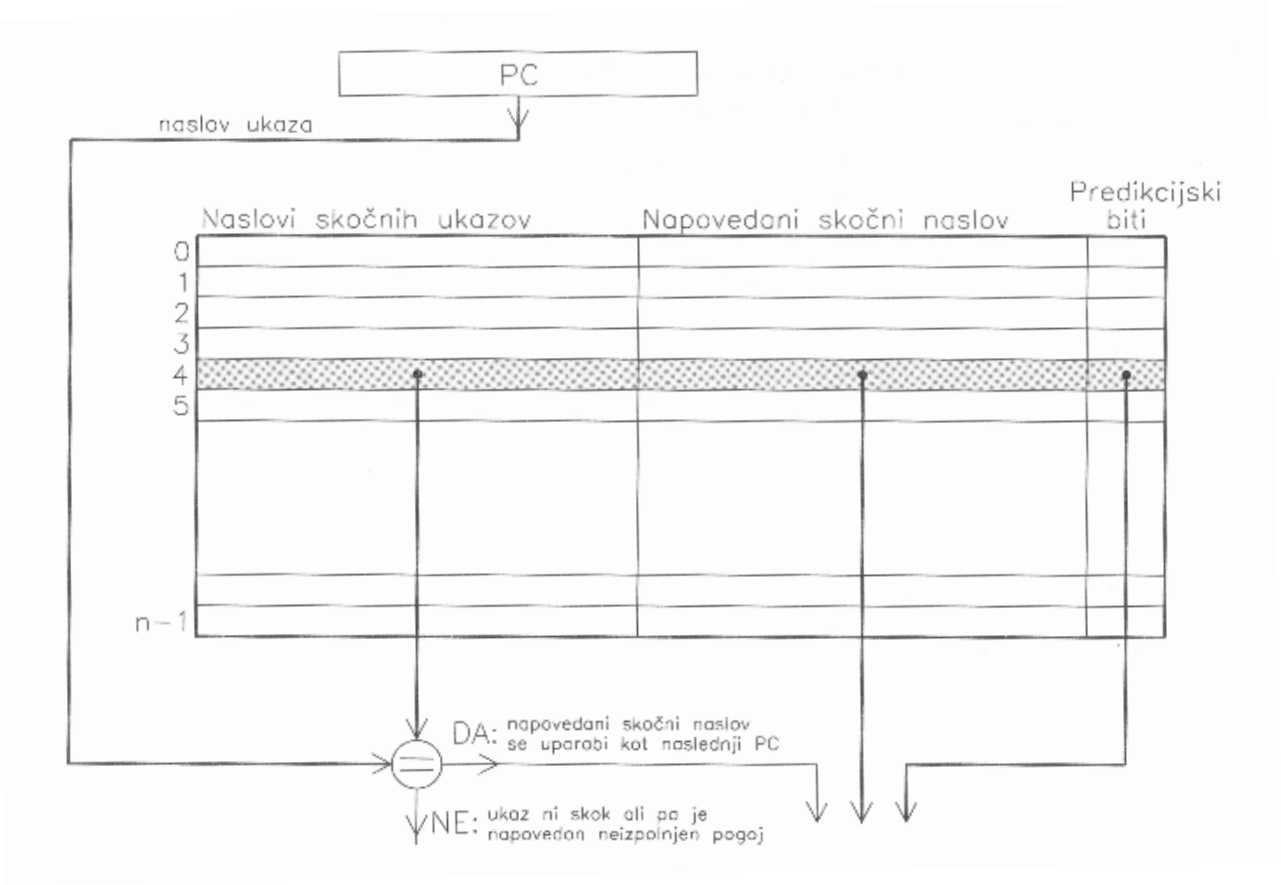
4. Turnirski prediktor

- tournament branch predictor
- Upošteva dejstvo, da globalni prediktor ni vedno boljši od lokalnega
- Paralelno delojoča lokalni in globalni prediktor tekmujeta
- Selektor določa, kateri bo uporabljen (glede na prejšnji uspeh)

5. Skočni predpomnilnik

- branch target buffer
- Tudi pri pravilni napovedi pogoja se vedno izgubi ena perioda
  - V stopnji IF ne poznamo skočnega naslova (ne poznamo niti ukaza, ker še ni dekodiran)
- Skočni PP vsebuje skočne naslove zadnjih skokov, pri katerih je bil pogoj izpolnjen
  - Naslovi se vanj shranijo v stopnji EX
- V IF se poleg ukaza bere tudi skočni PP
  - V primeru zadetka (in potencialnih prediktorskih bitov) se skočni naslov takoj vpiše v PC
  - Pri pravilni napovedi ni treba čakati 1 periodo
  - Pri napačni napovedi (ali skočnem naslovu) je treba vstavljati mehurčke

Skočni predpomnilnik:



Skočni PP je bolj zapleten od prediktorjev na osnovi tabel

- Npr. PP 1024x32 potrebuje 1024 32-bitnih komparatorjev (primerjalnikov)

V skočni PP se shrani skočni naslov le, kadar je pogoj izpolnjen

- Sicer je naslov poznan (naslednji po vrsti)

6. Vrnitveni prediktor

- return address predictor
- Težavna vrsta posrednih skokov
- Ista procedura se lahko kliče z zelo različnih mest v programu (npr. funkcija printf v C-ju)
  - Težko napovedati
- Običajno majhen sklad (npr. 16 naslovov)

7. Enota za prevzem ukazov

- integrated instruction fetch unit
- Današnji računalniki lahko istočasno prevzemajo in izvršujejo več ukazov (superskalarnost)
  - Prevzem ukazov bolj zapleten
- Enota deluje samostojno in dostavlja ukaze ostalim stopnjam
  - Dela tudi predikcijo, dostopa do PP, pri zgrešitvah menjava bloke v PP, ...

Vejitve brez skokov

Pri RISC procesorjih skoki zaradi cevovodnih kontrolnih nevarnosti niso zaželeni, zato se jim lahko pri implementaciji vejitev, kadar je možno, odrečemo. Podobne operacije izvajajo vektorski procesorji in GPE – reče se jim tudi *predikatne* vejitve.

