UVOD

Arhitektura računalniških sistemov

Kako je zgrajen in kako deluje računalnik?
- Rač. sistem

Arhitektura računalnika
- računalnik, kakor ga vidi programer na nivoju strojnega jezika

Organizacija računalnika
- zgradba, sestavni deli in povezave

Isto arhitekturo se da realizirati z različnimi organizacijami

Vsebina

1. Računanje
2. Zgodovina
3. Osnovni principi delovanja
4. Predstavitev informacije in aritmetika
5. Ukazna arhitektura (ISA)
6. Ukazi
7. Centralna procesna enota
8. Paralelizem na nivoju ukazov
9. Glavni pomnilnik in predpomnilniki
10. Navidezni pomnilnik

Literatura

Osnovna:

Dušan KODEK: **Arhitektura in organizacija računalniških sistemov**, BI-TIM, Ljubljana, januar 2008, ISBN 978-961-6046-08-4

Dodatna:

David A. PATTERSON & John L. HENNESSY: **Computer Organization and Design - The Hardware/Software Interface**, 3th ed., Morgan Kaufmann.

John L. HENNESSY & David A. PATTERSON: **Computer Architecture - A Quantitative approach**, 4th ed., Morgan Kaufmann.

Obveznosti

Ocena predmeta:

- vaje: 1/3
- pisni izpit: 1/3
- teoretični izpit: 1/3

Za uspešno opravljen predmet mora biti vsak od posameznih treh deležev ocenjen pozitivno!

Vaje

Oceno vaj pridobite z dvema kolokvijema.

Ocena je pozitivna, če je:

- povprečje kolokvijev vsaj 30 % in
- vsak kolokvij vsaj 20 %.

Ocena vaj velja le za tekoče šolsko leto.

Za podrobnosti glej spletno učilnico (stran Pravila).

Pisni izpit

Pisni izpit lahko opravite s kolokviji, če je

- povprečje kolokvijev vsaj 60 % in
- na vsakem kolokviju zberete vsaj 20 % točk.

Če opravite pisni del izpita s kolokviji, se lahko za teoretični izpit prijavite na kateregakoli od razpisanih izpitnih rokov v tekočem šolskem letu.

Asistenti

Miha Janež (miha.janez@fri.uni-lj.si, R3.56)
Nejc Ilc (nejc.ilc@fri.uni-lj.si, R2.41)
Ratko Pilipović (ratko.pilipovic@fri.uni-lj.si, R2.41)
Davor Sluga (davor.sluga@fri.uni-lj.si, R2.41)
Rok Češnovar (rok.cesnovar@fri.uni-lj.si, R2.41)

Opozorila

Pri kolokvijih in pisnih izpitih ni dovoljeno uporabljati literature
- dovoljen list z ukazi, en A4 list s formulami, kalkulator, pisalo

Na izpitu iz teorije imate lahko le pisalo

Ocena iz kolokvijev kot pisni del izpita velja samo za tekoče šolsko leto do prvega opravljanja teoretičnega dela izpita!
- Torej: Ocena iz kolokvijev vam propade, če teoretičnega dela izpita ne opravljate v istem šolskem letu kot ste opravili kolokvije, oziroma, če na njem dobite negativno oceno.

Ocena iz vaj velja le za tekoče šolsko leto!
- Torej: Ocena iz vaj vam propade, če v istem šolskem letu, ko ste vaje opravili, ne opravite tudi izpita.

1
Narava računanja in stroji za računanje

Razlogi za strojno računanje

Čemu strojno računanje?

Ročno računanje, 2 problema:

1. počasnost
2. nezanesljivost

Povezava med ročnim in strojnim računanjem

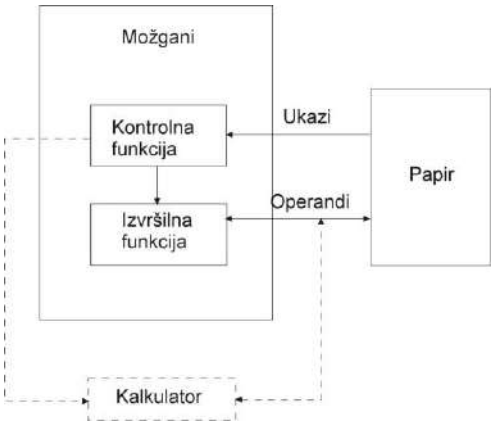
Ročno računanje
- papir (→ pomnilnik)
- možgani (→ procesor)

Papir
- ukazi (navodila)
- operandi

Možgani pri računanju opravljajo 2 funkciji:
- kontrolna funkcija
  - prevzema ukaze in skrbi za pravilen vrstni red izvrševanja ukazov
- izvršilna funkcija
  - npr. seštevanje, množenje, itd.

Papir lahko delimo v 2 vrsti:
- knjiga z navodili (→ ukazi)
- papir za vmesne in končne rezultate (→ operandi)

Ročno računanje



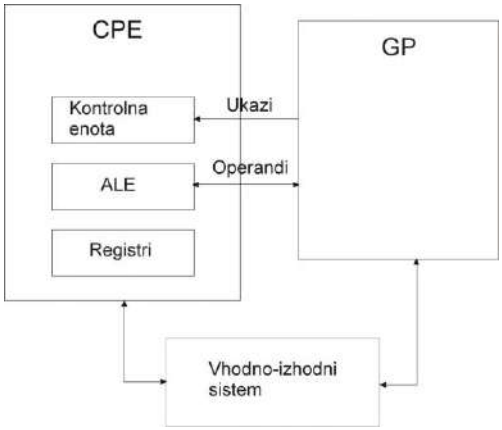
Strojno računanje

Današnji računalniki računajo na podoben način kot človek

Tudi računalnik ima lahko pomnilnik ločen na 2 dela:

- del za ukaze
- del za operande

Strojno računanje



Računanje in izračunljivost

Kakšni naj bodo stroji, ki znajo računati?

- Potrebno je najprej natančno definirati, kaj sploh je računanje

Tudi teoretično zanimiv problem:

- Kakšen naj bo stroj, da bo znal izračunati vse, kar se da izračunati?
- Kaj sploh pomeni, da se nekaj da izračunati?

Kako definirati računanje?

Računanje lahko definiramo kot določanje vrednosti funkcije $z = f(x)$

- funkcija f je mišljena zelo široko
- x so vhodni podatki, z pa izhodni

Beseda *računanje* (v slovenskem jeziku) ima 2 pomena:

- numerično računanje (calculation)
- računanje v širšem pomenu (computing)

Definicija izračunljivosti:

Funkcija *f(x)* je **izračunljiva**, če obstaja postopek, s katerim lahko določimo njeno vrednost (*z*) za vse možne vhodne podatke (*x*), nad katerimi je definirana.

Ta postopek je lahko zaporedje več korakov

Rečemo mu tudi algoritem

Algoritem je navodilo, ki v končnem številu korakov pripelje do želenega rezultata

- npr. Evklidov algoritem za izračun NSD 2 števil
- algoritem ni nujno povezan z računalniki
  - Npr.: recept iz kuharske knjige

Definicija izračunljivosti je torej tudi:

Funkcija je izračunljiva, če zanjo obstaja algoritem

Ali za vsak problem obstaja algoritem?

oz. Ali je vsak problem izračunljiv?

Teoretični modeli računanja:
- Turingov stroj (Alan Turing), 1936

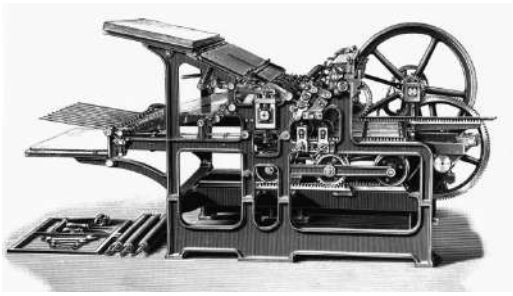
Church-Turingova hipoteza:

Problem je izračunljiv, če ga je možno v končnem številu korakov izračunati na Turingovem stroju

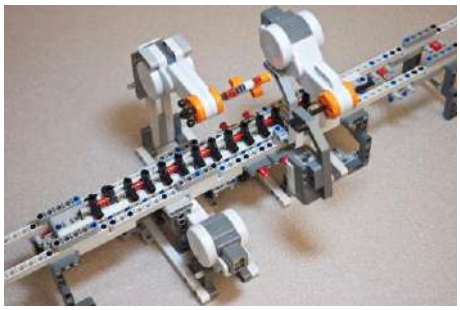
Turingovi stroji

Turingov stroj (Turing machine, TM) sestavljajo:
- procesor
- bralno-pisalna glava
- neskončno dolg trak
- mehanizem za pomik traku

- "Stroj" je mišljen kot abstrakten model računanja
  - ne kot neka mehanska naprava, npr.:



Kar pa ne pomeni, da ga ni možno fizično realizirati (v približku)



Pisanje programov za TM ni enostavno

- primitivni ukazi

Za vsako kombinacijo stanja avtomata in vhodne črke (na traku) definiramo, kaj glava zapiše na trak in smer pomika

Program za TM lahko ponazorimo s tabelo ali diagramom prehajanja stanj (DPS)

Računalniki in Turingovi stroji

Današnji rač. delujejo po von Neumannovem modelu

- ta je ekv. TM (če bi bil pomnilnik neskončen)
- manj primitiven, hitrejši
- TM je abstrakten (matematičen) model
  - enostavnost je v funkciji lažjega teoretičnega dokazovanja

Če je trak TM končen, a dolg, se da rešiti večino praktičnih problemov

Omejitve računalnikov

2 vrsti “težavnih” problemov:

- Neizračunljivi problemi
- Neobvladljivi problemi

Neizračunljivi problemi

Ustavitveni problem (Halting problem)

- Turing je dokazal, da ni mogoče napisati algoritma, ki bo ugotovil, ali se bo poljuben TM s poljubnim podatkom kdaj ustavil

Teoretične raziskave izračunljivosti

- Prevedba problema ustavljanja na problem, ki ga raziskujemo

Neobvladljivi problemi

To so izračunljivi problemi, ki pa jih ne moremo rešiti zaradi
- omejenega pomnilnika, in/ali
- omejenega časa

Teorija kompleksnosti
- prostorska kompleksnost
- časovna kompleksnost (običajno hujša)
  - polinomska: $O(n)$, $O(n*\log n)$, $O(n^2)$, $O(n^3)$, ...
  - eksponentna: $O(2^n)$, $O(n!)$, $O(n^n)$, ...

2
DOSEDANJI RAZVOJ STROJEV ZA RAČUNANJE

Digitalni princip

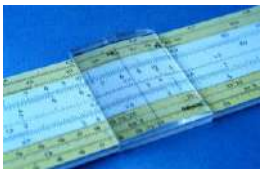
- digit (ang. številka, prst) iz latinščine
- neka fizikalna veličina diskretno predstavlja števila
  - npr. območja napetosti (ali nivoja tekočine ...)
- omejeno število stanj, npr. 10 (0, ..., 9) ali 2 (0, 1)
- natančnost se da povečati z uvedbo več številskih mest

- abak

Analogni princip

- fizikalna veličina zvezno predstavlja števila
  - mehanska (dolžina, kot), električna (napetost, upornost), ...
- omejena natančnost
- analognih rač. danes praktično ni več

- logaritmično računalo (Rechenschieber)



Analogni računalniki

Obdobje mehanike

Prvi kalkulatorji

Kalkulator je naprava (stroj), ki izvaja aritmetične operacije

- prvi kalkulatorji so izvajali le osnovne operacije
  - + in -, morda tudi * in /

Schickard, 1623

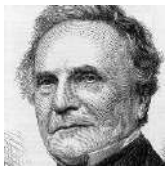
- zobata kolesa (10 zobnikov)
- mehanizem za prenos naprej
- ročen pogon
- operacije
  - seštevanje, odštevanje
  - množenje, deljenje z nekaj dela

Pascal, 1642

- 2 skupini koles po 6
  - ena je akumulator
  - druga za prištevanje ali odštevanje od števila v akumulatorju

Leibniz, 1671

Charles Babbage



Njegovi stroji precej podobni današnjim računalnikom

- tehnologija primitivna

Diferenčni stroj (Difference engine), 1823

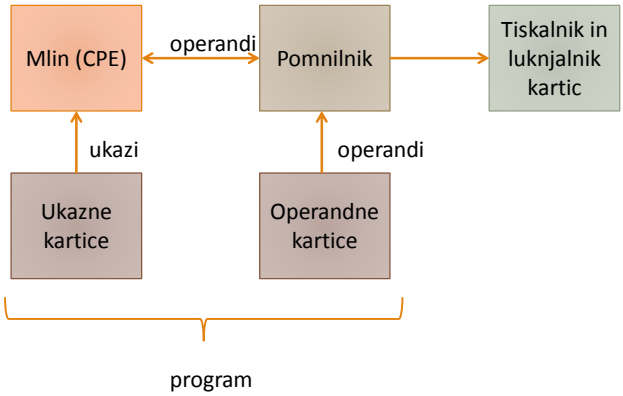
- aproksimacija funkcij s polinomi (na osnovi metode končnih diferenc)
- zaporedje fiksnih operacij

Analitični stroj (analytical engine), okrog 1835

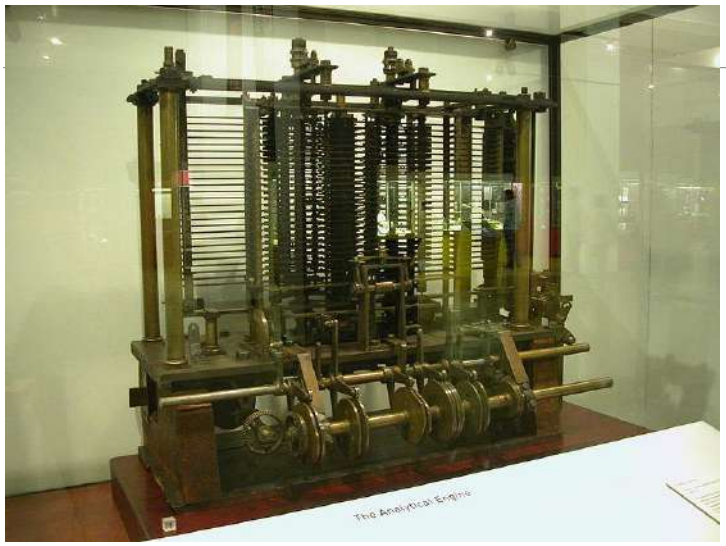
- Prvi računalnik
- Ni bil realiziran zaradi velike zahtevnosti in stroškov
- Računski del
  1. Mlin (mill): izvedba operacij
  2. Pomnilnik (store): shranjuje operande
- Luknjane kartice 2 vrst
  1. Ukazne kartice (s programi)
  2. Operandne kartice

Babbage za 100 let utonil v pozabo

Zgradba analitičnega stroja



Analitični stroj (zgrajen kasneje)



Elektromehanski stroji

Elektrotehnika ponuja nove možnosti

- elektromotorji za pogon mehanskih kalkulatorjev
- električno branje luknjanih kartic

Rele (relay)

- električno-krmiljeno stikalo

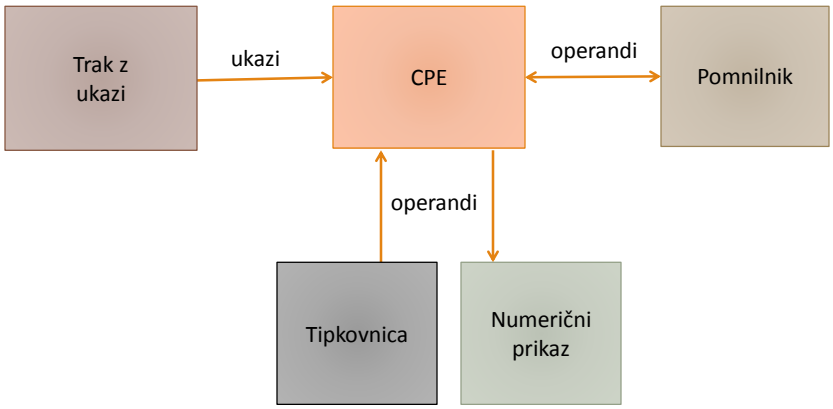
Konrad Zuse zgradil prvi delujoči računalnik

Zusejevi računalniki

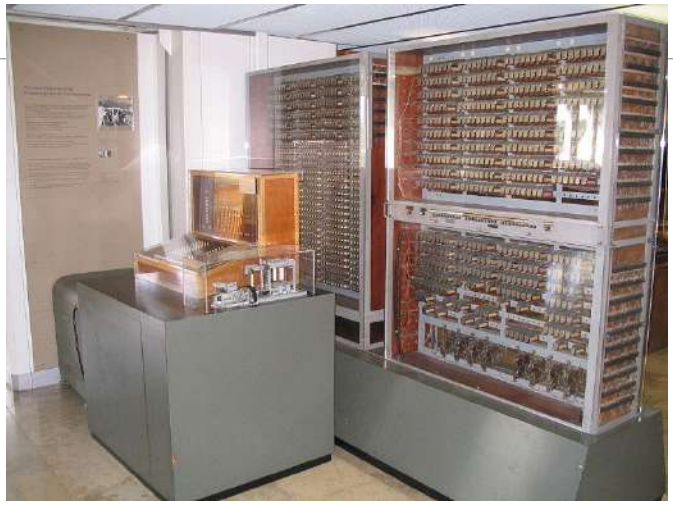
- Z1, 1938, mehanski
- Z2
- Z3, 1941, prvi delujoči (splošnonamenski) računalnik
  - 2600 relejev
  - pomnilnik 64 22-bitnih besed (releji)
  - 8-bitni ukazi
  - luknjan trak
  - plavajoča vejica: 14-bitna mantisa, 7-bitni eksp. + predznak
  - Tipkovnica
  - Hiba: ni imel pogojnih skokov
  - Frekvenca 5-10 Hz
  - Uničili so ga 1943 med bombardiranjem Berlina

Z1

Zgradba Z3



Z3 (kopija)



Z4 (Deutsches Museum, Muenchen)

Harvard Mark I

- Howard Aiken, izdelava IBM 1943
- 15m v dolžino
- elektromeh. desetiška števna kolesa
- pomnilnik 72 x 23 desetiških mest
- luknjan trak (24 stolpcev - bitov)
- Ukazi oblike A1 A2 OP
  - pomn. naslova + op., vsi 8-bitni

Elektromeh. stroji (40. leta) so bili uresničitev zamisli Babbagea

Njihov problem je mehanika, ki omejuje

- hitrost (vztrajnost gibljivih delov)
- zanesljivost (veliko zobnikov in vzvodov)

Hitro so zastareli zaradi pojava nove tehnologije, ki ne uporablja mehanike

- elektronika

Prvi elektronski računalniki

Zakaj je elektronika hitrejša?

- rele potrebuje vsaj nekaj ms za preklop
- elektroni so bistveno hitrejši

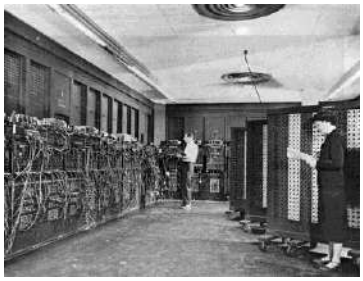
Elektronka ('vakuumska cev')

ENIAC

(Electronic Numerical Integrator And Calculator), 1945, vojaško financiran

- pomnilnik 20 x 10 desetiških števil
  - pomnilni element 10-bitni krožni števec iz 10 FF (2 elektronki na FF)
  - skupno 4000 elektronk
- funkcijska tabela (104 x 12 desetiških mest)
  - stikala
- fiksna vejica
- operacije +, -, *, /, sqrt
  - +, - 0.2ms, * 3ms, / 30ms

- ročno programiranje (stikala, prevezovanje kablov)
  - 6000 stikal
  - zzzelo zamudno
- podatki na luknjanih karticah
- 18000 elektronk, 1500 relejev, 30 m, 30 ton, 140kW
- programiranje je lahko trajalo tudi več dni
  - zato so razmišljali (von Neumann) o shranjenem programu



Elektronski računalniki s shranjenim programom

John von Neumann napisal predlog za EDVAC (Electronic Discrete Variable Computer)

- po njem von Neumannovi računalniki

Stroj voden *od znotraj*

Prednosti shranjenega programa

- dostop do ukazov enako hiter kot dostop do operandov
- program lahko kot vhodni podatek vzame drug program in ga spremeni v tretji
  - prevajalniki, zbirniki

EDVAC, 1951

- pomnilnik 1K 16-bitnih besed, s krožnim dostopom
  - + 20K besed v pomožnem pomnilniku
  - dvonivojska pom. hierarhija
- 3000 elektronk
- dvojiški stroj
- serijsko (bit za bitom)
- ukazi

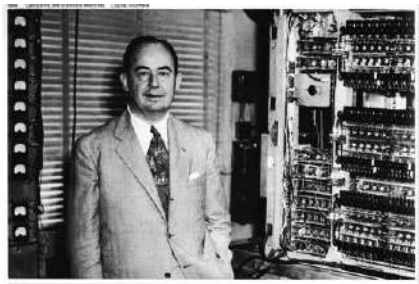
  A1 A2 A3 A4 OP
  - A1, A2: naslova vhodov
  - A3: naslov izhoda
  - A4: naslov nasl. ukaza

IAS, 1951

- o njem dostopne vse informacije!
- dvojiški
- pomnilnik na osnovi variante katodne cevi
  - čas dostopa neodvisen od prejšnjega naslova
  - 1K x 40
- hkratni dostop do bitov besede
- ukazi

  OP A

- akumulator, AC 40-bitni
- 1-operandni, 1-naslovni računalnik
- ukazi si sledijo po naraščajočih naslovih (razen pri skokih)
  - 12-bitni programski števec (PC ← PC +1)
- beseda
  - 40-bitno število v 2'K
  - dva 20-bitna ukaza
    - 8(OP) + 12(A)
- 40-bitni pomožni akumulator MQ



Razvoj po letu 1950

Komercialni interes
- serijska proizvodnja, nižja cena
- razlog za razmah niso več numerični problemi

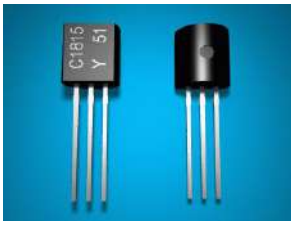
Mejniki pri razvoju
1. mehanski kalkulatorji
2. programsko voden rač. za splošne namene (Babbage, realizacija 1940. leta)
3. elektronika (ENIAC, 1945)
4. von Neumannovi rač. (shranjen program), (EDVAC, IAS, ...)

po 1951 je razvoj bolj tehnološki, ne toliko arhitekturni

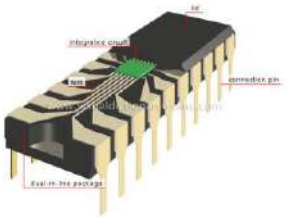
Razvoj tehnologije

Tranzistor, 1947
- Bell Labs (Shockley)

Uporaba tranzistorja
- ojačevalnik
- stikalo

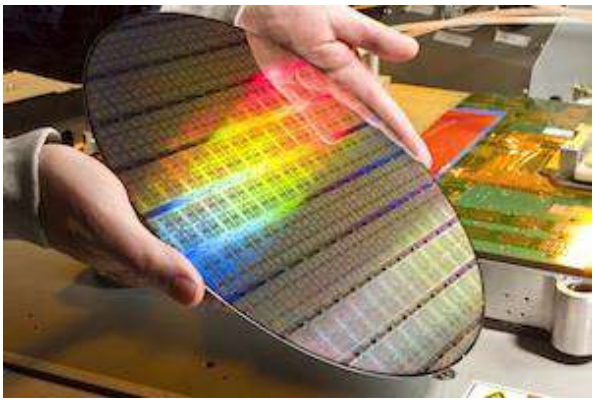


Integrirana vezja (čipi), 1958



Moorov zakon
- podvojitev števila transistorjev na čipu vsakih 18 mesecev
- 2000 (1971), nekaj milijard (danes)

Silicijeva rezina (wafer)



Razvoj programiranja

Nalaganje programa iz zunanjega (pomožnega) v glavni pomnilnik

Bootstrap

Nekdaj programskih orodij, ki olajšajo programiranje (OS, zbirniki, prevajalniki, urejevalniki), ni bilo

- programiranje je potekalo z vpisovanjem ničel in enic (strojni jezik)

Programski jeziki

Simbolični zapis: Zbirni jezik (Assembly language)

Zbirnik (Assembler) je program, ki pretvarja programe iz zbirnega jezika v strojni jezik

Višji programski jeziki
- prvi: FORTRAN, ALGOL, COBOL, LISP, ...
- kasneje: Pascal, C, C++, Java, ...

Primerjava
- koda v zbirnem oz. strojnem jeziku hitrejša
- programiranje v zbirnem jeziku počasnejše

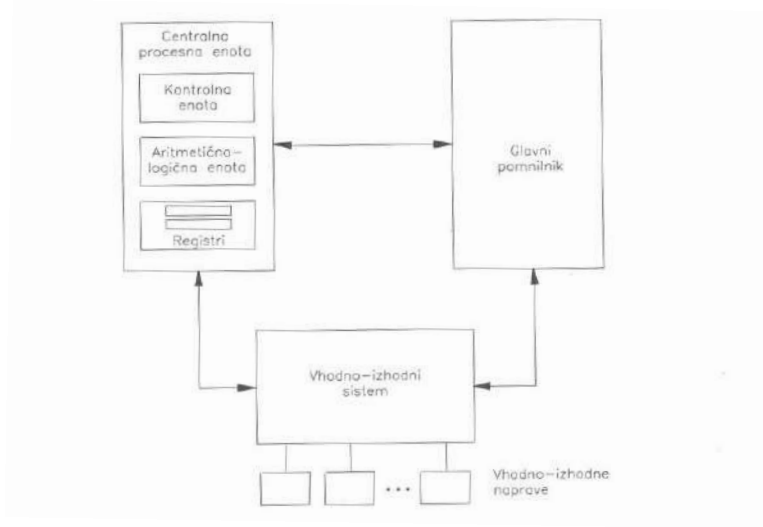
3
OSNOVNI PRINCIPI DELOVANJA RAČUNALNIKOV

- 2 ugotovitvi iz prvih dveh poglavij:
  - Definicija izračunljivosti po Church-Turingovi hipotezi
  - lastnosti stroja, ki je zmožen izračunati vse, kar se da izračunati
- Von Neumannov računalnik
  - ekvivalenca* s TM
  - to ni edini možen tak stroj

Von Neumannov računalniški model

- Von Neumann-ov računalnik:
  1. Sestavljajo ga
     - centralna procesna enota (CPE)
     - glavni pomnilnik (GP)
     - vhodno/izhodni (V/I) sistem
  2. Ima program shranjen v GP
  3. CPE jemlje ukaze programa iz GP in jih zaporedoma izvršuje

Zgradba von Neumannovega računalnika



Glavni deli von Neumannovega računalnika

1. **CPE oz. procesor**
   - zakaj centralna
   - mikroprocesor
   - vodi dogajanje v računalniku
   - osnovna naloga CPE je jemanje ukazov iz pomnilnika in njihovo izvrševanje
   - CPE delimo na tri dele:
     1. **kontrolna enota** nadzoruje aktivnosti
        - prevzem ukazov in operandov
        - aktiviranje operacij
     2. **aritmetično-logična enota (ALE)** izvršuje večino ukazov
     3. **registri** začasno shranjujejo podatke

2. **Glavni pomnilnik**
   - zakaj glavni
   - v njem so shranjeni ukazi in operandi
   - GP sestavljajo pomnilniške besede (vsaka ima svoj naslov)
   - tehnologija DRAM

3. **Vhodno/izhodni (V/I, ang. I/O) sistem**
   - namenjen prenosu informacije iz in v zunanji svet
   - vhodno/izhodne oz. periferne naprave so fizično najvidnejši del računalnika
     - tipkovnica, miška, monitor, modem, disk, tiskalnik, ...
     - pretvarjajo informacijo iz CPE v obliko, primerno za človeka ali druge naprave
       - nekatere služijo kot pomožni pomnilnik

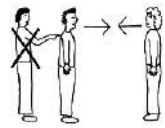
Ukaz

- Ukaz je shranjen v eni ali več (sosednih) pomnilniških besedah
- Vsak ukaz vsebuje
  - operacijsko kodo (katera operacija naj se izvrši)
  - informacijo o operandih, nad katerimi naj se izvrši operacija
- Format ukaza pove, kako so biti ukaza razdeljeni na operacijsko kodo in operande

- Naslov prvega ukaza (po vklopu računalnika) je vnaprej določen
- Pri vsakem ukazu sta 2 koraka:
  1. **Prevzem ukaza iz pomnilnika (fetch)**
     - to so **ukazi strojnega jezika** ali **strojni ukazi** (zaporedje ukazov je **program)**
     - strojni ukaz se bere iz tiste besede v pomnilniku, na katero kaže **programski števec** (PC, Program Counter)
  2. **Izvrševanje ukaza (execute)**
     - ukaz vsebuje operacijo in operande
     - CPE (običajno ALE) ukaz izvrši
     - PC nato vsebuje naslov naslednjega ukaza
       - običajno PC ← PC + 1 (razen pri **skočnih ukazih**)

Prekinitve

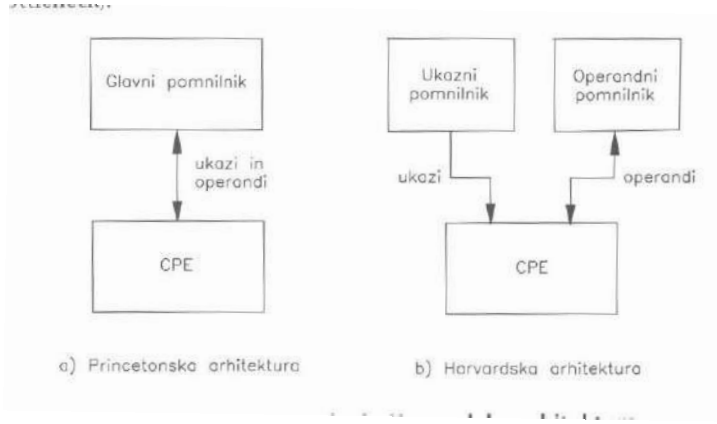
- Zaporedje teh 2 korakov se ponavlja ves čas delovanja računalnika
  - izjema so **prekinitve** (interrupt) in **pasti** (trap)



  - takrat se izvrši skok na prvi ukaz **prekinitvenega servisnega programa** (PSP)
    - pred tem se shrani vrednost PC

Glavni pomnilnik

- V glavni pomnilnik (GP) se shranjujejo ukazi in operandi
- GP je pasiven
- Za zmogljivost računalnika je pomembno, da se med CPE in GP lahko prenese dovolj informacije
  - "promet": prenosi med CPE in GP
  - ozko grlo von Neumann-ovega računalnika
  - ena od rešitev je Harvardska arhitektura (po Harvard Mark I-IV)
    - ima pomnilnik za ukaze in pomnilnik za operande
    - običajna arhitektura se imenuje Princetonska (zaradi IAS)

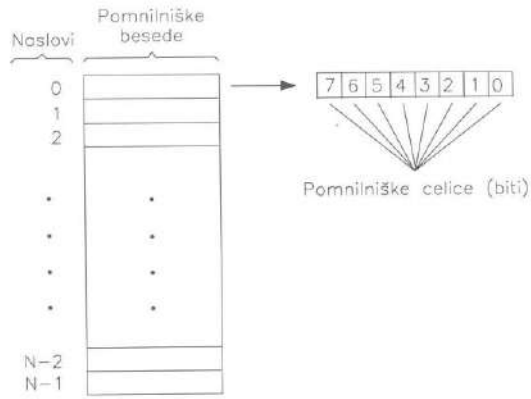


Danes prevladuje Princetonska arhitektura, vendar z ločenima *predpomnilnikoma* za ukaze in operande

Pomnilniške besede

- GP je zaporedje **pomnilniških besed** oz. **pomnilniških lokacij**
  - **dolžina pomnilniške besede** je število pomnilnih celic v njej (vsaka hrani 1 bit informacije)
    - dolžina pomnilniške besede je najpogosteje 8 bitov (1 **byte** oz. **bajt,** 1B)
  - vsaka lokacija ima svoj naslov
    - pom. beseda je def. kot najmanjše število bitov s svojim naslovom
    - iz pomnilnika ni možno prebrati (ali vanj vpisati) manj kot eno besedo

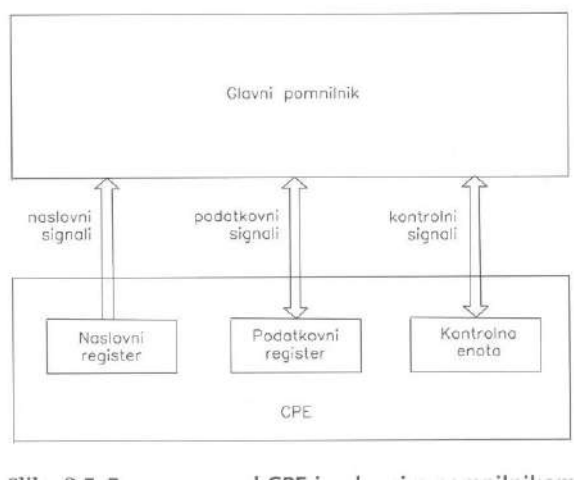
GP z dolžino besede 8 bitov:



Naslovni prostor

- velikost **naslovnega prostora** = 2 **dolžina naslova** (v bitih)
  - npr. pri 12-bitnem naslovu je naslovni prostor velikosti 2^{12} = 4096 pomnilniških besed oz. 4K
    - 2^{10} = 1024 = 1K (kilo),
    - 2^{20} = 1 048 576 = 1M (mega),
    - 2^{30} = 1 073 741 824 = 1G (giga)
- Vsebina pom. besede se lahko spreminja
  - v 8-bitno besedo lahko shranimo 2^8 različnih vsebin
- Če so registri večji kot pomnilniška beseda, je možen dostop tudi do več besed naenkrat (vsaj pri večini računalnikov)
  - npr. 32-bitni registri in 8-bitna beseda: dostop do 4 zaporednih besed hkrati (GP v obliki 4 pom.)

- CPE uporablja GP tako, da poda naslov besede in smer prenosa (lahko pa tudi št. besed)
- **Dostop** do pomnilnika (glede na smer prenosa):
  - **branje** iz pomnilnika (5x bolj pogosto)
  - **pisanje** v pomnilnik
- Informacije potujejo po *vodilih*
- CPE da naslov *na naslovno vodilo* in s kontrolnimi signali pove pomnilniku, da želi dostopiti do pomnilniške besede s tem naslovom
  - Pri branju pričakuje, da bo pomnilnik dal podatek na *podatkovno vodilo*
  - Pri pisanju da CPE na podatkovno vodilo podatek, ki se zapiše v pomnilnik



- CPE običajno vsebuje tudi
  - **naslovni register** oz. **MAR** (memory address register)
    - vsebuje naslov pomnilniške besede, do katere želimo dostopiti
  - **podatkovni register** oz. **MDR** (memory data register)
    - sem se pri branju zapiše iz pomnilnika prebrana vrednost
    - pri pisanju je v njem vrednost, ki naj se zapiše v pomnilnik
- MAR in MDR sta povezana s pomnilnikom preko naslovnih oz. podatkovnih signalov (vodil)
  - poleg teh obstajajo tudi kontrolni signali (smer prenosa (branje/pisanje), število besed, časovni parametri, ...)

- Dolžina MAR je enaka dolžini naslova
  - isto dolžina PC
  - če naslovni prostor postane premajhen, je to lahko velik problem
    - naslovi nastopajo tudi kot operandi
    - povečanje naslova pomeni drugačno zgradbo ukazov in s tem nekompatibilnost za nazaj (kar kažejo tudi ☹ izkušnje proizvajalcev)
- Dolžina MDR določa število bitov, ki se lahko naenkrat prenesejo med CPE in GP
  - enaka večkratniku dolžine pom. besede
  - njeno povečanje ni tako težavno
  - dolžina MDR vpliva na število dostopov za operand določene velikosti (npr. 64=2*32)
    - programer tega ne vidi

Semantični prepad

- Pri von Neumann-ovem računalniku iz vsebine pomnilniške besede ni mogoče vedeti, ali gre za ukaz ali operand oz. kakšne vrste je operand
  - CPE ne more zaznati nesmiselnih operacij (npr. množenje črk)
- Semantični prepad je razlika med opisom v višjem in v strojnem jeziku

Povzetek

- CPE da naslov na naslovno vodilo in s kontrolnimi signali pove pomnilniku, da želi dostopiti do pom. besede s tem naslovom
- Pri branju pričakuje, da bo pomnilnik dal podatek na podatkovno vodilo
- Pri pisanju da CPE na podatkovno vodilo podatek, ki se zapiše v pomnilnik

4
ZAPIS INFORMACIJE IN ARITMETIKA

Informacija

- Informacija v računalniku
  - Ukazi
  - Operandi
    - Numerični
      - Fiksna vejica
        - Predznačena
        - Nepredznačena
      - Plavajoča vejica
        - Enojna natančnost
        - Dvojna natančnost
    - Nenumerični
      - Logične spremenljivke
      - Znaki

Zapis nenumeričnih operandov

- Pri prvih rač. so bili operandi samo numerični
  - danes je veliko nenumeričnih
- Običajno so nenumerični operandi znaki oz. nizi znakov (strings)
- Vsak znak (character) je predstavljen z neko abecedo

Abeceda BCDIC

- BCDIC (Binary Coded Decimal Interchange Code)
- do leta 1964
- 6-bitna
- 10 številk, 26 črk, 28 posebnih znakov
- hitro je postala premajhna

000000 ... 0
000001 ... 1
000010 ... 2
...
001001 ... 9

010001 ... A
010010 ... B
010011 ... C
...

	000	001	010	011	100	101	110	111
000	0	1	2	3	4	5	6	7
001	8	9		#	@			
010	&	A	B	C	D	E	F	G
011	H	I	+0	.	¤			
100	-	J	K	L	M	N	O	P
101	Q	R	-0	$	*			
110	space	/	S	T	U	V	W	X
111	Y	Z	‡	,	%			
	0	1	2	3	4	5	6	7

Abeceda EBCDIC

- Extended Binary Coded Decimal Interchange Code
- IBM, 1964
- 8-bitna
- razširitev abecede BCD

Abeceda ASCII

- ASCII - American Standard Code for Information Interchange
- 1968
- originalno 7-bitna (128 znakov), razširjena 8-bitna
- od tega 95 natisljivih znakov in 33 kontrolnih znakov
  - A ... 1000001 (65), B ... 1000010 (66), ...
  - a ... 1100001 (97), b ... 1100010 (98), ...
  - 0 ... 0110000 (48), 1 ... 0110001 (49), ...
  - ! ... 0100001 (33), " ... 0100010 (34), ...
- kontrolni znaki za rač. komunikacije in krmiljenje V/I naprav

Koda BCD

- Spodnji 4 biti znakov za desetiške cifre v abecedah BCDIC, EBCDIC in ASCII ustrezajo njihovi dvojiški numerični vrednosti
  - to je koda **BCD** (**Binary Coded Decimal**), 4-bitna binarna predstavitev desetiških cifer

Unicode

- Unicode
  - neprofitni konzorcij, 1991
  - abecede UTF-8, UTF-16, UTF-32
  - UTF-8
    - posamezen znak zavzame od 1 do 4 bajtov
    - prvih 128 znakov isto kot ASCII (kompatibilnost)

Število bajtov	Št. bitov kode	Prva koda	Zadnja koda	Bajt 1	Bajt 2	Bajt 3	Bajt 4
1	7	00	7F	0xxxxxxx			
2	11	0080	07FF	110xxxxx	10xxxxxx		
3	16	0800	FFFF	1110xxxx	10xxxxxx	10xxxxxx	
4	21	10000	10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

Zapis numeričnih operandov v fiksni vejici

- Števila
- Pozicijska notacija
  - vsaka pozicija ima svojo težo
  - $192{,}73 = 1\times10^{2} + 9\times10^{1} + 2\times10^{0} + 7\times10^{-1} + 3\times10^{-2}$

Pozicijska notacija

- Ta zapis lahko posplošimo na uteži oblike r^{i}, kjer je r **baza** ali **radix** številskega sistema

$$V = \sum_{i=-m}^{n-1} b_i r^i$$

  - $215{,}36_{7} = 2\times7^{2} + 1\times7^{1} + 5\times7^{0} + 3\times7^{-1} + 6\times7^{-2}$
- V računalnikih se uporablja baza $r = 2$
  - nekdaj se je tudi baza $r = 10$
    - BCD-kodiranje

Dvojiški zapis števil

- Dvojiški (binarni) zapis: baza $r = 2$
  - $b_{n-1} \ldots b_2 b_1 b_0 , b_{-1} b_{-2} \ldots b_{-m}$ $\quad b_i = 0$ ali 1

Vrednost: $$V(b) = \sum_{i=-m}^{n-1} b_i 2^i$$

- Primer: pretvori $110101{,}101_2$ v desetiško število.