- V ta namen so uporabni set-ukazi

1. Bolj enostaven primer (npr. štetje števil, ki ustrezajo nekemu pogoju):

```
if ( x1 < 5)
   x2 = x2 + 1;

slti x3, x1, 5     # x3 = (x1 < 5)? 1 : 0
add x2, x2, x3
```

2. Bolj splošen primer:

```
if ( x1 < 5)       // x2 = (x1 < 5)? x7: x8;
   x2 = x7;
else
   x2 = x8;
```

- Mux 2/1 izvaja logično funkcijo y = !a&d0 v a&d1
- Potrebujemo pa 32 takih funkcij (za vse bite):

  32x mux 2/1: x2 = (x1<5)&x7 V !(x1<5)&x8
- Vendar mora biti pogoj (x1<5) 1 ali 0 za vse bite! (..1111 oz. ..0000)
- Pogoj P je treba spremeniti tako, da bodo v primeru enice šli na 1 tudi ostali biti

  0...0000 -> 0...0000

  0...0001 -> 1...1111
- To lahko naredimo z operacijo –P, če je P pogoj: not(A)+1

  not(0...0000)=1...1111, 1...111+1=0000

  not(0...0001)=1...1110, +1=1...1111
- Koda

```
slti x3, x1, 5                      # x3 = (x1 < 5)? 1: 0

# Razširitev enic/ničel: 0001 -> 1111, 0000 -> 0000
# Način a:
xori x3, x3, -1                     # 0001 -> 1110, 0000 -> 1111
addi x3, x3, 1                      # 1110 -> 1111, 1111 -> 0000        x3 = maska

xori x4, x3, -1                     # x4 = !maska

# mux:
and x5, x7, x3                      # maska & x7
and x6, x8, x4                      # !maska & x8
or x2, x5, x6
```

Operacije, ki trajajo več urinih period

Ko ukaz s tako operacijo pride v stopnjo EX, se cevovod ustavi in čaka, da se operacija izvrši

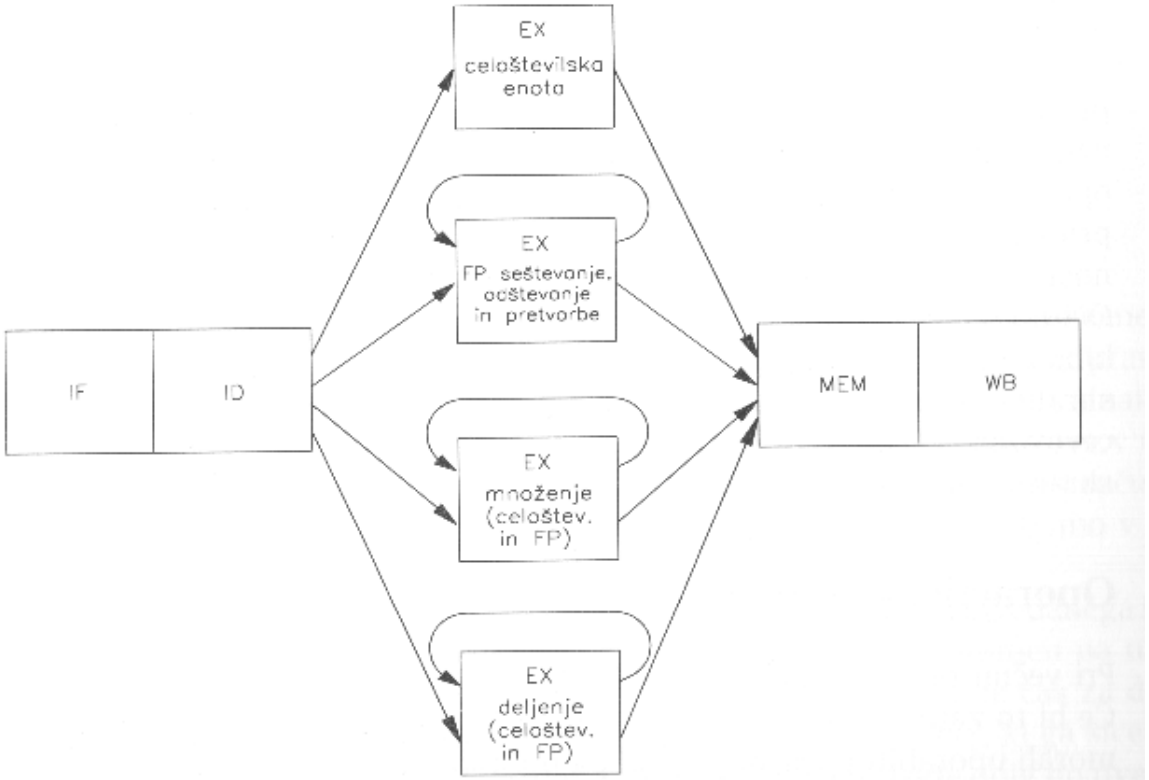
- cevovod bi pri mnogih programih postal prepočasen

Zato so uvedli funkcijske enote:

- **Celoštevilska enota** (integer unit)
  - celošt. ALE ukazi, skoki, load, store
  - pri preprostih RISC je le ta
- **Enota za operacije v plavajoči vejici** (floating-point unit)
  - seštevanje, odštevanje, pretvorbe
- **Enota za množenje**
  - celoštevilsko in v FP
- **Enota za deljenje**
  - celoštevilsko in v FP

Predpostavimo, da FE niso cevovodne

- naslednji ukaz lahko uporabi neko enoto šele, ko jo prejšnji zapusti (strukturne nevarnosti)
- samo celošt. enota rabi 1 periodo, ostale več