  $110101{,}101_2 =$

  $1*2^5 + 1*2^4 + 0*2^3 + 1*2^2 + 0*2^1 + 1*2^0 + 1*2^{-1} + 0*2^{-2} + 1*2^{-3} = 53{,}625_{10}$

Pretvorba desetiških števil v bazo *r*

- Algoritem:
  1. $N : r = Q_1 + b_0$
  2. Ponavljaj 1. za $Q_i : r = Q_{i+1} + b_i$ za i = 1, 2, 3, ...
  3. Končaj, ko $Q_i = 0$

- Primer: pretvorba 98_{10} v bazo r=3
  - $98_{10} = 10122_3$
- Posebno nas zanima pretvorba v bazo r=2 (pretvorba desetiškega števila v dvojiško)
  - $27_{10} = 11011_2$

Pretvorba ulomkov v bazo *r*

- Algoritem:
  1. $N * r = b_{-1} + F_1$
  2. Ponavljaj 1. za $F_i * r = b_{-(i+1)} + F_{i+1}$ za $i = 1, 2, \ldots$
  3. Končaj, ko $F_i = 0$

- Primer: pretvorba $0{,}375_{10}$ v bazo $r = 2$
  - $0{,}011_2$

- Kadar število *N* odrežemo na *k* decimalk, dobimo približek *N'*
  - napaka *N'* - *N*, absolutna napaka |*N'* - *N*|
  - Abs. napaka ne more preseči r^{-k}

    $r^{-k} \leq E_{max}$
  - Poiščemo tak *k*, da neenačba velja (običajno lahko tudi brez kalkulatorja)

    $k \geq \log_r (1/E_{max})$

    $k = \lceil \log_r(1/E_{max}) \rceil$

- Če logaritma z bazo *r* ne znamo izračunati, ga pretvorimo v bazo *e* ali 10:

  $\log_a c = \log_a b * \log_b c$ (pravilo)

  (na ta način se znebimo baze *b*, v našem primeru *r*,
  za a pa vzamemo kako znano bazo)

  $\log_e c = \log_e r * \log_r c$

  $\log_r c = \ln c / \ln r$

  $k = \lceil \ln(1/E_{max}) / \ln r \rceil$

- Primer: pretvorba $N = 0{,}8_{10}$ v bazo $r = 3$. Vzemi toliko decimalk, da napaka ne preseže $E_{max} = 0{,}01$.

  $0{,}8_{10} = 0,\ 2101\ 2101\ \ldots_{3}$

  Če upoštevamo *k* decimalk, napaka ne preseže r^{-k}

  $r^{-k} <= E_{max}$

  Brez kalkulatorja lahko ocenimo primeren *k*:

  $3^{-5} = 1/243 = 0{,}004\ldots$, $3^{-4} = 1/81 = 0{,}012\ldots$

  S kalkulatorjem:

  $k = \lceil \ln(100) / \ln(3) \rceil = \lceil 4{,}19 \rceil = 5$

  $0{,}8_{10} = 0{,}21012_3$

- Pri $r = 2$ imamo kar dvojiški logaritem (lb)

  $k = \lceil \log_2(1/E\text{max}) \rceil$

- Primer: $0{,}8_{10}$ v bazo 2, $E_{max} = 0{,}01$

  $0{,}8 = 0, 1100\ 1100\ \ldots_2$

  $k = 7$: $0{,}8 = 0{,}1100110_2$ (N' = 0,796875, E = -0,003125)

- Primer: $N = 159{,}3_{10}$ v bazo $r = 16$. $|N'-N| \leq 10^{-3}$

  9(15),4(12)(12)(12)...16

  $16^{-3} < 10^{-3}$

  $k = 3$

  159,310 = 9(15),4(12)(12)16

Pretvorba med poljubnima bazama

- Pretvorba r' v r:
  - r' v 10
  - 10 v r

- Npr. $26{,}5_8$ v r=3
  - $211{,}12\ 12\ \ldots_3$

Osmiška in šestnajstiška baza

- Poleg dvojiške se v računalništvu pogosto uporabljata tudi **osmiška** (oktalna) in še posebno **šestnajstiška** (heksadecimalna) baza
  - v 16-iški bazi so poleg 0 .. 9 še dodatne cifre:
    - A (10), B (11), C (12), D (13), E (14), F (15)
  - Primer:
    - $3C7_{16} = 3*16^2 + 12*16^1 + 7*16^0 = 768 + 192 + 7 = 967_{10}$
  - Različni načini zapisa:
    - $3C7_{16} = 3C7_H$ = 0x3C7 = $3C7

- Ker sta ti bazi sorodni bazi 2, je pretvorba enostavna
  - Pri osmiški bazi ena cifra predstavlja 3 bite (dvojiške baze)
    - $1110010101_2 = 1\ 110\ 010\ 101_2 = 1625_8$,
    - $327_8 = 011\ 010\ 111_2$
  - Pri šestnajstiški bazi ena cifra predstavlja 4 bite (dvojiške baze)
    - $1110010101_2 = 11\ 1001\ 0101_2 = 395_{16}$ oz. 0x395
    - $A15_{16} = 1010\ 0001\ 0101_2$

- Z *n* biti lahko zapišemo nepredznačena števila od 0 do 2^n-1 (z *n* biti lahko v kateremkoli formatu zapišemo 2^n števil!)
  - npr. *n* = 3, števila od 0 (000) do 7 (111)
  - npr. *n* = 10, števila od 0 (000...) do 1023 (111...)
- Kadar rezultat neke operacije preseže obseg števil, se pojavi **prenos** (**carry**)
  - rezultat na podanem številu cifer ni pravilen

101 + 100 = (1)001

Zapisi predznačenih števil

- Predznačeno število lahko zapišemo na več načinov
- V vseh primerih imamo *n*-bitno število: $b_{n-1} \ldots b_2 b_1 b_0$, njegova vrednost pa se v različnih načinih zapisa razlikuje
- Primer: Zapisi 3-bitnih predznačenih števil

b_2	b_1	b_0	PV	PO	1'K	2'K
0	0	0	+0	-4	+0	0
0	0	1	1	-3	1	1
0	1	0	2	-2	2	2
0	1	1	3	-1	3	3
1	0	0	-0	0	-3	-4
1	0	1	-1	1	-2	-3
1	1	0	-2	2	-1	-2
1	1	1	-3	3	-0	-1

Predznak-veličinski zapis

1. **Predznak-veličinski zapis**

$$V(b) = (-1)^{b_{n-1}} \sum_{i=0}^{n-2} b_i 2^i$$

- prvi bit (b_{n-1}) predstavlja predznak, ostali velikost
- Hibe:
  - predznak je treba obravnavati posebej
  - ima dve ničli: -0 in +0
- PV zapis ni primeren za seštevanje/odštevanje
- Primeren za množenje/deljenje (ki pa sta manj pogosti operaciji)

Zapis z odmikom

2. **Zapis z odmikom**

$$V(b) = \sum_{i=0}^{n-1} b_i 2^i - 2^{n-1}$$

- odmik je (običajno) 2^{n-1}
- nekoč priljubljen zapis
- Hibe:
  - pri seštevanju je treba odmik odšteti
  - pri odštevanju je treba odmik prišteti
    - v oboje se lahko hitro prepričamo

Eniški komplement

3. Eniški komplement (1'K)

$$V(b) = \sum_{i=0}^{n-1} b_i 2^i - b_{n-1}(2^n - 1)$$

- b_{n-1} je predznak
- pozitivna števila (b_{n-1}=0) enako kot pri PV
- negativno število dobimo iz pozitivnega z invertiranjem vseh bitov
  - ekvivalentno odštevanju od $2^n - 1$ (same enice)
- predznaka ni treba obravnavati posebej! ☺
- hibe: ☹
  - 2 ničli (-0, +0)
  - pri prenosu z najvišjega mesta je treba na najnižjem mestu prišteti 1 (End Around Carry - EAC)

Dvojiški komplement

4. Dvojiški komplement (2'K)

$$V(b) = \sum_{i=0}^{n-1} b_i 2^i - b_{n-1} 2^n$$

- Tudi tu se pozitivna števila začnejo z 0:
  - 0000 (0), 0001 (1), ..., 0110 (6), 0111 (7)=max
- Negativna števila se začnejo z 1:
  - 1000 (-8), 1001 (-7), ..., 1110 (-2), 1111 (-1)
  - ni pa takoj razvidno, za katero število gre ☹

- Negativno število dobimo tako, da invertiramo vse bite pozitivnega števila (eniški komplement) in prištejemo 1 (to je ekvivalentno odštevanju od 2^n)
  - npr.

    0010 (2)
    1101 (-2 v 1'K)
    + 1
    1110 (-2 v 2'K)

- Velja pa tudi obratno: če želimo ugotoviti, za katero negativno število gre, spet naredimo 2'K (1'K in prištevanje enice)
  - 10110 = ?, 1'K: 01001 + 1 = 01010, kar je 10, torej je 10110 enako -10 (minus deset)

- Razlikovanje med pojmoma *zapis v 2'K* in *2'K nekega števila*

- Bit prenosa pri 2'K ignoriramo!

```
  011
 +110
 ----
(1)001
```

- Pri razširitvi števila na več bitov je potrebno **razširiti predznak**:
  - **0**101 v **000**101
  - **1**100 v **111**100
  - 01011111 v 0000000001011111
  - 11001100 v 1111111111001100

- 2'K je najpogosteje uporabljan zapis
  - primeren za seštevanje/odštevanje
    - nima EAC
  - le ena predstavitev za ničlo
  - predznaka ni treba obravnavati posebej

$$a-c = a+(-c) = a+2^n-c = a-c\mathbf{+2^n}$$

```
   011
  +110          110=1000-001
  ----
(1)001
```


Primer

- Zapiši -37 kot predznačeno 10-bitno število v PV, PO, 1'K in 2'K
  - PV: 1000100101
  - PO: 0111011011
  - 1'K: 1111011010
  - 2'K: 1111011011

Preliv

- Obseg števil v *n*-bitnem 2'K:

  $-2^{n-1} \leq x \leq 2^{n-1} - 1$

- Če je (pravi) rezultat operacije izven tega območja: **preliv** (**overflow**)
  - rezultat je napačen
  - preliv se da detektirati

- Preliv ni isto kot **prenos** (**carry**) z najvišjega mesta!
  - le-ta se nanaša na operacije z *nepredznačenim*i števili
    - območje $0 \leq x \leq 2^n - 1$
  - pri 2'K se prenos ignorira

- Kdaj pride do preliva?
  - potreben pogoj je, da imata števili enak predznak
  - zadosten pogoj pa je, da ima vsota drugačen predznak kot števili

- Pogoj za preliv (OF) bi lahko zapisali kot

$$OF = x_{n-1}\, y_{n-1}\, \overline{s_{n-1}} \vee \overline{x_{n-1}}\ \ \overline{y_{n-1}}\, s_{n-1}$$

vendar je možno tudi enostavneje (kot bomo videli)

- Primeri operacij v 4-bitnem 2'K:

```
  0100 (4)     0101 (5)     1100 (-4)    1010 (-6)
+ 0011 (3)   + 0100 (4)   + 0101 (5)   + 1011 (-5)
  0111 (7)     1000 (-8)  1 0001 (1)   1 0101 (5)
```


Primeri

- Seštej 21 in -7 v 6-bitnem 2'K

```
  010101
+ 111001
(1)001110
```

Primeri aritmetičnih operacij v različnih bazah

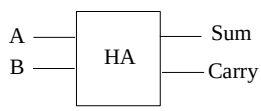
- $02345_9 + 16250_9 = 18605_9$
- $21202_3 + 12012_3 = (1)10221_3$, pojavi se prenos
- $11001_2 + 01011_2 = (1)00100_2$, pojavi se prenos

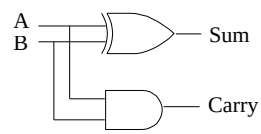
- $4102_5 - 2430_5 = 1122_5$
- $3306_7 - 0615_7 = 2361_7$
- $10110_2 - 01101_2 = 01001_2$

- $324_5 * 023_5 = 014112_5$
- $1101_2 * 0101_2 = 01000001_2$

Vezja za aritmetiko

- **1-bitni seštevalnik**
  - **Polovični seštevalnik** (**Half Adder, HA**)
    - sešteva 2 bita, izračuna vsoto (*s*, sum) in (izhodni) prenos (*c*, carry)

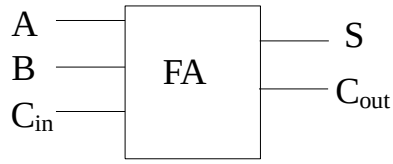




x	*y*	*c*	*s*
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

s = xy' V x'y = x ⊕ y
c = xy (= x&y)

- **Polni seštevalnik** (**Full Adder, FA**)
  - sešteva 3 bite, izračuna vsoto in (izhodni) prenos



$s = x \oplus y \oplus z$ $(= x'y'z \vee x'yz' \vee xy'z' \vee xyz)$
$c = xy \vee xz \vee yz$

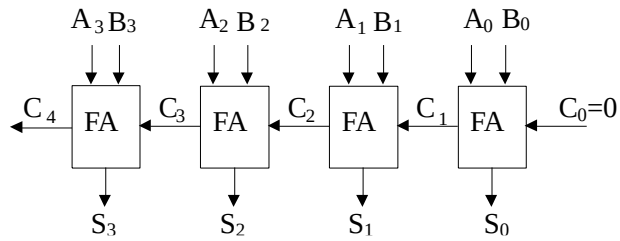
Večbitni seštevalnik

Večbitni seštevalnik

- **Seštevalnik z razširjanjem prenosa** (Ripple Carry Adder, RCA)
  - zaporedna vezava 1-bitnih FA
  - izhodni prenos nižjega vezan na enega od vhodov višjega
    - običajno se en vhod imenuje kar vhodni prenos (c_{in})

$$s = x \oplus y \oplus c_{in}$$
$$c_{out} = xy + xc_{in} + yc_{in}$$

  - hiba: zakasnitev
    - V najslabšem primeru se prenos razširja čez vse FA
    - Dejanska zakasnitev je odvisna od operandov
    - Maksimalna zakasnitev pa narašča praktično linearno



- **Seštevalnik z vnaprejšnjim prenosom** (Carry-Lookahead Adder, CLA)
  - hiter izračun vseh prenosov
    - le na osnovi vhodov x, y in c_0
  - dodatna logika
    - sprememba večnivojske oblike v dvonivojsko

- Seštevanje in odštevanje predznačenih števil v 2'K z enim vezjem
  - signal M (Add'/Sub) določa operacijo
    - 0: +
    - 1: -
  - odštevanje kot prištevanje 2'K
    - X – Y = X + Y' + 1
      - -Y kot dvojiški komplement Y
      - Y' = (y_{n-1}' ... y_1' y_0') ... 1'K
      - $y_i \oplus M$
      - XOR dela kot krmiljen negator (a ⊕ 0 = a, a ⊕ 1 = a')
      - +1: M vežemo na c_0

- Detekcija preliva
  - enak predznak operandov
  - drugačen predznak vsote

(glej prejšnjo formulo za OF)

  - pri prvem produktu je c_{n-1}=0 in c_n=0, pri drugem obratno, zato

$$OF = c_{n-1} \oplus c_n$$

Binarno množenje

- Binarno množenje
  - tvorba delnih (parcialnih) produktov (n*n konjunkcij)
  - seštevanje delnih produktov

$$
\begin{array}{cccccccc}
x_2 & & x_1 & & x_0 & \times & y_2 & y_1 & y_0 \\
\hline
x_2y_2 & & x_1y_2 & & x_0y_2 & & & & \\
 & x_2y_1 & & x_1y_1 & & x_0y_1 & & & \\
 & & x_2y_0 & & x_1y_0 & & x_0y_0 & & \\
\hline
\end{array}
$$

  - Delni produkt je enak množencu, če je ustrezni bit množitelja enak 1, sicer je enak 0

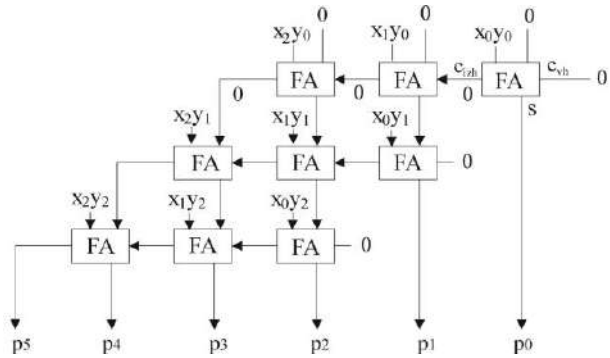
- 2 vrsti metod:
  - pomikanje in seštevanje
    - 1 bit / cikel ure
    - poceni, a ne prav hitro
    - registri
  - kombinacijski množilniki
    - brez ure
    - dragi, a hitri

- Množenje s pomiki in seštevanjem
  - Postopek iz n korakov:
    - Če je najnižji bit množitelja B enak 1, prištej množenec A registru P (na začetku 0)
      - sicer prištej 0
    - Pomik desno registrov P in B (kaskadno vezanih)

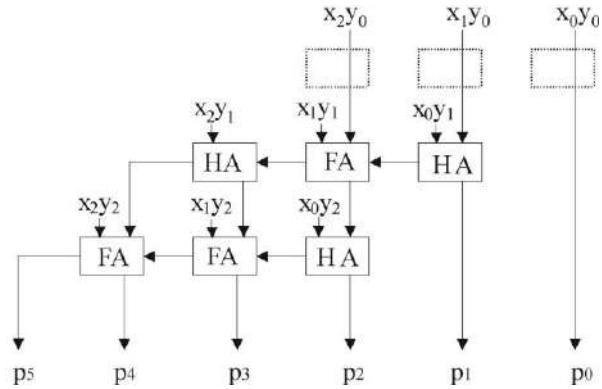
- Primer: A=5, B=6

	P	B	
0	0000	0110	začetek
1	0000	0110	P ← P + 0
	0000	0011	P,B >> 1
2	0101	0011	P ← P + A
	0010	1001	P,B >> 1
3	0111	1001	P ← P + A
	0011	1100	P,B >> 1
4	0011	1100	P ← P + 0
	0001	1110	P,B >> 1

- Matrični množilnik
  - na primeru 3x3



- Nekateri FA so odveč



- Zakasnitev ~ linearna
  - $(3n-2)\Delta FA$
  - $(3n-4)\Delta FA$
- Obstajajo tudi metode za hitro seštevanje več sumandov, t.i. paralelni števniki (parallel counters)
  - Wallace, Dadda, ...
  - glavna aplikacija je množenje

Množenje v 2'K

- Booth-ov algoritem

Binarno deljenje

- 2 osnovna načina:
  - zaporedje odštevanj in pomikov
  - matrični delilnik
    - enobitni odštevalniki

Problemi pri vključitvi aritmetike v računalniški sistem

- Preliv
  - 2 rešitvi:
    - postavitev posebnega bita
    - sprožitev pasti (nek bit lahko določa, ali se sproži, ali pa se ignorira)
- Dolžina produkta
  - produkt dveh števil je shranjen v spremenljivki enake velikosti kot števili
- Izvajanje operacij v eni urini periodi
  - množenje in deljenje sta zahtevnejši operaciji
  - 2 rešitvi:
    - ukazi korak-množenja
    - množenje izvaja posebna enota
      - lahko FPU (floating point unit)
      - CPU čaka na izračun

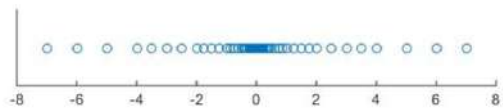
Zapis števil v plavajoči vejici

- Obseg števil v fiksni vejici je za določene probleme premajhen
  - potrebovali bi tudi zelo velika ali zelo majhna števila
- Znanstvena notacija omogoča krajši zapis
  - npr. 1×10^{18} namesto 1 000 000 000 000 000 000
- Število lahko zapišemo kot $m \times r^e$
  - m je **mantisa**, r je **baza** (običajno 2), e je **eksponent**
  - s spreminjanjem eksponenta vejica plava vzdolž mantise levo in desno (odtod ime plavajoča vejica)

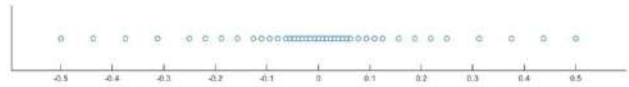
- V plavajoči vejici lahko zapišemo bistveno večja, pa tudi bistveno manjša števila kot v fiksni
  - kljub temu pa je možnih števil enako mnogo (2^n)

- Primer: plavajoča vejica v mini (6-bitnem) formatu
  - predznak: 1bit, mantisa: 3 biti, eksponent: 3 biti
  - $(-1)^S * m * 2^{E-7}$,
    - max: $111 * 2^0 = 7$
    - min abs.: $0 * 2^{-3} = 0$
      - $1 * 2^{-7} = 0{,}0078$, $2 * 2^{-7} = 0{,}016$, ...
    - min: $-111 * 2^0 = -7$

celoten obseg števil:



del obsega:



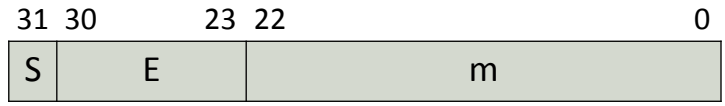
- Vsako število lahko v plavajoči vejici zapišemo na več načinov:
  - npr. $1\times10^{18} = 10\times10^{17} = 0{,}1\times10^{19}$ ...
  - npr. $1\times2^{3} = 10\times2^{2} = 0{,}1\times2^{4}$ ...
  - zato mantiso normiramo:
    - prvi bit je 1 (normalni bit), implicitno predstavljen
    - npr.: mantisa 01001... pomeni 1,01001...
  - zelo majhnih števil pa ni mogoče predstaviti v normirani obliki
    - denormirana števila
    - **podliv** (underflow)
- Eksponent je predstavljen v **predstavitvi z odmikom**

- Nekdaj je vsak proizvajalec je uporabljal svoj format zapisa v plavajoči vejici
  - isti program je lahko na različnih računalnikih dajal različne rezultate

- Standard IEEE 754 (1985)
  - IEEE: Institute of Electrical and Electronics Engineers
  - 2 formata:
    - enojna natančnost (single precision), 32 bitov
    - dvojna natančnost (double precision), 64 bitov

Enojna natančnost

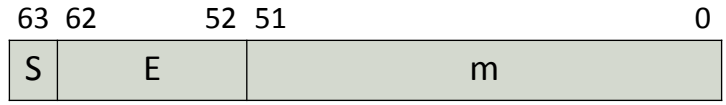
- Enojna natančnost (single precision), 32 bitov



- predznak *S* (0: +, 1: –)
- 8-biten eksponent *e* z odmikom 127 (*e* = *E* - 127)
- 23-bitna mantisa *m* (7-mestna desetiška natančnost)
- normirana vrednost je $\mathbf{(-1)^S \cdot 1{,}m \cdot 2^{E-127}}$, *E* = 1, 2, ..., 254
- obseg: $\pm$ 1,18*10^{-38}, $\pm$ 3,40*10^{38} (v norm. obliki)

Dvojna natančnost

- Dvojna natančnost (double precision), 64 bitov



- predznak *S* (0: +, 1: –)
- 11-biten eksponent *e* z odmikom 1023 (*e* = *E* - 1023)
- 52-bitna mantisa *m* (16-mestna desetiška natančnost)
- normirana vrednost je $\mathbf{(-1)^S \cdot 1{,}m \cdot 2^{E-1023}}$, *E* = 1, 2, ..., 2046
- obseg: $\pm$2,22*10^{-308}, $\pm$ 1,80*10^{308} (v norm. obliki)

- Primer: število 2
  - $2 = +1.0*2^1$
  - S = 0, m = 0, e = 1
  - enojna: E = e + 127 = 128 = 10000000

31	30 23	22 0
0	10000000	00000000000000000000000

  - dvojna: E = e + 1023 = 1024 = 10000000000

63	62 52	51 0
0	10000000000	00000000000000000000000 00000

- Primer: število -8.25
  - $-8.25 = -1000.01 = -1.00001*2^3$
  - S = 1, m = 0000100 ..., e = 3
  - enojna: e = 3, E = e + 127 = 130 = 10000010

31	30 23	22 0
1	10000010	00001000000000000000000

  - dvojna: e = 3, E = e + 1023 = 1026 = 10000000010

63	62 52	51 0
1	10000000010	00001000000000000000000 00000

Denormirana števila

- **Denormirana števila** (zelo majhna števila)
  - E=0
  - implicitni normalni bit je enak 0
  - vrednost v 32-bitnem formatu je $(-1)^S \cdot 0,m \cdot 2^{-126}$
    - eksponent je -126 namesto -127, ker imamo (0,m) namesto (1,m)
  - vrednost v 64-bitnem formatu je $(-1)^S \cdot 0,m \cdot 2^{-1022}$,
    - eksponent je -1022 namesto -1023, ker imamo (0,m) namesto (1,m)
  - tudi 0 je denormirano število, ki ima mantiso enako 0

Neskončnosti in NaN

- Še dve posebni vrsti števil:
  - **Neskončnosti**
    - E = 255 (v 32-bitnem formatu) oz. E = 2047 (v 64-bitnem formatu), vsi biti E so 1
    - če m=0, imamo $+\infty$ in $-\infty$
    - pojavijo se, kadar je rezultat prevelik (npr. 1/0 da $+\infty$)
  - **NaN**
    - ravno tako E = 255 oz. 2047
    - $m \neq 0$
    - pojavijo se kot rezultat nedefiniranih operacij
      - npr. $0 \times \infty$, $0/0$, $\infty - \infty$, kvadratni koren negativnega števila, ...
    - rezultat operacije, ki vsebuje operand NaN, je tudi NaN

Aritmetika v plavajoči vejici

- Aritmetika v plavajoči vejici se obravnava in realizira ločeno od aritmetike v fiksni vejici
  - bolj zapletena
- Zaokroževanje
  - zaokrožujemo od matematično natančne vrednosti k najbližjemu še predstavljivemu številu
  - kadar je vrednost enako oddaljena od dveh najbližjih števil, se po standardu IEEE 754 zaokroži k sodemu številu
  - pri računanju mantiso podaljšamo za 3 dodatne bite
    - varovalni bit (guard bit)
    - zaokroževalni bit (round bit)
    - lepljivi bit (sticky bit)

- **Varovalni bit** je potreben, ker je vsota lahko za eno mesto daljša od operandov
- **Zaokroževalni bit** omogoča bolj natančno zaokroževanje
- Primer: desetiška predstavitev z mantiso dolžine 3
  - dodamo varovalno in zaokroževalno mesto
  - pri seštevanju/odštevanju po pravilu število z manjšim eksponentom zapišemo z večjim eksponentom (mantisa se pomakne desno)
  - $1{,}01*10^4 - 3{,}76*10^2 = (1{,}0100 - 0{,}0376)*10^4 = 0{,}9724*10^4 =$ zaokr. $9{,}72*10^3$
  - če bi uporabili le 4-mestno mantiso, bi dobili napačno $9{,}73*10^3$

- **Lepljivi bit** se uporablja zaradi zaokroževanja k sodemu številu
- Primer: (brez lepljivega bita)
  - $4{,}56*10^0 + 5{,}01*10^{-3} = (4{,}5600 + 0{,}0050)*10^0 = 4{,}5650*10^0 =$ zaokr. $4{,}56*10^0$
  - natančna vrednost bi bila 4,56501 (zato bi bil bolj pravilen rezultat 4,57), vendar zaradi pomika mantise v desno zadnja enica izpade
  - lepljivi bit pove, ali je desno od zaokroževalnega mesta še kako od nič različno mesto
    - v tem primeru je treba zaokrožiti navzgor (ne navzdol zaradi morebitnega najbližjega sodega števila)
    - izračuna se kot funkcija ALI izpadlih bitov

Seštevanje v plavajoči vejici

- Seštevanje (in odštevanje) v plavajoči vejici
  - prvo število naj bo tisto z večjim eksponentom (začasni eksponent)
  - pomik mantise drugega števila
  - seštevanje (odštevanje) mantis
  - Če preliv, zmanjšaj mantiso (pomik) in povečaj začasni eksponent
  - Zaokrožitev mantise, bita *r* in *s* ...

- Primer 1. Seštej binarno 3,25 + 30, če je mantisa 3-bitna, imamo pa tudi bite g, r in s. Določi njihovo vrednost.
  - $11{,}01*2^0 + 11110{,}0*2^0 = 1{,}101000*2^1 + 1{,}111000*2^4 = 1{,}111000*2^4 + 0{,}001101*2^4 = 10{,}000101*2^4 = 1{,}000|0101*2^5 = 1{,}000|011*2^5 = 1{,}000*2^5$
  - grs=011, vsota je 32
- Primer 2. Odštej binarno 30 - 4,5, če je mantisa 3-bitna, imamo pa tudi bite g, r in s. Določi njihovo vrednost.
  - $11110{,}0*2^0 - 100{,}1*2^0 = 1{,}111000*2^4 - 1{,}001000*2^2 = 1{,}111000*2^4 - 0{,}01001000*2^4 = 1{,}100|110*2^4 = 1{,}101*2^4$
  - grs=110, vsota je 26

Množenje v plavajoči vejici

- Množenje v plavajoči vejici
  - eksponenta seštejemo (dobimo začasni eksponent)
  - mantisi zmnožimo z množilnikom (fixed-point)
  - po potrebi normiramo rezultat
  - predznak produkta je XOR obeh predznakov
- Primer 1: A*B, $A = 1{,}01*2^2$, $B = 1{,}11*2^0$,
  - začasni eksponent = 2 + 0 = 2
  - množimo mantisi 1,01*1,11 = 10,0011
  - $10{,}0011*2^2$, normiramo: $1{,}00011*2^3$
  - predznak: $0 \oplus 0 = 0$, tj. +

- Primer2: C = A*B v enojni natančnosti

```
A = 0x326C8000
B = 0xBF200000
0x326C8000 = 1,11011001*2^-27
0xBF200000 = -1,01*2^-1

Zmnožimo mantisi:
  1,11011001*1,01
  --------------
  1,11011001
   ,0111011001
  --------------
 10,0100111101
-10,0100111101*2^-28 = -1,00100111101 *2^-27

C = B213D00016
```

- Deljenje v plavajoči vejici
  - odštevanje eksponentov, deljenje mantis

Sklad in delo s podprogrami

Patricio Bulić, Damjan Šonc, Andrej Štrancar

Univerza v Ljubljani, FRI

ARS

Vsebina

Sklad

- Sklad: podatkovna struktura (linearni seznam), organizirana v smislu LIFO.
- Za delo s skladom potrebujemo **skladovni kazalec** (SP) : kazalec, ki kaže na vrh sklada.
- Za delo s skladom uporabljamo dve operaciji: PUSH in POP.
- Več možnih načinov implementacije sklada in operacij PUSH/POP.
- Sklad uporabljamo predvsem za shranjevanje začasnih spremenljivk (npr. lokalnih v podprogramih) za prenos parametrov v podprograme, za shranjevanje registrov v podprogramih in za shranjevanje povratnega naslova pri klicu podprogramov.

Sklad

- Sklad: podatkovna struktura (linearni seznam), organizirana v smislu LIFO.
- Za delo s skladom potrebujemo **skladovni kazalec** (SP) : kazalec, ki kaže na vrh sklada.
- Za delo s skladom uporabljamo dve operaciji: PUSH in POP.
- Več možnih načinov implementacije sklada in operacij PUSH/POP.
- Sklad uporabljamo predvsem za shranjevanje začasnih spremenljivk (npr. lokalnih v podprogramih) za prenos parametrov v podprograme, za shranjevanje registrov v podprogramih in za shranjevanje povratnega naslova pri klicu podprogramov.

Sklad

- Sklad: podatkovna struktura (linearni seznam), organizirana v smislu LIFO.
- Za delo s skladom potrebujemo **skladovni kazalec** (SP) : kazalec, ki kaže na vrh sklada.
- Za delo s skladom uporabljamo dve operaciji: PUSH in POP.
- Več možnih načinov implementacije sklada in operacij PUSH/POP.
- Sklad uporabljamo predvsem za shranjevanje začasnih spremenljivk (npr. lokalnih v podprogramih) za prenos parametrov v podprograme, za shranjevanje registrov v podprogramih in za shranjevanje povratnega naslova pri klicu podprogramov.

Sklad

- Sklad: podatkovna struktura (linearni seznam), organizirana v smislu LIFO.
- Za delo s skladom potrebujemo **skladovni kazalec** (SP) : kazalec, ki kaže na vrh sklada.
- Za delo s skladom uporabljamo dve operaciji: PUSH in POP.
- Več možnih načinov implementacije sklada in operacij PUSH/POP.
- Sklad uporabljamo predvsem za shranjevanje začasnih spremenljivk (npr. lokalnih v podprogramih) za prenos parametrov v podprograme, za shranjevanje registrov v podprogramih in za shranjevanje povratnega naslova pri klicu podprogramov.

Sklad

- Sklad: podatkovna struktura (linearni seznam), organizirana v smislu LIFO.
- Za delo s skladom potrebujemo **skladovni kazalec** (SP) : kazalec, ki kaže na vrh sklada.
- Za delo s skladom uporabljamo dve operaciji: PUSH in POP.
- Več možnih načinov implementacije sklada in operacij PUSH/POP.
- Sklad uporabljamo predvsem za shranjevanje začasnih spremenljivk (npr. lokalnih v podprogramih) za prenos parametrov v podprograme, za shranjevanje registrov v podprogramih in za shranjevanje povratnega naslova pri klicu podprogramov.

Sklad : arhitekturna podpora

- Kaj mora vsebovati arhitektura nekega procesorja, da bo omogočeno delo s skladom?
- Imeti mora register, ki deluje kot skladovni kazalec (SP): lahko je to namenski register ali eden izmed splošnih.
- Pri procesorju, ki operande hrani v registrih moramo imeti podprti operaciji `PUSH reg` in `POP reg`.
- Operaciji `PUSH reg` / `POP reg` sta lahko implementirani kot dva mikroprocesorska ukaza ali kot zaporedje mikroprocesorskih ukazov.

Sklad : arhitekturna podpora

- Kaj mora vsebovati arhitektura nekega procesorja, da bo omogočeno delo s skladom?
- Imeti mora register, ki deluje kot skladovni kazalec (SP): lahko je to namenski register ali eden izmed splošnih.
- Pri procesorju, ki operande hrani v registrih moramo imeti podprti operaciji `PUSH reg` in `POP reg`.
- Operaciji `PUSH reg` / `POP reg` sta lahko implementirani kot dva mikroprocesorska ukaza ali kot zaporedje mikroprocesorskih ukazov.

Sklad : arhitekturna podpora

- Kaj mora vsebovati arhitektura nekega procesorja, da bo omogočeno delo s skladom?
- Imeti mora register, ki deluje kot skladovni kazalec (SP): lahko je to namenski register ali eden izmed splošnih.
- Pri procesorju, ki operande hrani v registrih moramo imeti podprti operaciji `PUSH reg` in `POP reg`.
- Operaciji `PUSH reg` / `POP reg` sta lahko implementirani kot dva mikroprocesorska ukaza ali kot zaporedje mikroprocesorskih ukazov.

Sklad : arhitekturna podpora

- Kaj mora vsebovati arhitektura nekega procesorja, da bo omogočeno delo s skladom?
- Imeti mora register, ki deluje kot skladovni kazalec (SP): lahko je to namenski register ali eden izmed splošnih.
- Pri procesorju, ki operande hrani v registrih moramo imeti podprti operaciji `PUSH reg` in `POP reg`.
- Operaciji `PUSH reg` / `POP reg` sta lahko implementirani kot dva mikroprocesorska ukaza ali kot zaporedje mikroprocesorskih ukazov.

Sklad : različica 1

- Sklad naj narašča v smeri naraščajočih naslovov in SP naj kaže na prvo prosto mesto na skladu.
- Predpostavimo, da registrski operand zaseda n pomnilniških besed.
- Operaciji `PUSH reg` / `POP reg`:

```
PUSH reg :
        M[SP] <- reg; SP <- SP + n;
POP reg :
        SP <- SP - n; reg <- M[SP];
```

Sklad : različica 1

- Sklad naj narašča v smeri naraščajočih naslovov in SP naj kaže na prvo prosto mesto na skladu.
- Predpostavimo, da registrski operand zaseda n pomnilniških besed.
- Operaciji `PUSH reg` / `POP reg`:

  PUSH reg :

  ```
  M[SP] <- reg; SP <- SP + n;
  ```

  POP reg :

  ```
  SP <- SP - n; reg <- M[SP];
  ```

Sklad : različica 1

- Sklad naj narašča v smeri naraščajočih naslovov in SP naj kaže na prvo prosto mesto na skladu.
- Predpostavimo, da registrski operand zaseda n pomnilniških besed.
- Operaciji `PUSH reg` / `POP reg`:

PUSH reg :

```
M[SP] <- reg; SP <- SP + n;
```

POP reg :

```
SP <- SP - n; reg <- M[SP];
```

Sklad : različica 1

- Sklad naj narašča v smeri naraščajočih naslovov in SP naj kaže na prvo prosto mesto na skladu.
- Predpostavimo, da registrski operand zaseda n pomnilniških besed.
- Operaciji `PUSH reg` / `POP reg`:

  PUSH reg :

  ```
  M[SP] <- reg; SP <- SP + n;
  ```

  POP reg :

  ```
  SP <- SP - n; reg <- M[SP];
  ```

Sklad : različica 1

- Sklad naj narašča v smeri naraščajočih naslovov in SP naj kaže na prvo prosto mesto na skladu.
- Predpostavimo, da registrski operand zaseda n pomnilniških besed.
- Operaciji `PUSH reg` / `POP reg`:

  PUSH reg :

  ```
  M[SP] <- reg; SP <- SP + n;
  ```

  POP reg :

  ```
  SP <- SP - n; reg <- M[SP];
  ```

Sklad : različica 2

- Sklad naj narašča v smeri naraščajočih naslovov in SP naj kaže na zadnji podatek na skladu.
- Predpostavimo, da registrski operand zaseda n pomnilniških besed.
- Operaciji `PUSH reg` / `POP reg`:

  PUSH reg :

  ```
  SP <- SP + n; M[SP] <- reg;
  ```

  POP reg :

  ```
  reg <- M[SP]; SP <- SP - n;
  ```

Sklad : različica 2

- Sklad naj narašča v smeri naraščajočih naslovov in SP naj kaže na zadnji podatek na skladu.
- Predpostavimo, da registrski operand zaseda n pomnilniških besed.
- Operaciji `PUSH reg` / `POP reg`:

  PUSH reg :

  ```
  SP <- SP + n; M[SP] <- reg;
  ```

  POP reg :

  ```
  reg <- M[SP]; SP <- SP - n;
  ```

Sklad : različica 2

- Sklad naj narašča v smeri naraščajočih naslovov in SP naj kaže na zadnji podatek na skladu.
- Predpostavimo, da registrski operand zaseda n pomnilniških besed.
- Operaciji `PUSH reg` / `POP reg`:

PUSH reg :

```
SP <- SP + n; M[SP] <- reg;
```

POP reg :

```
reg <- M[SP]; SP <- SP - n;
```

Sklad : različica 2

- Sklad naj narašča v smeri naraščajočih naslovov in SP naj kaže na zadnji podatek na skladu.
- Predpostavimo, da registrski operand zaseda n pomnilniških besed.
- Operaciji `PUSH reg` / `POP reg`:

  PUSH reg :

  ```
  SP <- SP + n; M[SP] <- reg;
  ```

  POP reg :

  ```
  reg <- M[SP]; SP <- SP - n;
  ```

Sklad : različica 2

- Sklad naj narašča v smeri naraščajočih naslovov in SP naj kaže na zadnji podatek na skladu.
- Predpostavimo, da registrski operand zaseda n pomnilniških besed.
- Operaciji `PUSH reg` / `POP reg`:

  PUSH reg :

  ```
  SP <- SP + n; M[SP] <- reg;
  ```

  POP reg :

  ```
  reg <- M[SP]; SP <- SP - n;
  ```

Sklad : različica 3

- Sklad naj narašča v smeri padajočih naslovov in SP naj kaže na prvo prosto mesto na skladu.
- Predpostavimo, da registrski operand zaseda n pomnilniških besed.
- Operaciji `PUSH reg` / `POP reg`:

  PUSH reg :

  ```
  M[SP] <- reg; SP <- SP - n;
  ```

  POP reg :

  ```
  SP <- SP + n; reg <- M[SP];
  ```

Sklad : različica 3

- Sklad naj narašča v smeri padajočih naslovov in SP naj kaže na prvo prosto mesto na skladu.
- Predpostavimo, da registrski operand zaseda n pomnilniških besed.
- Operaciji `PUSH reg` / `POP reg`:

  PUSH reg :

  ```
  M[SP] <- reg; SP <- SP - n;
  ```

  POP reg :

  ```
  SP <- SP + n; reg <- M[SP];
  ```

Sklad : različica 3

- Sklad naj narašča v smeri padajočih naslovov in SP naj kaže na prvo prosto mesto na skladu.
- Predpostavimo, da registrski operand zaseda n pomnilniških besed.
- Operaciji `PUSH reg` / `POP reg`:

PUSH reg :

```
M[SP] <- reg; SP <- SP - n;
```

POP reg :

```
SP <- SP + n; reg <- M[SP];
```

Sklad : različica 3

- Sklad naj narašča v smeri padajočih naslovov in SP naj kaže na prvo prosto mesto na skladu.
- Predpostavimo, da registrski operand zaseda n pomnilniških besed.
- Operaciji `PUSH reg` / `POP reg`:

  PUSH reg :

  ```
  M[SP] <- reg; SP <- SP - n;
  ```

  POP reg :

  ```
  SP <- SP + n; reg <- M[SP];
  ```

Sklad : različica 3

- Sklad naj narašča v smeri padajočih naslovov in SP naj kaže na prvo prosto mesto na skladu.
- Predpostavimo, da registrski operand zaseda n pomnilniških besed.
- Operaciji `PUSH reg` / `POP reg`:

  PUSH reg :

  ```
  M[SP] <- reg; SP <- SP - n;
  ```

  POP reg :

  ```
  SP <- SP + n; reg <- M[SP];
  ```

Sklad : različica 4

- Sklad naj naraščа v smeri padajočih naslovov in SP kaže na zadnji podatek na skladu.
- Predpostavimo, da registrski operand zaseda n pomnilniških besed.
- Operaciji `PUSH reg` / `POP reg`:

  PUSH reg :

  ```
  SP <- SP - n; M[SP] <- reg;
  ```

  POP reg :

  ```
  reg <- M[SP]; SP <- SP + n;
  ```

Sklad : različica 4

- Sklad naj naraščča v smeri padajočih naslovov in SP kaže na zadnji podatek na skladu.
- Predpostavimo, da registrski operand zaseda n pomnilniških besed.
- Operaciji `PUSH reg` / `POP reg`:

  PUSH reg :

  ```
  SP <- SP - n; M[SP] <- reg;
  ```

  POP reg :

  ```
  reg <- M[SP]; SP <- SP + n;
  ```

Sklad : različica 4

- Sklad naj naraščа v smeri padajočih naslovov in SP kaže na zadnji podatek na skladu.
- Predpostavimo, da registrski operand zaseda n pomnilniških besed.
- Operaciji `PUSH reg` / `POP reg`:

PUSH reg :

```
SP <- SP - n; M[SP] <- reg;
```

POP reg :

```
reg <- M[SP]; SP <- SP + n;
```

Sklad : različica 4

- Sklad naj naraščа v smeri padajočih naslovov in SP kaže na zadnji podatek na skladu.
- Predpostavimo, da registrski operand zaseda n pomnilniških besed.
- Operaciji `PUSH reg` / `POP reg`:

  PUSH reg :

  ```
  SP <- SP - n; M[SP] <- reg;
  ```

  POP reg :

  ```
  reg <- M[SP]; SP <- SP + n;
  ```

Sklad : različica 4

- Sklad naj naraščča v smeri padajočih naslovov in SP kaže na zadnji podatek na skladu.
- Predpostavimo, da registrski operand zaseda n pomnilniških besed.
- Operaciji `PUSH reg` / `POP reg`:

  PUSH reg :

  ```
  SP <- SP - n; M[SP] <- reg;
  ```

  POP reg :

  ```
  reg <- M[SP]; SP <- SP + n;
  ```

HIP - sklad

- HIP nima skladovnega kazalca - zato se moramo dogovoriti, kateri izmed splošnonamenskih registrov naj bo skladovni kazalec. **Dogovor**: naj bo to register `R30`.
- Zaradi zaheteve po poravnanosti pomnilniških operandov, dovolimo prenos samo med registri (32-bitna vsebina) in skladom, tj. le operaciji `PUSH reg` / `POP reg`.
- **Dogovor**: **naj sklad narašča v smeri padajočih naslovov in naj skladovni kazalec kaže na prvo prosto mesto na skladu**.
- Sklad naj se začne na dnu podatkovnega segmenta.

HIP - sklad

- HIP nima skladovnega kazalca - zato se moramo dogovoriti, kateri izmed splošnonamenskih registrov naj bo skladovni kazalec. **Dogovor**: naj bo to register `R30`.
- Zaradi zaheteve po poravnanosti pomnilniških operandov, dovolimo prenos samo med registri (32-bitna vsebina) in skladom, tj. le operaciji `PUSH reg` / `POP reg`.
- **Dogovor**: **naj sklad narašča v smeri padajočih naslovov in naj skladovni kazalec kaže na prvo prosto mesto na skladu**.
- Sklad naj se začne na dnu podatkovnega segmenta.

HIP - sklad

- HIP nima skladovnega kazalca - zato se moramo dogovoriti, kateri izmed splošnonamenskih registrov naj bo skladovni kazalec. **Dogovor**: naj bo to register `R30`.
- Zaradi zaheteve po poravnanosti pomnilniških operandov, dovolimo prenos samo med registri (32-bitna vsebina) in skladom, tj. le operaciji `PUSH reg` / `POP reg`.
- **Dogovor**: **naj sklad narašča v smeri padajočih naslovov in naj skladovni kazalec kaže na prvo prosto mesto na skladu**.
- Sklad naj se začne na dnu podatkovnega segmenta.

HIP - sklad

- HIP nima skladovnega kazalca - zato se moramo dogovoriti, kateri izmed splošnonamenskih registrov naj bo skladovni kazalec. **Dogovor**: naj bo to register `R30`.
- Zaradi zaheteve po poravnanosti pomnilniških operandov, dovolimo prenos samo med registri (32-bitna vsebina) in skladom, tj. le operaciji `PUSH reg` / `POP reg`.
- **Dogovor**: **naj sklad narašča v smeri padajočih naslovov in naj skladovni kazalec kaže na prvo prosto mesto na skladu**.
- Sklad naj se začne na dnu podatkovnega segmenta.

HIP - sklad

- Na začetku vsakega programa **moramo nastaviti skladovni kazalec**:

```
addui r30,r0, #0x4fc ;
```

- HIP nima ukazov PUSH/POP, zato se moramo dogovoriti, kako bomo izvedli sklad in operaciji, tj. makro ukaza PUSH in POP. Makro ukaza PUSH in POP naj imata samo en eksplicitni registrski operand. Izvedemo ju kot zaporedje dveh ukazov procesorja HIP:

PUSH reg :

```
sw 0(r30), reg
subui r30,r30,#4
```

POP reg :

```
addui r30,r30,#4
lw reg, 0(r30)
```

HIP - sklad

- Na začetku vsakega programa **moramo nastaviti skladovni kazalec**:

```
addui r30,r0, #0x4fc ;
```

- HIP nima ukazov PUSH/POP, zato se moramo dogovoriti, kako bomo izvedli sklad in operaciji, tj. makro ukaza PUSH in POP. Makro ukaza PUSH in POP naj imata samo en eksplicitni registrski operand. Izvedemo ju kot zaporedje dveh ukazov procesorja HIP:

PUSH reg :

```
sw 0(r30), reg
subui r30,r30,#4
```

POP reg :

```
addui r30,r30,#4
lw reg, 0(r30)
```

HIP - sklad

- Na začetku vsakega programa **moramo nastaviti skladovni kazalec**:

```
addui r30,r0, #0x4fc ;
```

- HIP nima ukazov PUSH/POP, zato se moramo dogovoriti, kako bomo izvedli sklad in operaciji, tj. makro ukaza PUSH in POP. Makro ukaza PUSH in POP naj imata samo en eksplicitni registrski operand. Izvedemo ju kot zaporedje dveh ukazov procesorja HIP:

PUSH reg :

```
sw 0(r30), reg
subui r30,r30,#4
```

POP reg :

```
addui r30,r30,#4
lw reg, 0(r30)
```

HIP - sklad

- Na začetku vsakega programa **moramo nastaviti skladovni kazalec**:

```
addui r30,r0, #0x4fc ;
```

- HIP nima ukazov PUSH/POP, zato se moramo dogovoriti, kako bomo izvedli sklad in operaciji, tj. makro ukaza PUSH in POP. Makro ukaza PUSH in POP naj imata samo en eksplicitni registrski operand. Izvedemo ju kot zaporedje dveh ukazov procesorja HIP:

PUSH reg :

```
sw 0(r30), reg
subui r30,r30,#4
```

POP reg :

```
addui r30,r30,#4
lw reg, 0(r30)
```

Prenos parametrov in klic podprogramov - kako?

- Kje so parametri (v registrih ali na skladu)?
- Kako in kaj morata storiti klicoči program in klicani podprogram?
- Kdo parametre na koncu pobriše oz. odstrani?
- Kako se vrnemo iz podprograma oz. kje je povratni naslov?
- Kako podprogram vrača vrednosti?
- Kako se ohranja vrednosti v registrih, ki jih je ustvaril klicoči program, med izvajanjem podprograma? Kdo mora te vrednosti shraniti in kam ter kdo in kako jih obnovi?

Prenos parametrov in klic podprogramov - kako?

- Kje so parametri (v registrih ali na skladu)?
- Kako in kaj morata storiti klicoči program in klicani podprogram?
- Kdo parametre na koncu pobriše oz. odstrani?
- Kako se vrnemo iz podprograma oz. kje je povratni naslov?
- Kako podprogram vrača vrednosti?
- Kako se ohranja vrednosti v registrih, ki jih je ustvaril klicoči program, med izvajanjem podprograma? Kdo mora te vrednosti shraniti in kam ter kdo in kako jih obnovi?

Prenos parametrov in klic podprogramov - kako?

- Kje so parametri (v registrih ali na skladu)?
- Kako in kaj morata storiti klicoči program in klicani podprogram?
- Kdo parametre na koncu pobriše oz. odstrani?
- Kako se vrnemo iz podprograma oz. kje je povratni naslov?
- Kako podprogram vrača vrednosti?
- Kako se ohranja vrednosti v registrih, ki jih je ustvaril klicoči program, med izvajanjem podprograma? Kdo mora te vrednosti shraniti in kam ter kdo in kako jih obnovi?

Prenos parametrov in klic podprogramov - kako?

- Kje so parametri (v registrih ali na skladu)?
- Kako in kaj morata storiti klicoči program in klicani podprogram?
- Kdo parametre na koncu pobriše oz. odstrani?
- Kako se vrnemo iz podprograma oz. kje je povratni naslov?
- Kako podprogram vrača vrednosti?
- Kako se ohranja vrednosti v registrih, ki jih je ustvaril klicoči program, med izvajanjem podprograma? Kdo mora te vrednosti shraniti in kam ter kdo in kako jih obnovi?

Prenos parametrov in klic podprogramov - kako?

- Kje so parametri (v registrih ali na skladu)?
- Kako in kaj morata storiti klicoči program in klicani podprogram?
- Kdo parametre na koncu pobriše oz. odstrani?
- Kako se vrnemo iz podprograma oz. kje je povratni naslov?
- Kako podprogram vrača vrednosti?
- Kako se ohranja vrednosti v registrih, ki jih je ustvaril klicoči program, med izvajanjem podprograma? Kdo mora te vrednosti shraniti in kam ter kdo in kako jih obnovi?

Prenos parametrov in klic podprogramov - kako?

- Kje so parametri (v registrih ali na skladu)?
- Kako in kaj morata storiti klicoči program in klicani podprogram?
- Kdo parametre na koncu pobriše oz. odstrani?
- Kako se vrnemo iz podprograma oz. kje je povratni naslov?
- Kako podprogram vrača vrednosti?
- Kako se ohranja vrednosti v registrih, ki jih je ustvaril klicoči program, med izvajanjem podprograma? Kdo mora te vrednosti shraniti in kam ter kdo in kako jih obnovi?

Prenos parametrov in klic podprogramov - procesor HIP - kako?

- Za prenos parametrov bomo uporabljali registra R24 in R25 ter sklad.
- Kako bi izvedli klice podprogramov in prenos parametrov preko sklada na procesorju HIP?
- Sprejeli bomo dogovor o klicu podprogramov in prenosu parametrov (*calling convention*) - **tega se moramo vedno držati!**.

Prenos parametrov in klic podprogramov - procesor HIP - kako?

- Za prenos parametrov bomo uporabljali registra R24 in R25 ter sklad.
- Kako bi izvedli klice podprogramov in prenos parametrov preko sklada na procesorju HIP?
- Sprejeli bomo dogovor o klicu podprogramov in prenosu parametrov (*calling convention*) - **tega se moramo vedno držati!**.

Prenos parametrov in klic podprogramov - procesor HIP - kako?

- Za prenos parametrov bomo uporabljali registra R24 in R25 ter sklad.
- Kako bi izvedli klice podprogramov in prenos parametrov preko sklada na procesorju HIP?
- Sprejeli bomo dogovor o klicu podprogramov in prenosu parametrov (*calling convention*) - **tega se moramo vedno držati!**.

HIP - dogovor o klicu podprogramov

- **Dogovor**: parametri se v podprogram prenašajo od desne proti levi glede na podpis podprograma.
- Prva dva parametra gledano z leve proti desni v podpisu funkcije bo klicoči program prenašal izključno preko registrov R24 in R25, ostale pa izključno preko sklada od desne proti levi.

HIP - dogovor o klicu podprogramov

- **Dogovor**: parametri se v podprogram prenašajo od desne proti levi glede na podpis podprograma.
- Prva dva parametra gledano z leve proti desni v podpisu funkcije bo klicoči program prenašal izključno preko registrov R24 in R25, ostale pa izključno preko sklada od desne proti levi.

HIP - dogovor o klicu podprogramov

- Pri HIP-u za klic podprograma uporabimo ukaz:

```
call rp, odmik(rb)
```

  pri čemer se povratni naslov shrani v register `rp`, v programski števec pa se vpiše naslov podprograma, ki je `rb + odmik`.

- **Dogovor**: naj bo `rp` register `R31`.
- **Dogovor**: za kratke klice naj bo `rb` register `R0`:

```
call r31, PODPROG(r0)
```

- **Dogovor**: za dolge klice naj bo `rb` register `R27`:

```
lhi   r27, #PODPROG
addui r27, r27, #PODPROG
call  r31, 0(r27)
```

HIP - dogovor o klicu podprogramov

- Pri HIP-u za klic podprograma uporabimo ukaz:

```
call rp, odmik(rb)
```

  pri čemer se povratni naslov shrani v register `rp`, v programski števec pa se vpiše naslov podprograma, ki je `rb + odmik`.

- **Dogovor**: naj bo `rp` register `R31`.
- **Dogovor**: za kratke klice naj bo `rb` register `R0`:

```
call r31, PODPROG(r0)
```

- **Dogovor**: za dolge klice naj bo `rb` register `R27`:

```
lhi   r27, #PODPROG
addui r27, r27, #PODPROG
call  r31, 0(r27)
```

HIP - dogovor o klicu podprogramov

- Pri HIP-u za klic podprograma uporabimo ukaz:

```
call rp, odmik(rb)
```

  pri čemer se povratni naslov shrani v register `rp`, v programski števec pa se vpiše naslov podprograma, ki je `rb + odmik`.

- **Dogovor**: naj bo `rp` register `R31`.
- **Dogovor**: za kratke klice naj bo `rb` register `R0`:

```
call r31, PODPROG(r0)
```

- **Dogovor**: za dolge klice naj bo `rb` register `R27`:

```
lhi   r27, #PODPROG
addui r27, r27, #PODPROG
call  r31, 0(r27)
```

HIP - dogovor o klicu podprogramov

- Pri HIP-u za klic podprograma uporabimo ukaz:

```
call rp, odmik(rb)
```

  pri čemer se povratni naslov shrani v register `rp`, v programski števec pa se vpiše naslov podprograma, ki je `rb + odmik`.

- **Dogovor**: naj bo `rp` register `R31`.
- **Dogovor**: za kratke klice naj bo `rb` register `R0`:

```
call r31, PODPROG(r0)
```

- **Dogovor**: za dolge klice naj bo `rb` register `R27`:

```
lhi   r27, #PODPROG
addui r27, r27, #PODPROG
call  r31, 0(r27)
```

HIP - dogovor o klicu podprogramov

- Splošen zapis ukaza za brezpogojni skok je:

```
j odmik(rb)
```

- **Dogovor**: Ker je po dogovoru povratni naslov shranjen v registru R31, naj bo `rb` za vrnitev iz podprograma register `R31`:

```
j 0(r31)
```

- **Dogovor**: za dolge skoke naj bo `rb` register `R26`.

```
lhi   r26, #SKOK
addui r26, r26, #SKOK
j     0(r26)
```

HIP - dogovor o klicu podprogramov

- Splošen zapis ukaza za brezpogojni skok je:

```
j odmik(rb)
```

- **Dogovor**: Ker je po dogovoru povratni naslov shranjen v registru R31, naj bo `rb` za vrnitev iz podprograma register `R31`:

```
j 0(r31)
```

- **Dogovor**: za dolge skoke naj bo `rb` register `R26`.

```
lhi   r26, #SKOK
addui r26, r26, #SKOK
j     0(r26)
```

HIP - dogovor o klicu podprogramov

- Splošen zapis ukaza za brezpogojni skok je:

```
j odmik(rb)
```

- **Dogovor**: Ker je po dogovoru povratni naslov shranjen v registru R31, naj bo `rb` za vrnitev iz podprograma register `R31`:

```
j 0(r31)
```

- **Dogovor**: za dolge skoke naj bo `rb` register `R26`.

```
lhi   r26, #SKOK
addui r26, r26, #SKOK
j     0(r26)
```

HIP - dogovor o klicu podprogramov

Klicoči program pred klicem podprograma:

1. prva dva parametra (od leve proti desni) shrani v registra `R24` in `R25`
2. na sklad porine preostale parametre od desne proti levi,
3. izvede klic podprograma z ukazom `call`.

Klicoči program po vrnitvi iz podprograma:

1. s sklada pobriše parametre : skladovnemu kazalcu (`R30`) prišteje štirikratnik števila prenešenih parametrov na sklad.

HIP - dogovor o klicu podprogramov

- Do parametrov na skladu znotraj klicanega podprograma dostopamo preko enega registra, ki mu pravimo **kazalec na okvir** (*frame pointer*). Ta se med izvajanjem podprograma ne spreminja! Tako vedno vemo, kje so parametri na skladu.
  Dogovor: naj bo kazalec na okvir v registru R29.
- Lokalne spremenljivke (le avtomatske) se hranijo izključno na skladu.
- Iz podprograma vračamo eno samo 32-bitno vrednost v registru.
  Dogovor: naj bo to register R28. Če je parameter, ki ga vračamo zapisan z (do) 32 biti, potem ga vračamo po vrednosti, sicer po referenci!

HIP - dogovor o klicu podprogramov

- Do parametrov na skladu znotraj klicanega podprograma dostopamo preko enega registra, ki mu pravimo **kazalec na okvir** (*frame pointer*). Ta se med izvajanjem podprograma ne spreminja! Tako vedno vemo, kje so parametri na skladu.
  Dogovor: naj bo kazalec na okvir v registru R29.
- Lokalne spremenljivke (le avtomatske) se hranijo izključno na skladu.
- Iz podprograma vračamo eno samo 32-bitno vrednost v registru.
  Dogovor: naj bo to register R28. Če je parameter, ki ga vračamo zapisan z (do) 32 biti, potem ga vračamo po vrednosti, sicer po referenci!

HIP - dogovor o klicu podprogramov

- Do parametrov na skladu znotraj klicanega podprograma dostopamo preko enega registra, ki mu pravimo **kazalec na okvir** (*frame pointer*). Ta se med izvajanjem podprograma ne spreminja! Tako vedno vemo, kje so parametri na skladu.
  Dogovor: naj bo kazalec na okvir v registru R29.
- Lokalne spremenljivke (le avtomatske) se hranijo izključno na skladu.
- Iz podprograma vračamo eno samo 32-bitno vrednost v registru.
  Dogovor: naj bo to register R28. Če je parameter, ki ga vračamo zapisan z (do) 32 biti, potem ga vračamo po vrednosti, sicer po referenci!

HIP - dogovor o klicu podprogramov

Klicani podprogram ob vstopu:

1. na sklad porine povratni naslov - register R31;
2. na sklad porine kazalec na okvir - register R29;
3. v register R29 (kazalec na okvir) prepiše skladovni kazalec;
4. po potrebi rezervira prostor na skladu za lokalne spremenljivke (spreminja skladovni kazalec!);
5. na sklad **shrani vse registre, ki jih spreminja**;
6. do prenešenih parametrov in lokalnih spremenljivk dostopamo preko kazalca na okvir (register R29): zadnji parameter na skladu je na naslovu R29+12, prva lokalna spremenljivka je na naslovu R29.

HIP - dogovor o klicu podprogramov

Klicani podprogram ob vstopu:

1. na sklad porine povratni naslov - register R31;
2. na sklad porine kazalec na okvir - register R29;
3. v register R29 (kazalec na okvir) prepiše skladovni kazalec;
4. po potrebi rezervira prostor na skladu za lokalne spremenljivke (spreminja skladovni kazalec!);
5. na sklad **shrani vse registre, ki jih spreminja**;
6. do prenešenih parametrov in lokalnih spremenljivk dostopamo preko kazalca na okvir (register R29): zadnji parameter na skladu je na naslovu R29+12, prva lokalna spremenljivka je na naslovu R29.

HIP - dogovor o klicu podprogramov

Klicani podprogram ob vstopu:

1. na sklad porine povratni naslov - register R31;
2. na sklad porine kazalec na okvir - register R29;
3. v register R29 (kazalec na okvir) prepiše skladovni kazalec;
4. po potrebi rezervira prostor na skladu za lokalne spremenljivke (spreminja skladovni kazalec!);
5. na sklad **shrani vse registre, ki jih spreminja**;
6. do prenešenih parametrov in lokalnih spremenljivk dostopamo preko kazalca na okvir (register R29): zadnji parameter na skladu je na naslovu R29+12, prva lokalna spremenljivka je na naslovu R29.

HIP - dogovor o klicu podprogramov

Klicani podprogram ob vstopu:

1. na sklad porine povratni naslov - register R31;
2. na sklad porine kazalec na okvir - register R29;
3. v register R29 (kazalec na okvir) prepiše skladovni kazalec;
4. po potrebi rezervira prostor na skladu za lokalne spremenljivke (spreminja skladovni kazalec!);
5. na sklad **shrani vse registre, ki jih spreminja**;
6. do prenešenih parametrov in lokalnih spremenljivk dostopamo preko kazalca na okvir (register R29): zadnji parameter na skladu je na naslovu R29+12, prva lokalna spremenljivka je na naslovu R29.

HIP - dogovor o klicu podprogramov

Klicani podprogram ob vstopu:

1. na sklad porine povratni naslov - register R31;
2. na sklad porine kazalec na okvir - register R29;
3. v register R29 (kazalec na okvir) prepiše skladovni kazalec;
4. po potrebi rezervira prostor na skladu za lokalne spremenljivke (spreminja skladovni kazalec!);
5. na sklad **shrani vse registre, ki jih spreminja**;
6. do prenešenih parametrov in lokalnih spremenljivk dostopamo preko kazalca na okvir (register R29): zadnji parameter na skladu je na naslovu R29+12, prva lokalna spremenljivka je na naslovu R29.

HIP - dogovor o klicu podprogramov

Klicani podprogram ob vstopu:

1. na sklad porine povratni naslov - register R31;
2. na sklad porine kazalec na okvir - register R29;
3. v register R29 (kazalec na okvir) prepiše skladovni kazalec;
4. po potrebi rezervira prostor na skladu za lokalne spremenljivke (spreminja skladovni kazalec!);
5. na sklad **shrani vse registre, ki jih spreminja**;
6. do prenešenih parametrov in lokalnih spremenljivk dostopamo preko kazalca na okvir (register R29): zadnji parameter na skladu je na naslovu R29+12, prva lokalna spremenljivka je na naslovu R29.

HIP - dogovor o klicu podprogramov

Klicani podprogram pred izstopom:

1. v register `R28` zapiše vrednost, ki jo vrača;
2. s sklada obnovi vse shranjene registre;
3. s sklada 'odstrani' lokalne spremenljivke tako, da register `R29` prepiše v skladovni kazalec `R30`;
4. s sklada obnovi (prebere) staro vrednost registra `R29`;
5. s sklada obnovi (prebere) povratni naslov v register `R31`;
6. z ukazom `jmp` se vrne na naslov `R31+0`.

HIP - dogovor o klicu podprogramov

Klicani podprogram pred izstopom:

1. v register `R28` zapiše vrednost, ki jo vrača;
2. s sklada obnovi vse shranjene registre;
3. s sklada 'odstrani' lokalne spremenljivke tako, da register `R29` prepiše v skladovni kazalec `R30`;
4. s sklada obnovi (prebere) staro vrednost registra `R29`;
5. s sklada obnovi (prebere) povratni naslov v register `R31`;
6. z ukazom `jmp` se vrne na naslov `R31+0`.

HIP - dogovor o klicu podprogramov

Klicani podprogram pred izstopom:

1. v register `R28` zapiše vrednost, ki jo vrača;
2. s sklada obnovi vse shranjene registre;
3. s sklada 'odstrani' lokalne spremenljivke tako, da register `R29` prepiše v skladovni kazalec `R30`;
4. s sklada obnovi (prebere) staro vrednost registra `R29`;
5. s sklada obnovi (prebere) povratni naslov v register `R31`;
6. z ukazom `jmp` se vrne na naslov `R31+0`.

HIP - dogovor o klicu podprogramov

Klicani podprogram pred izstopom:

1. v register `R28` zapiše vrednost, ki jo vrača;
2. s sklada obnovi vse shranjene registre;
3. s sklada 'odstrani' lokalne spremenljivke tako, da register `R29` prepiše v skladovni kazalec `R30`;
4. s sklada obnovi (prebere) staro vrednost registra `R29`;
5. s sklada obnovi (prebere) povratni naslov v register `R31`;
6. z ukazom `jmp` se vrne na naslov `R31+0`.

HIP - dogovor o klicu podprogramov

Klicani podprogram pred izstopom:

1. v register `R28` zapiše vrednost, ki jo vrača;
2. s sklada obnovi vse shranjene registre;
3. s sklada 'odstrani' lokalne spremenljivke tako, da register `R29` prepiše v skladovni kazalec `R30`;
4. s sklada obnovi (prebere) staro vrednost registra `R29`;
5. s sklada obnovi (prebere) povratni naslov v register `R31`;
6. z ukazom `jmp` se vrne na naslov `R31+0`.

HIP - dogovor o klicu podprogramov

Klicani podprogram pred izstopom:

1. v register `R28` zapiše vrednost, ki jo vrača;
2. s sklada obnovi vse shranjene registre;
3. s sklada 'odstrani' lokalne spremenljivke tako, da register `R29` prepiše v skladovni kazalec `R30`;
4. s sklada obnovi (prebere) staro vrednost registra `R29`;
5. s sklada obnovi (prebere) povratni naslov v register `R31`;
6. z ukazom `jmp` se vrne na naslov `R31+0`.

HIP klic podprogramov - zgled

Predpostavimo, da želimo klicati naslednji podprogram:

```
int sestej(int a, int b) {
    int SUM;
    SUM = a + b;

    return SUM;
}
```

HIP klic podprogramov - zgled

- klicoči program :

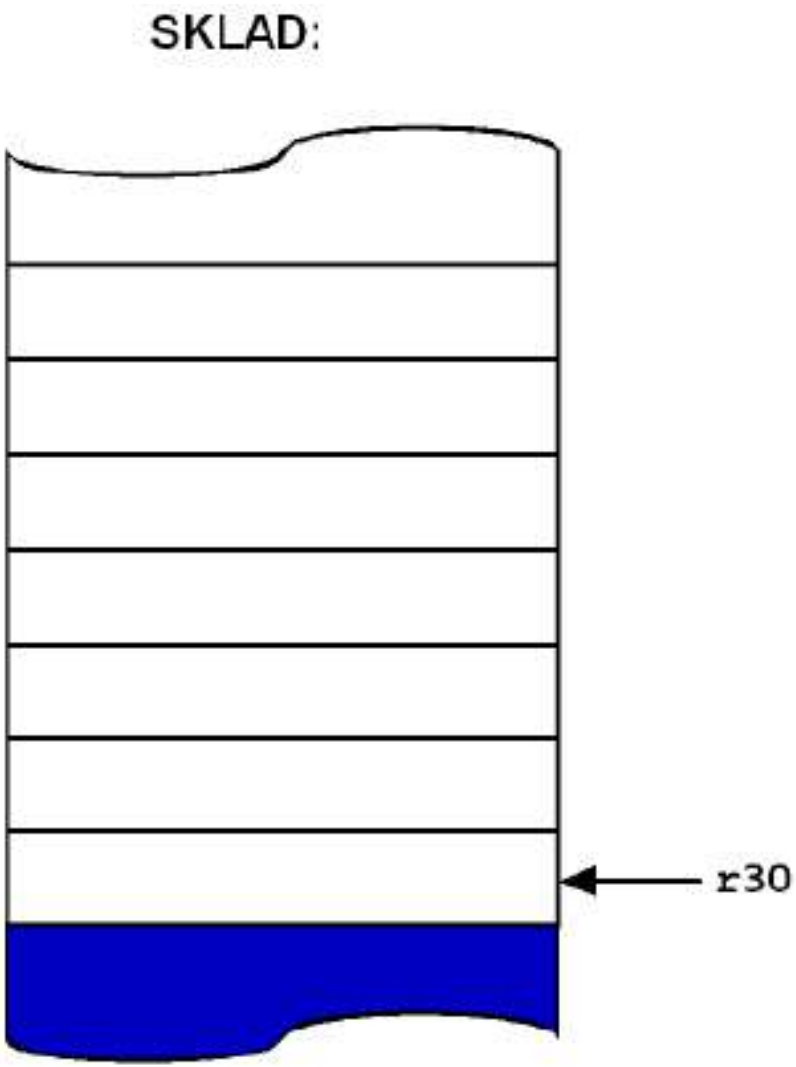
```
addui r30, r0, #0x4fc  ; nalozi skladovni kazalec
...
lw r24, a(r0)          ; prvi parameter prenasamo preko r24
lw r3, b(r0)           ; drugi parameter (tokrat za potrebe zgleda)
push r3                ; prenasamo preko sklada; sicer preko registra
call r31, sestej(r0)   ; klici podprogram in shrani povratni naslov v r31
                       ; Pozor: v nasem primeru se mora podprogram nahajati v prvih
                       ; 32K pomnilnika. Zakaj?
                       ; v splosnem moramo namesto r0 uporabiti drugi bazni
                       ; register, s cimer omogocimo postavitev podprograma
                       ; na poljuben (poravnani) naslov v pomnilniku
addui r30,r30,#4       ; pocisti parameter s sklada
...
```

HIP klic podprogramov - zgled

- klicani podprogram :

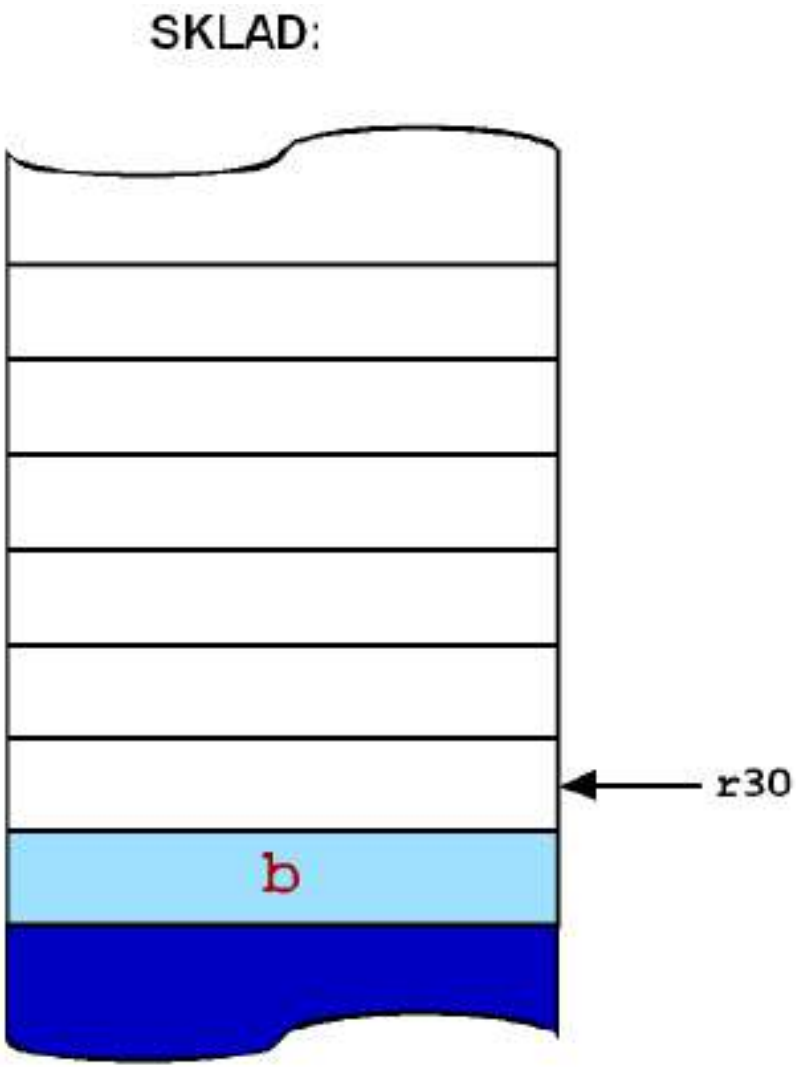
```
; VSTOPNA TOCKA: -----------------------------------
push r31          ; shrani povratni naslov
push r29          ; shrani r29
add r29,r0,r30    ; R29 <- SP : nastavi kazalec na okvir;
                  ;  ...sedaj lahko spreminjamo SP
;-------------------------------------------------

; PROCEDURA: ---------------------------------------
subui r30,r30,#4 ; rezerviraj na skladu prostor za
                  ; 32-bitno spremenljivko SUM
push r6           ; shrani register, ki se
                  ; v podprogramu spreminja
lw r6,12(r29)     ; r6 <- b
add r6,r6,r24     ; r6 = a + b
sw 0(r29), r6     ; SUM <- r6
lw r28, 0(r29)    ; vrednosti vracamo v r28!
pop r6            ; pred izstopom obnovimo r6
;-------------------------------------------------

; IZSTOPNA TOCKA: ----------------------------------
add r30,r0,r29    ; pobrisemo lokalne spremenljivke
                  ; s sklada
pop r29           ; obnovi r29
pop 31            ; povratni naslov v r31
j 0(r31)          ; povratek v klicoci program
```

Sklad - HIP - zgled.



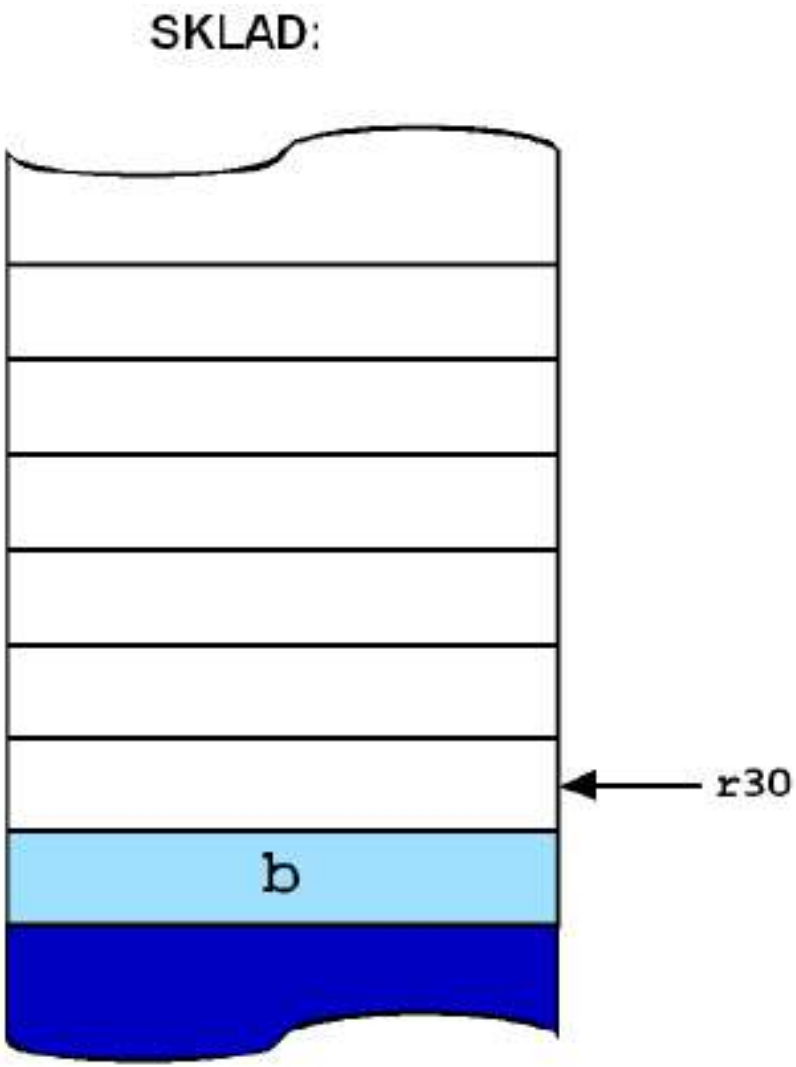

```
; KLICOCI PROGRAM:
lw r24, a(r0)
lw r3, b(r0)
push r3
call r31, sestej(r0)
addui r30,r30,#4
```

Sklad - HIP - zgled.