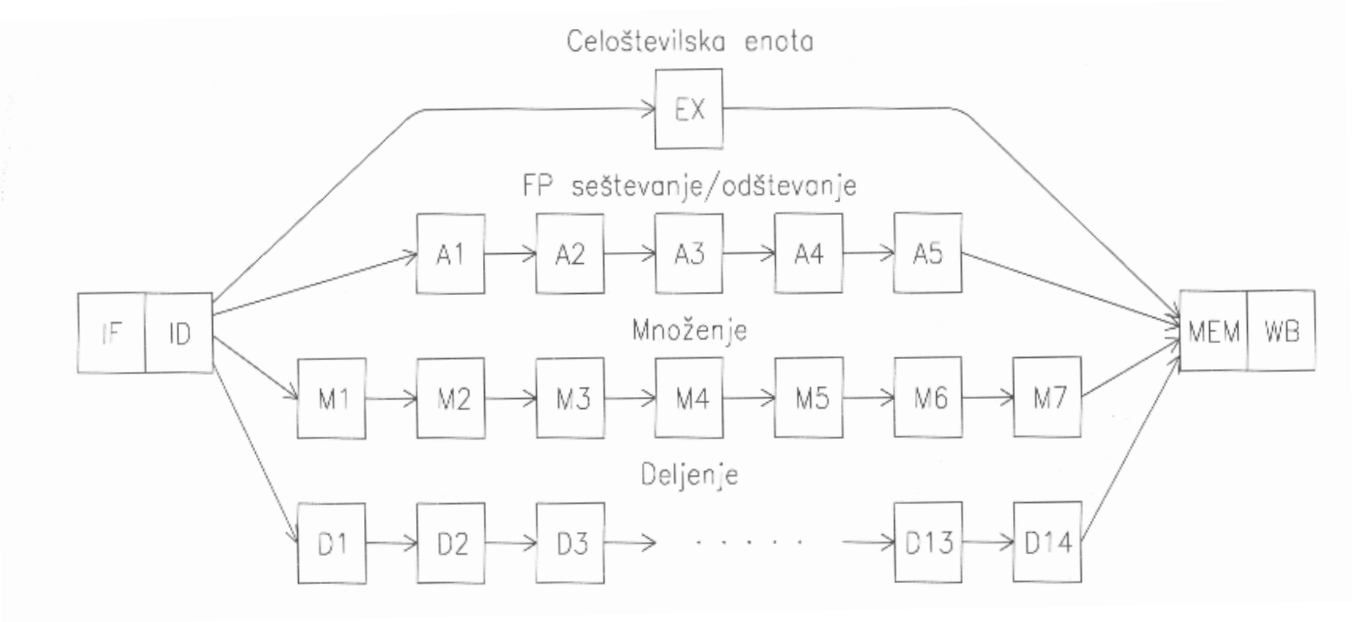


Če so FE cevovodne (danes običajno), lahko odpravimo strukturne nevarnosti v ID in EX

- lahko pa se SN pojavijo v MEM in WB

Dodatni problemi:

- V MEM in WB pride hkrati lahko več rezultatov
  - reg. blok mora omogočati več pisanj vanj hkrati
- poveča se tudi verjetnost podatkovnih nevarnosti
  - v MEM in WB prihajajo ukazi v spremenjenem vrstnem redu
  - pojavijo se PN tipov WAW in WAR



3 vrste PN:

- **RAW** (read after write): ukaz *j* bere operand, preden ga ukaz *i* shrani
- **WAR** (write after read): ukaz j piše v reg., še preden ga *i* prebere
- **WAW** (write after write): ukaz j piše v reg., preden vanj piše *i*
- RAR ne more povzročiti PN

Pri preprostih RISC je edina možnost RAW

- s premoščanjem jo običajno odpravimo (razen pri load)

Primer: zaporedje ukazov v plavajoči vejici

- enota (FPU) ima kar svojo množico registrov
  - poenostavi ugotavljanje nevarnosti
  - to rešitev uporablja večina CPE

	Urina perioda																
Ukaz	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
`FLD F4, 0(x2)`	IF	ID	EX	ME	WB													
`FMUL F0, F4, F6`		IF	ID	○	M1	M2	M3	M4	M5	M6	M7	ME	WB					
`FADD F2, F0, F8`			IF	○	ID	○	○	○	○	○	○	A1	A2	A3	A4	A5	ME	WB
`FST 0(x2), F2`					IF	○	○	○	○	○	○	ID	EX	○	○	○	○	ME

Odpravljanje podatkovnih nevarnosti z dinamičnim razvrščanjem

Dinamično razvrščanje:

- strojna sprememba vrstnega reda izvrševanja ukazov (da se zmanjša št. čakalnih period)

Primer PN:

- čakanje na "počasen" ukaz, ki se izvršuje v neki FE
- npr.:

```
fdiv   f0,f5,f6          ; f0 ← f5/f6
fadd   f4,f0,f2          ; f4 ← f0 + f2
fsub   f8,f2,f1          ; f8 ← f2 - f1
```

  - ukaz `fsub` mora čakati (cevovod se ustavi zaradi odvisnosti med `fdiv` in `fadd`)
  - ker pa je `fsub` neodvisen od prejšnjih ukazov, ga lahko pomaknemo gor (da se izogne čakanju)

ID moramo razdeliti na 2 stopnji:

1. Izstavljanje (issue)
   - dekodiranje
   - ugotavljanje SN
     - pri SN izvrševanje ukaza ni možno (ne glede na PN)
2. Branje operandov
   - ugotavljanje PN
     - v primeru nevarnosti se čaka
     - v tej stopnji lahko pride do spremembe vrstnega reda ukazov

Spremenjen vrstni red izvrševanja lahko pripelje do PN tipa WAR in WAW

Tomasulov algoritem (1967)

- uvede *rezervacijske postaje* za dinamično razvrščanje ukazov

Podatkovne odvisnosti lahko delimo na

- prave podatkovne odvisnosti
  - ukaz potrebuje kot vhodni operand rezultat enega od prejšnjih ukazov
- imenske odvisnosti

```
fdiv    f0,f5,f6        # f0 ← f5/f6
fadd    f4,f0,f2        # f4 ← f0+f2
fsw     f4,0(x1)        # M[x1] ← f4
fsub    f2,f3,f7        # f2 ← f3-f7
fmul    f4,f3,f2        # f4 ← f3*f2
```