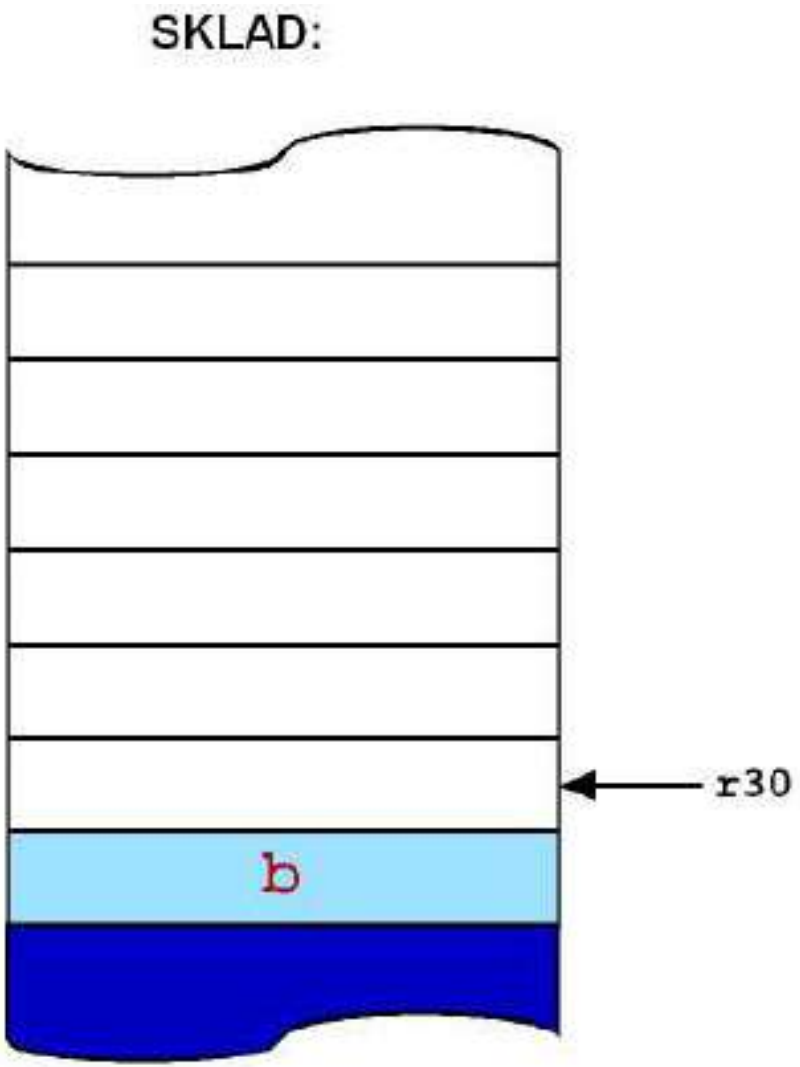

```
; KLICOCI PROGRAM:
lw r24, a(r0)
lw r3, b(r0)
push r3
call r31, sestej(r0)
addui r30,r30,#4
```

Sklad - HIP - zgled.



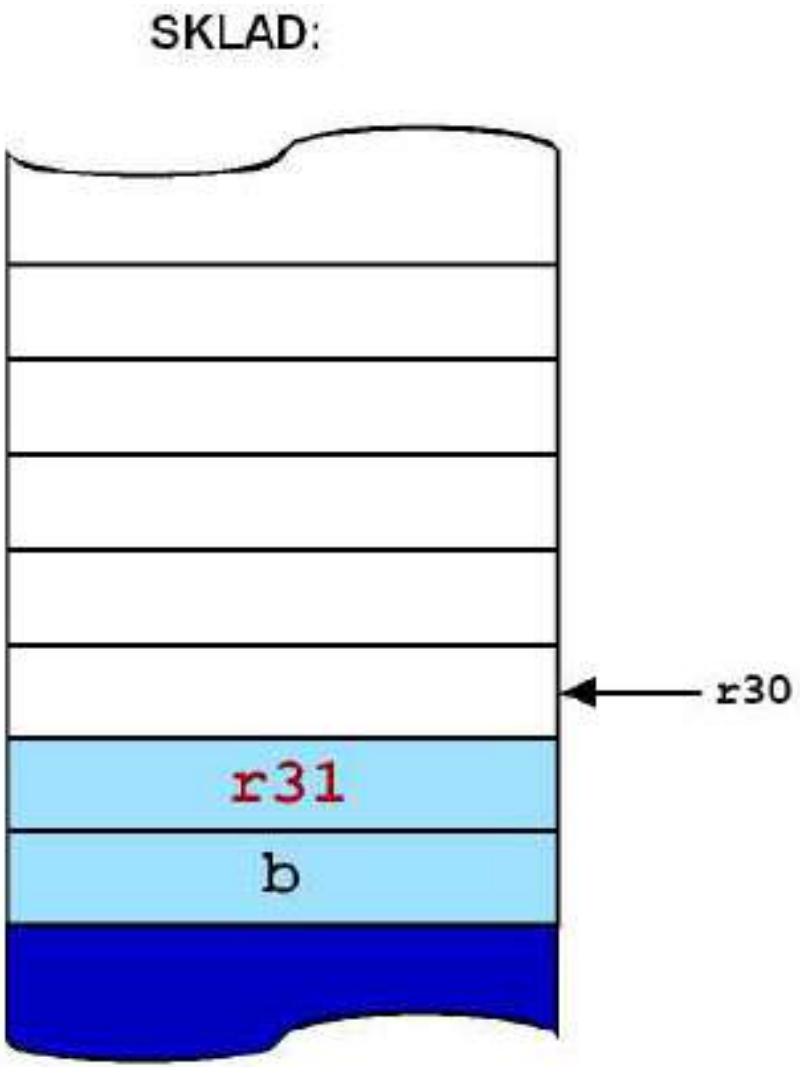

```
; KLICOCI PROGRAM:
lw r24, a(r0)
lw r3, b(r0)
push r3
call r31, sestej(r0)
addui r30,r30,#4
```

Sklad - HIP - zgled.



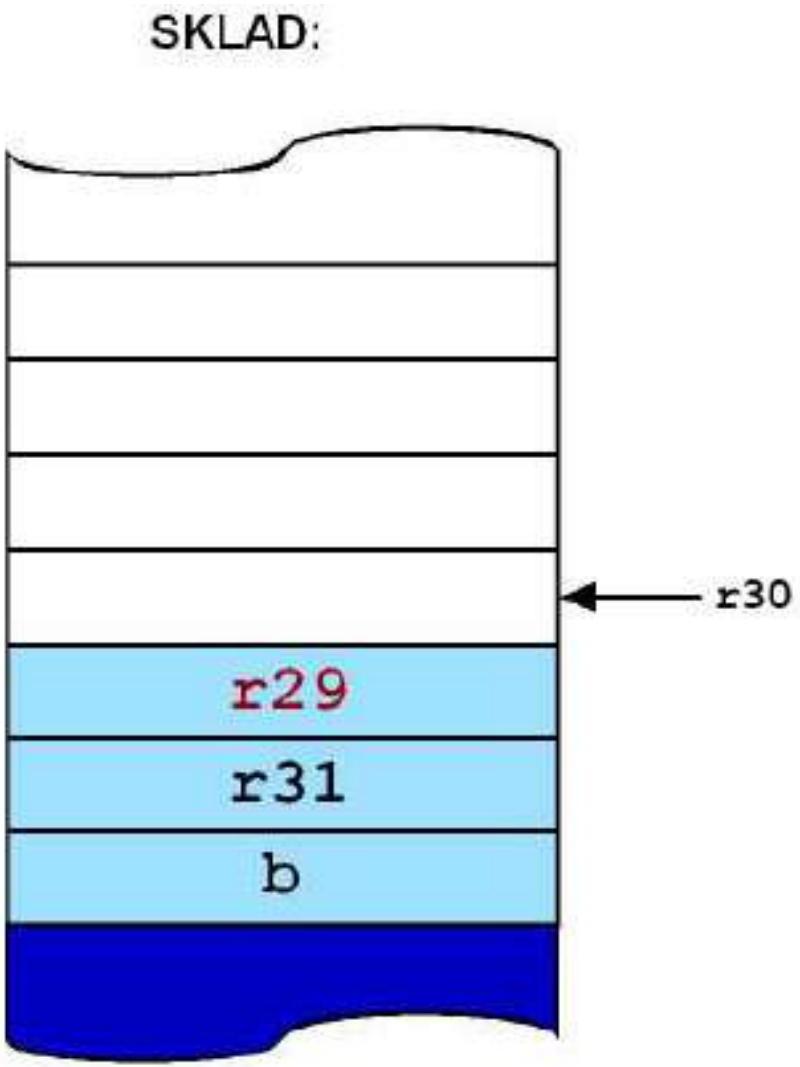

```
; PODPROGRAM:
; VSTOPNA TOCKA:
push r31
push r29
add r29,r0,r30

; PROCEDURA:
subui r30,r30,#4
push r6
lw r6,12(r29)
add r6,r6,r24
sw 0(r29), r6
lw r28, 0(r29)
pop r6

; IZSTOPNA TOCKA:
add r30,r0,r29
pop r29
pop 31
j 0(r31)
```

Sklad - HIP - zgled.



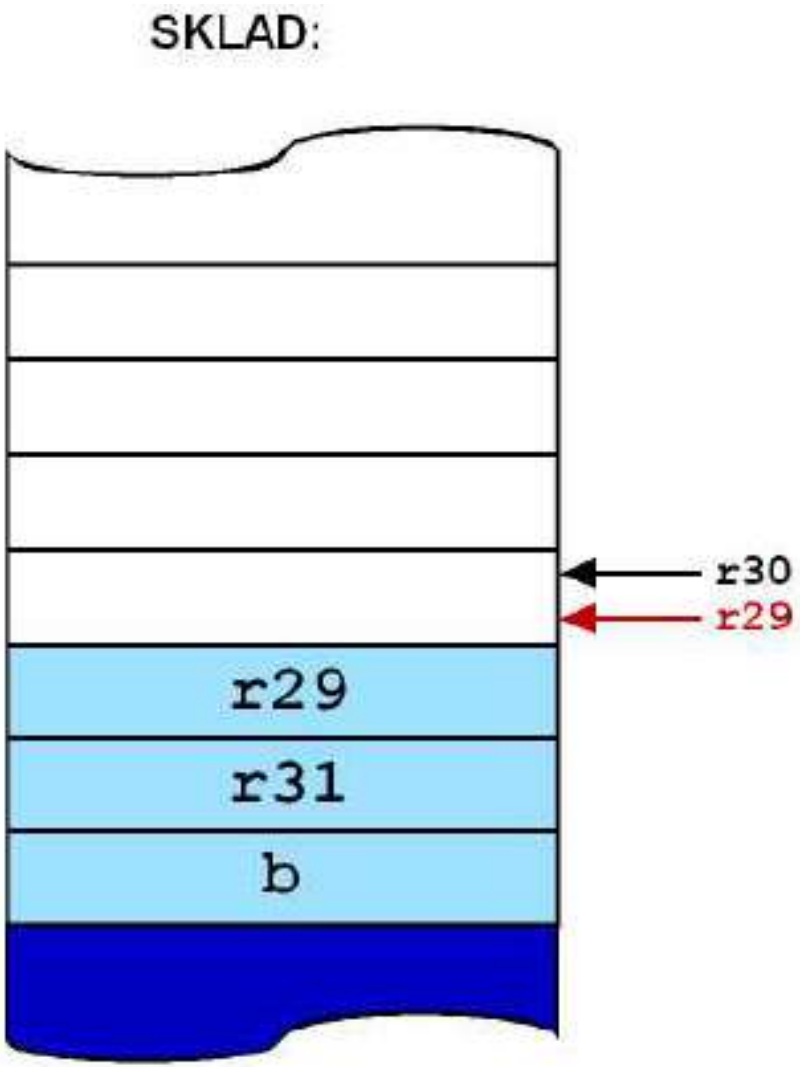

```
; PODPROGRAM:
; VSTOPNA TOCKA:
push r31
push r29
add r29,r0,r30

; PROCEDURA:
subui r30,r30,#4
push r6
lw r6,12(r29)
add r6,r6,r24
sw 0(r29), r6
lw r28, 0(r29)
pop r6

; IZSTOPNA TOCKA:
add r30,r0,r29
pop r29
pop 31
j 0(r31)
```

Sklad - HIP - zgled.



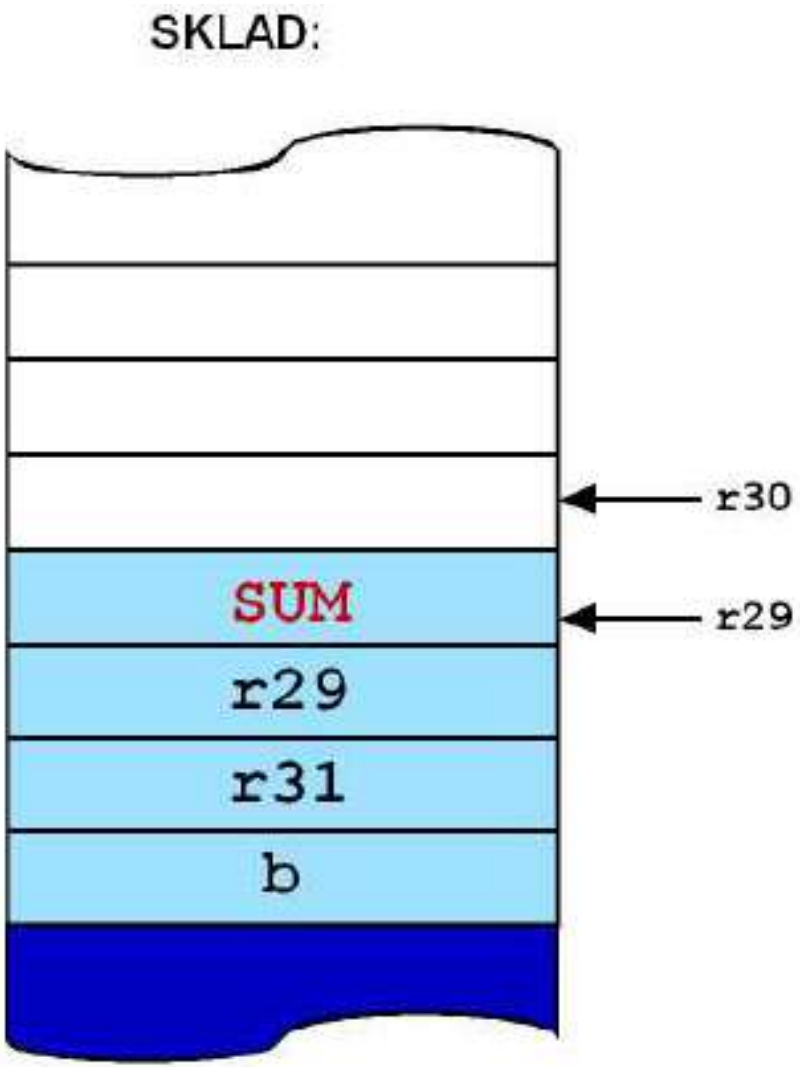

```
; PODPROGRAM:
; VSTOPNA TOCKA:
push r31
push r29
add r29,r0,r30

; PROCEDURA:
subui r30,r30,#4
push r6
lw r6,12(r29)
add r6,r6,r24
sw 0(r29), r6
lw r28, 0(r29)
pop r6

; IZSTOPNA TOCKA:
add r30,r0,r29
pop r29
pop 31
j 0(r31)
```

Sklad - HIP - zgled.



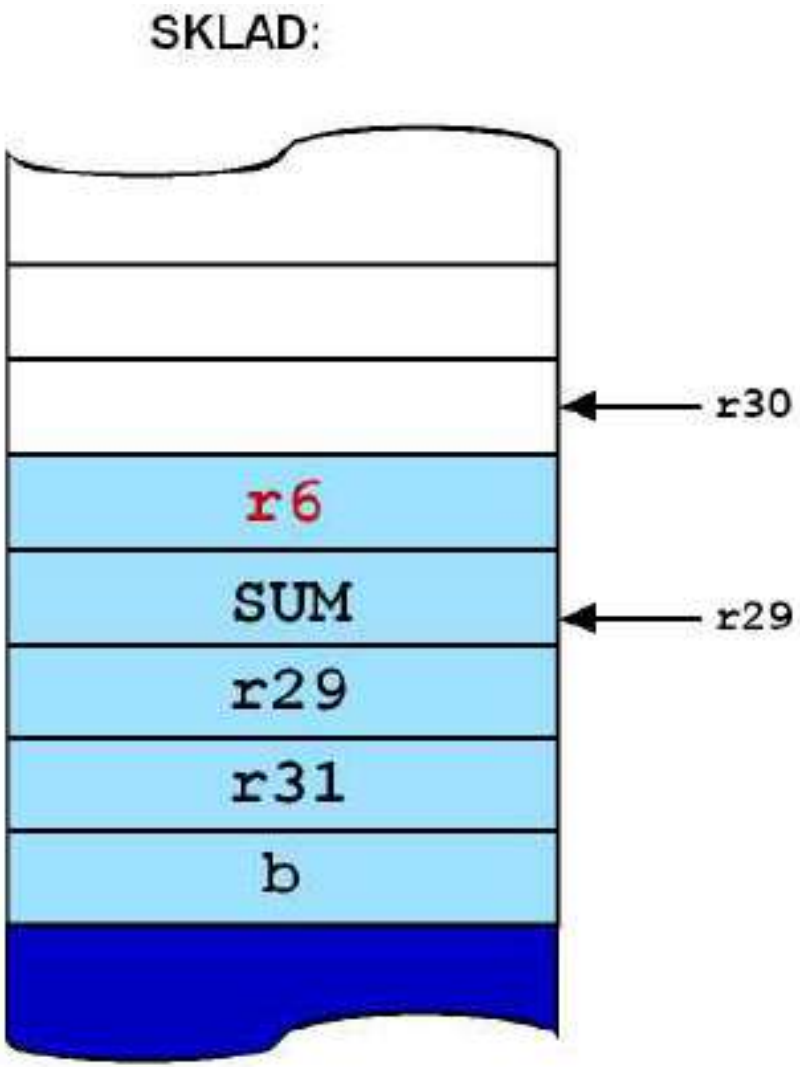

```
; PODPROGRAM:
; VSTOPNA TOCKA:
push r31
push r29
add r29,r0,r30

; PROCEDURA:
subui r30,r30,#4
push r6
lw r6,12(r29)
add r6,r6,r24
sw 0(r29), r6
lw r28, 0(r29)
pop r6

; IZSTOPNA TOCKA:
add r30,r0,r29
pop r29
pop 31
j 0(r31)
```

Sklad - HIP - zgled.



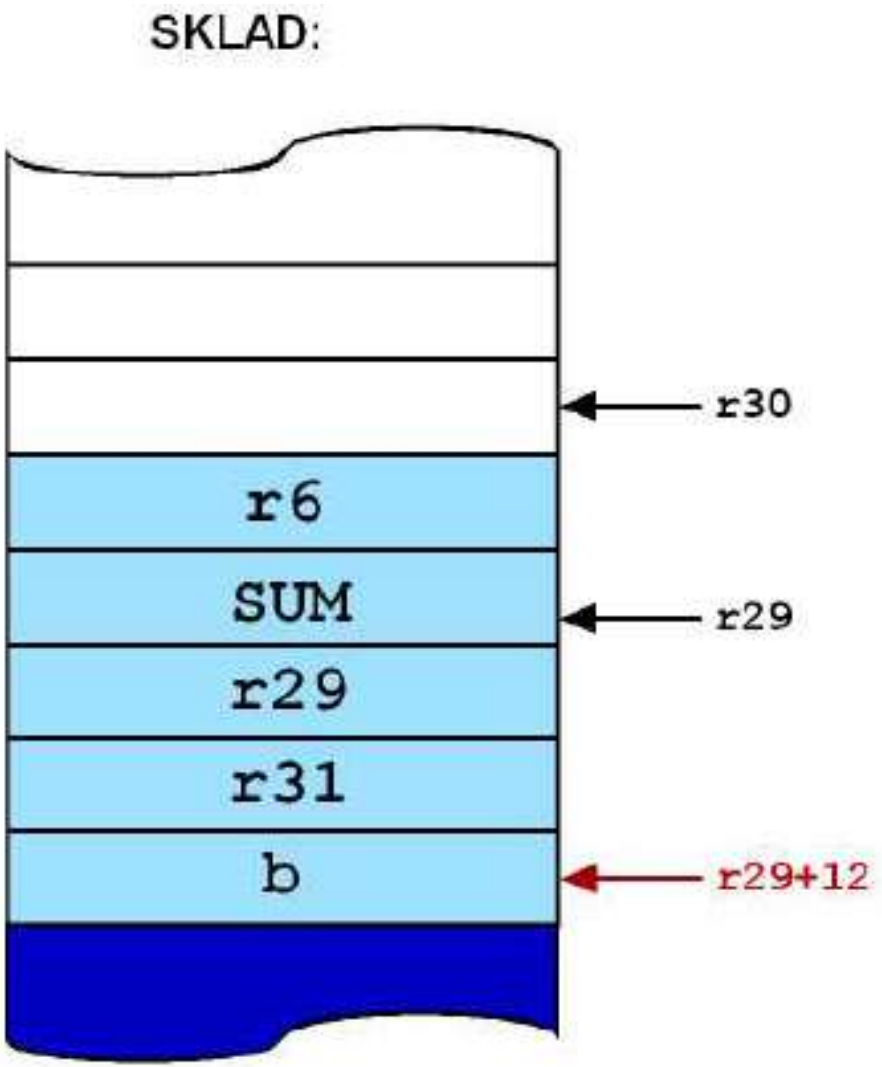

```
; PODPROGRAM:
; VSTOPNA TOCKA:
push r31
push r29
add r29,r0,r30

; PROCEDURA:
subui r30,r30,#4
push r6
lw r6,12(r29)
add r6,r6,r24
sw 0(r29), r6
lw r28, 0(r29)
pop r6

; IZSTOPNA TOCKA:
add r30,r0,r29
pop r29
pop 31
j 0(r31)
```

Sklad - HIP - zgled.



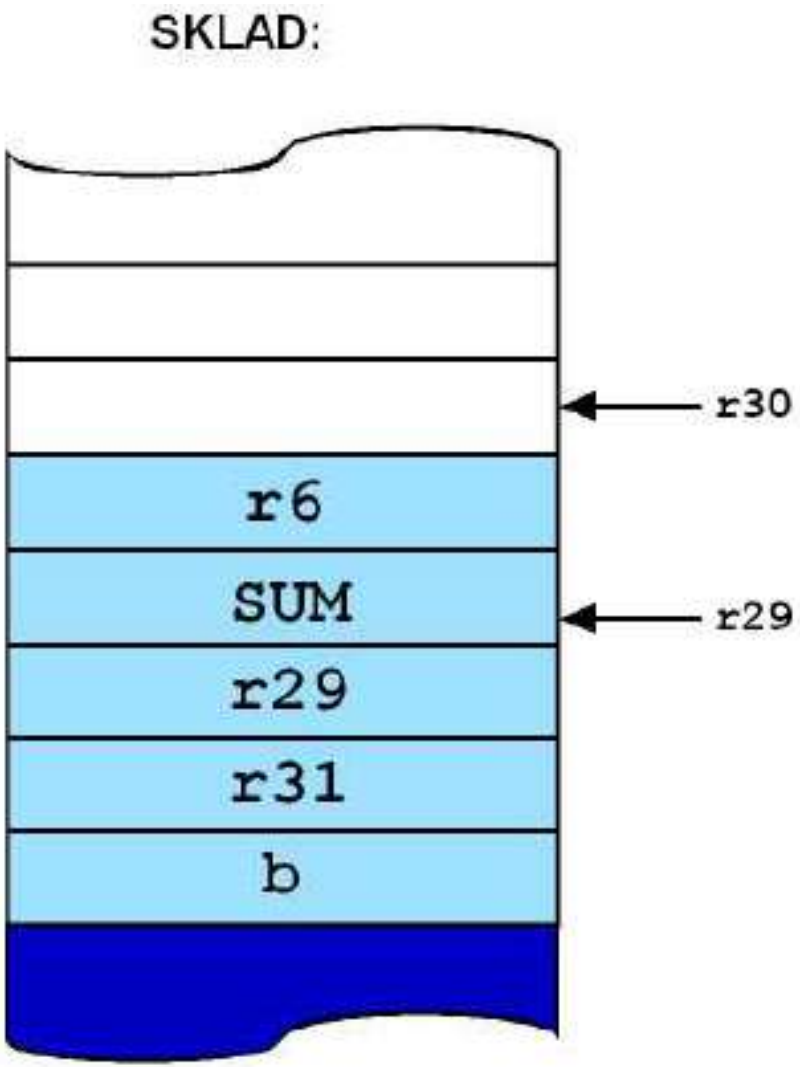

```
; PODPROGRAM:
; VSTOPNA TOCKA:
push r31
push r29
add r29,r0,r30

; PROCEDURA:
subui r30,r30,#4
push r6
lw r6,12(r29)
add r6,r6,r24
sw 0(r29), r6
lw r28, 0(r29)
pop r6

; IZSTOPNA TOCKA:
add r30,r0,r29
pop r29
pop 31
j 0(r31)
```

Sklad - HIP - zgled.



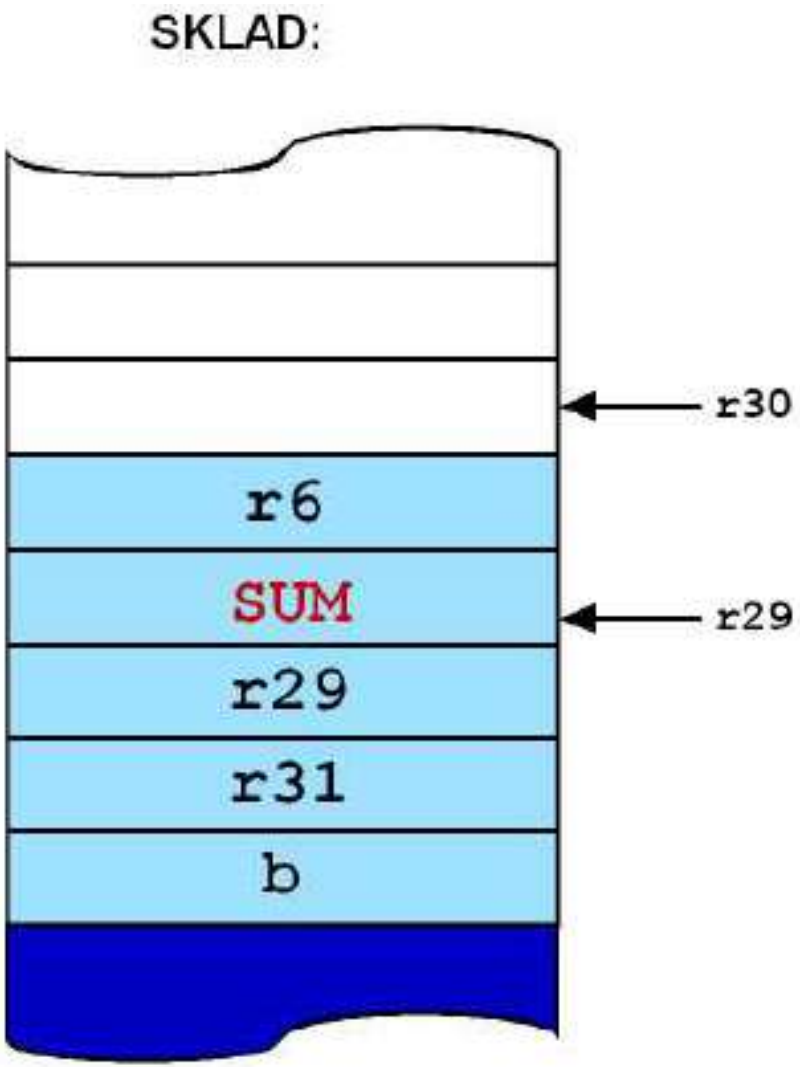

```
; PODPROGRAM:
; VSTOPNA TOCKA:
push r31
push r29
add r29,r0,r30

; PROCEDURA:
subui r30,r30,#4
push r6
lw r6,12(r29)
add r6,r6,r24
sw 0(r29), r6
lw r28, 0(r29)
pop r6

; IZSTOPNA TOCKA:
add r30,r0,r29
pop r29
pop 31
j 0(r31)
```

Sklad - HIP - zgled.



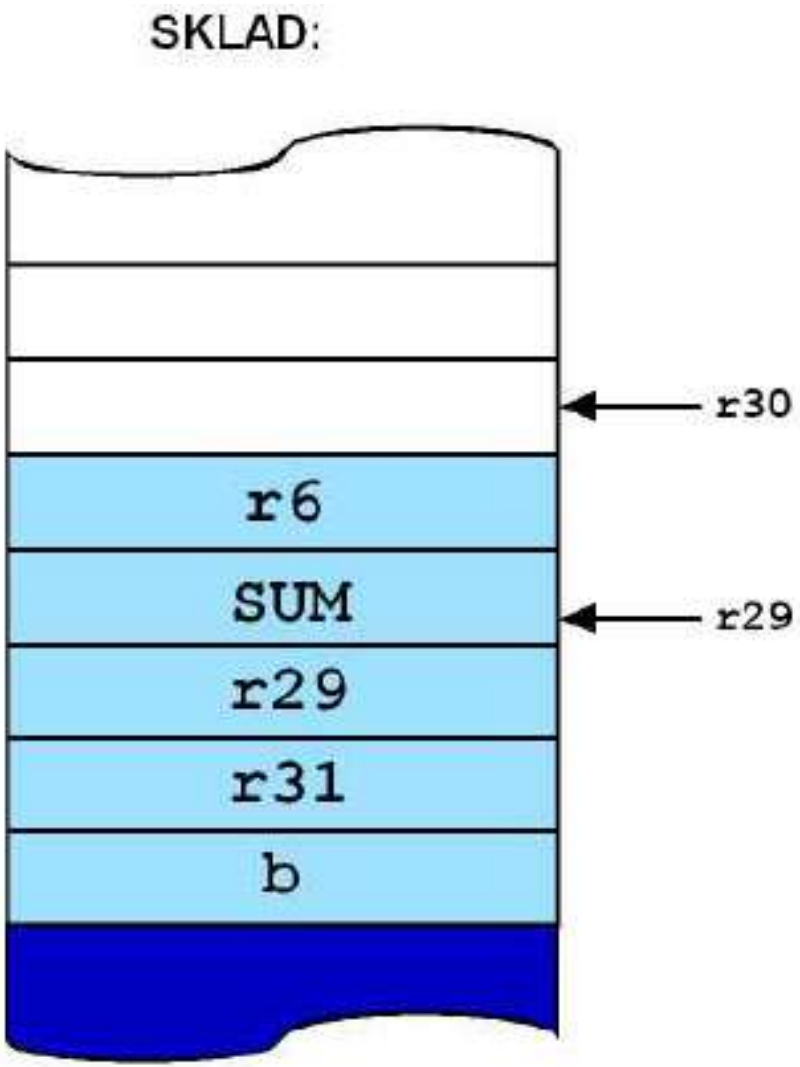

```
; PODPROGRAM:
; VSTOPNA TOCKA:
push r31
push r29
add r29,r0,r30

; PROCEDURA:
subui r30,r30,#4
push r6
lw r6,12(r29)
add r6,r6,r24
sw 0(r29), r6
lw r28, 0(r29)
pop r6

; IZSTOPNA TOCKA:
add r30,r0,r29
pop r29
pop 31
j 0(r31)
```

Sklad - HIP - zgled.



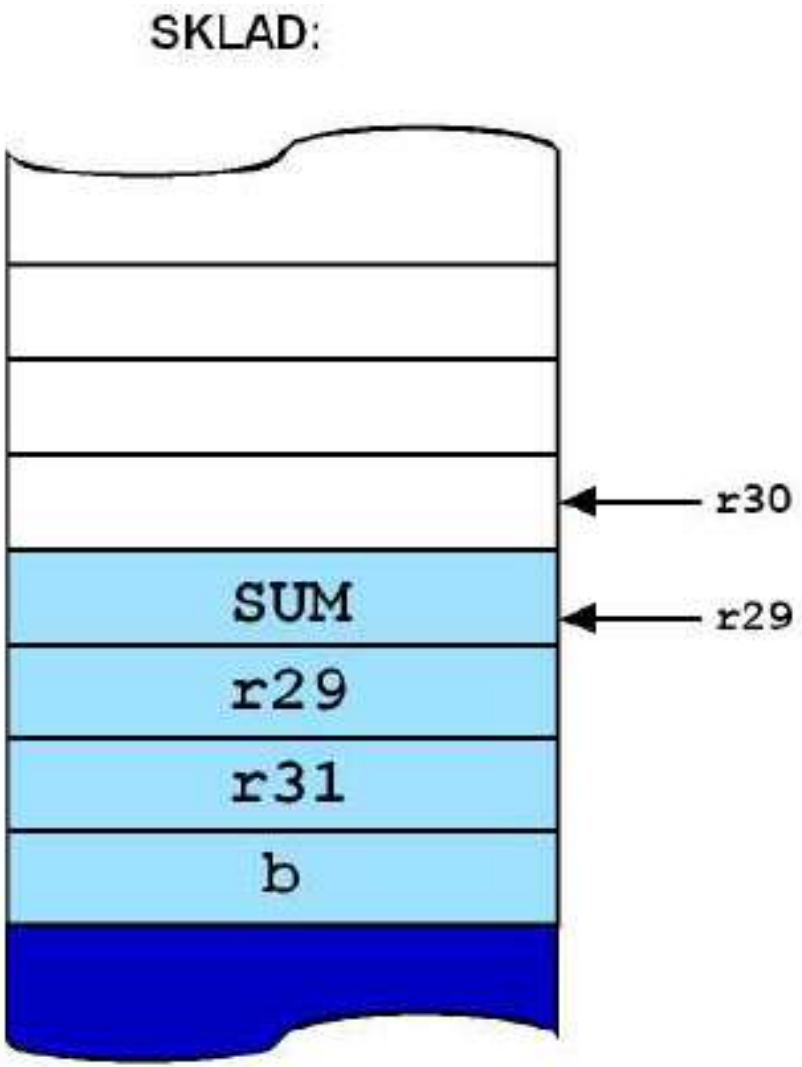

```
; PODPROGRAM:
; VSTOPNA TOCKA:
push r31
push r29
add r29,r0,r30

; PROCEDURA:
subui r30,r30,#4
push r6
lw r6,12(r29)
add r6,r6,r24
sw 0(r29), r6
lw r28, 0(r29)
pop r6

; IZSTOPNA TOCKA:
add r30,r0,r29
pop r29
pop 31
j 0(r31)
```

Sklad - HIP - zgled.



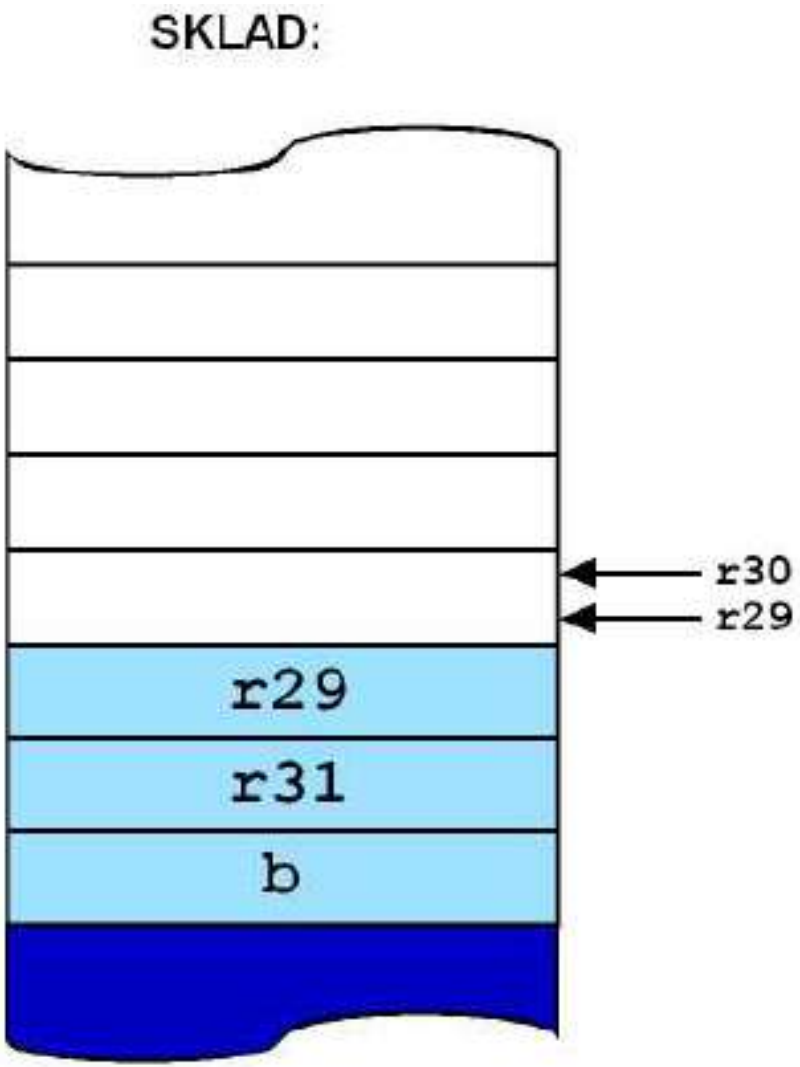

```
; PODPROGRAM:
; VSTOPNA TOCKA:
push r31
push r29
add r29,r0,r30

; PROCEDURA:
subui r30,r30,#4
push r6
lw r6,12(r29)
add r6,r6,r24
sw 0(r29), r6
lw r28, 0(r29)
pop r6

; IZSTOPNA TOCKA:
add r30,r0,r29
pop r29
pop 31
j 0(r31)
```

Sklad - HIP - zgled.