- Imenske odvisnosti: WAR in WAW
  - med `fadd` in `fsub` zaradi R2
    - nevarnost WAR (*antiodvisnost*)
  - med `fadd` in `fmul` zaradi F4
    - nevarnost WAW (*izhodna odvisnost*)
- Prave podatkovne odvisnosti: RAW
  - med `fdiv` in `fadd`
  - med `fadd` in `fsw`
  - med `fsub` in `fmul`

Imenske odvisnosti lahko vedno odpravimo s preimenovanjem registrov (če imamo na voljo dodatne registre)

```
fdiv   f0,f5,f6      f0 ← f5/f6
fadd   ft2,f0,f2     ft2 ← f0+f2
fsw    ft2,0(x1)     M[x1] ← ft2
fsub   ft1,f3,f7     ft1 ← f3-f7
fmul   f4,f3,ft1     f4 ← f3*ft1
```

Tomasulov algoritem pa odpravi nevarnosti, ki izvirajo iz imenskih odvisnosti (WAR in WAW), brez preimenovanja registrov

Špekulativno izvajanje ukazov

Pri dinamičnem razvrščanju ukazov se problemi zaradi KN zelo povečajo

- ker se v vsaki periodi izvršuje več ukazov, je v primeru napačne predikcije težko ugotoviti, kateri se morajo razveljaviti
- cevovod se mora ustavljati

Špekulativno izvajanje ukazov (speculative execution)

- predpostavi se, da je napoved skokov z dinamično predikcijo pravilna
- potreben pa je mehanizem, ki v primeru napačne napovedi odstrani vse, kar so naredili napačno napovedani ukazi
  - izvršitev ukaza ne sme vplivati na registre, dokler ni potrjena pravilnost napovedi skoka
  - **preureditveni izravnalnik** (reorder buffer, ROB)
    - začasno hrani rezultate ukazov

Preureditveni izravnalnik je realiziran kot FIFO vrsta v obliki krožnega bufferja

- ukazi so v njem v pravilnem vrstnem redu

Vsako polje v njem ima 4 parametre:

1. vrsta ukaza
   - skoki, store ali registrski ukazi
2. ponor
   - register ali pomnilniška beseda
3. vrednost
   - rezultat ukaza, ki naj se shrani
4. veljavnost
   - 1, če je v parametru vrednost že rezultat ukaza
   - 0, če se na rezultat ukaza še čaka

Velikost preureditvenega izravnalnika se imenuje *ukazno okno* (instruction window)

- določa, koliko ukazov se lahko izvede špekulativno
- če je velika, se porabi več energije za izbris vsega izračunanega

Večizstavitveni procesorji

Približevanje CPI vrednosti 1

- dinamična predikcija skokov
- dinamično razvrščanje
- špekulativno izvrševanje ukazov

CPI < 1:

- v vsaki urini periodi se mora prevzeti in izstaviti več kot 1 ukaz:

-> večizstavitveni procesorji (multiple issue processors)

- dejanska paralelnost na osnovi več enot

- običajno se uporablja IPC = 1 / CPI

Vidiki prevzema in izstavljanja ukazov

1. Prevzem ukazov
   - izstavitev *n* ukazov zahteva, da je ukazni PP sposoben dostavljati *n* ukazov v periodi
   - treba je povečati širino dostopa do čakalne vrste in zmogljivost pomnilnika
2. Izstavljanje ukazov
   - če je med (*n*) ukazi skok z napovedanim skočnim pogojem, se preostali ukazi ne izstavijo
     - prevzem ukazov v naslednji periodi pa se začne z napovedanega skočnega naslova
   - potrebno je tudi preveriti medsebojne odvisnosti med operandi
     - pri *n* ukazih s 3 reg. operandi je potrebnih $n(n-1)$ primerjav $(2(n-1) + 2(n-2) ...)$

Strojno ugotavljanje podatkovnih odvisnosti je zahtevno za realizacijo, zato sta se pojavili 2 rešitvi in s tem 2 vrsti večizstavitvenih procesorjev:

1. Superskalarnost (dinamično ‘več-izstavljanje’)
2. VLIW (statično ‘več-izstavljanje’ - prevajalnik)

Superskalarni procesorji

Dinamično določanje, kateri ukazi se v dani periodi ure izstavijo
- če se jih lahko izstavi največ n, je to n-kratni superskalarni procesor (n-way superscalar processor)

n(n-1) primerjav je težko izvesti v eni periodi
- pri superskalarnih procesorjih se primerjave razdeli med več stopenj cevovoda

Ukazi se sicer izvajajo špekulativno
- le da jih je več hkrati
- Št. FE običajno > n, da se zmanjšajo SN

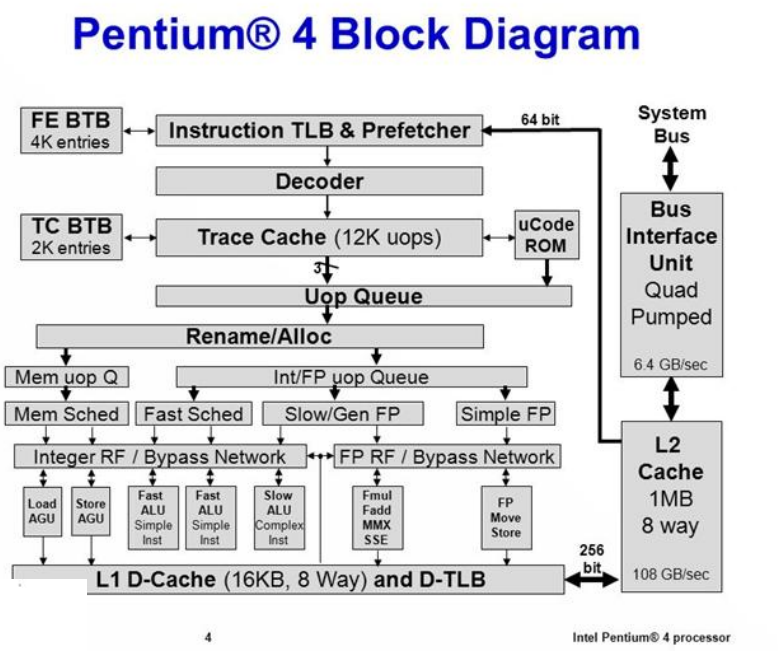
Potrebujejo pa večjo zmogljivost:
- prenosnih poti,
- preureditvenega izravnalnika,
- dostopa do registrov

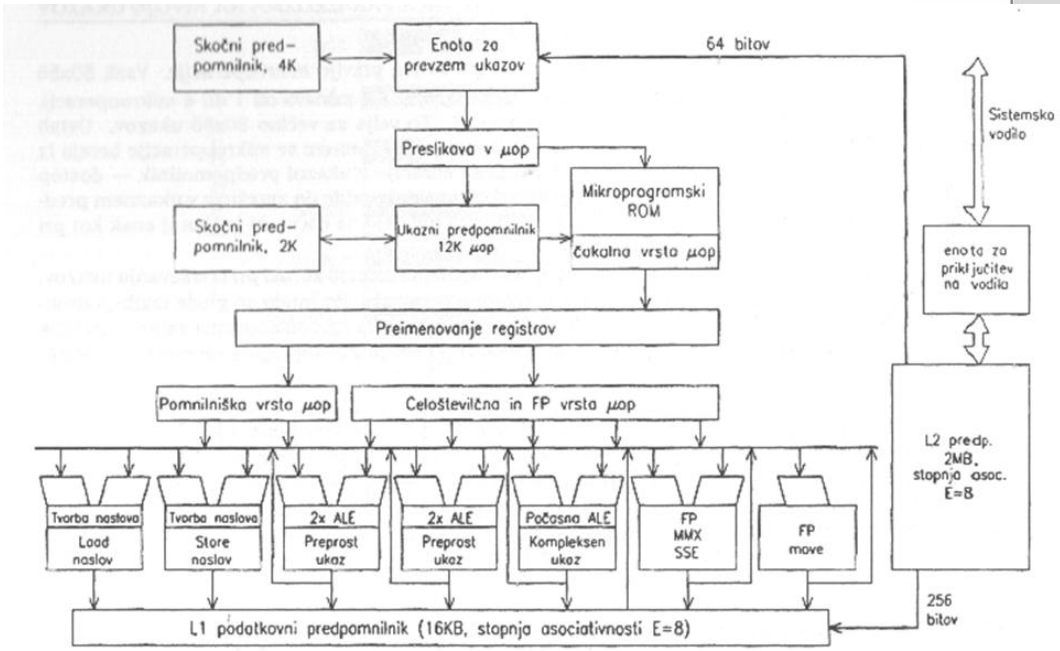
Najbolj zapleteni del superskalarnega procesorja je ROB

Zato procesorji po letu 2000 uporabljajo **eksplicitno preimenovanje registrov**

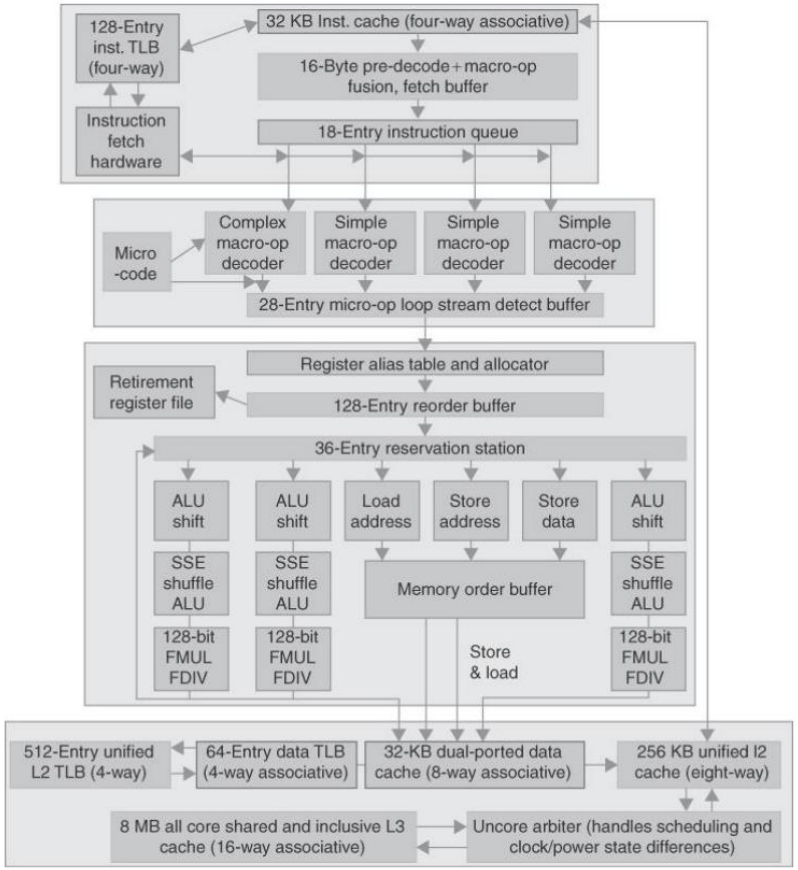
- preureditveni izravnalnik je preprostejši
  - skrbi le za vrstni red ukazov, ne pa tudi za operande iz registrov
- procesor ima še dodatne registre
  - *razširjena množica registrov* (lahko tudi nekaj sto)
  - *preimenovalna tabela* določa, kateri so v neki periodi programsko dostopni
- korak izstavljanja je drugačen:
  - Iz čakalne vrste se vzame *n* ukazov
  - Izhodni register vsakega ukaza se preimenuje v enega od prostih registrov
    - S tem se odpravijo nevarnosti WAW in WAR, ki izvirajo iz imenskih odvisnosti
  - Preveri se medsebojna odvisnost operandov (kot že prej opisano)
    - Po potrebi se popravijo številke vhodnih registrov
  - Ukazi se prenesejo v ROB
    - Globina se določi na osnovi števila registrov
  - Ukazi se prenesejo v FE