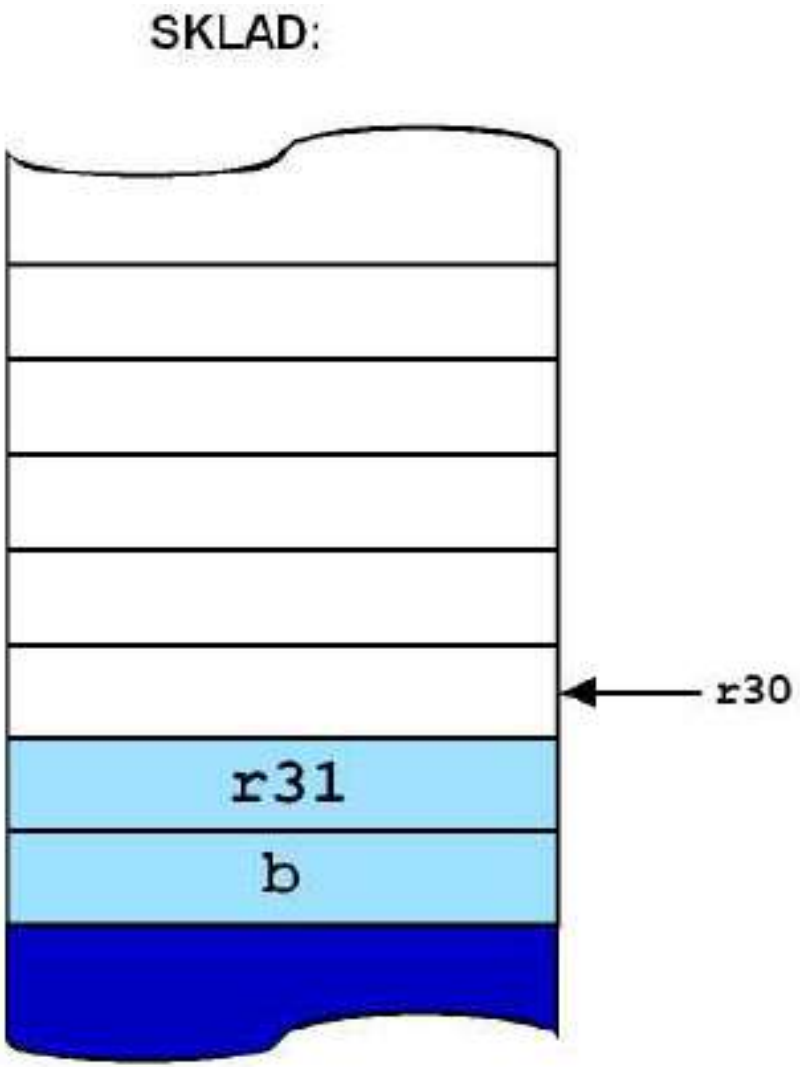

```
; PODPROGRAM:
; VSTOPNA TOCKA:
push r31
push r29
add r29,r0,r30

; PROCEDURA:
subui r30,r30,#4
push r6
lw r6,12(r29)
add r6,r6,r24
sw 0(r29), r6
lw r28, 0(r29)
pop r6

; IZSTOPNA TOCKA:
add r30,r0,r29
pop r29
pop 31
j 0(r31)
```

Sklad - HIP - zgled.



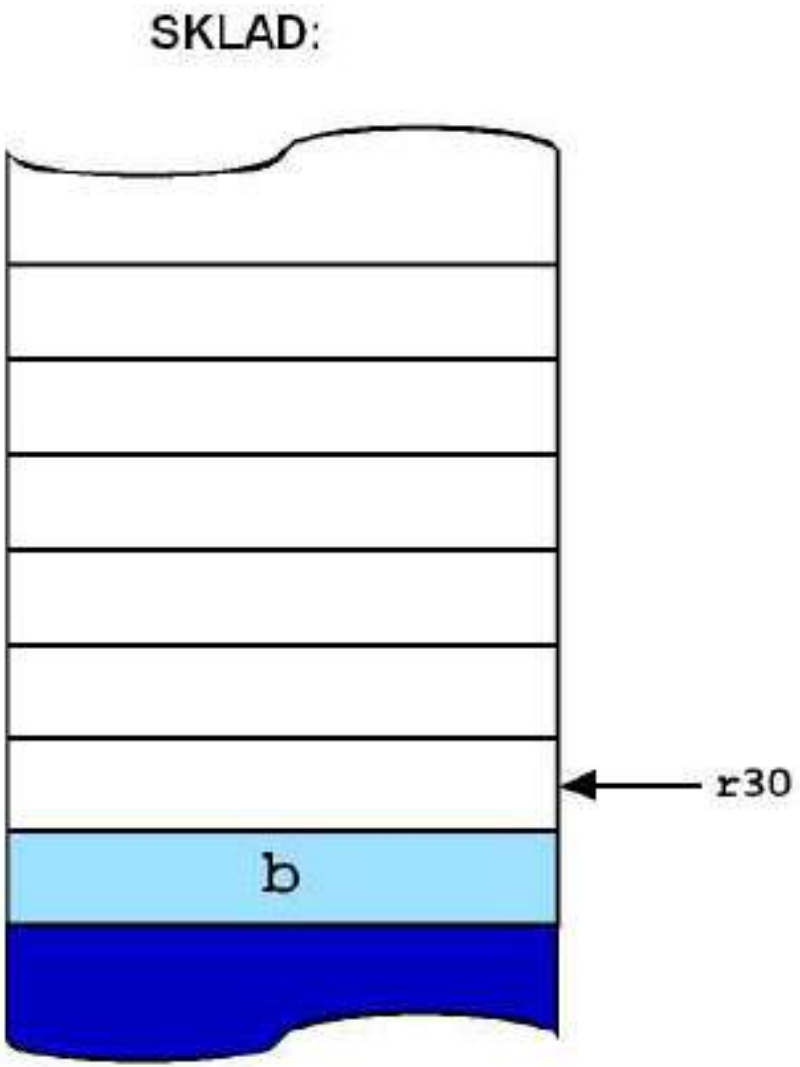

```
; PODPROGRAM:
; VSTOPNA TOCKA:
push r31
push r29
add r29,r0,r30

; PROCEDURA:
subui r30,r30,#4
push r6
lw r6,12(r29)
add r6,r6,r24
sw 0(r29), r6
lw r28, 0(r29)
pop r6

; IZSTOPNA TOCKA:
add r30,r0,r29
pop r29
pop 31
j 0(r31)
```

Sklad - HIP - zgled.



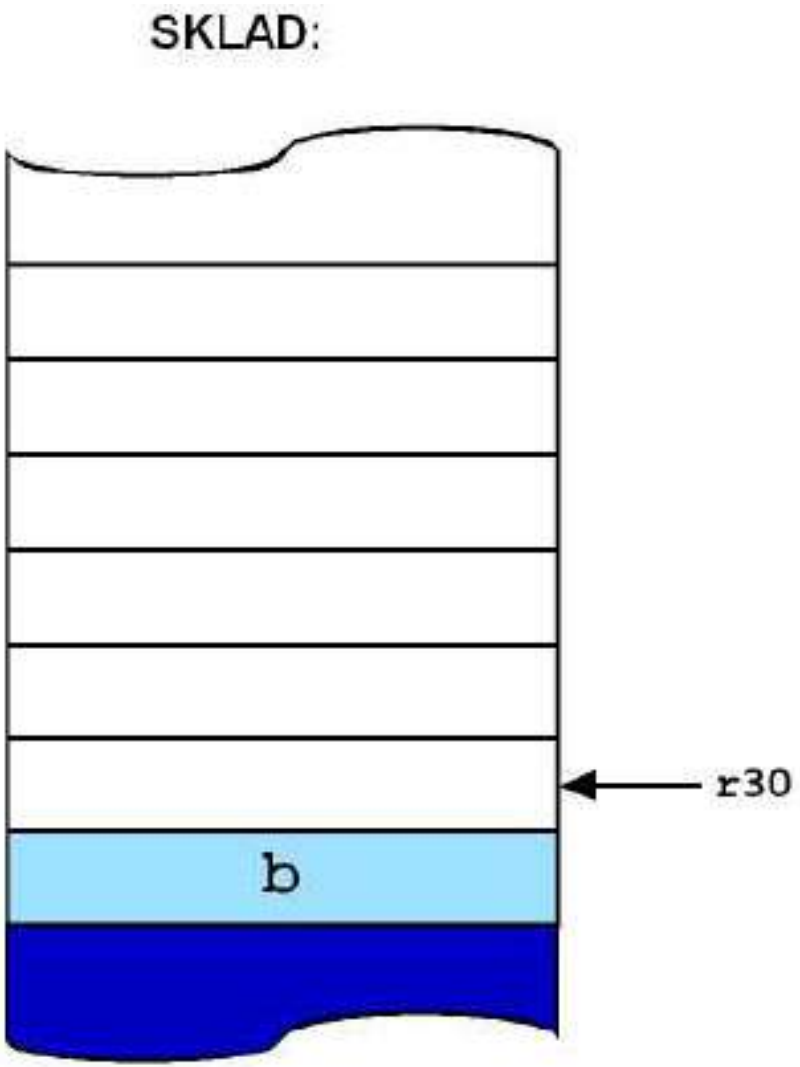

```
; PODPROGRAM:
; VSTOPNA TOCKA:
push r31
push r29
add r29,r0,r30

; PROCEDURA:
subui r30,r30,#4
push r6
lw r6,12(r29)
add r6,r6,r24
sw 0(r29), r6
lw r28, 0(r29)
pop r6

; IZSTOPNA TOCKA:
add r30,r0,r29
pop r29
pop 31
j 0(r31)
```

Sklad - HIP - zgled.



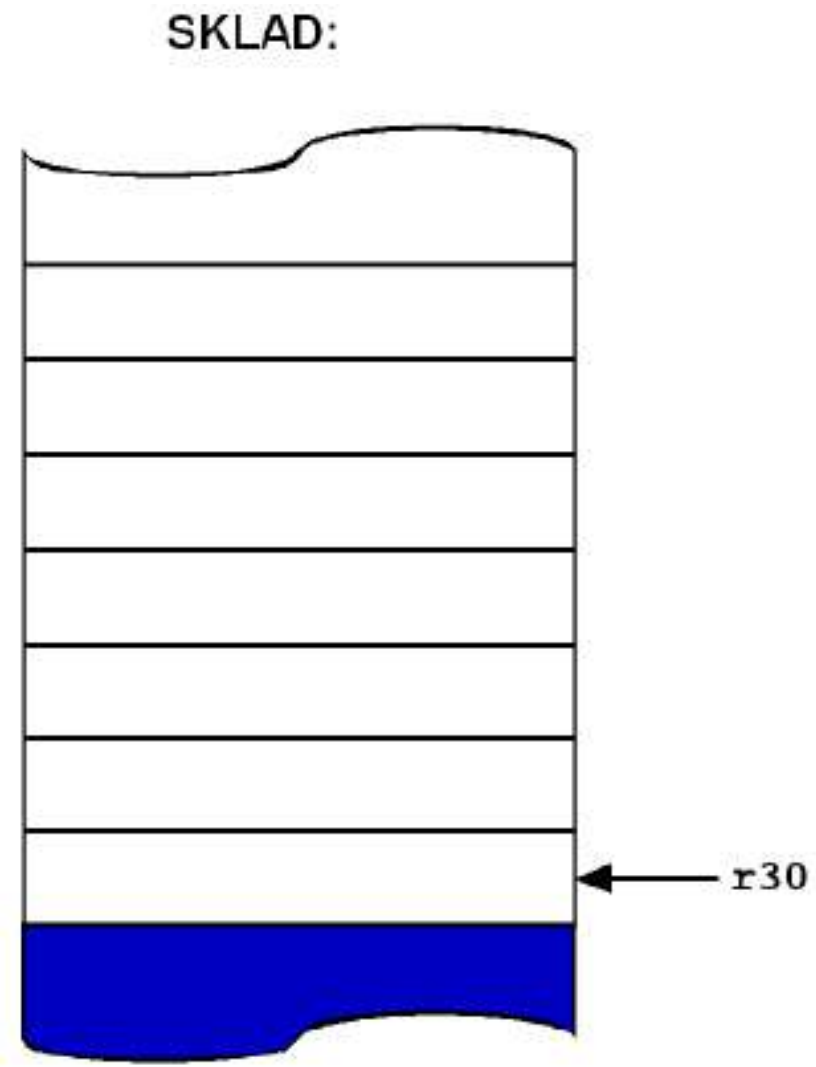

```
; PODPROGRAM:
; VSTOPNA TOCKA:
push r31
push r29
add r29,r0,r30

; PROCEDURA:
subui r30,r30,#4
push r6
lw r6,12(r29)
add r6,r6,r24
sw 0(r29), r6
lw r28, 0(r29)
pop r6

; IZSTOPNA TOCKA:
add r30,r0,r29
pop r29
pop 31
j 0(r31)
```

Sklad - HIP - zgled.




```
; PODPROGRAM:
; VSTOPNA TOCKA:
push r31
push r29
add r29,r0,r30

; PROCEDURA:
subui r30,r30,#4
push r6
lw r6,12(r29)
add r6,r6,r24
sw 0(r29), r6
lw r28, 0(r29)
pop r6

; IZSTOPNA TOCKA:
add r30,r0,r29
pop r29
pop 31
j 0(r31)
```

Sklad - HIP - zgled.




```
; KLICOCI PROGRAM:
lw r24, b(r0)
lw r3, a(r0)
push r3
call r31, sestej(r0)
addui r30,r30,#4
```

HIP - povzetek uporabe registrov

Register	Namen uporabe
R0	Ničla
R1-R23	Splošno namenski registri
R24	Prvi parameter z leve, ki se prenese v podprogram
R25	Drugi parameter z leve, ki se prenese v podprogram
R26	Bazni register za dolge skoke
R27	Bazni register za dolge klice
R28	Vrednost, ki jo vrača podprogram
R29	Kazalec na okvir
R30	Skladovni kazalec
R31	Povratni naslov

Za tiste, ki želijo znati več:

- Dušan Kodek. Arhitektura računalniških sistemov, 2. popravljena in razširjena izdaja, BI-TIM, 2000.
- David Patterson, John Hennessy. Computer Organization and Design, The Hardware/Software Interface, Third Edition (Appendix A: Assemblers, Linkers, and the SPIM Simulator)
- Procedure Call Standard for the ARM Architecture *http://www.arm.com/miscPDFs/8031.pdf*
- MicroBlaze Processor Reference Guide *http://www.xilinx.com/ise/embedded/edk_docs.htm*

5
Ukazi

Prevajanje ukazov

Prevajalnik programe, napisane v višjem programskem jeziku, prevede v zbirni jezik (zbirnik pa nato v strojni jezik), ali pa kar neposredno v strojni jezik

- Primer 1 (iz jezika C v zbirni jezik):

```
a = b + c;    // predpostavimo, da je a v r1, b v r2 in c v r3
add   r1, r2, r3   ; r1 ← r2 + r3
```

- Primer 2:

```
a = b + c + d + e;    // r1: a, r2: b, r3: c, r4: d, r5: e
add r1, r2, r3
add r1, r1, r4
add r1, r1, r5
```

- Primer 3:

```
A[12] = h + A[8];          // r1: A, r3: h

lw   r2, 32(r1)    ; r2 ← M[r1+32]
add  r2, r2, r3    ; r2 ← r2 + r3
sw   r2, 48(r1)    ; M[r1+48]← r2
```

- Operand je lahko tudi konstanta
  - **takojšnji** (**immediate**) operand

```
addi   r1, r2, 5          ; r1 ← r2 + 5
```

(add immediate)

Splošne lastnosti ukazov

- Vsak ukaz vsebuje
  - Informacijo o operaciji, ki naj se izvrši (operacijska koda)
  - Informacijo o operandih, nad katerimi naj se izvrši operacija
- Ukaz je shranjen v eni ali več (sosednih) pomnilniških besedah
- Format ukaza pove, kako so biti ukaza razdeljeni na operacijsko kodo in operande

5 dimenzij lastnosti ukazov

Dimenzija
1. Način shranjevanja operandov v CPE
2. Število eksplicitnih operandov v ukazu
3. Lokacija operandov in načini naslavljanja
4. Operacije
5. Vrsta in dolžina operandov

D1. Načini shranjevanja operandov v CPE

- 3 načini shranjevanja operandov v CPE:
  1. **Akumulator**
     - najstarejši način
     - edini register
       - zato ga v ukazih ni treba eksplicitno navajati
     - ukaza LOAD, STORE za prenos v in iz akumulatorja
     - veliko prometa z GP (shranjevanje vmesnih rezultatov), zato počasnost

2. Sklad (stack)

- v danem trenutku je dostopna samo najvišja lokacija
  - podobno kot sklad pladnjev
- LIFO
- ukaza PUSH, POP (ali PULL)
- podobno akumulatorju (takoj dostopen le 1 operand)
  - preprosta realizacija, kratki ukazi, preprosti prevajalniki
  - vendar je prostora za več operandov

3. Množica registrov (register set)

- Najbolje (danes edina rešitev)
  - nekdaj dragi, pa tudi prevajalniki jih niso znali dobro uporabljati
- Register je skupina pomnilniških celic, ki imajo skupne krmilne signale
  - Vsak register ima svoj naslov
- Namen: shranjevanje vmesnih rezultatov
  - pri skladu: v pomnilnik
- 2 rešitvi:
  - splošnonamenski registri (vsi ekvivalentni)
  - 2 skupini: za operande, za naslove
- 2 vrsti:
  - programsko nedostopni
  - programsko dostopni
    - programer jih lahko uporablja kot nek hiter pomnilnik

- **Programsko dostopni registri**
  - majhen pomnilnik, v katerega lahko shranimo enega ali več operandov
  - prednosti pred GP:
    1. Hitrost
       - registri so hitrejši od GP
       - bližji so aritmetično-logični in kontrolni enoti
       - možen je istočasen dostop do več registrov naenkrat
    2. Krajši ukazi
       - krajši naslov (ker je registrov malo) kot pri GP

D2: Število eksplicitnih operandov v ukazu

- **m-operandni** računalnik
  - običajno se podajajo naslovi operandov
  - danes m največ 3
- 4 skupine:
  - **3-operandni**

$$OP3 \leftarrow OP2 + OP1$$
$$PC \leftarrow PC + 1$$

    - operandi so običajno v registrih

- **2-operandni**
  - enostavnejši, a malo počasnejši

$$OP2 \leftarrow OP2 + OP1$$
$$PC \leftarrow PC + 1$$

- **1-operandni**
  - akumulator

$$AC \leftarrow AC + OP1$$
$$PC \leftarrow PC + 1$$

  - mikroprocesorji iz 70. in 80. let
    - Intel 8080, Motorola 6800, Zilog Z80
    - Intel 8086, Intel 80186, Intel 80286

- **Brez-operandni (skladovni)**
  - najkrajši ukazi

$$Sklad_{VRH} \leftarrow Sklad_{VRH} + Sklad_{VRH-1}$$
$$PC \leftarrow PC + 1$$

  - toda: potrebna sta vsaj 2 ukaza z ekspl. operandom!
    - PUSH, POP (prenos med GP in skladom)

D3: Lokacija operandov in načini naslavljanja

- 2 vprašanji:
  - Kje so operandi?
  - Kako je v ukazu podana informacija o njih?

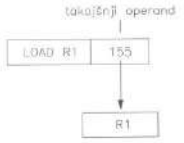
- **Lokacija operandov**
  - registri CPE
  - GP (oz. predpomnilnik)
  - (registri krmilnika V/I naprave)

- 2- in 3-operandni računalniki se delijo še na:
  - **registrsko-registrske** računalnike
    - najbolj razširjeni
    - vsi operandi v registrih CPE
    - reče se tudi **load/store** računalniki (ker rabimo load in store)
  - **registrsko-pomnilniške**
    - 1 operand *lahko* v pomnilniku, drugi v registru
  - **pomnilniško-pomnilniške**
    - vsak operand *lahko* v pomnilniku
    - zapleteni ukazi, CISC (npr. VAX)

Načini naslavljanja

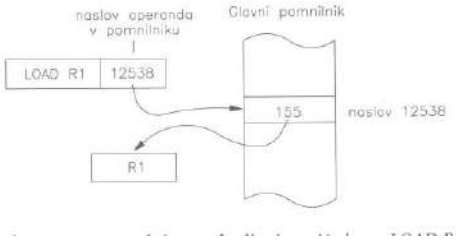
- Načini naslavljanja: Kako je v ukazu podana informacija o operandih
  - Tičejo se predvsem pomnilniških operandov
    - pri registrskih je enostavno

1. **Takojšnje naslavljanje** (immediate addressing)
   - operand je v ukazu podan z vrednostjo (je del ukaza)
   - **takojšnji operandi (literali)** so kar konstante
     - LOAD R1,#155, (R1 ← 155
     - ADD R1,#3 (R1 ← R1 + 3)



2. **Neposredno naslavljanje** (direct addressing)
   - operand je podan z naslovom
     - če je to naslov registra, je to **registrsko naslavljanje**
     - če je to naslov v GP, je to **(neposredno) pomnilniško naslavljanje**
   - primerno za operande, ki se jim ne spreminjajo naslovi

Registrsko: ADD R1, R2
Pomnilniško: LOAD R1, (12538) ali pa ADD R1, (1001)



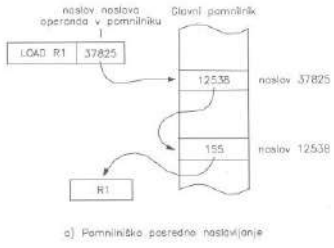
- Težave:
  - velik naslovni prostor → dolg naslov → dolgi ukazi
  - povečanje pom. prostora → drugačni ukazi → nezdružljivost za nazaj
  - primeri, ko operand ni na stalnem naslovu

3. **Posredno naslavljanje** (indirect addressing)

- v ukazu je naslov lokacije, na kateri je shranjen naslov operanda
  - **Pomnilniško posredno naslavljanje**, če gre za naslov pomnilniške lokacije (nerodno, ni pogosto)
    - ADD R1,@(1001) R1 ← R1 + M[M[1001]]
  - **Registrsko posredno naslavljanje**, če gre za naslov registra
    - uporablja se tudi **odmik** (displacement)
      - iz obojega se izračuna pomnilniški naslov
    - imenuje se tudi **relativno naslavljanje**
      - naslov operanda določen relativno na vsebino registra
    - najpogostejši način naslavljanja

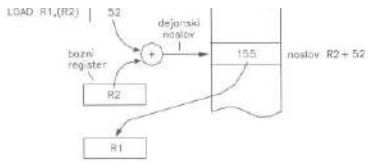
Posredno naslavljanje:

pomnilniško



a) Pomnilniško posredno naslavljanje

registrsko



Glavne vrste relativnega naslavljanja

3.1 Bazno naslavljanje (base addressing)

- reče se tudi **naslavljanje z odmikom** (displacement addressing)
- najpogostejše
- naslov operanda A = R2 + D
  - k vsebini registra R2 prištejemo odmik D
- R2 je **bazni register**, A pa **dejanski naslov** (effective address)
- Npr.: ADD R1,100(R2) R1 ← R1 + M[R2+100]
- Če D=0: Bazno brez odmika
  - ADD R1,(R2) R1 ← R1 + M[R2]

3.2 Indeksno naslavljanje (indexed addressing)

- odmik D
- $A = R2 + R3 + D = R2 + D_1$
- R3 je indeksni register
- glavno področje uporabe so polja, strukture in seznami
  - elementi se običajno obdelujejo zaporedoma po naraščajočih (ali padajočih) indeksih, zato sta pogosti operaciji

    $R3 \leftarrow R3 + \Delta$ in $R3 \leftarrow R3 - \Delta$
  - Δ je dolžina operanda, merjena v številu pomnilniških besed (*korak indeksiranja*)
- Npr.:
  - ADD R1,100(R2+R3), R1 ← R1 + M[R2+R3+100] (dostop do elementov polja)

3.3 Pred-dekrementno naslavljanje (pre-decrement addressing)

- $R3 \leftarrow R3 - \Delta$
- $A = R2 + D$ ali $A = R2 + R3 + D$
- bazno ali indeksno

3.4 Po-inkrementno naslavljanje (post-increment addressing)

- $A = R2 + D$ ali $A = R2 + R3 + D$
- $R3 \leftarrow R3 + \Delta$

3.5 Velikostno indeksno naslavljanje (scaled indexed addressing)

- $A = R2 + R3 \times \Delta + D$
- dovolj je inkrementirati R3

- Pred-dekrementno in po-inkrementno naslavljanje v paru tvorita **skladovno naslavljanje** (stack addressing)
  - sklad je v GP
  - določeni računalniki imajo register **skladovni kazalec** (stack pointer)

Še 2 pojma:

- **Pozicijsko neodvisno naslavljanje**
  - pozicijsko neodvisni programi
    - lahko jih premestimo v drug del pomnilnika
    - ne smejo vsebovati absolutnih naslovov
      - neposredno, pomnilniško posredno nasl.
  - možna rešitev je preslikovanje naslovov
    - če program ni pozicijsko neodvisen
- **PC-relativno naslavljanje**
  - kot bazni register služi kar programski števec (PC)

D4: Operacije

- Operacije niso ključnega pomena
  - Npr., možno je narediti računalnik, ki ima en sam ukaz:

    SBN A,B,C

    Pomen: M[A] ← M[A] - M[B]; če M[A] < 0, skoči na C

- Operacij je manj kot ukazov
- Imena ukazov so **mnemoniki**
  - okrajšava ang. imena ukaza
    - vsebuje tudi operacijo
    - npr. A, D, AD, ADD, S ... za seštevanje v fiksni vejici

Skupine operacij

1. **Aritmetične in logične operacije**
   - izvajajo se v ALE (nad operandi v fiksni vejici)
   - Aritmetične operacije: seštevanje, odštevanje, množenje, deljenje, aritm. negacija, absolutna vrednost, inkrement, dekrement
     - za vsako je več ukazov (različne dolžine operandov)
   - Logične operacije: AND, OR, NOT, XOR, pomiki

2. **Prenosi podatkov** (data transfer)

- izvor, ponor
- v resnici gre za kopiranje
- Običajni **mnemoniki**:
  - LOAD: GP → R
  - STORE: R → GP
  - MOVE: R → R ali GP → GP
  - PUSH: GP ali R → Sklad
  - POP (PULL): Sklad → GP ali R
- tudi CLEAR in SET

3. Kontrolne operacije

- spreminjajo vrstni red ukazov

3.1 Pogojni skoki (conditional branches). 3 načini za izpolnjenost pogoja:

- **Pogojni biti** se postavijo kot rezultat določenih operacij.
  - Z (zero), N (negative), C (carry), V (overflow), itd.
  - Npr. ukaz BEQ (branch if equal) skoči, če je Z=1
- **Pogojni register**
  - poljuben register
  - Npr. ali je njegova vsebina 0
- **Primerjaj in skoči** (compare and branch)
  - skok, če je primerjava izpolnjena

3.2 Brezpogojni skoki (uncond. branch, jump)

3.3 Klici in vrnitve iz podprogramov

- ukaz za klic podprograma mora shraniti **povratni naslov** (return address)
- tipična mnemonika sta CALL in JSR (jump to subroutine)
- RET (return) za vrnitev

4. **Operacije v plavajoči vejici**.
   - izvaja jih posebna enota (FPU – Floating Point Unit), ki ni del ALE
   - poleg osnovnih štirih operacij so še koren, logaritem, eksponentna in trigonometrične funkcije

5. **Sistemske operacije**.
   - vplivajo na način delovanja računalnika
   - običajno spadajo med **privilegirane ukaze**

6. **Vhodno/izhodne operacije.**
   - obstajajo na nekaterih računalnikih
     - na drugih se uporabljajo običajni ukazi za prenos podatkov
   - prenosi med GP in V/I ter med CPE in V/I

- Ukaze lahko delimo tudi na
  - **skalarne** in
  - **vektorske**
    - na vektorskih računalnikih se lahko ista operacija izvrši na *N* skupinah operandov
    - pri skalarnih je treba za to uporabiti zanko
    - vektorske ukaze srečamo na superračunalnikih

D5: Vrsta in dolžina operandov

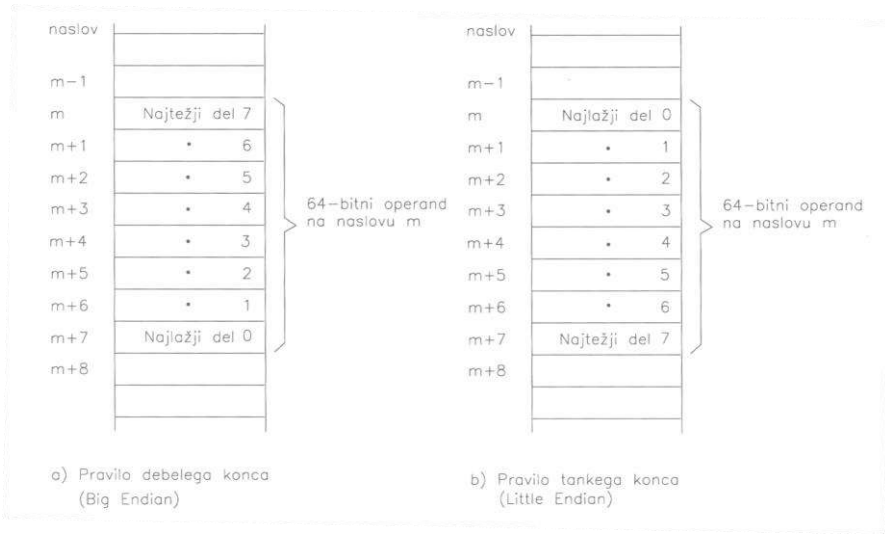
- Vrste operandov:
  1. **bit**
     - v višjih jezikih jih običajno ni
     - koristno pri sistemskih operacijah
  2. **znak**
     - običajno 8-bitni ASCII
     - več znakov tvori **niz** (string)
  3. **celo število**
     - predznačeno ali nepredznačeno
     - dolžine 8, 16, 32, 64 bitov
  4. **realno število**
     - št. v plavajoči vejici (običajno po standardu IEEE 754)
     - enojna natančnost 32 bitov, dvojna natančnost 64 bitov; obstajajo tudi 128-bitna
  5. **desetiško število**
     - v 8 bitih 2 BCD števili ali 1 ASCII znak

- Operandi dolžin večkratnikov 2 imajo posebna imena:

  8 Bajt (byte)
  16 Polovična beseda (halfword)
  32 Beseda (word)
  64 Dvojna beseda (double word)
  128 Štirikratna beseda (quad word)

  - to sicer ne velja za vse računalnike

- **Sestavljeni pomnilniški operandi** so sestavljeni iz več pomnilniških besed
  - v pomnilniku morajo biti na zaporednih lokacijah, sicer bi težko podali naslov takega operanda
- Obstajata 2 načina (glede na vrstni red), kako jih shranimo v pomnilnik:
  - **pravilo debelega konca** (Big Endian Rule)
    - najtežji del operanda na najnižjem naslovu
  - **pravilo tankega konca** (Little Endian Rule)
    - najlažji del operanda na najnižjem naslovu



- **Problem poravnanosti**
  - pomnilnik, ki omogoča dostop do 8 8-bitnih besed hkrati, je narejen kot 8 paralelno delujočih pomnilnikov
  - istočasen dostop do *s* besed dolgega operanda na naslovu *A* je možen le, če je *A* deljiv z *s* (*A* mod *s* = 0)
    - pri 8-bitni pomnilniški besedi mora imeti 64-bitni operand zadnje 3 bite enake 0
      - **poravnan** (aligned) operand
      - sicer **neporavnan** (misaligned)
        - potreben več kot en dostop
        - pri nekaterih računalnikih se sproži past

Ukazna arhitektura (ISA)

- **Ukazna arhitektura** (**Instruction Set Architecture**, **ISA**)
  - natančno definira vse ukaze (nabor ukazov) nekega procesorja
  - ne govori pa o implementaciji
- HIP je malo poenostavljena verzija procesorjev MIPS
  - MIPS spada med najbolj priljubljene RISC arhitekture

HIP

- HIP je model za študij CPE
  - temelji na procesorju MIPS R2000 oz. R3000
- Lastnosti:
  - 8-bitna pomnilniška beseda
  - 32-bitni pomnilniški naslov
  - dostop do PP traja pri zadetku en cikel ure, pri zgrešitvi 11
  - pomnilniško preslikan vhod/izhod

Ostale lastnosti HIP

- Način shranjevanja operandov v CPE
  - 32 32-bitnih splošnonamenskih registrov R0, R1, …, R31
    - Vsebina R0 je vedno 0 (pri pisanju vanj se ne zgodi nič)
- Število eksplicitnih operandov v ukazu
  - vsi ALE ukazi imajo 3 eksplicitne operande
    - pri dveh se tretji ignorira (NOT, LHI)
- Lokacija operandov in načini naslavljanja
  - Lokacija operandov
    - registrsko-registrski (load/store) računalnik
      - pomnilniški operandi nastopajo samo v ukazih load in store
    - pri ALE ukazih 2 operanda v registrih
      - tretji v registru ali takojšnji
    - dostop do operandov v pomnilniku le z load in store

- Naslavljanje: samo 2 načina
  - **Takojšnje** naslavljanje
    - takojšnji operand je 16-biten
    - razširitev predznaka (ali ničle) na 32 bitov
  - **Bazno** naslavljanje
    - odmik Di je 16-bitno predznačeno število v 2'K
    - dejanski naslov A = Rb + Di
    - kot bazni register Rb katerikoli splošnonamenski
      - če R0: neposredno naslavljanje s 16-bitnim naslovom, ki se razširi na 32 bitov z razširitvijo predznaka
      - če je odmik 0: bazno naslavljanje brez odmika

- Operacije in operandi
  - pomnilniški operandi so lahko 8-, 16- ali 32-bitni (bajt, polbeseda, beseda)
  - 16- in 32-bitni operandi so v pomnilniku shranjeni po **pravilu debelega konca** (**big endian rule**) in morajo biti obvezno poravnani (sicer past)
  - tudi biti (GP in R) po pravilu debelega konca
  - vse ALE operacije so 32-bitne
    - 8- in 16-bitni operandi se pri load pretvorijo v 32-bitne
      - Razširitev ničle pri nepredznačenih (LBU, LHU)
      - Razširitev predznaka pri predznačenih (LB, LH)
  - vse ALE operacije se izvršijo v eni urini periodi

Zgradba ukazov pri HIP:

- vsi ukazi so 32-bitni
- 2 formata s 6-bitno operacijsko kodo
  - Format 2: bita 30 in 31 enaka 1
  - Format 1: sicer

Format 1:

31 – 26	25 – 21	20 – 16	15 – 0
Op. koda	Rs1	Rd	Takojšnji operand ali odmik
6	5	5	16

Format 2:

31 – 26	25 – 21	20 – 16	15 – 11	10 – 0
Op. koda	Rs1	Rs2	Rd	func
6	5	5	5	11

- če bit 30 enak 1, imamo bazno naslavljanje, sicer takojšnje
- v formatu 2 parameter func razširja operacijsko kodo
  - v formatu 2 je možnih $2^4 * 2^{11}$ ukazov ($=2^{15}$), HIP jih uporablja 21
- kot Rs1, Rs2 ali Rd se lahko uporabi vsak register (R0 do R31)

➢ Število ukazov

- vseh ukazov je 52
  - 31 v formatu 1, 21 v formatu 2
- ni ukazov za množenje, deljenje
- ni ukazov v plavajoči vejici

Vrste ukazov

- Ukaze HIP delimo v več skupin:
  1. ukazi za prenos podatkov (load, store)
     - gre za prenos *operandov* med registri in pomnilnikom
     - nalaganje (iz pomnilnika) in shranjevanje (v pomnilnik)
  2. ALE ukazi
     - aritmetične in logične operacije
  3. kontrolni ukazi
     - skoki
  4. sistemski ukazi

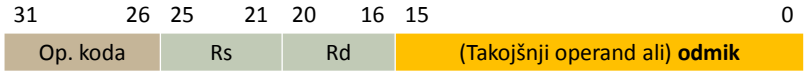
Ukazi za prenos podatkov (load/store)

- Uporabljajo format 1 z baznim naslavljanjem (bazni register je Rs1)

Load word:

lw r1, 30(r2) ; r1 $\leftarrow_{32}$ M[30 + r2]

Format 1:



6 bitov	5 bitov	5 bitov	16 bitov
100110	00010	00001	0000000000011110
lw	r2	r1	30
38	2	1	30

Celoten ukaz v strojni kodi: 0x9841001E

- Kaj pa, če je vrednost, ki jo želimo naložiti v (32-bitni) register, krajša od njegove dolžine?
  - Na katero vrednost postavimo preostale bite?
- 2 možnosti:
  - razširitev predznaka
    - lb (load byte (signed))
    - lh (load halfword (signed))
  - razširitev ničle
    - lbu (load byte unsigned)
    - lhu (load halfword unsigned)

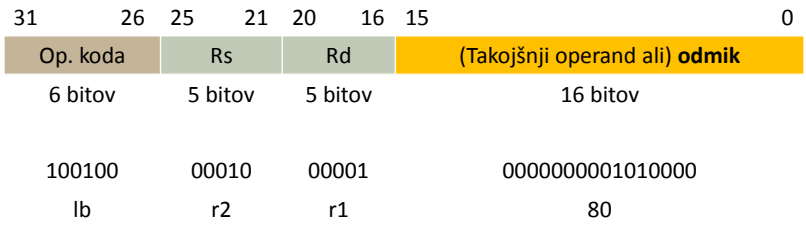
Load byte:

lb r1, 80(r2)

pomen: $r1_{31..8} \leftarrow_{raz} M[80 + r2]_7$, $r1_{7..0} \leftarrow_8 M[80 + r2]$

npr.: 0x6F se razširi drugače kot 0x94

Format 1:



Load byte unsigned

lbu r1, 80(r2)

pomen: $r1_{31..8} \leftarrow_{raz} 0$, $r1_{7..0} \leftarrow_{8} M[80 + r2]$

tu se 0x6F razširi enako kot 0x94

Podobno za 16-bitne besede:

- **Load halfword**
  - halfword (polbeseda) je 16 bitov: 2B
- **Load halfword unsigned**

Ukazi za prenos podatkov (load/store):

Format	Op. koda	Ukaz	Opis
1	100000	LBU	Load byte unsigned
1	100001	LHU	Load halfword unsigned
1	100100	LB	Load byte
1	100101	LH	Load halfword
1	100110	LW	Load word
1	101000	SB	Store byte
1	101001	SH	Store halfword
1	101010	SW	Store word

- Odmik je 16-biten
- Pri ukazih load za 8- in 16-bitne operande sta 2 varianti:
  - običajna (signed): razširitev predznaka (do 32 bitov)
  - unsigned: razširitev ničle (do 32 bitov)

Primeri ukazov load/store

Primer ukaza		Ime ukaza	Opis
LW	R1, 30(R2)	Load word	$R1 \leftarrow_{32} M[30 + R2]$
LW	R1, 1000(R0)	Load word	$R1 \leftarrow_{32} M[1000]$
LB	R1, 80(R3)	Load byte	$R1_{0..7} \leftarrow_{8} M[80 + R3]$, $R1_{8..31} \leftarrow_{raz} M[80 + R3]_{7}$
LBU	R1, 80(R3)	Load byte unsigned	$R1_{0..7} \leftarrow_{8} M[80 + R3]$, $R1_{8..31} \leftarrow 0$
LH	R1, 80(R3)	Load halfword	$R1_{0..15} \leftarrow_{16} M[80 + R3]$, $R1_{16..31} \leftarrow_{raz} M[80 + R3]_{15}$
LHU	R1, 80(R3)	Load halfword unsigned	$R1_{0..15} \leftarrow_{16} M[80 + R3]$, $R1_{16..31} \leftarrow 0$
SW	70(R5), R6	Store word	$M[70 + R5] \leftarrow_{32} R6$
SB	70(R5), R6	Store byte	$M[70 + R5] \leftarrow_{8} R6_{0..7}$
SH	70(R5), R6	Store halfword	$M[70 + R5] \leftarrow_{16} R6_{0..15}$

- M[x] je vsebina pomnilniške besede na naslovu x
- Znak $\leftarrow_{32}$ pomeni 32-bitni prenos iz (ali v) naslovov x, x+1, x+2, x+3 po pravilu debelega konca
- Znak $\leftarrow_{16}$ pomeni 16-bitni prenos iz (ali v) naslovov x, x+1
- Znak $\leftarrow_{8}$ pomeni 8-bitni prenos iz (ali v) naslov x
- Znak $\leftarrow_{raz}$ pomeni razširitev bita

Pri ukazih store je *Rd* izvor

Primer programa v zbirnem jeziku za procesor HIP

```
        .data
        .org 0x400
var1:   .byte 5
var2:   .byte 6
sum:    .space 1
        .align 2        ; zahtevana je poravnanost
ABC:    .word16 -33, 0x1C

        .code
        .org 0
        lb r1, var1(r0)
        lb r2, var2(r0)
        add r3, r1, r2
        sb sum(r0), r3
        halt
```


Psevdo-ukazi

.data – začetek podatkovnega segmenta

.text, **.code** – začetek ukaznega segmenta

.org <n> – določen začetni naslov

.space <n> – rezerviraj n bajtov prostora (naključne vred.)

.word <n1>,<n2>.. – določi zaporedna 32-bitna števila

.word16 <n1>,<n2>.. – določi zaporedna 16-bitna števila

.byte <n1>,<n2>.. – določi zaporedna 8-bitna števila

.align <n> – poravnaj naslov, da bo deljiv z n

Psevdo-ukazi so namenjeni zbirniku (programu), ne procesorju!

- Glej dokument (pdf) ***Nabor ukazov procesorja HIP***, ki je na spletni učilnici (v poglavju Druga gradiva in programi)
  - ta vsebuje tudi psevdo-ukaze zbirnika za procesor HIP

ALE ukazi

1. **aritmetične operacije** (+, –)
   - add, addu (+ addi, addui)
   - sub, subu (+ subi, subui)
2. **logične operacije** (&, V, ⊕, not)
   - and, or, xor (+ andi, ori, xori), not
3. **pomiki** (shift) (levi, desni; logični, aritmetični)
   - sll, srl, sra (+ slli, srli, srai)
   - lhi
4. **ukazi za primerjavo oz. set operacije** (pogoji: =, ≠, <, >, ≤, ≥)
   - seq, sne, slt, sgt, sltu, sgtu (+ seqi, snei, slti, sgti, sltui, sgtui)

- Logične operacije:
  - & (and), V (or), ⊕ (xor), not
  - Delujejo po bitih (*bitwise operations*)
    - 0110 and 1010 = 0010
      - 0xABCD and 0x1357
    - 0110 or 1010 = 1110
      - 0xABCD or 0x1357
    - 0110 xor 1010 = 1100
      - 0xABCD xor 0x1357
    - not 0110 = 1001
  - Uporabljajo se tudi za branje in vpisovanje posameznih bitov
    - branje bita: xxxx & 0100 primerjamo z 0000
    - nastavljanje bita: xxxx | 0100
    - brisanje bita: xxxx & 1011

- Pomiki:
  - **navadni pomiki** (shift)
  - **rotacije** (rotate)
- Navadni pomiki:
  - levi (0110 v 1100)
  - desni
    - **logični** (0110 v 0011)
      - v izpraznjena mesta gredo ničle
    - **aritmetični** (0110 v 0011, 1011 v 1101)
      - najbolj levi bit se ne spreminja in se vstavlja v izpraznjena mesta (število smatramo kot predznačeno – ta bit je predznak)

- Levi pomik (za n mest) predstavlja tudi množenje z 2^n
  - 00000101 << 3 = 00101000
- Desni pomik (za n mest) pa je deljenje z 2^n
  - 00110010 >> 4 = 00000011
- Aritmetični pomik ohrani predznak
  - število obravnava kot predznačeno
    - 11000 >> 1 = 11100
  - ni pa to več pravo celoštevilsko deljenje!
    - 11001 >> 1 = 11100 (-7 >> 1 = -4)
- S pomiki in seštevanjem/odštevanjem je možno realizirati tudi poljubno množenje/deljenje
- Tudi pomiki (logični) se uporabljajo za izločanje/vstavljanje bitov
  - npr. 0x1 << 2 = 0100

Seznam vseh ALE ukazov

- ALE ukazi so 3-operandni
  - razen NOT in LHI, ki sta v resnici 2-operandna, zato se eden od treh operandov ignorira
- 2 operanda sta v registrih
  - tretji je lahko v registru ali takojšnji (immediate)

Rd ← Rs1 op Rs2
Rd ← Rs1 op Takojšnji operand

ALE ukazi (1): aritmetične in logične operacije

Format	Op. koda	func	Ukaz	Opis	Tip operacije
2	110000	0	ADD	Add	Aritmetične
2	110001	0	SUB	Subtract	
2	110010	0	ADDU	Add unsigned	
2	110011	0	SUBU	Subtract unsigned	
2	110100	0	AND	And	Logične
2	110101	0	OR	Or	
2	110110	0	XOR	Exclusive or	
2	111110	0	NOT	Not (1's complement)	
1	000000	x	ADDI	Add immediate	Aritmetične takojšnje
1	000001	x	SUBI	Subtract immediate	
1	000010	x	ADDUI	Add unsigned imm.	
1	000011	x	SUBUI	Sub. unsigned imm.	
1	000100	x	ANDI	And immediate	Logične takojšnje
1	000101	x	ORI	Or immediate	
1	000110	x	XORI	Exclusive or imm.	

ALE ukazi (2): Ukazi za primerjavo

Format	Op. koda	func	Ukaz	Opis	Tip operacije
2	111000	0	SEQ	Set if equal	set
2	111001	0	SNE	Set if not equal	set
2	111010	0	SLT	Set if less than	set
2	111011	0	SGT	Set if greater than	set
2	111100	0	SLTU	Set if less than unsigned	set
2	111101	0	SGTU	Set if greater than unsigned	set
1	001000	x	SEQI	Set if equal immediate	set imm.
1	001001	x	SNEI	Set if not equal immediate	set imm.
1	001010	x	SLTI	Set if less than immediate	set imm.
1	001011	x	SGTI	Set if greater than imm.	set imm.
1	001100	x	SLTUI	Set if less than unsig. imm.	set imm.
1	001101	x	SGTUI	Set if greater than uns. imm.	set imm.

Če je pogoj izpolnjen, se v *Rd* zapiše 1, sicer 0

ALE ukazi (3): Pomiki

Format	Op. koda	func	Ukaz	Opis	Tip operacije
2	110000	1	SLL	Shift left logical	shift
2	110001	1	SRL	Shift right logical	shift
2	110010	1	SRA	Shift right arithmetic	shift
1	010000	x	SLLI	Shift left logical immediate	shift imm.
1	010001	x	SRLI	Shift right logical immediate	shift imm.
1	010010	x	SRAI	Shift right arithmetic imm.	shift imm.
1	000111	x	LHI	Load high immediate	

Ukazi za pomike uporabljajo pomikalnik (barrel shifter)

- kombinacijsko vezje, ki izvede poljuben pomik (za 0, ..., 31 mest) v eni urini periodi
- število mest pomika je podano v *Rs2* ali v takojšnjem operandu

Load high immediate (Lhi)

- poseben ukaz, ki 16-bitno (konstantno) vrednost naloži v gornjo polovico registra
- Zakaj sploh potrebujemo tak ukaz?
  - Problem je, kako naložiti 32-bitno konstanto v register
  - z enim 32-bitnim ukazom ni možno
  - zato to storimo v 2 korakih:
    1. naložimo zgornjih 16 bitov
    2. naložimo spodnjih 16 bitov
  - Npr.: 0x12345678

```
lhi   r1, 0x1234
addui   r1, r1, 0x5678
```

(lahko tudi `ori`)

- Kaj pa, če konstante ne poznamo?
  - npr., da gre za oznako (labelo)
    - ta predstavlja naslov
    - npr. oznaka ABC, ki ima vrednost 0x12345678
    - to vrednost izračuna zbirnik (assembler)
  - v tem primeru lahko zbirnik zasnujemo tako, da
    - v ukazu LHI podamo 32-bitno vrednost, zbirnik pa upošteva samo zgornjo polovico

      `lhi  r1, ABC` se tolmači kot `lhi r1, 0x1234`
    - v ukazu ADDUI podamo 32-bitno vrednost, zbirnik pa upošteva samo spodnjo polovico

      `addui r1,r1,ABC` se tolmači kot `addui r1,r1,0x5678`

- To velja tudi, če so podatki na naslovu, ki se ne da zapisati s 16 biti (“visok” naslov)
  - v tem primeru je treba visok naslov naložiti v register
  - to je ekvivalentno nalaganju (poljubne) 32-bitne konstante v register

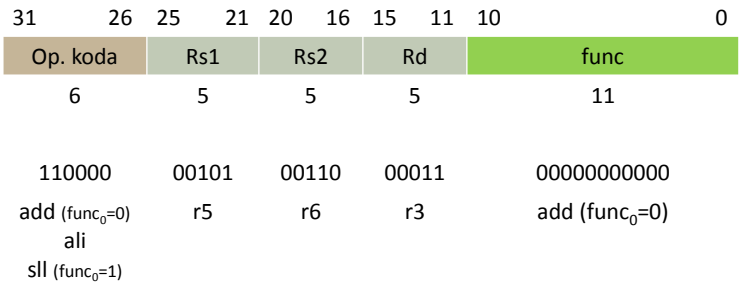
- Primer programa v zbirnem jeziku za procesor HIP, ki vrednost 32-bitne spremenljivke A prepiše v 32-bitno spremenljivko B. Ukazi naj se nahajajo od naslova 0x0 naprej, spremenljivka A je na naslovu 0x400, B pa na naslovu 0x25000000.

```
        .data
        .org 0x400
A:      .word 2014

        .data
        .org 0x25000000
xy:     .space 4
B:      .space 4

        .code
        .org 0
        lw r1, A(r0)
        lhi r2, B           ; assembler: lhi r2, 0x2500
        addui r2, r2, B     ; assembler: addui r1, r2, 0x0004
        sw 0(r2), r1
        halt
```


Add:

add r3, r5, r6 ; r3 ← r5 + r6

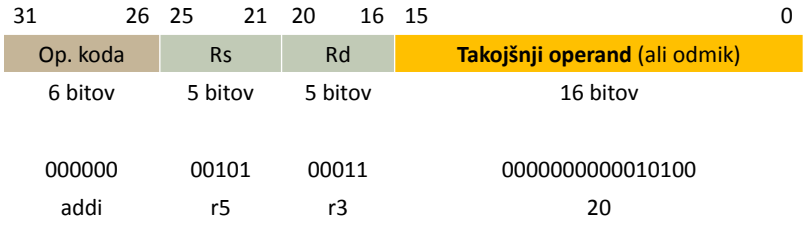
Format 2:



Add immediate:

addi r3, r5, 20 ; r3 ← r5 + 20

Format 1:



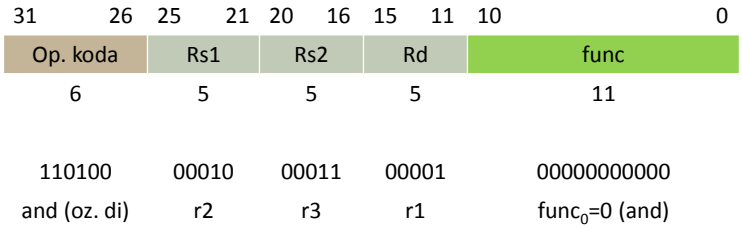
Zakaj imata add in addi različen format?

- Ukaza **addu** in **addui** sta enaka kot **add** in **addi**, le da se operanda tolmačita kot nepredznačena (unsigned)
  - zato ne more priti do preliva, sam izračun pa je enak
- Ukazi **sub**, **subi**, **subu** in **subui** so namenjeni odštevanju (i ... immediate, u ... unsigned)

And:

and r1, r2, r3 ; r1 ← r2 & r3

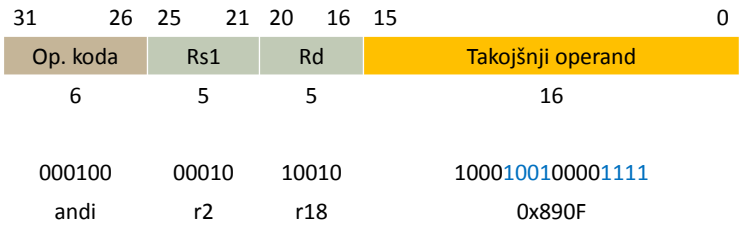
Format 2:



And immediate:

andi r18, r2, 0x890F ; r18 ← r2 & 0x890F

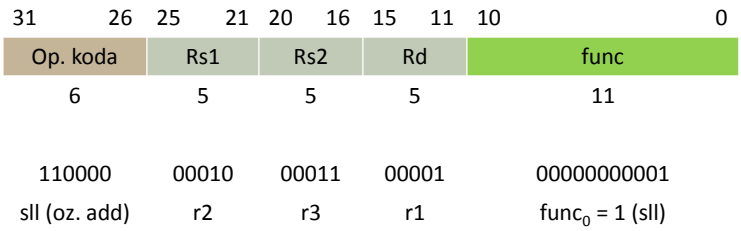
Format 1:



Shift left logical:

```
sll  r1, r2, r3          ; r1 ← r2 << r3 (oz. r2 × 2^r3)
```

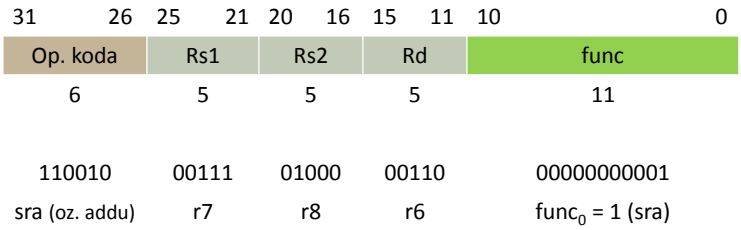
Format 2:



Shift right arithmetic:

```
sra  r6, r7, r8          ; r6 ← r7 >> r8
                         ; r6_31 ← r7_31
```

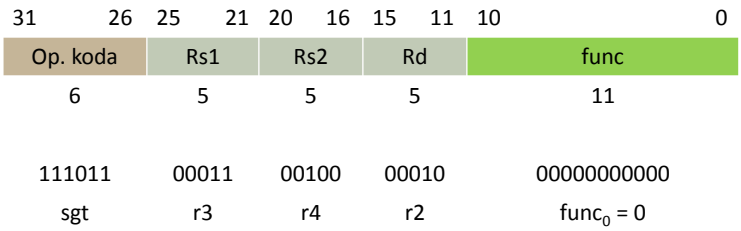
Format 2:



Set if greater than:

sgt r2, r3, r4 ; r2 ← (r3 > r4)

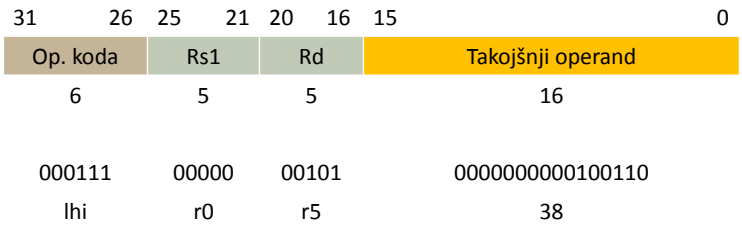
Format 2:



Load high immediate:

lhi r5, 38 ; $r5_{31..16} \leftarrow 38, r5_{15..0} \leftarrow 0$

Format 1:



- Primer: program, ki na osnovi pomikov in seštevanja 32-bitno nepredznačeno spremenljivko A množi z 10 in jo shrani v B:

```
        .data
        .org 0x400
A:      .space 4      ( ali .word vrednost)
B:      .space 4

        .code
        .org 0x0
        lw r1, A(r0)
        slli r2, r1, 3
        slli r3, r1, 1
        add r4, r2, r3
        sw B(r0), r4
        halt
```


- Register R0, lahko uporabimo za tvorbo vrste novih operacij iz obstoječih:
  - MOV R1,R2 (Move one register to another) z ukazom ADD R1,R0,R2.
    - Vsebina R2 se naloži v R1
  - NOP (No operation) z ukazom ADDI R0,R0,#0
  - LI R1,#const (Load halfword immediate) z ukazom ADDI R1,R0,#const
    - 16-bitna predznačena konstanta const se naloži v R1 z razširitvijo predznaka
  - LUI R1,#const (Load halfword unsigned immediate) z ukazom ADDUI R1,R0,#const
    - 16-bitna nepredznačena konstanta const se naloži v R1 z razširitvijo ničle

Kontrolni ukazi

- Kontrolni ukazi omogočajo spremembo vrstnega reda izvajanja ukazov
  - takim ukazom rečemo skoki
- 2 vrsti skokov:
  - brezpogojni
    - vedno se izvede
    - omogoča preskok dela programa, pa tudi vrnitev nazaj
  - pogojni
    - izvede se, če je izpolnjen določen pogoj
    - omogoča pogojni preskok dela programa in končne zanke
- Kontrolni ukazi omogočajo vejitve in zanke

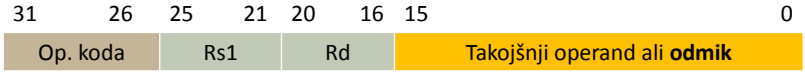
- Skoki pri HIP:
  - Brezpogojni skok:
    - j (jump)
  - Pogojna skoka:
    - beq (branch if equal zero) pri Rd = 0
    - bne (branch if not equal zero) pri Rd ≠ 0
  - Kontrolni ukazi pri HIP uporabljajo format 1
  - Pogojna skoka uporabljata PC-relativno naslavljanje
    - za bazni register je uporabljen PC + 4
    - register Rs1 se ignorira

Jump:

j LABEL(Rs1) ; PC ← LABEL + Rs1

Npr.: j Nekam(r0) ; PC ← Nekam + r0

Format 1:



6 bitov	5 bitov	5 bitov	16 bitov
101100	00000	00000	0000000000111000
j	r0	r0	56
		(*Rd* se ignorira)	

Branch if equal (to) zero:

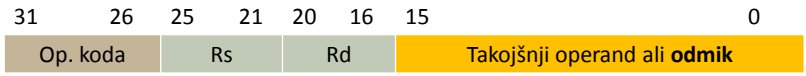
beq Rd, LABEL (PC-relativno)

(pogoj se nanaša na vsebino registra *Rd*)

Npr.:

beq r4, 24 ; če r4 == 0, potem PC ← PC + 4 + 24, sicer PC ← PC + 4

Format 1:



6 bitov	5 bitov	5 bitov	16 bitov
100111	00000	00100	0000000000011000
beq	r0	r4	24
	(*Rs* se ignorira)		

Vejitve

```
if ( pogoj)
      blok1
else
      blok2
```

- Če pogoj ni izpolnjen, je treba skočiti na blok2

- Ukaz beq izvaja pogojni skok

```
beq  Rd, LABEL
```

  - Če je register Rd enak 0, CPE skoči na naslov LABEL
- Podoben ukaz je

```
bne  Rd, LABEL
```

  - Če je register Rd različen od 0, CPE skoči na naslov LABEL

- Primer:

```
if (x > 5)
      a = b + 1;
else
      a = b + 2;
```

Predpostavimo r1: x, r2: a, r3: b

- 2 možnosti:

```
        sgti   r4, r1, 5    ; if (x > 5) r4 = 1;
        bne    r4, Blk1     ; if (r4!=0) goto Blk1;
        addi   r2, r3, 2    ; a = b + 2;
        j      Ven(r0)      ; goto Ven;
Blk1:   addi   r2, r3, 1    ; Blk1: a = b + 1;
Ven:    naslednji ukaz      ; Ven: naslednji ukaz
```

```
        sgti   r4, r1, 5    ; if (x > 5) r4 = 1;
        beq    r4, Blk2     ; if (r4==0) goto Blk2;
        addi   r2, r3, 1    ; a = b + 1;
        j      Ven(r0)      ; goto Ven;
Blk2:   addi   r2, r3, 2    ; Blk2: a = b + 2;
Ven:    naslednji ukaz      ; Ven: naslednji ukaz
```


Zanke

```
while ( pogoj)
     Blok;
```

Pogosto je zanka WHILE take oblike:

```
i = I1;
while ( i < I2)
{
   ...
   i = i + K;
}
```

V takem primeru lahko uporabimo tudi zanko FOR:

```
for (i = I1; i < I2; i=i+K)
{
   ...
}
```

Primer 1

Jezik C	Zbirni jezik za HIP (r1: sum, r2: i)

sum = 0;
i = 5;
while (i > 2)
{
  sum = sum + i;
  i--;
}
	addi r1, r0, #0
	addi r2, r0, #5
	Loop: sgti r3, r2, #2
	beq r3, Ven
	add r1, r1, r2
	subi r2, r2, #1
	j Loop(r0)
	Ven: ...

sum = 0;
for (i=5; i>2; i--)
  sum = sum + i;
	isto kot zgoraj

sum = 0;
i = 5;
do {
  sum = sum + i;
  i--;
} while (i > 2);
	addi r1, r0, #0
	addi r2, r0, #5
	Loop: add r1, r1, r2
	subi r2, r2, #1
	sgti r3, r2, #2
	bne r3, Loop

- Zanka do-while je v zbirnem jeziku enostavnejša od while
  - Seveda pa while in do-while v splošnem nista ekvivalentna!
    - pri slednjem se blok prvič vedno izvede

Primer 2

```
            while ( a[i] == j)
                  i++;

            (r1: i, r2: j, r3: bazni naslov a, tj. naslov od a[0])

Loop:       sll   r4, r1, 2
            add   r4, r4, r3
            lw    r5, 0(r4)    ; v r4 je naslov a[i]
            sub   r5, r5, r2
            bne   r5, Exit
            addi  r1, r1, #1
            j     Loop(r0)
Exit:       ...
```

Nabor vseh kontrolnih ukazov

Format	Op. koda	Ukaz	Opis
1	100011	BNE	Branch if Rd not equal to zero
1	100111	BEQ	Branch if Rd equal to zero
1	101100	J	Jump
1	101101	CALL	Jump to subroutine
1	101110	TRAP	Jump to vector address
1	101111	RFE	Return from exception

- Ukaz za klic procedure CALL shrani naslov naslednjega ukaza (PC + 4) v Rd, v PC pa se zapiše Rs+odmik
  - vrnitev iz proc. je brezpogojni skok z odmikom 0 (bazni reg. Rd)

- Ukaz TRAP
  - EPC ← PC
  - onemogoči se prekinitve (I ← 0)
  - skok na servisni program
    - njegov naslov je na naslovu FFFFFF00 + 4×n (to je vektorski naslov ali vektor, n pa je številka vektorja)
    - shrani se v PC

- Pri RFE se EPC zapiše v PC

Primeri kontrolnih ukazov

Primer ukaza		Ime ukaza	Opis
J	84 (R8)	Jump	PC ← R8 + 84
BEQ	R7, 800	Branch if equal to zero	če je R7 = 0, potem PC ← PC + 4 + 800 sicer PC ← PC + 4
BNE	R7, 800	Branch if not equal to zero	če je R7 ≠ 0, potem PC ← PC + 4 + 800 sicer PC ← PC + 4
CALL	R9, 84(R8)	Jump to subroutine	R9 ← PC + 4, PC ← R8 + 84
TRAP	#n	Jump to vector address	EPC ← PC + 4, PC $\leftarrow_{32}$ M[FFFFFF00 + 4×n], I ← 0, 0 ≤ n ≤ 63
RFE		Return from exception	PC ← EPC

Sistemski ukazi

Format	Op. koda	func	Ukaz	Opis
2	110011	1	EI	Enable interrupts
2	110100	1	DI	Disable interrupts
2	110101	1	MOVER	Move from EPC to register
2	110110	1	MOVRE	Move from register to EPC

- Pri sistemskih ukazih se registri ignorirajo (ali pa se uporablja le eden)

Zgradba ukazov

Ukaz je shranjen v eni ali več (sosednih) pomnilniških besedah

Vsak ukaz vsebuje

1. Operacijsko kodo (informacijo o operaciji, ki naj se izvrši)
2. Informacijo o operandih, nad katerimi naj se izvrši operacija

- **Zgradba ali format ukaza**
  - pove, kako so biti ukaza razdeljeni na operacijsko kodo in operande
    - število polj, njihova velikost in pomen posameznih bitov v njih
- Možni so različni formati
- Parametri, ki najbolj vplivajo na format:
  1. Dolžina pom. besede
     - pri 8: dolžina ukaza večkratnik 8
     - pri dolgih pom. besedah: dolžina ukaza ½ ali ¼ besede
  2. Število eksplicitnih operandov v ukazu
  3. Vrsta in število registrov v CPE
     - št. registrov vpliva na št. bitov za naslavljanje
  4. Dolžina pom. naslova
     - predvsem, če se uporablja neposredno naslavljanje
  5. Število operacij

- Optimalne rešitve za format ukazov ni
  - kaj je kriterij?
  - neke vrste umetnost
  - medsebojna odvisnost parametrov
  - možno je minimizirati velikost programov
    - pogostost ukazov, Huffmanovo kodiranje
    - v praksi se ni izkazalo (Burroughs)

- 3 načini:

1. Spremenljiva dolžina
   - št. eksplicitnih operandov spremenljivo
   - različni načini naslavljanja
   - veliko formatov
     - npr. 1..15 bajtov pri 80x86, 1..51 VAX
   - kratki formati za pogoste ukaze

Op. koda	Način naslavljanja 1	Naslovno polje 1	. . .	Način naslavljanja n	Naslovno polje n

2. Fiksna dolžina
   - št. eksplicitnih operandov fiksno
   - majhno št. formatov (RISC)
     - Alpha, ARM, MIPS, PowerPC, SPARC

Op. koda	Naslovno polje 1	Naslovno polje 2	Naslovno polje 3

3. Hibridni način

Op. koda	Način naslavljanja	Naslovno polje

Op. koda	Naslovno polje 1	Način naslavljanja 2	Naslovno polje 2

Op. koda	Način naslavljanja	Naslovno polje 1	Naslovno polje 2

➢ **Ortogonalnost ukazov** (medsebojna neodvisnost parametrov ukaza)

1. Informacija o operaciji neodvisna od info. o operandih
2. Informacija o enem operandu neodvisna od info. o ostalih operandih

Število ukazov in RISC

- **CISC računalniki**
  - Complex Instruction Set Computer
  - imajo veliko število ukazov
  - IBM 370, VAX, Intel
- **RISC računalniki**
  - Reduced Instruction Set Computer
  - imajo majhno število ukazov
  - MIPS, ARM, DEC Alpha, IBM/Motorola Power PC
- Oboji imajo svoje prednosti in slabosti
  - na začetku so bili računalniki tipa CISC, RISC pa so se pojavili kasneje
  - RISC so enostavnejši in imajo hitrejše ukaze, vendar pa program potrebuje več ukazov

- 2 ugotovitvi v 80. letih:
  1. **Stalno povečevanje števila ukazov**
     - IAS (1951): 23 ukazov in 1 način nasl.
     - 70. leta: stotine ukazov
  2. **Velik del ukazov redko uporabljan**

Razlogi za povečevanje števila ukazov

- Semantični prepad
  - v 60. letih so proizvajalci zato povečevali št. ukazov
- Mikroprogramiranje
  - dodajanje novih ukazov preprosto
- Razmerje med hitrostjo CPE in GP
  - faktor vsaj 10
  - kompleksen ukaz hitrejši kot zaporedje preprostih ukazov

Razlogi za zmanjševanje števila ukazov

- Težave prevajalnikov
  - velik del ukazov redko uporabljan
- Pojav predpomnilnikov
  - v primeru zadetka v PP je dostop skoraj enako hiter kot do mikroukazov
- Uvajanje paralelizma v CPE
  - cevovod (lažja realizacija pri preprostih ukazih)

Definicija arhitekture RISC

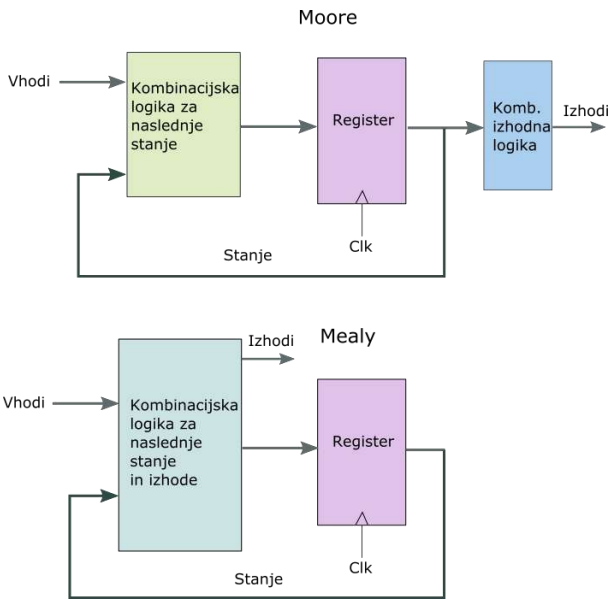
- Večina ukazov se izvrši v enem ciklu CPE
  - lažja real. cevovoda
- Registrsko-registrska zasnova (load/store)
  - zaradi zahteve 1
- Ukazi realizirani s trdo ožičeno logiko
  - ne mikroprogramsko
- Malo ukazov in načinov naslavljanja
  - hitrejše in enostavnejše dekodiranje in izvrševanje
- Enaka dolžina ukazov
- Dobri prevajalniki
  - upoštevajo zgradbo CPE

6 Centralna procesna enota

Splošno

- CPE je digitalen sistem.
- Vsebuje **kombinacijska** in **sekvenčna** digitalna vezja
- **Stanje CPE:**
  - stanje sekvenčnih (pomnilnih) elementov
- Delovanje CPE je odvisno od
  - trenutnega stanja in
  - trenutnih vhodov

Konceptualna predstavitev sinhronskih digitalnih vezij



Delovanje CPE

1. **Dobava ukaza iz pomnilnika** (fetch)
2. **Izvrševanje ukaza**
   a) dekodiranje ukaza
   b) prenos operandov v CPE (po potrebi)
   c) izvedba operacije
   d) shranjevanje rezultata (po potrebi)
   e) PC ← PC + 1 (razen pri skokih)

- Ta cikel dveh korakov se ponavlja, dokler računalnik deluje
  - Izjema so prekinitve (interrupt) in pasti (trap)
    - skok na nek drug ukaz

- Vsak od 2 korakov je sestavljen iz bolj elementarnih korakov
  - vsak traja eno ali več period ure CPE
  - urin signal
- Pri sinhronih sekvenčnih vezjih se spremembe stanja prožijo ob aktivni fronti ure (prednja ali zadnja)
  - perioda ure CPE (t_{CPE}) je čas med dvema sosednima frontama

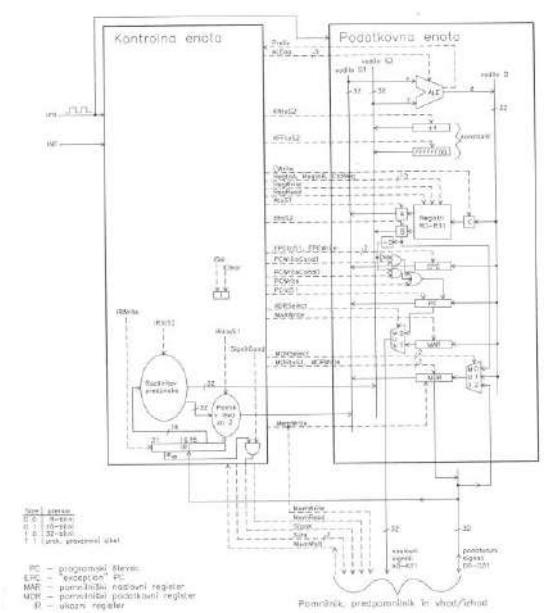
$f_{CPE} = 1/t_{CPE}$

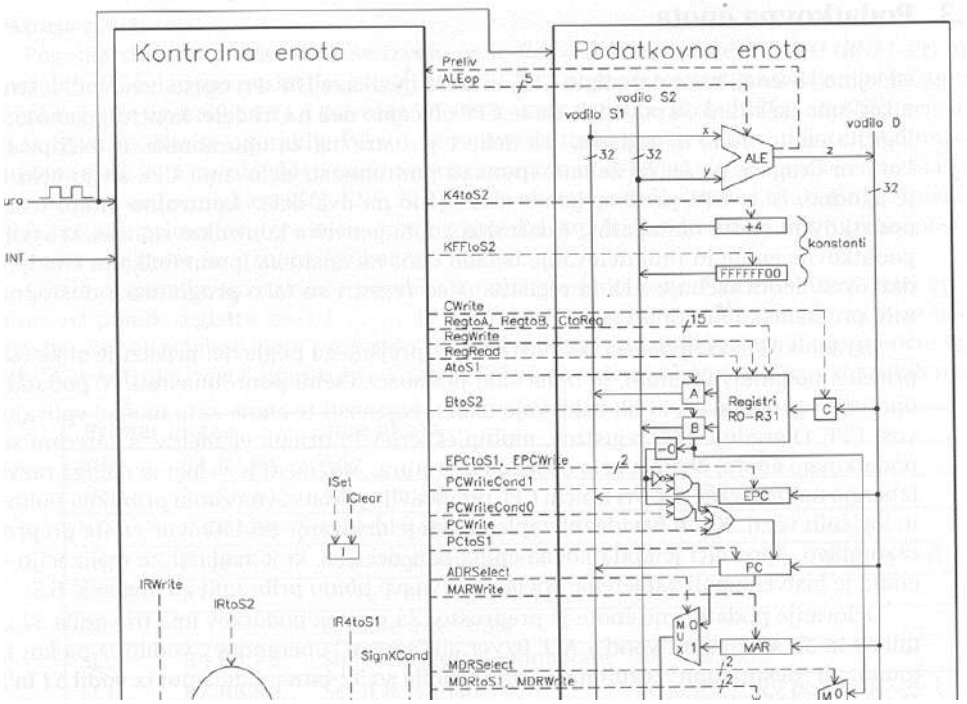
- Perioda je navzdol omejena z zakasnitvami kombinacijskih vezij
  - če bi bila krajša, se novo stanje ne bi imelo časa vzpostaviti
  - zato je frekvenca omejena navzgor
- Opcija so tudi asinhronska sekvenčna vezja
  - hitrejša (ni ure)
    - MIPS R3000 4x hitrejši kot sinhronski
  - težavna za načrtovanje

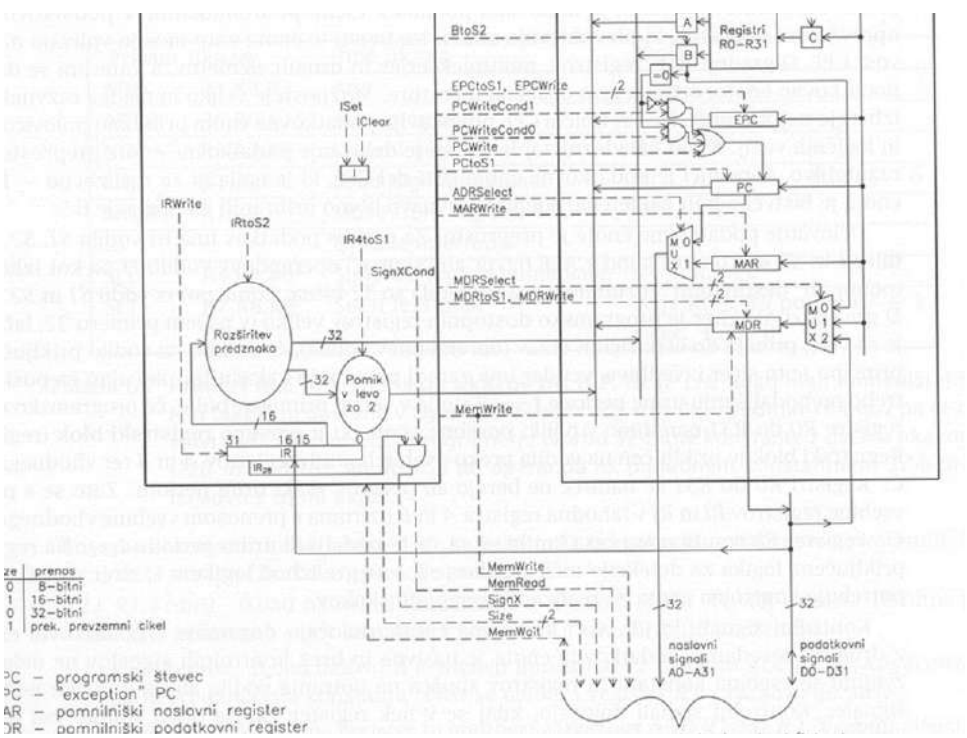
Podatkovna enota

- CPE lahko razdelimo na dva dela:
  - **kontrolna enota** (control unit), KE
    - generira kontrolne signale, ki vodijo delovanje (podatkovne enote, pomnilnika, V/I)
  - **podatkovna enota** (datapath), PE
    - ALE, registri

CPE (HIP)




Kontrolna enota
Podatkovna enota
Preliv
ALEop
vodilo S1
vodilo S2
vodilo D
ALE
K4toS2
+4
konstanti
KFFtoS2
FFFFFF00
CWrite
RegtoA, RegtoB, CtoReg
RegWrite
RegRead
AtoS1
BtoS2
Registri R0-R31
EPCtoS1, EPCWrite
PCWriteCond1
PCWriteCond0
PCWrite
PCtoS1
EPC
PC
ADRSelect
MARWrite
MAR
MDRSelect
MDRtoS1, MDRWrite
ISet
IClear
IRWrite
IRtoS2
IR4toS1
SignXCond
INT


BtoS2
Registri R0-R31
EPCtoS1, EPCWrite
PCWriteCond1
PCWriteCond0
PCWrite
PCtoS1
EPC
PC
ADRSelect
MARWrite
MAR
MDRSelect
MDRtoS1, MDRWrite
MDR
ISet
IClear
IRWrite
IRtoS2
IR4toS1
SignXCond
Razširitev predznaka
Pomik v levo za 2
MemWrite
MemRead
SignX
Size
MemWait
naslovni signali A0-A31
podatkovni signali D0-D31
ze | prenos
0 | 8-bitni
1 | 16-bitni
0 | 32-bitni
1 | prek. prevzemni cikel
PC – programski števec
PC – "exception" PC
AR – pomnilniški naslovni register
DR – pomnilniški podatkovni register

Podatkovna enota

- 3 (32-bitna) vodila (za prenos podatkov):
  - *S1* in *S2:* vhod v ALE
  - *D:* izhod iz ALE
- Vse ALE operacije se izvršijo v eni urini periodi
- 32-bitni konstanti, priključeni na *S2*:
  - +4 (za povečevanje *PC*) in
  - FFFFFF00 (za izjeme)

Registrski blok

- Registrski blok (register file): registri *R0* do *R31*
  - priključen na vodila preko izhodnih registrov A in B ter vhodnega registra C
  - če bi registre *R0* do *R31* direktno priključili na vodilo, bi dobili večje zakasnitve in s tem daljšo urino periodo t_{CPE}
  - na register B je priključena logika za detekcijo ničle
- Kontrolni signali za registrski blok
  - RegtoA izbere register, ki se bo prebral v A, RegtoB ...
  - CtoReg določa register, kamor se bo vpisala vsebina C
  - prenos sprožita signala
    - RegRead (prenos v A in B) in RegWrite (prenos iz C)
  - pisanje ob aktivni fronti ure
- Signali AtoS1, BtoS2, PCtoS1 prenašajo vsebine registrov A, B in PC na vodili S1 in S2
- Signali za pisanje v registre: Cwrite, PCWrite, MDRWrite, ...