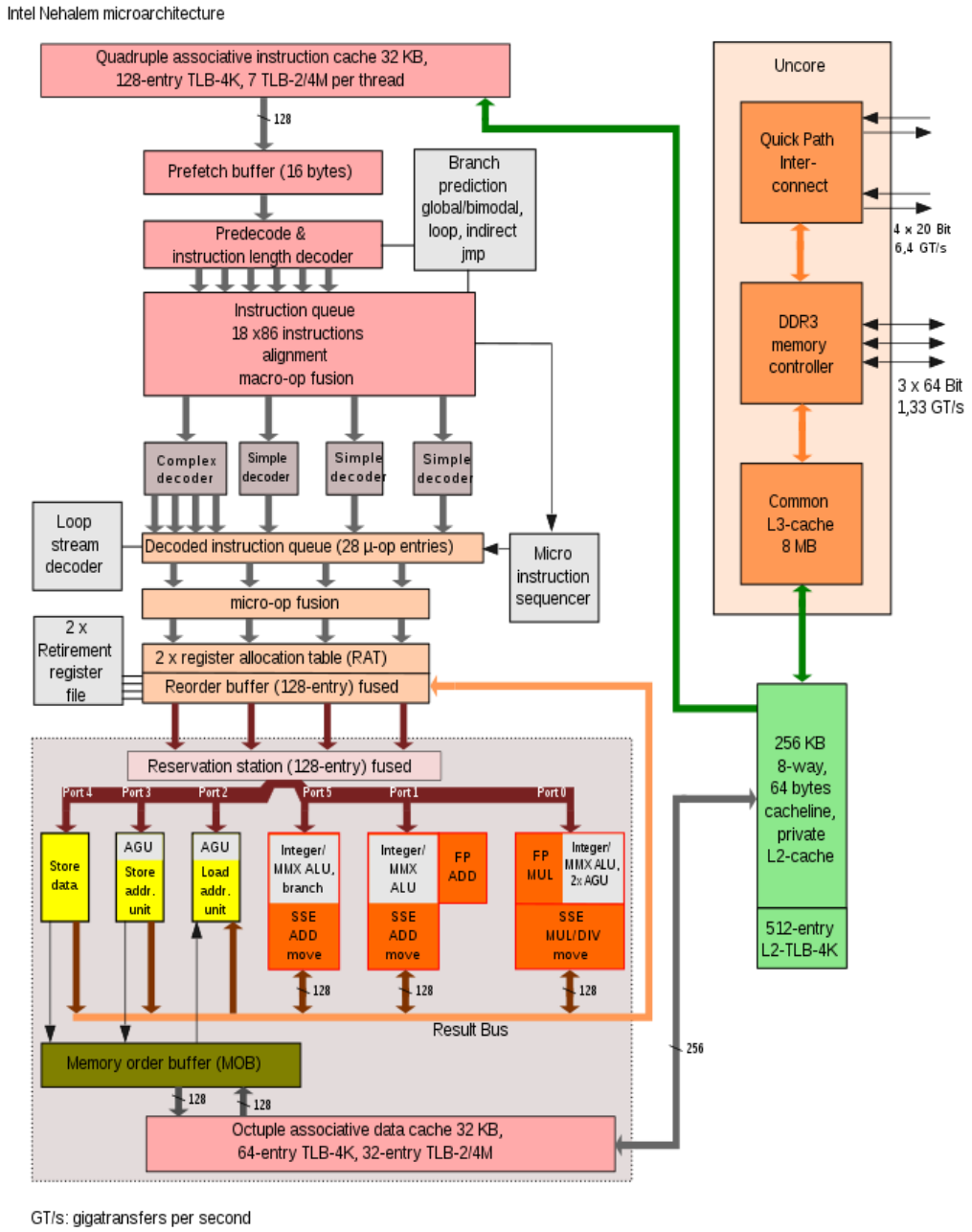
- Primer starejšega superskalarnega procesorja: Intel Pentium 4
  - mikroarhitektura NetBurst
  - 7 FE:
    - load
    - store
    - preproste celoštevilske operacije (x2)
    - zahtevne celoštevilske operacije
    - FP operacije
    - prenosi FP operandov iz/v pomnilnik





Intel Core i7 (prvi)





Procesorji VLIW

Procesorji VLIW (very long instruction word) imajo dolge ukaze

- Vsebujejo več običajnih ukazov, ki se lahko izvršujejo paralelno
  - Npr. da vsak zaposli eno FE
- Tipičen ukaz:
  - 3 celošt. ukazi
  - 2 FP ukaza
  - 2 pomnilniška dostopa
  - 1 skok
- CPE ne ugotavlja odvisnosti in nevarnosti
  - To je delo prevejalnika
  - Če ne uspe najti dovolj neodvisnih ukazov za vse enote, se nekaterim FE da ukaz NOP

Dvo-izstavitveni cevovod:

- hkrati se izvajata 2 ukaza različne vrste

Vrsta ukaza	1	2	3	4	5	6	7	8
ALE ali skok	IF	ID	EX	ME	WB			
Load ali store	IF	ID	EX	MEM	WB			
ALE ali skok		IF	ID	EX	MEM	WB		
Load ali store		IF	ID	EX	MEM	WB		
ALE ali skok			IF	ID	EX	MEM	WB	
Load ali store			IF	ID	EX	MEM	WB	
ALE ali skok				IF	ID	EX	MEM	WB
Load ali store				IF	ID	EX	MEM	WB

Kako bi podano zanko razporedili na dvo-izstavitveni CPE s prejšnje strani?

```
Zanka:   lw x31, 0(x20)
         add x31, x31, x21
         sw x31, 0(x20)
         addi x20, x20, -4
         blt x22, x20, Zanka
```

spremenili vrstni red ukazov, da bi bilo čim manj čakalnih period?

	ALE/skok	Load/store
Zanka:	-- (nop)	lw x31, 0(x20)
	addi x20, x20, -4	-- (nop)
	add x31, x31, x21	-- (nop)
	blt x22, x20, Zanka	sw x31, 0(x20)

CPI = 4/5 = 0.8 (IPC = 5/4 = 1.25)

Potencialne prednosti VLIW:

- Prevajalnik vidi celoten program
  - Zato lahko odkrije več paralelnosti kot logika v procesorju, ki vidi le ukazno okno
  - Odkrivanje paralelnosti se izvede samo enkrat
- Procesor je lahko preprostejši
  - Ne rabi logike za odkrivanje paralelnosti
  - Zato je frekvenca ure lahko višja

Digitalno procesiranje signalov

- Veliko paralelnosti

EPIC (explicitly parallel instruction computing)

- Intel 1997
- *predikatni ukazi*
- Itanium 1 (2000), 2 (2002)

Omejitve paralelizma na nivoju ukazov

Količina paralelnosti v programih je omejena

- S povečevanjem količine logike lahko pridobimo le do neke meje

Koliko paralelnosti na nivoju ukazov je v nekem programu?

- zamislimo si idealni superskalarni procesor
- lastnosti:
    1. ni strukturnih nevarnosti
        - neomejeno število registrov za preimenovanje
        - neomejeno število FE, vse izvršijo operacijo v 1 periodi
        - torej se v 1 periodi lahko izstavi in izvrši neomejeno število ukazov
    2. ni kontrolnih nevarnosti
        - popolno napovedovanje skokov (vsi napovedani 100%)
        - neomejeno ukazno okno
        - do izbrisa zaradi napačne špekulacije nikoli ne pride

3. naslovi vseh pomnilniških operandov znani vnaprej
   - ukazi load se lahko prestavijo pred store (če ne gre za isti naslov)
4. predpomnilniki nimajo zgrešitev
   - vsi pomnilniški dostopi trajajo 1 periodo

- ostanejo le prave podatkovne nevarnosti

- izvajamo različne programe in merimo dosegljivi IPC
  - na 6 programih iz SPEC92
    - IPC od 18 do 150
    - povprečni IPC okrog 80
  - z upoštevanjem bolj realnih lastnosti dosegljivi IPC pade na okrog 5
  - realni IPC pa je manjši

Paralelizem na nivoju niti

Paralelizem na višjem nivoju, ki ga na nivoju ukazov ni mogoče izkoristiti

- izvrševanje se razdeli v več neodvisnih poti (niti)
  - thread-level parallelism
- Pri večnitnosti (multithreading) si niti delijo FE enega procesorja
- vsaka nit ima svoje stanje
  - ločeno in neodvisno od drugih niti
  - nit ima svojo kopijo registrov, svoj PC, svoje tabele strani in nekatere programsko nevidne registre
- niti pa si delijo GP in PP
- nit vidi procesor, kakor da je namenjen le njej sami
  - en fizični procesor je videti kot več *logičnih procesorjev*
- problem: niti je treba definirati (paralelno programiranje)
  - eksplicitni paralelizem
  - obstoječe programe je (bi bilo) potrebno predelati!

Več oblik večnitnosti:

1. **Časovna večnitnost** (temporal multithreading)
   - preklapljanje, niti se izmenjujejo

   a. *Drobno-zrnata večnitnost*
      - preklop med nitmi vsako urino periodo
      - treba je shraniti celotno stanje cevovoda
      - če bi posamezna nit morala čakati, se jo v tem ciklu izpusti (da se ne izgublja časa)
      - hiba je upočasnitev posameznih niti

   b. *Grobo-zrnata večnitnost*
      - preklop samo, kadar pride pri niti do daljšega čakanja
      - ni treba shraniti stanja cevovoda (čakamo, da se izprazni)

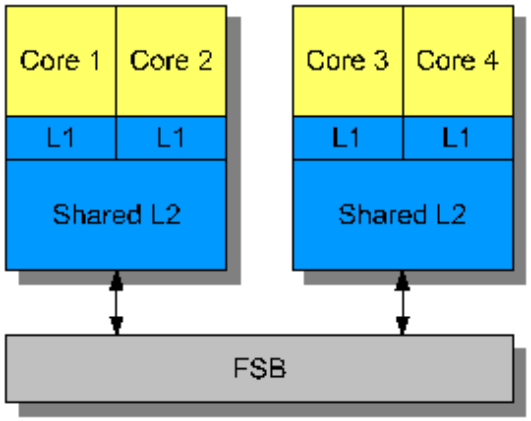
2. **Istočasna večnitnost** (simultaneous multithreading, SMT)
   - pri večizstavitvenih procesorjih
     - Intel Pentium 4: Hyper-threading (običajno 2 niti)
   - ni potrebno veliko sprememb
   - prednost: ni medsebojnih odvisnosti
   - hiba: v določenih primerih se zmogljivost tudi poslabša
     - programerji morajo preverjati, ali se pri neki aplikaciji SMT obnese, ali ne