- ADRSelect:
  - ADRSelect = 0: PC na naslovno vodilo
  - ADRSelect = 1: MAR na naslovno vodilo
- MDRSelect podobno
- Signali za dostop do pomnilnika
  - z MemWrite in MemRead se izbere pisanje ali branje
  - Size (2-bitni) pove, ali se prenaša 8, 16 ali 32 bitov
  - SignX pri pretvorbah v 32 bitov:
    - SignX = 1: razširitev predznaka
    - SignX = 0: razširitev ničle
  - MemWait: čakanje na pomnilnik pri zgrešitvi v predpomnilniku

PC in EPC

- Za uporabnika sta vidna še programski števec PC in EPC
  - PCtoS1 prepušča vsebino PC na vodilo S1
  - PCWrite sproži pisanje iz vodila D v PC
  - izhod PC je priključen tudi na MUX, preko katerega lahko pride na pomnilniške naslovne signale A0-A31
- EPC shranjuje PC ob prekinitvah in pasteh (izjeme - exceptions)
  - EPCtoS1, EPCWrite
  - 1-bitni register I je v kontrolni enoti (pri I = 0 so prekinitve onemogočene)

MAR in MDR

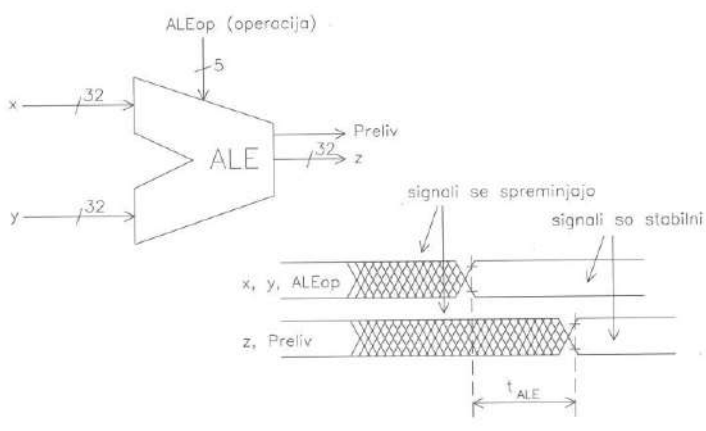
- MAR (Memory address register) in MDR (Memory data register) služita komunikaciji s pomnilnikom
  - MDR je vezan na S1 in na pomnilniške podatkovne signale D0-D31
    - MDRtoS1
    - MemWrite
  - Pisanje v MDR (MDRWrite). Z MDRSelect (2 bita) izberemo vhod
    - pomnilnik (D0-D31)
    - vodilo D
    - register B
  - MAR je vezan preko MUXa na pomnilniške naslove A0-A31
  - MARWrite: D v MAR

- Ukazni register IR je v kontrolni enoti
- Zakaj imamo 3 vodila?
  - želimo, da se ALE operacije izvršijo v eni periodi ure
- Zakaj pa imamo sploh vodila in ne kar dvotočkovnih povezav?
  - porabili bi mnogo več elementov in s tem prostora na čipu

ALE

- ALE operacija se izvrši na 32-bitnih vhodnih operandih *x* in *y*, ki sta na vodilih *S1* in *S2*
- Rezultat je 32-bitni izhod *z* (pojavi se na vodilu *D*) in *Preliv*, ki je speljan v kontrolno enoto
- Signali ALEop (5 bitov) pridejo iz kontrolne enote
  - ta jih tvori iz bitov operacijske kode (in njenega podaljška *func*) ukaza
  - pri HIP ukazih za spodnje 4 bite ALEop lahko uporabimo kar spodnje 4 bite operacijske kode (biti 26-29 v registru *IR*)

ALE



ALE operacije (1)

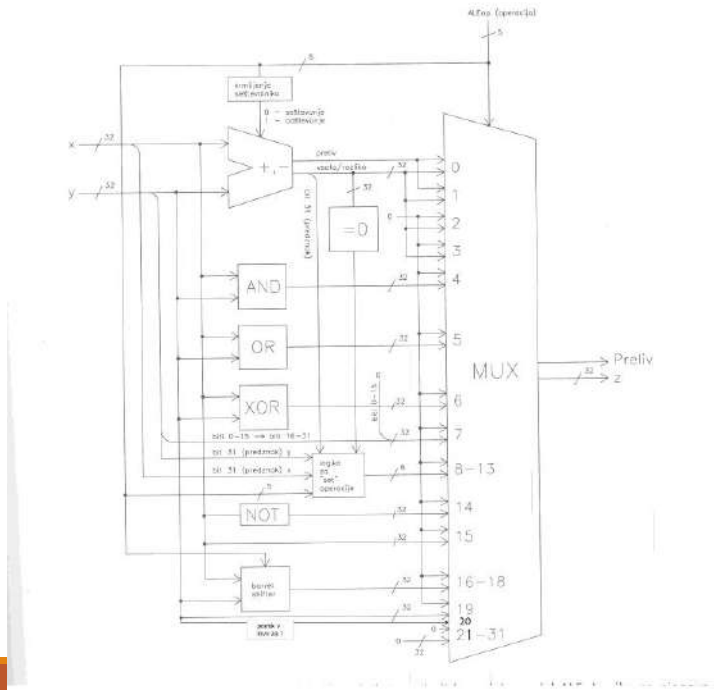
Št.	ALEop	Operacija	Opis
0	00000	ADD	z = x + y in Preliv
1	00001	SUB	z = x – y in Preliv
2	00010	ADDU	z = x + y
3	00011	SUBU	z = x – y
4	00100	AND	z = xy
5	00101	OR	$z = x \vee y$
6	00110	XOR	$z = x \oplus y$
7	00111	LHI	$z = y \times 2^{16}$
8	01000	SEQ	če je x = y, potem z = 1, sicer z = 0
9	01001	SNE	če je $x \neq y$, potem z = 1, sicer z = 0
10	01010	SLT	če je x < y, potem z = 1, sicer z = 0
11	01011	SGT	če je x > y, potem z = 1, sicer z = 0

ALE operacije (2)

Štev.	ALEop	Operacija	Opis
12	01100	SLTU	če je $x \leq y$, potem z = 1, sicer z = 0
13	01101	SGTU	če je $x \geq y$, potem z = 1, sicer z = 0
14	01110	NOT	z = not x
15	01111	S1 → D	z = x (edina pot iz S1 in S2 na D gre skozi ALE)
16	10000	SLL	z = x, logično pomaknjen za y mest v levo
17	10001	SRL	z = x, logično pomaknjen za y mest v desno
18	10010	SRA	z = x, aritmetično pomaknjen za y mest v desno
19	10011	S2 → D	z = y (edina pot iz S1 in S2 na D gre skozi ALE)
20	10100	2*S2 → D	z = 2y
21	10101	ZER	z = 0
22..31	10110 do 11111	rezervirano	z = 0

Najbolj zapleteni operaciji za realizacijo sta seštevanje in odštevanje (še posebno pri Carry Lookahead)

Zgradba ALE pri HIP



Koraki pri izvrševanju ukazov

- Izvrševanje ukazov razdelimo na 5 korakov:
  1. Prevzem ukaza (IF – Instruction Fetch)
  2. Dekodiranje ukaza (ID – Instruction Decode)
  3. Izvrševanje operacije (EX - Execute)
  4. Dostop do pomnilnika (MEM - Memory)
  5. Shranjevanje rezultata (WB – Write Back)
- En korak traja eno ali več urinih period
- Vsi ukazi ne potrebujejo vseh 5 korakov

1. korak: Prevzem

1. Prevzem ukaza

$$IR \leftarrow_{32} M[PC]$$

- Vsebina *PC* se preko MUXa prenese na pomnilniški naslov
- Iz pomnilnika se prebere 32 bitov in prenese v *IR*
  - Vsebina *PC* je vedno mnogokratnik 4
- Prenos traja v primeru zadetka v predpomnilniku 1 periodo, v primeru zgrešitve pa 11 period
  - dokler je aktiven MemWait, kontrolna enota čaka

2. korak: Dekodiranje

2. Dekodiranje ukaza

A ← *Rs1*;
B ← *Rs2* (ali *Rd*);
PC ← *PC* + 4

- to troje se izvede hkrati
- povečanje *PC* za 4 izvaja ALE
  - na *S1* gre vsebina *PC*
  - na *S2* gre konstanta +4
  - ALE operacija ADDU
  - vsebina *D* gre v *PC*
- korak traja eno periodo

3. korak: Izvrševanje

3. Izvrševanje operacije

- ALE operacija ali računanje dejanskega naslova

3.1 **Ukazi za prenos podatkov** (load/store)

$MAR \leftarrow A + \text{raz}(IR_{0..15})$;
$MDR \leftarrow B$

- razširjeni $IR_{0..15}$ je odmik (z IRtoS2 gre na *S2*)
- *A* (*Rs1*) gre na *S1* (AtoS1)
- ALE operacija ADDU
- rezultat *D* gre v *MAR*
- hkrati gre *B* v *MDR* (pri branju ni potrebno, vendar ne škodi, obenem pa poenostavlja načrtovanje kontrolne enote)
- porabi se 1 perioda

3.2 **Klic procedure** (ukaz CALL)

$C \leftarrow PC$.
$PC \leftarrow A + \text{raz}(IR_{0..15})$

- 2 periodi:
  - v prvi se *PC* preko *S1*, ALE ($S1 \rightarrow D$) in *D* zapiše v *C*
  - v drugi se vsota *A* in odmika zapiše v *PC*

3.3 **ALE ukazi**

$C \leftarrow A$ op B ali $C \leftarrow A$ op $\text{raz}(IR_{0..15})$

(glede na format (2 ali 1))

- *A* je *Rs1*
- *Rs2* je *B* (format 2) ali pa takojšnji operand $IR_{0..15}$ (format 1)
- Kontrolna enota aktivira BtoS2 oz. IRtoS2
- Traja 1 periodo

3.4 **Ukaz TRAP**

$EPC \leftarrow PC; \qquad I \leftarrow 0$

$MAR \leftarrow \text{FFFFFF00} + 4 \times \text{raz}(IR_{0..15})$

- naslov servisnega programa je na naslovu FFFFFF00 + 4 × *n*
- 2 periodi:
  - v prvi gre *PC* preko *S1*, ALE (*S1* → *D*) in *D* v *EPC,* 0 pa gre v *I*, da se onemogočijo prekinitve
  - v drugi se sešteje konstanta in IR (številka vektorja *n*), pomaknjen za 2 v levo

3.5 **Brezpogojni skok** (ukaz J)

$PC \leftarrow A + \text{raz}(IR_{0..15})$

- zapis v *PC* z PCWrite
- 1 perioda

3.6 **Pogojni skoki** (ukaza BEQ in BNE)

BEQ: če je $B = 0$, potem $PC \leftarrow PC + \text{raz}(IR_{0..15})$

BNE: če je $B \neq 0$, potem $PC \leftarrow PC + \text{raz}(IR_{0..15})$

- Kot pogoj se uporablja vsebina *Rd*, ki se je v koraku 2 prenesel v *B*
- Na *B* je priključena logika za ugotavljanje ničle
- PCWriteCond0 piše v *PC*, če je $B = 0$
- PCWriteCond1 piše v *PC*, če je $B \neq 0$
- Logika z vrati AND in OR (na sliki) določa vpis v *PC*:
  - $\text{PCWriteCond0} \cdot (B=0) \vee \text{PCWriteCond1} \cdot (B\neq 0) \vee \text{PCWrite}$
- 1 perioda

3.7 **Ukaz RFE**

za vrnitev iz prekinitve ali pasti

$PC \leftarrow EPC$

- 1 perioda

3.8 **Ukaza MOVER in MOVRE**

- MOVER prenese vsebino *EPC* v *Rd*, MOVRE pa vsebino *Rs1* v *EPC*

MOVER: $C \leftarrow EPC$

MOVRE: $EPC \leftarrow A$

3.9 **Ukaza EI in DI**

- omogočanje/onemogočanje prekinitev
- signala Iset in Iclear postavita register *I* na 1 oz. na 0

EI: $I \leftarrow 1$

DI: $I \leftarrow 0$

4. korak: Dostop

4. Dostop do pomnilnika

ukazi load in TRAP: $MDR \leftarrow M[MAR]$

ukazi store: $M[MAR] \leftarrow MDR$

- naslov se je v prejšnjem koraku prenesel v *MAR*
- 1 perioda + morebitne čakalne periode

5. korak: Shranjevanje

5. Shranjevanje rezultata

5.1 **ALE ukazi, CALL in MOVER**

$Rd \leftarrow C$

5.2 **Ukaz TRAP**

$PC \leftarrow MDR$

- naslov servisnega programa, ki se je v koraku 4 prebral iz pomnilnika v *MDR*, gre preko ALE ($S2 \rightarrow D$) v *PC*

5.3 **Ukazi load**

$C \leftarrow MDR$ (preko ALE)

$Rd \leftarrow C$

- 2 periodi

Čas izvrševanja ukazov

- Čas izvrševanja različnih ukazov je različen
  - nekateri ukazi ne potrebujejo vseh korakov
  - trajanje nekaterih korakov se od ukaza do ukaza lahko razlikuje
- Čas izvrševanja je odvisen tudi od časa za dostop do pomnilnika, ta pa od pogostosti zgrešitev v predpomnilniku
- Vsi ukazi potrebujejo en dostop do pomnilnika (za prevzem ukaza), ukazi za prenos podatkov (load in store) in TRAP pa še enega

- **Število urinih period na ukaz** (pri HIP)
  - najmanjše
  - povprečno (upošteva tudi zgrešitve v predpomnilniku)

Vrsta ukazov	Število pomnilniških dostopov	Najmanjše število urinih period na ukaz	Povprečno število urinih period na ukaz
Load	2	6	7
Store	2	4	5
TRAP	2	6	7
ALE	1	4	4,5
J, BEQ, BNE	1	3	3,5
CALL	1	5	5,5
MOVER	1	4	4,5
MOVRE, EI, DI	1	3	3,5

Povprečno število urinih period na ukaz

- Povprečno število urinih period pa upošteva še povprečno število čakalnih urinih period, ki so zaradi zgrešitev v predpomnilniku potrebne pri dostopih do pomnilnika
  - Povprečno št. period = najmanjše št. period + povprečno št. čakalnih period (št. čakalnih period × verjetnost zgrešitve × št. pom. dostopov)
    - pri vsaki zgrešitvi je 10 čakalnih urinih period
  - če je verjetnost zadetka v predpomnilniku 95%, je povprečno število čakalnih urinih period enako 10 × 0,05 × število pomnilniških dostopov

Povprečno število urinih period na ukaz za vse ukaze skupaj

- **CPI** (Clocks Per Instruction)

$$CPI = \sum_{i=1}^{n} CPI_i \cdot p_i$$

  - CPI_i je število urinih period za ukaz vrste i
  - p_i je relativna pogostost (verjetnost) posamezne vrste ukaza
  - če za CPI_i vzamemo najmanjše možno število urinih period, dobimo $CPI_{idealni}$, ki ne vključuje izgubljenih urinih period zaradi zgrešitev v predpomnilniku

- Pogostost posameznih skupin ukazov je precej odvisna tudi od programa, ki ga poganjamo
  - Npr. prevajalnik za C uporablja 46% ALE ukazov, 36% ukazov za prenos podatkov in 18% kontrolnih ukazov
  - Če predpostavimo, da je število load ukazov trikrat večje od števila store ukazov in da je 5% od vseh skokov klicev procedur, dobimo:

$CPI_{idealni}$ = 5,5*0,36 + 4*0,46 +
(3*0,95 + 5*0,05)*0,18 = 4,38

- Če upoštevamo še 95% verjetnost zadetka, dobimo

$CPI_{resnični}$ = 6,5*0,36 + 4,5*0,46 + (3,5*0,95 + 5,5*0,05)*0,18
= 5,06
= $CPI_{idealni}$ + 0,68

- Obstaja tudi parameter **MIPS** (Million Instructions Per Second):

$$MIPS = \frac{f_{CPE}}{CPI \cdot 10^6} = \frac{1}{CPI \cdot t_{CPE} \cdot 10^6}$$

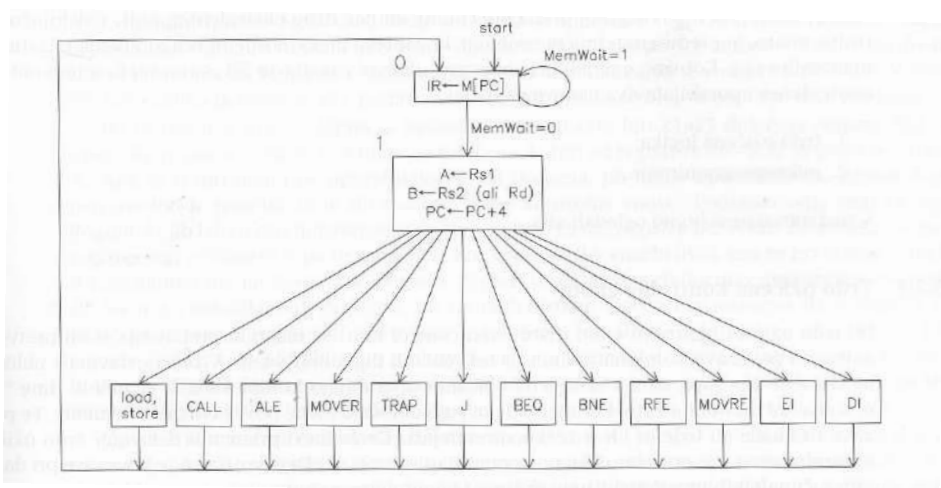
- Na MIPS vpliva frekvenca ure
  - pri 2GHz bi dobili za prevajalnik za C 395,2 MIPS

Kontrolna enota

- Podatkovna enota je pasivna
  - naredi samo tisto, kar od nje zahtevajo kontrolni signali
- Kontrolna enota (KE) mora vedeti za vsak ukaz, kateri koraki so potrebni in katere signale je treba aktivirati v določeni periodi
  - KE je zapletena
    - večina napak pri razvoju novega računalnika je v KE
- Delovanje KE lahko podamo z **diagramom prehajanja stanj** (DPS)
  - med stanji se seveda prehaja ob aktivni fronti ure
- Temu ustreza **končni avtomat** (finite state machine)

- V vsakem stanju sta definirani dve funkciji:
  1. **Funkcija naslednjega stanja.**
     - določa, pri katerih pogojih (vhodni signali) se izvrši prehod v vsako od možnih stanj
  2. **Funkcija izhodnih signalov**
     - določa, kateri izhodni signali so v danem stanju aktivni

Poenostavljen diagram prehajanja stanj

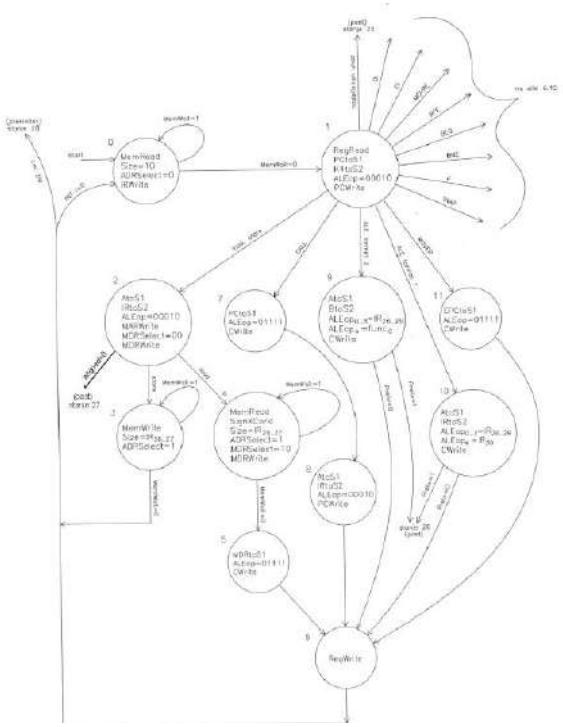


- CPE lahko izvršuje ukaze na 2 načina (to vpliva na realizacijo kontrolne enote):
  1. **Trdo ožičena logika**
     - vezje (logična vrata, pomnilne celice, povezave)
     - spremembe so možne le s fizičnim posegom
  2. **Mikroprogramiranje**
     - pri vsakem ukazu se aktivira ustrezno zaporedje **mikroukazov** (**mikroprogram**)
     - mikroprogrami so shranjeni v kontrolnem pomnilniku CPE
     - mikroukazi so primitivnejši od običajnih in jih izvršuje trdo ožičena logika
     - počasnejše, vendar lahko spreminjamo ali dodajamo ukaze, ne da bi spreminjali vezje
- Uporabnika način izvajanja ukazov v resnici ne zanima

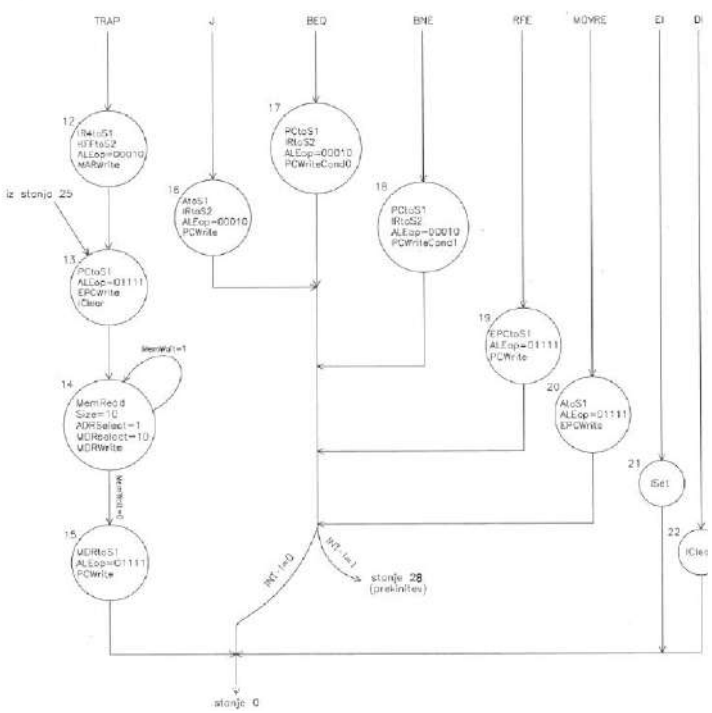
Trdo ožičena kontrolna enota

- Avtomat realiziramo s kombinacijskimi in sekvenčnimi vezji
  - npr. vrata in flip-flopi
- Trdo ožičena pomeni, da so povezave fiksne
  - če hočemo kaj spremeniti, jih je potrebno fizično spremeniti
- Najprej potrebujemo popoln DPS

DPS,
1.del

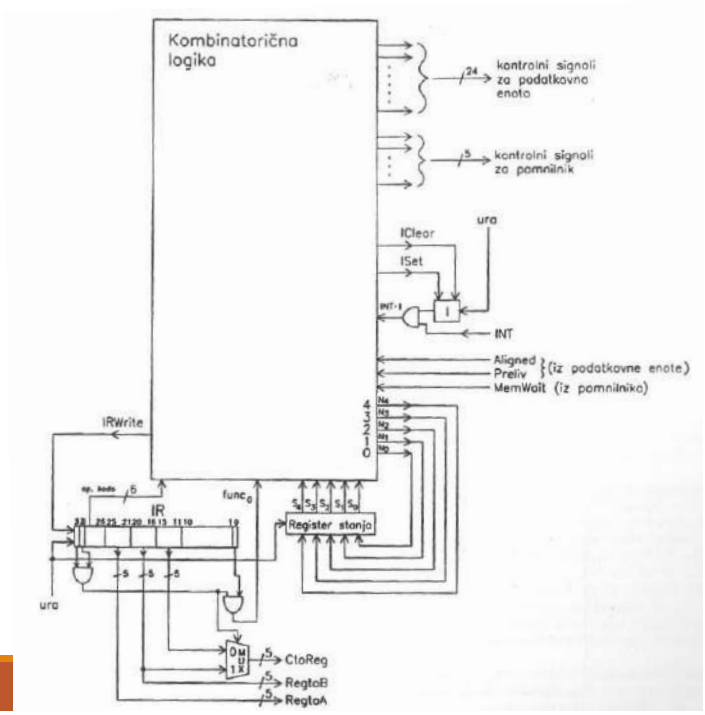


DPS,
2.del



- DPS ima 28 stanj
  - 5-bitni register stanja
  - kombinacijsko logiko, ki iz stanja, registra IR in vhodnih signalov MemWait, Preliv in INT (pogojen z *I*) tvori naslednje stanje in izhodne signale
- Zaradi preproste zgradbe ukazov lahko precejšen del bitov IR peljemo mimo KE v PE
  - npr. biti 21-25 določajo register *Rs1* in se uporabljajo samo v koraku 1
    - določajo, kateri od registrov *R0-R31* se prebere v *A*
    - lahko jih uporabimo za signale RegtoA neposredno iz *IR*
  - bite 16-20 registra *IR* lahko peljemo direktno na RegtoB
  - pri CtoReg potrebujemo še dodaten MUX, kajti
    - register *Rd* (kamor se piše iz *C*) določajo v formatu 1 biti 16-20, v formatu 2 pa biti 11-15
    - zato je treba izbirati med dvema 5-bitnima signaloma $IR_{11..15}$ (če sta $IR_{30..31}$ oba 1) in $IR_{16..20}$ (sicer)
- Kontrolna enota ima 39 izhodnih signalov

Realizacija KE



- Primer: izhodni signal IRWrite je aktiven samo v stanju 0, zato ga definira funkcija

  $$IRWrite = S_4'S_3'S_2'S_1'S_0' ,$$

  kjer so $S_{0..4}$ biti stanja

- Primer: izhodni signal PCtoS1 je aktiven v stanjih 1, 7, 13 in 24, zato ga definira funkcija

  $$PCtoS1 = S_4'S_3'S_2'S_1'S_0 + S_4'S_3'S_2S_1S_0 + S_4'S_3S_2S_1'S_0 + S_4S_3S_2'S_1'S_0'$$

- Za realizacijo kombinacijske logike se pogosto uporabljajo kar bralni pomnilniki (ROM) ali PAL

Mikroprogramska kontrolna enota

- Pri nekaterih računalnikih je število ukazov, formatov in načinov naslavljanja lahko zelo veliko
  - za Intelove procesorje 80x86 bi potrebovali več tisoč stanj
- **Mikroprogramska kontrolna enota** je narejena kot majhen računalnik
  - diagram prehajanja stanj se pretvori v **mikroprogram**
    - vsako stanje je en **mikroukaz**
  - mikroprogram je shranjen v bralnem pomnilniku (ROM)
    - **firmware**
    - pri HIP zadostuje ROM s 512 lokacijami (9-bitni naslov)
  - za vsak ukaz obstaja majhen mikroprogram (iz mikroukazov, ki so bolj primitivni)
  - pri potencialnem spreminjanju ukazov je mnogo lažje spremeniti mikroprogram, kot pa vezje

- Format mikroukazov pri HIP:

48 0

ALEop	S1	S2	Wreg	R	Mux	Pom-nilnik	Next	Skočni naslov
5	5	4	10	2	3	5	6	9

- Pri mikroprog. KE vsak ukaz traja 1 urino periodo
- Mikroukazi aktivirajo kontrolne signale
  - pri HIP večina bitov mikroukaza predstavlja signale (vsak enega)
- Mikroprog. števec μPC je 9-biten, mikroprog. pomnilnik velikosti 512 x 49
  - ukazov nekaj čez 50, povprečen ukaz rabi ≈ 2 mikroukaza

- Polja v mikroukazu (vsak bit predstavlja en signal):
  - ALEop:
    - ALE operacija v tej urini periodi
      - tudi pri vseh ostalih poljih gre za trenutno urino periodo, zato tega ne bomo posebej omenjali
  - S1:
    - določa, kateri register gre na vodilo S1
    - Biti: AtoS1, EPCtoS1, PCtoS1, MDRtoS1, IR4toS1 (le en bit lahko 1)
  - S2:
    - določa, kateri register ali konstanta gre na vodilo S2
    - Biti: K4toS2, KFFtoS2, BtoS2, IRtoS2 (le en bit lahko 1)
  - Wreg:
    - delovni register, v katerega se piše
    - Biti: Cwrite, EPCWrite, PCWrite, PCWriteCond0, PCWriteCond1, MARWrite, MDRWrite, IRWrite, ISet, IClear

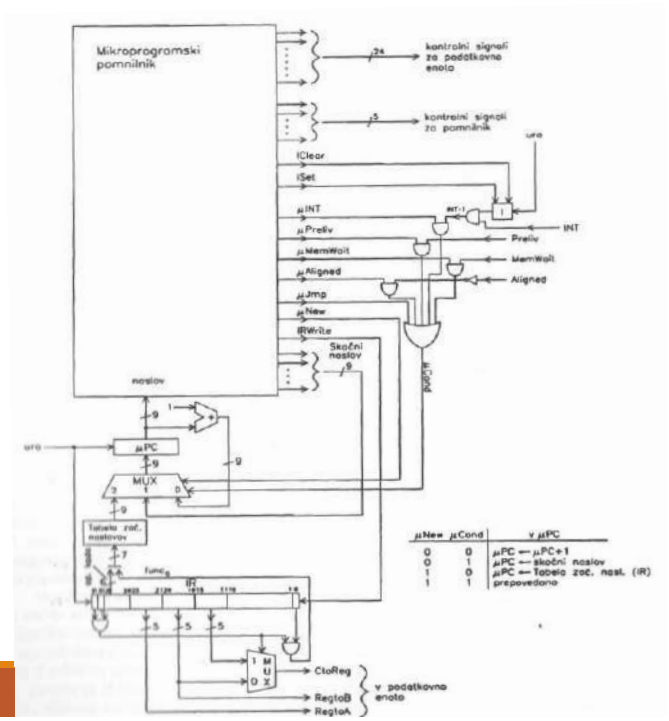
- R:
  - določa, ali se bere ali piše v registre R0-R31
  - Bita: RegWrite, RegRead
- Mux:
  - določa krmiljenje obeh mux
  - Biti: ADRSelect, MDRSelect
- Pomnilnik:
  - določa, kaj se dogaja s pomnilnikom
  - Biti: MemRead, MemWrite, SignXCond, Size
- Next:
  - določa način izbire naslova naslednjega ukaza
  - Biti: μNew, μJmp, μINT, μPreliv, μAligned, μMemWait
    - največ eden na 1
- Skočni naslov:
  - naslov, ki se bo zapisal v μPC, če tako določajo biti v Next
  - Biti: naslov v mikroprog. pomnilniku

- Določanje naslova naslednjega mikroukaza:
  1. Če μNew = 1, se v μPC iz tabele začetnih naslovov (dispatch table) prenese naslov prvega mikroukaza novega mikroprograma
     - tabela preslika $IR_{26..31}$ in IR_0 v 9-bitni mikroprogramski naslov
     - zaradi 7 bitov je lokacij v tabeli precej več kot ukazov
       - nedefinirani ukazi se v tabeli preslikajo v mikroprogram, ki sproži past
  2. Če μJmp = 1, brezpogojni skok na Skočni naslov
     - sicer se izračuna logična funkcija μINT · INT + μPreliv · Preliv + μAligned · Aligned' + μMemWait · MemWait
       - če 1, skok na Skočni naslov
  3. Sicer μPC ← μPC + 1

Primeri mikroprogramov za ukaze LW, SH, SUB, ADDI in ADDU

št.	Oznaka	ALEop	S1	S2	Wreg	R	Mux	Pomnilnik	Next	Skočni naslov
1	**LW:**	ADDU	AtoS1	IRtoS2	MARWrite					
2	LW1:				MDRWrite		ADRSelect=1 MDRSelect=10	MemRead Size=10	µMemWait	LW1
3		S1 → D	MDRtoS1		Cwrite					
4	regw:					RegWrite				
5	fetch:				IRWrite		ADRSelect=0	MemRead Size=10	µMemWait	fetch
6		ADDU	PCtoS1	K4toS2	PCWrite	RegRead			µNew	
1	**SH:**	ADDU	AtoS1	IRtoS2	MARWrite MDRWrite		MDRSelect=00			
2	SH1:						ADRSelect=1	MemWrite Size=01	µMemWait	SH1
3	SH2:				IRWrite		ADRSelect=0	MemRead Size=10	µMemWait	SH2
4		ADDU	PCtoS1	K4toS2	PCWrite	RegRead			µNew	
1	**SUB:**	SUB	AtoS1	BtoS2	CWrite				µJmp	regw
1	**ADDI:**	ADD	AtoS1	IRtoS2	CWrite				µJmp	regw
1	**ADDU:**	ADDU	AtoS1	BtoS2	CWrite				µJmp	regw

Mikroprog-ramska KE



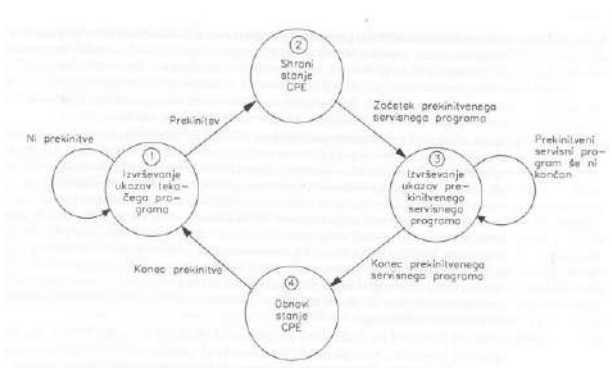
- Ker je mikroprog. pomnilnik najdražji del mikroprog. KE, so želeli zmanjšati
  - širino mikroukazov
    - kodiranje (namesto 1-od-N)
    - poleg tega:
      - določena polja niso vedno aktivna,
      - določena niso aktivna naenkrat, ...
  - število mikroukazov
- Delitev glede na širino ukazov oz. stopnjo kodiranja:
  1. **Horizontalno mikroprogramiranje**
     - malo (ali nič) kodiranja
     - dolgi mikroukazi
     - hitrejše, dražje
  2. **Vertikalno mikroprogramiranje**
     - veliko kodiranja
     - kratki mikroukazi (a bolj številni)
     - počasnejše, cenejše

Prekinitve in pasti

- **Prekinitev** (interrupt) je dogodek, ki povzroči, da CPE začasno preneha izvajati tekoči program ter prične izvajati t.i. **prekinitveni servisni program** (PSP)
  - Zahteva za prekinitev pride v CPE od zunaj, npr. od neke vhodno/izhodne naprave
- **Past** (trap) je posebna vrsta prekinitve, ki jo zahteva sama CPE ob nekem nenavadnem dogodku ali celo na zahtevo programerja
  - pasti pridejo od znotraj
- Če ne bi bilo prekinitev in pasti, bi morala CPE stalno preverjati stanje mnogih naprav

- Primeri uporabe:
  - zahteve V/I naprav ob različnih dogodkih
  - napaka v delovanju nekega dela računalnika
  - aritmetični preliv
  - napaka strani ali segmenta (pri navideznem pomnilniku)
  - dostop do zaščitene pomnilniške besede
  - dostop do neporavnane pomnilniške besede
  - uporaba nedefiniranega ukaza
  - klic programov operacijskega sistema

- Pri prekinitvah razlikujemo 4 stanja:
  - Normalno izvrševanje ukazov programa
  - Shranjevanje stanja CPE ob pojavu zahteve za prekinitev
  - Skok na prekinitveni servisni program in njegovo izvajanje
  - Vrnitev iz prekinitvenega servisnega programa in obnovitev stanja CPE



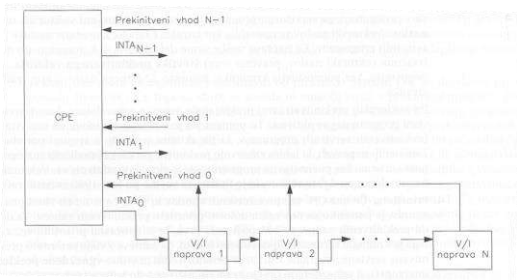
- 5 dejavnikov:

1. **Kdaj CPE reagira na prekinitveno zahtevo**
   - najenostavneje je po izvrševanju tekočega ukaza
     - v tem primeru se mora ohraniti samo stanje programsko dostopnih registrov (*R0-R31, PC, EPC, I*)
   - programer lahko onemogoči odziv CPE na prekinitvene zahteve (bit *I*, ukaza DI in EI)
     - po vklopu so V/I prekinitve onemogočene, dokler se V/I naprave ne inicializirajo
     - če pride do nove prekinitve, preden prekinitveni servisni program shrani registre, lahko pride do izgube *PC*, ki se ob prekinitvi shrani v *EPC*
2. **Kako zagotoviti "nevidnost" prekinitev**
   - treba je zagotoviti, da je stanje (registrov) CPE enako kot prej

2. **Kje se dobi naslov prekinitvenega servisnega programa**
   - to je pomembno pri prekinitvah, ki prihajajo od zunaj
   - najprej je treba ugotoviti, katera naprava je zahtevala prekinitev
     - če je na vsakem prekinitvenem vhodu samo ena naprava, je problem trivialen
     - drugače je, če je na enem prek. vhodu več naprav, ali če ima ista naprava več PSP
   - najpreprostejše je **programsko izpraševanje** (polling)
     - CPE bere registre vsake V/I naprave, v katerih je bit, ki pove, ali je naprava zahtevala prekinitev
     - če je, izvrši skok na njen prek. servisni program
     - polling je zamuden
   - običajen način pa so **vektorske prekinitve**
     - naprava pošlje v CPE naslov njenega PSP v **prekinitvenem prevzemnem ciklu**, s katerim CPE obvesti V/I naprave, naj pošljejo informacijo o izvoru prekinitve
     - ta naslov se imenuje **prekinitveni vektor** ali **vektorski naslov**, ki se običajno izračuna iz **številke prekinitvenega vektorja** po nekem pravilu (tu zadošča npr. že 8-bitno število)
     - možno je tudi, da ima ena naprava več PSP

4. Prioriteta

- če ima CPE več prek. vhodov in več naprav na posameznem vhodu, potrebujemo neko prioriteto.
- **vgnezdene prekinitve** (nested interrupts), pri katerih zahteve z višjo prioriteto prekinejo prek. servisne programe z nižjo prioriteto
- **prekinitveni krmilnik** omogoča računalnikom, ki imajo CPE z enim samim bitom za omogočanje prekinitev, bolj fleksibilno obravnavo prioritete
- določanje prioritete je možno izvesti tudi z **marjetično verigo** (daisy chain): naprava, ki ni zahtevala prekinitve, spusti določen signal v naslednjo napravo, tista pa, ki jo je, zapre signalu pot in vrne CPE ustrezno informacijo, da jo CPE lahko prepozna



5. Potrjevanje prekinitve

- potrebno zato, da naprava spusti prekinitveni vhod (da se prekinitev ne servisira večkrat)
- dva načina:
  - programsko: prekinitveni servisni program piše v nek register krmilnika naprave
  - strojno: z nekim signalom (ali kombinacijo večih) se obvesti napravo