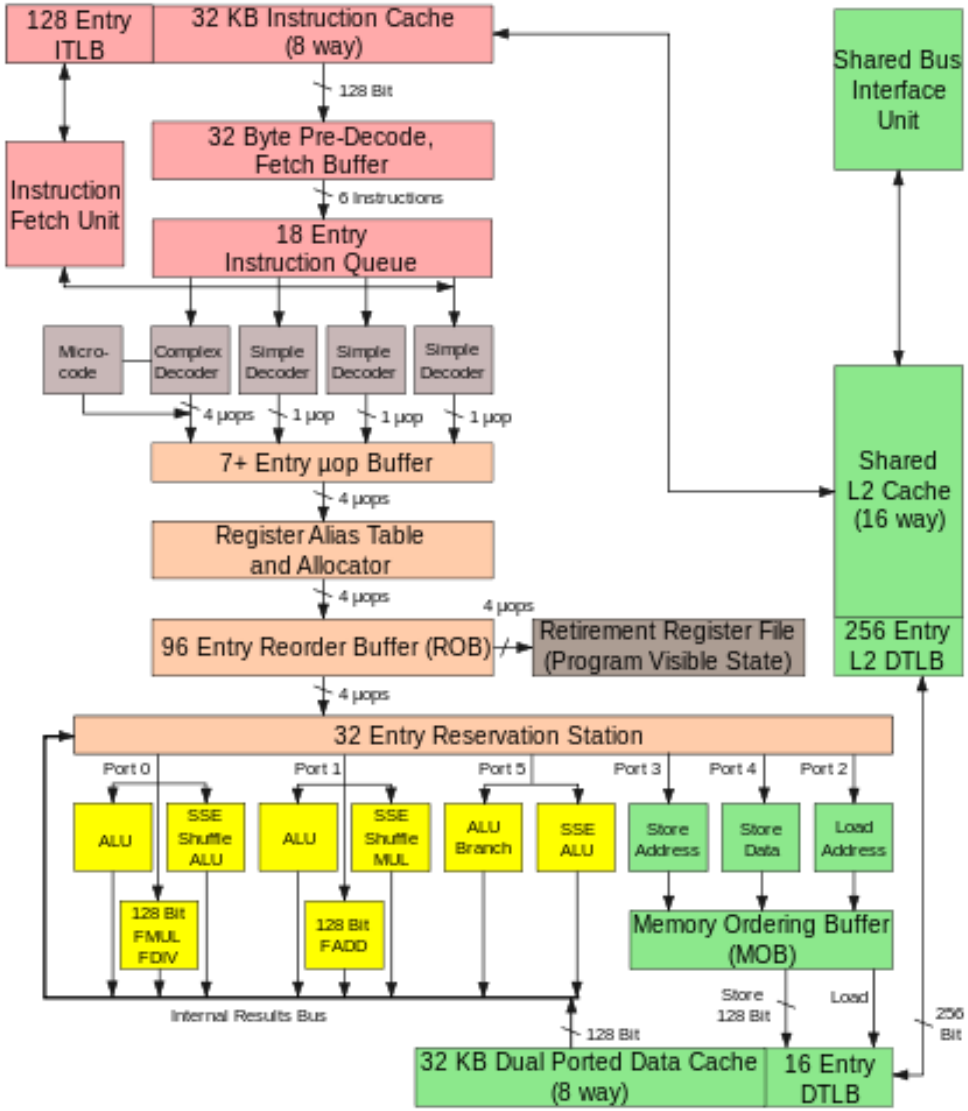
Večjedrni procesorji (multicore)

- več CPE (jeder) na istem čipu
- pogosto imajo CPE svoje PP L1, L2 in višje pa si delijo
  - zato CPE niso čisto neodvisne
  - jedra so običajno tudi večnitna (pogosto dvonitna)
- množična proizvodnja večjedrnih procesorjev
  - predvsem v interesu proizvajalcev
    - ceneje kot vlagati v razvoj novih rešitev
  - uporabniki redko lahko uporabijo veliko število jeder
  - Npr., procesor z IPC = 4 bi bil verjetno bolj koristen kot 8-jedrni
  - “uporabniki se bodo pač morali naučiti pisanja večnitnih programov” ?!

Primer:

- Intel Core 2 Quad





Intel Core 2 Architecture

Večprocesorji (multiprocesorji)

Multiprocesorji s skupnim pomnilnikom (shared memory):

- večjedrniki, simetrični multiprocesorji (*symmetric multiprocessors* - SMP), UMA (*uniform memory access*)
  - dostop do vseh delov pomnilnika enako hiter
- več večjedrnih čipov, porazdeljen skupni pomnilnik (*distributed shared memory* - DSM), NUMA (non-*uniform memory access*)
  - dostop do vseh delov pomnilnika ni enako hiter
- problem pomnilniške koherence in konsistence: protokoli skladnosti (koherence) – MSI, MESI, ... pri pisanju v PP se predpomnilnikom ostalih jeder razveljavijo kopije skupnih PP-blokov

Multiračunalniki (Cray X, IBM Blue Gene):

- več večjedrnih čipov,
- povezovalna omrežja

Gruče (clusters)

- veliko število nalog, ki se izvajajo vzporedno
- oblak

Paralelizem na nivoju podatkov

Arhitekture SIMD:

- vektorski procesorji
  - vektorski ukazi (npr. `vadd v3,v1,v2`)
  - vektorski registri, skalarni registri
  - vektorske FE
  - vektorske enote load/store globoko cevovodne
- (multimedijske) razširitve nabora ukazov (x86: MMX, SSE, AVX)
- grafične procesne enote (GPE), GPGPU
  - visoka stopnja vzporednosti
  - GPE vsebuje večje število SM ('Streaming multiprocessor') – en SM na sliki desno
  - vsak SM izvaja vektorske ukase (npr. na 32 procesnih enotah – SIMD lanes)
  - kontrolna enota (scheduler) pošilja v izvajanje niti, ki so pripravljene – pogosto menjavanje
  - 'skrivanje latence' na osnovi velikega števila čakajočih niti

Domensko specifične arhitekture

DSA

- izvajajo le specifične naloge, a te zelo učinkovito (npr. množenje matrik) – pospeševalniki (accelerators)
- CPE izvaja ostale naloge

Tenzorska procesna enota (TPE, TPU)

- Google, ASIC, v1 2017
- faza sklepanja (inference) za globoke nevronske mreže (Deep neural networks, DNN)
- v1: 256x256 8-bitna matrična množilna enota (MMU)

TPE deluje kot koprocesor

- ukaze dostavlja CPE preko vodila PCIe x16
- uteži so v DRAMu izven čipa
- MMU kot sistolično polje (kot nek 2D cevovod)