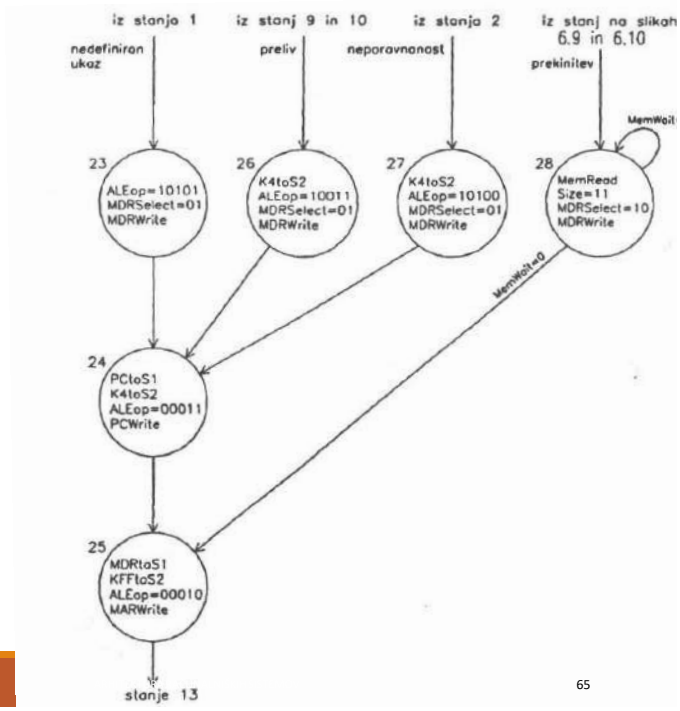
Prekinitve in pasti pri HIP

- HIP ima en sam prekinitveni vhod INT in uporablja vektorske prekinitve
- CPE se odzove na prekinitveno zahtevo s prekinitveno prevzemnim ciklom
  - podobno branju iz pomnilnika, le da signal Size=11 povzroči, da se pomnilnik ne odzove
  - V/I naprave pa se odzovejo tako, da tista z najvišjo prioriteto da na podatkovne signale D0-D7 s 4 pomnoženo številko vektorja *n*
    - to je 6-bitno število
    - *n* gre na D2-D7, ker je množenje s 4 enako pomiku za 2 bita, D0 in D1 pa gresta na 0
    - s tem so pomnilniški naslovi FFFFFF00 + 4 x n (vektorski naslovi), na katerih so shranjeni začetni naslovi PSP, vedno večkratniki 4 (poravnanost)

- Poleg prekinitev ima HIP 3 pasti:
  1. Nedefiniran ukaz (*n*=0)
  2. Preliv (pri ADD, ADDI, SUB, SUBI) (*n*=1)
  3. Neporavnan operand (pri 16- ali 32-bitnih operandih pri load oz. store) (*n*=2)
- Poleg tega ima ukaz *TRAP*, ki je programska past
  - program skoči na naslov, ki je shranjen na enem od vektorskih naslovov
- Pri prekinitvi ali pasti se v *PC* naloži 32-bitni naslov, ki je shranjen v pomnilniku na vektorskem naslovu

Številka vektorja n	Vrsta pasti ali prekinitve	Vektorski naslov
0	nedefiniran ukaz	FFFF $FF00_H$
1	preliv	FFFF $FF04_H$
2	neporavnan operand	FFFF $FF08_H$
3-63	V/I naprave	FFFF $FF00_H + 4 \times n$

- Vključitev prekinitev in pasti v DPS



Merjenje zmogljivosti CPE

- Zmogljivost CPE ni isto kot zmogljivost računalnika!
  - vplivata tudi zmogljivost pomnilniškega in V/I sistema
  - zmog. CPE in zmog. računalnika lahko enačimo le, če sta pomnilniški in V/I sistem dovolj zmogljiva (da CPE ne čaka), kar pa je problemsko odvisno
- Za zmogljivost CPE je merodajen čas izvrševanja programa
- Če zanemarimo V/I, je čas izvrševanja programa enak času, ki ga potrebuje CPE (CPE čas)

$$\textit{CPE čas} = \textit{Število ukazov programa} \times \textit{CPI} \times t_{CPE}$$

CPI ... povprečno št. urinih period na ukaz (Clocks Per Instruction)

- Te tri lastnosti so medsebojno odvisne (pa tudi sredstva za njihovo izboljšanje):
  - t_{CPE} (f_{CPE}): odvisna od hitrosti in števila digitalnih vezij, pa tudi od zgradbe CPE
  - CPI: zgradba CPE in ukazi
  - Število ukazov, v katere se prevede program: ukazi in lastnosti prevajalnika
- Posamezna od teh lastnosti ni merilo!
- Čas je seveda odvisen tudi od programa, vhodnih podatkov in velikosti problema

- Marsikdo primerja različne CPE kar na osnovi frekvence ure (f_{CPE})
  - slabo, ker je lahko zelo zavajajoče
  - neka CPE nižje frekvence ima lahko krajše CPE čase kot neka druga CPE višje frekvence
- Pogosto se uporablja **MIPS** (Million Instructions Per Second):

$$MIPS = \frac{1}{CPI \cdot t_{CPE} \cdot 10^6} = \frac{f_{CPE}}{CPI \cdot 10^6}$$

  - Z njim se CPE čas izrazi takole:

$$CPE\ čas = \frac{Število\ ukazov}{MIPS \cdot 10^6}$$

  - Tudi MIPS nezanesljiv
    - odvisen od števila in vrste ukazov
      - pri enostavnejših ukazih je MIPS večji (čeprav jih potrebujemo več)
    - odvisen od programa
    - Meaningless Indication of Processor Speed

- **MFLOPS** (Million FLoating point Operations Per Second)
  - operacije v plavajoči vejici so si (na različnih računalnikih) bolj podobne kot ukazi
  - ima smisel samo za programe, ki uporabljajo operacije v plavajoči vejici
  - proizvajalci so začeli navajati maksimalno (teoretično) zmogljivost
    - dosti večja kot na realnih programih
- **Sintetični "benchmarki"**
  - 1976: Whetstone, Linpack
  - 1984: Dhrystone (brez FP)
  - Quicksort, Sieve, Puzzle, ...
  - proizvajalci tudi tu niso stali križem rok ... ☺
    - npr. "optimizacija prevajalnikov"
- **SPEC** (Standard Performance Evaluation Corporation)
  - več programov, pogosto spreminjanje

7
Paralelizem na nivoju ukazov

1

- 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

2

- 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 kompliciran: 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)

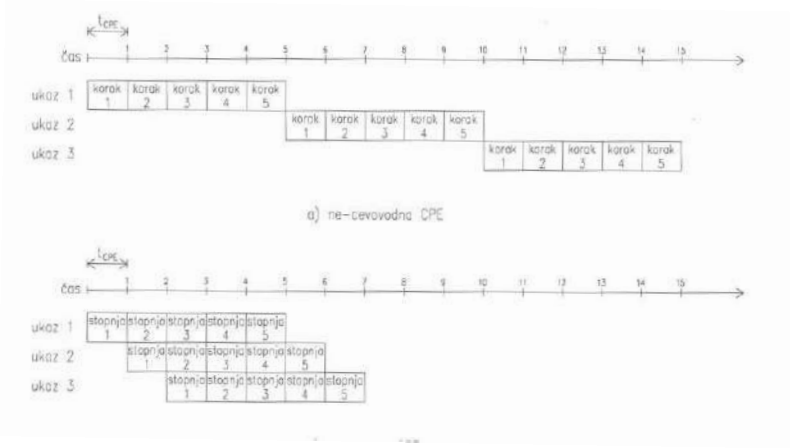
3

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

4

- Primer: ne-cevovodna CPE in cevovodna CPE
  - v drugem primeru se izvrši 5x več ukazov (če zanemarimo začetno zakasnitev)



5

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

6

- Število izvršenih ukazov v danem času se poveča zaradi 2 vzrokov:
  1. manjši CPI
     - čeprav je trajanje ukaza (**latenca**) enako ($N*t_{CPE}$)
  2. krajša t_{CPE}
     - če uspemo narediti enostavne podoperacije
     - $t_{CPE} = t_{shranjevanje} + t_{podoperacija}$

       $t_{shranjevanje}$ ... čas shranjevanja rezultata podoperacije v registre
     - z več stopnjami lahko zmanjšamo $t_{podoperacija}$, $t_{shranjevanje}$ pa ne

7

- 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

8

– BTW: najvišja dosežena frekvenca je nekaj nad 8GHz (AMD)
  » s pomočjo navijanja frekvence (overclocking), pa tudi polivanja čipa s tekočim dušikom

9

- Zakaj se ne uporabljajo poljubno dolgi cevovodi?
  – pač povečujemo N in višamo hitrost CPU

10

- Razlog so **cevovodne nevarnosti** (pipeline hazards), zaradi katerih se mora cevovod ustaviti in počakati, da nevarnost 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!

11

- 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)

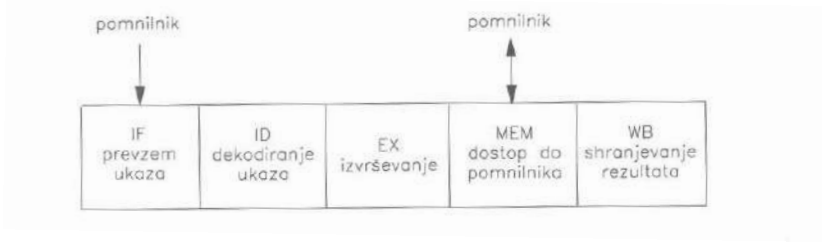
12

Cevovodna podatkovna enota

- Cevovodna realizacija je pri računalnikih RISC enostavnejša kot pri CISC
  - preprostejši ukazi
- Cevovodno CPE si bomo ogledali na primeru računalnika HIP
- 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

13

- Petstopenjska cevovodna podatkovna enota na primeru računalnika HIP:

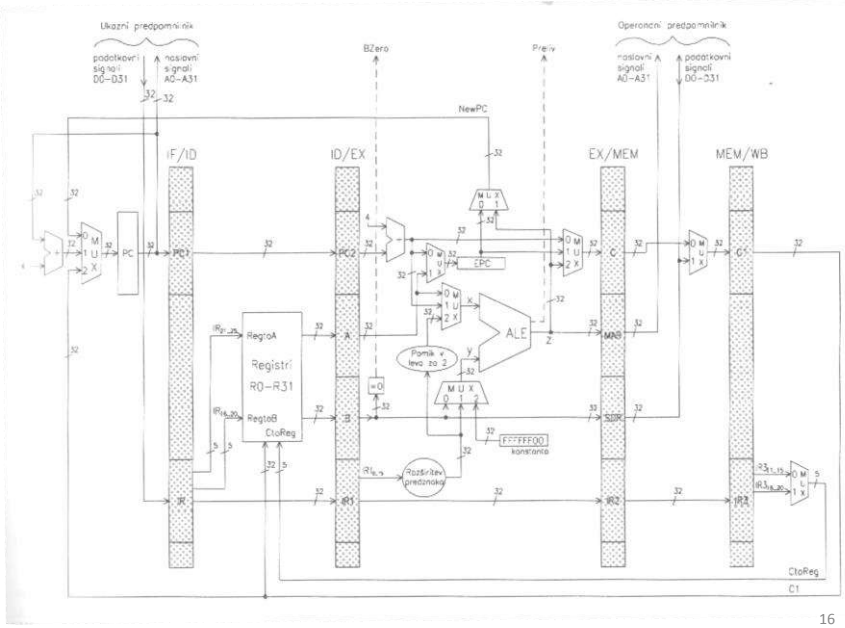


14

- Včasih sta potrebna dva hkratna dostopa do pomnilnika
- Cevovodni HIP ima zato 2 predpomnilnika:
  - ukazni
  - operandni

15

Cevovodna podatkovna enota



16

1. Stopnja IF

$IR \leftarrow_{32} M[PC]$; $PC1 \leftarrow PC$;

$PC \leftarrow$ PC + 4 (ali PC ← newPC ali C1, če zahteva stopnja EX ali stopnja WB)

2. Stopnja ID

IR1 ← IR; PC2 ← PC1;

A ← Rs1; B ← Rs2 (ali Rd)

(Rd ← C1, če zahteva stopnja WB)

17

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:

3.1 **Ukazi za prenos podatkov**

IR2 ← IR1; SDR ← B; MAR ← A + raz($IR1_{0..15}$)

18

3.2 **ALE ukazi**

IR2 ← IR1; SDR ← B;

C ← A op B (ali raz($IR1_{0..15}$))

3.3 **Klic in brezpogojni skok**

IR2 ← IR1; C ← PC2 + 4;

NewPC ← A + raz($IR1_{0..15}$)

3.4 **Pogojni skoki**

IR2 ← IR1; NewPC ← PC2 + 4 + raz($IR1_{0..15}$)

3.5 **TRAP**

IR2 ← IR1; EPC ← PC2 + 4;

MAR ← FFFFFF00 + 4xraz($IR1_{0..15}$); I ← 0

19

3.6 **RFE**

IR2 ← IR1; NewPC ← EPC.

3.7 **MOVER in MOVRE**

IR2 ← IR1;

C ← EPC (pri MOVER); EPC ← A (pri MOVRE)

3.8 **EI in DI**

IR2 ← IR1; I ← 1 (pri EI); I ← 0 (pri DI).

20

4. Stopnja MEM

4.1 **load in TRAP**

IR3 ← IR2; C1 ← M[MAR].

branje iz operandnega PP

4.2 **store**

IR3 ← IR2; M[MAR] ← SDR.

SDR se shrani v operandni PP

4.3 **Ostali ukazi**

IR3 ← IR2; C1 ← C. (zaradi WB)

pri zgrešitvah se ustavijo vse stopnje cevovoda (čakanje na MemWait)

21

5. Stopnja WB

5.1 **ALE ukazi, load, CALL, MOVER**

Rd ← C1.

5.2 **TRAP**

PC ← C1.

22

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

23

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
    - ekv. čakanju eno urino periodo
    - pri višjih prog. jezikih jih vstavlja kar prevajalnik
  - 2 slabosti:
    - potrebno je drugačno programiranje
    - upočasnjeno delovanje
  - ni več posebno aktualno

24

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
- Pri HIP do SN ne prihaja
  - Če PP ne bi bil razdeljen, bi lahko prihajalo (load, store)

25

Podatkovne nevarnosti

- PN (data hazard): kadar ukaz potrebuje kot vhodni operand rezultat še ne dokončanega ukaza
  - medsebojna odvisnost ukazov, ki so blizu skupaj
- Npr.

ADDI	R20, R0, #0	R20 ← 0
SUB	R3, R4, R5	R3 ← R4 – R5
ADD	R1, R3, R6	R1 ← R3 + R6
AND	R2, R3, R7	R2 ← R3 ∧ R7
XOR	R8, R3, R9	R8 ← R3 ⊕ R9
OR	R10, R3, R12	R10 ← R3 ∨ R12

- **Rešitev 1**: **cevovodna zaklenitev** (pipeline interlock)
  - stopnja ID se ustavi za 3 periode (zato se mora tudi IF, ker bi se sicer izgubil ukaz, ki je v njej). EX,MEM in WB morajo delovati naprej (sicer se vzrok za nevarnost ne bo odstranil).

26

- Vsaka stopnja, kjer lahko pride do napake, mora imeti vgrajeno logiko za cev. zaklenitev
  - mehurček (bubble) je strojni ekvivalent operacije NOP
    - v tem primeru ga “izvaja” stopnja EX
    - mehurček potuje od stopnje EX naprej
- Pri HIP lahko pride do PN le v stopnji ID
  - prisotnost nevarnosti se ugotovi s primerjavo bitov $IR_{16..20}$ in $IR_{21..25}$ (Rs1, Rs2) z biti $IRx_{16..20}$ in $IRx_{21..25}$ (format 1, 2), x=1..3
  - to velja le za ukaze, ki shranjujejo v Rd (v stopnjah EX, MEM in WB)
  - mehurček bomo naredili z “ukazom” NOP (32 ničel, ADDI ...)

27

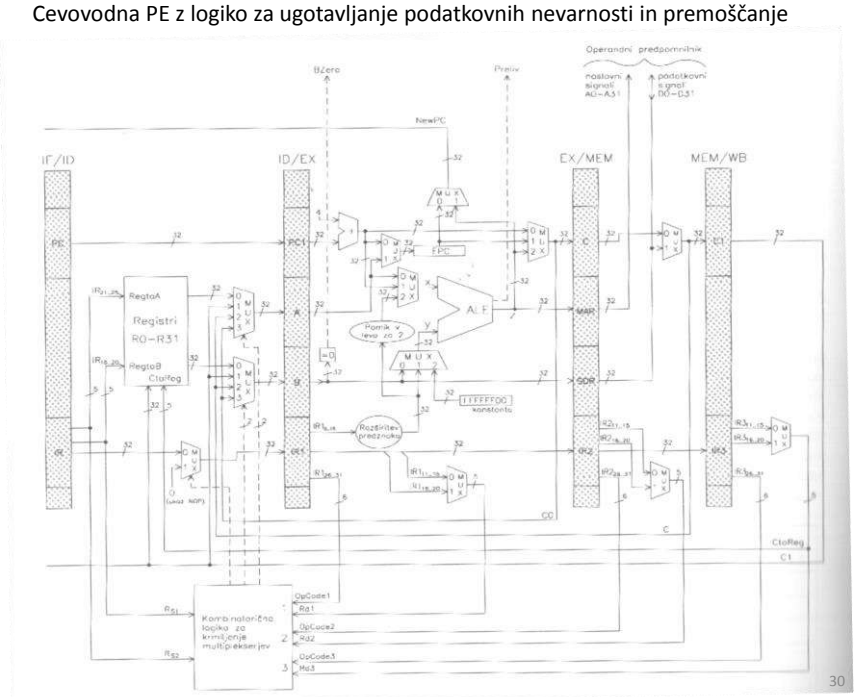
	Urina perioda												
Ukaz	1	2	3	4	5	6	7	8	9	10	11	12	13
ADDI R20,R0,#0	IF	ID	EX	MEM	WB								
SUB R3,R4,R5		IF	ID	EX	MEM	WB							
ADD R1,R3,R6			IF	○	○	○	ID	EX	MEM	WB			
AND R2,R3,R7							IF	ID	EX	MEM	WB		
XOR R8,R3,R9								IF	ID	EX	MEM	WB	
OR R10,R3,R12									IF	ID	EX	MEM	WB

28

- **Rešitev 2**: **premoščanje** (bypassing, data forwarding)
  - rezultat ukaza ADD iz EX prenesemo v ID in ustavljanje ni potrebno
  - vendar premoščanje le iz EX v ID ni dovolj:
    - tudi AND in XOR rabita rezultat ukaza ADD (v R3 ga še ni, v EX pa ga ni več)
  - logika za premoščanje mora omogočati prenos iz stopenj EX, MEM in WB v stopnjo ID
  - PN se ugotavljajo s primerjavo Rs1 in Rs2 v stopnji ID z Rd, ki ga uporabljajo ukazi v stopnjah EX do WB (če gre za ukaze, ki pišejo v Rd)
  - če PN ni mogoče odpraviti (pri HIP možno le pri load), logika ustavi IF, v ID pa vstavi mehurček v IR1

29

Cevovodna PE z logiko za ugotavljanje podatkovnih nevarnosti in premoščanje



30

- Tako se bistveno zmanjša izguba zaradi PN, ni pa popolnoma odpravljena
  - Včasih premoščanje ni možno, ker operanda ni v cevovodu
  - Npr.

LW	R3, 56(R4)	$R3 \leftarrow_{32} M[56 + R4]$
SUB	R1, R3, R6	$R1 \leftarrow R3 - R6$
ADD	R2, R3, R7	$R2 \leftarrow R3 + R7$
XOR	R8, R3, R9	$R8 \leftarrow R3 \oplus R9$

  - 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)

31

	Urina perioda								
Ukaz	1	2	3	4	5	6	7	8	9
LW R3,56(R4)	IF	ID	EX	MEM	WB				
SUB R1,R3,R6		IF	○	ID	EX	MEM	WB		
ADD R2,R3,R7				IF	ID	EX	MEM	WB	
XOR R8,R3,R9					IF	ID	EX	MEM	WB

32

- **Cevovodno razvrščanje** (pipeline scheduling)
  - Prevajalnik lahko s spreminjanjem vrstnega reda ukazov pogosto odpravi nevarnost
  - Npr.:

    a = b + c
    d = e – f

    (naslovi naj bodo v registrih Ra, ..., Rf)

LW	R2,0(Rb)	R2 ← b
LW	R3,0(Rc)	R3 ← c
LW	R4,0(Re)	R4 ← e ukaz prestavljen naprej
ADD	R5,R2,R3	R5 ← b + c
LW	R6,0(Rf)	R6 ← f ukaz prestavljen naprej
SW	0(Ra),R5	a ← b + c
SUB	R7,R4,R6	R7 ← e – f
SW	0(Rd),R7	d ← e – f

33

- Večina prevajalnikov danes uporablja cev. razvrščanje
  - čakanja pa se vedno ne da odpraviti
  - delež ukazov load, kjer se kljub temu pojavi PN:
    - 4 – 40% (odvisno od programa), povprečno pa 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

34

- 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 HIP je edina možnost RAW
  - s premoščanjem jo običajno odpravimo (razen pri load)

35

Kontrolne nevarnosti

- KN: pri ukazih, ki spremenijo PC drugače kot PC ← PC + 4
  - to so kontrolni ukazi oz. skoki:
    - brezpogojni skoki
    - pogojni skoki
    - klici (procedur)
    - vrnitve
  - skočni naslov se prenese v PC (v stopnji EX, razen pri TRAP (WB))
    - J, BEQ, BNE, CALL, TRAP, RFE

36

- 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
- Enostavna (a bolj slaba) rešitev je vstavljanje mehurčka v IF in ID
  - **skočna zakasnitev** (branch delay), čakanje 2 periodi
    - le pri TRAP je treba čakati 5 period

	Urina perioda									
Št. ukaza	1	2	3	4	5	6	7	8	9	10
Skočni ukaz	IF	ID	EX	MEM	WB					
Skočni ukaz + 1		IF	○	IF	ID	EX	MEM	WB		
Skočni ukaz + 2					IF	ID	EX	MEM	WB	
Skočni ukaz + 3						IF	ID	EX	MEM	WB
Skočni ukaz + 4							IF	ID	EX	MEM

IF v periodi 4 dobi drug ukaz!

37

- 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
  - 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
- KN so najhujše od 3 vrst nevarnosti

38

Odpravljanje kontrolnih nevarnosti

- Prvi korak je zmanjšanje skočne zakasnitve:
  1. *Preverjanje pogoja za skok* naj se izvaja čim bližje prvi stopnji cevovoda
  2. *Izračun skočnega naslova* naj se izvaja čim bližje prvi stopnji cevovoda
- HIP: preverjanje skočnega pogoja je v BEQ in BNE
  - 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
    - komparator za B=0

39

- Drug način je predikcija izpolnitve skočnega pogoja in napoved skočnega naslova (če se skok izvede)
  - branch predictor je vezje, ki napoveduje (ne)izpolnjenost pogoja
- 2 skupini:
  - s statično predikcijo
  - z dinamično predikcijo

40

Statična predikcija z zakasnjenimi skoki

- Prevajalnik skuša napovedati bolj verjeten rezultat preverjanja skočnega pogoja
  - med izvrševanjem programa se zato ne spreminja $\rightarrow$ statična predikcija
  - tudi že omenjeni primer (ki predpostavi neizpolnjenost pogoja) je preprost primer statične predikcije
  - **skočne reže** (branch slots)
    - v njih so ukazi, ki so v programu za skokom
    - enako številu stopenj cevovoda, ki so pred stopnjo, v kateri se v PC zapiše skočni naslov
      - pri HIP je to EX, pred njo sta 2 stopnji (zato 2 skočni reži)

41

- Pri uporabi zakasnjenih skokov se (ne glede na izpolnjenost pogoja) izvršijo vsi prevzeti ukazi
  - ukaza (oz. ukazi) v skočnih režah se ne nadomestita z mehurčki
  - ker se vedno izvršita (izvršijo), izgleda kakor da se skok izvede kasneje

42

Primer

Prvotno zaporedje ukazov		Spremenjeno zaporedje ukazov pri zakasnjenih skokih		
XOR	R1,R2,R3	XOR	R1,R2,R3	
ADD	R4,R6,R7	JMP	88(R20)	
SUB	R8,R5,R10	ADD	R4,R6,R7	skočna reža 1
JMP	88(R20)	SUB	R8,R5,R10	skočna reža 2
AND	R2,R2,R3	AND	R2,R2,R3	
OR	R7,R4,R9	OR	R7,R4,R9	

- Ukaza ADD in SUB se prestavita v skočni reži
- Pri desnem zaporedju ni čakalnih period

43

- Pri pogojnih skokih je težje:
  - ukaza, ki vpliva na pogoj, ne smemo dati v režo!
  - namesto njega damo ukaz NOP (to dela prevajalnik)
    - prebere se iz ukaznega PP
    - možno je tudi, da bi bila oba ukaza NOP
      - lahko pa tudi nobeden

Prvotno zaporedje ukazov	Spremenjeno zaporedje ukazov pri zakasnjenih skokih	
L1: XOR R1,R2,R3	L1: XOR R1,R2,R3	
ADD R4,R6,R7	SUB R8,R5,R10	
SUB R8,R5,R10	BEQ R8,L1	
BEQ R8,L1	ADD R4,R6,R7	skočna reža 1
AND R2,R2,R3	NOP	skočna reža 2
OR R7,R4,R9	AND R2,R2,R3	
	OR R7,R4,R9	

44

- Izboljšava cevovoda z zakasnjenimi skoki je uvedba *razveljavitvenih skokov* (cancelling branches)
  - To ni nič drugega kot vstavljanje mehurčkov v IF in ID (kar smo že spoznali)
    - To je smiselno, kadar pri zakasnjenem skoku ne moremo koristno uporabiti nobene od skočnih rež (to pomeni 2 NOP)
    - Pri neizpolnjenih pogojih privarčujemo
  - Toda: potrebujemo dva nova ukaza (BEQC, BNEC) + logiko za vstavljanje mehurčkov pri razv. skokih

45

- Meritve (na cevovodih z eno skočno režo):
  - pri pogojnih skokih se koristno zapolni ~ 70% rež
  - če upoštevamo še brezpogojne skoke (kjer so reže vedno koristno zasedene), se št. čakalnih urinih period zmanjša na 25%
- Ugotovljeno je bilo, da se druga reža koristno zapolni 2x redkeje kot prva
  - pri HIP bi pričakovali zmanjšanje čakalnih period na ~ 40%
  - pri skokih se namesto 2 izgubi 0,8 periode

46

- CPI:

  $CPI_{idealni} = (1 – 0{,}109)*1 + 0{,}109*1{,}8 = 1{,}087$

  – dosti bolje od 1,218
  – če bi uporabili še razvelj. skoke, bi bilo še bolje
  – pri dolgih cevovodih pa je koristno zapolniti reže težko
- Statična predikcija
  – prednost: večino dela opravi prevajalnik
  – hiba: večino dela opravi prevajalnik
    - zahteva drugačno programiranje → problemi s kompatibilnostjo za nazaj
- Danes se bolj uporabljajo strojni načini za dinamično predikcijo skokov

47

Dinamična predikcija skokov

- Dinamična predikcija
  – prilagaja se 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

48

- služi kot napoved izpolnjenosti pogoja
- če je napovedan izpolnjen pogoj, potrebujemo še skočni naslov
  - ta je dostopen šele v stopnji ID
  - zato privarčujemo le en urino periodo (pri HIPu)
  - 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

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

49

- Možna tudi n-bitna prediktorska tabela, n > 2
  - Vendar ni dosti boljša kot 2-bitna
- Tabele so velikosti največ 4096
  - 12 bitov naslova

3. Korelacijski prediktor

- Correlating branch prediction table
- Primer:

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

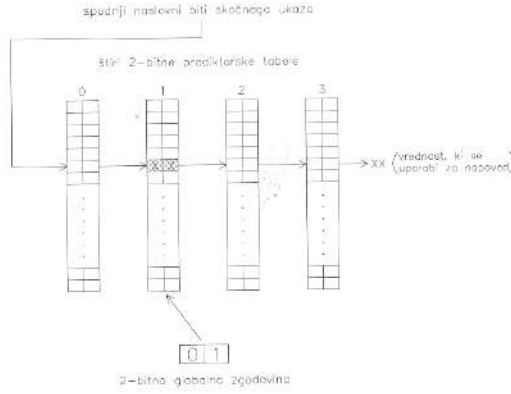
50

```
        SUBI  R3,R1,#2
        BNE   R3,L1       ; skok s1
        ADD   R1,R0,R0    ; a ← 0
L1:     SUBI  R3,R2,#2
        BNE   R3,L2       ; skok s2
        ADD   R2,R0,R0    ; b ← 0
L2:     SUBI  R3,R1,R2    ; R3 ← a – b
        BEQ   R3,L3       ; skok s3
```

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

51

– 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)

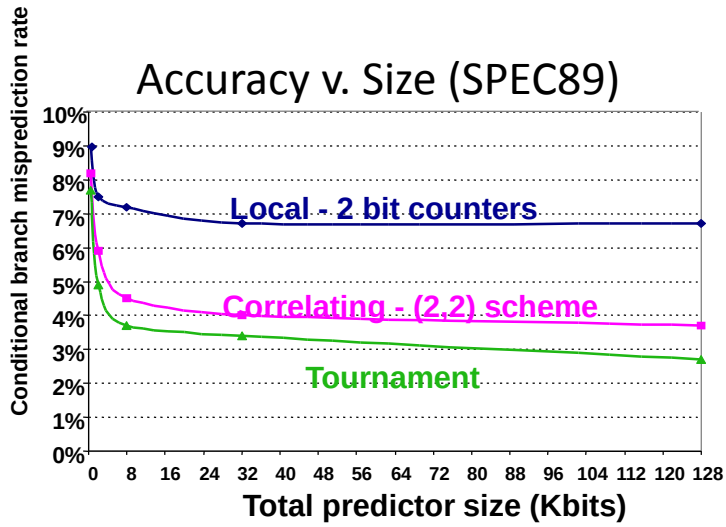


52

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)
- Najbolj znan primer je procesor Alpha 21264
  - Globalni prediktor je 2-bitna prediktorska tabela velikosti 4096 (do nje se dostopa z zgodovino prejšnjih 12 skokov)
  - Dvonivojski lokalni prediktor:
    - » Tabela 1024x10 (naslov je spodnjih 10 bitov ukaza, vrednost pa 10-bitna zgodovina)
    - » 3-bitna prediktorska tabela (1024x3) (naslov je zgodovina iz prve tabele)
  - Selektor je tabela 4096x2 (naslov je spodnjih 12 bitov ukaza)
    - » 0, 1: globalni; 2, 3: lokalni
  - Ko je znana izpolnjenost pogoja, se osvežijo vsi trije

53



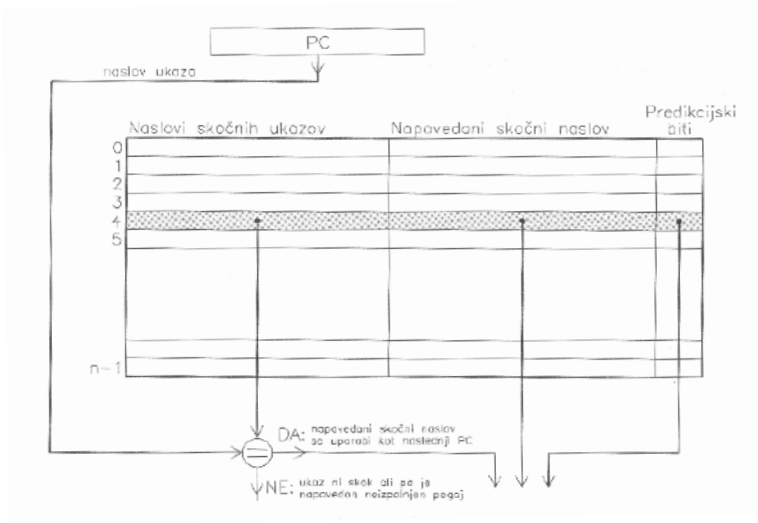
54

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

55

Skočni predpomnilnik:



56

- 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)

57

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

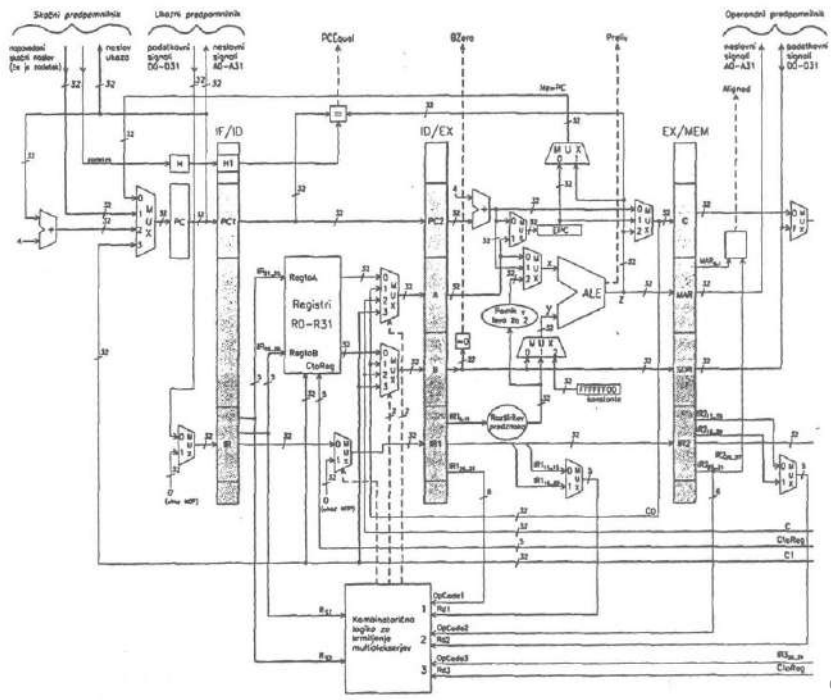
- integrated instruction fetch unit
- Današnji računalniki lahko istočasno prevzemajo in izvršujejo več ukazov (superskalarnost)
  - Prevzem ukazov bolj zapleten
- Enota deluje samostojno in dostavlja ukaze ostalim stopnjam
  - Dela tudi predikcijo, dostopa do PP, pri zgrešitvah menjava bloke v PP, ...

58

Vključitev dinamične predikcije v HIP

- HIP ima skočno zakasnitev 2 urini periodi
- Odločimo se npr. za skočni predpomnilnik (SPP) z enim prediktorskim bitom
  - ta bit prepreči, da bi se informacija o skoku izbrisala iz PP že ob prvi napačni napovedi
- V stopnji IF gre vsebina PC v ukazni in v SPP
- Če je v SPP zadetek, se napovedani naslov iz njega prenese v PC
  - s tega naslova naj bi stopnja IF prevzela naslednji ukaz
  - skočni ukaz pa se izvršuje naprej (v ID in EX), ker napoved morda ni pravilna

59



60

- Modifikacije podatkovne enote:
  - mux 4/1 (namesto 3/1) zaradi SPP
  - 1-bitni register H
    - ob zadetku se vanj vpiše 1, ob zgrešitvi 0
    - H je potreben zato, ker se mora v EX preveriti pravilnost skočnega naslova (ob zadetku se je vpisal v PC)
  - komparator v ID
    - primerja napovedani (v PC1, stopnja ID) in izračunani (na izhodu ALE, stopnja EX) skočni naslov
    - njegov izhod je PCEqual
      - 0 pri neenakosti, če je hkrati H1=1
      - 1 sicer

61

- kadar je PCEqual=1, je bila napoved naslova pravilna
  - skočnemu ukazu v EX se ne dovoli pisanje v PC
- sicer je bila napačna
  - tedaj sta napačna tudi ukaza v IF in ID
  - v IF in ID se vstavita mehurčka
  - v PC se prenese NewPC = PC2 + 4 + $razIR1_{0..15}$
- v stopnji EX se mora osvežiti informacija v skočnem PP (v primeru skoka)
  - način odvisen od
    - » izpolnitve pogoja
    - » zadetka v skočnem PP (H1)
    - » prediktorskega bita

62

1. Skočni pogoj je izpolnjen
   - v primeru zadetka v SPP se prediktorski bit postavi na 1
   - sicer se naslov skočnega ukaza in izračunani skočni naslov shranita v SPP (predikt. bit 1)
2. Skočni pogoj ni izpolnjen
   - v primeru zadetka v SPP
     - in predikt. bita 0, se info. o tem skočnem ukazu izbriše iz SPP
     - in predikt. bita 1, gre le-ta v 0, info. pa ostane v SPP
   - sicer nič

- stanje predikt. bita se torej uporablja le pri odločitvi o brisanju info. iz SPP
- zadetek v SPP pri HIPu pomeni, da je napovedan izpolnjen pogoj
  - ne glede na stanje predikt. bita

63

- Uspešnost dinamične predikcije pri HIP
  - verjetnost zadetka v SPP H_c naj bo 90%
    - za to zadošča že majhen SPP, velikosti 64
  - verjetnost, da zadetek v SPP pomeni pravilno napoved, H_p naj bo 92%
  - pogoj za skok naj bo izpolnjen z verjetnostjo 67,3%

Zadetek v SPP	Napoved izpolnjenosti pogoja	Dejanska izpolnjenost pogoja	Število čakalnih period
da	izpolnjen	izpolnjen	0
da	izpolnjen	neizpolnjen	2
ne	neizpolnjen	izpolnjen	2
ne	neizpolnjen	neizpolnjen	0

64

Čakalne periode $_{\text{pogojni skok}}$
= H_c * (1-H_p) * 2 + (1-H_c) * verj. za izpolnjen skočni pogoj * 2
= 0,9*0,08*2 + (1-0,9)*0,673*2
= 0,279

Čakalne periode $_{\text{brezpogojni skok}}$ = (1-H_c)*2 = (1-0,9)*2 = 0,2

Čakalne periode $_{\text{skok}} = \frac{0{,}125 \times 0{,}279 + 0{,}025 \times 0{,}2}{0{,}125 + 0{,}025} = 0{,}266$

$CPI_{idealni}$ = (1 – 0,15)*1 + 0,15*1,266 = 1,04

4% je približno 2x manj kot pri statični predikciji

Škoda zaradi zgrešitev v PP je znatno večja (v gornjih enačbah ni upoštevana)

65

Prekinitve in pasti pri cevovodu

- Kdaj skočiti na servisni program?
  - istočasno se izvaja več ukazov
  - delno izvršeni ukazi lahko povzročijo napake
- 3 primeri
  1. **Vhodno/izhodne prekinitve**
     - običajno je, da cevovod izvrši ukaze (ki so že v njem) do konca
       - V/I prekinitve so razmeroma redki dogodki, zato izguba ni velika
     - pri HIP je prekinitveno-prevzemni cikel treba izvesti izven cevovoda (sicer bi rabili 6 stopenj v cevovodu)

66

2. Programske pasti

- TRAP je v bistvu kot klic procedure
  - poseben brezpogojni skok

3. Pasti, do katerih pride med izvrševanjem ukaza

- najtežje
- zgodijo se na sredi ukaza
  - ukaz se ne more dokončati
  - potrebno ga je ustaviti, izvršiti servisni program in ga ponovno začeti
    - » treba je tudi paziti, da del ukaza, ki se je (bil) že izvršil, ne povzroči napake

67

Stopnja cevovoda	Problematične pasti pri HIP
IF	napaka strani (pri branju ukaza), zaščita pomnilnika
ID	nedefiniran ukaz
EX	preliv
MEM	napaka strani (pri dostopu do operanda), zaščita pomnilnika neporavnan operand,
WB	nobena

- napaka strani: pri navideznem (virtualnem) pomnilniku, kadar stran ni (fizično) v GP
  - ne gre za resnično napako
- zaščita pomnilnika: dostop do naslova, ki ne pripada programu

- pri napaki strani se po servisiranju program nadaljuje na prekinjenem mestu
- pri ostalih pasteh se običajno zaključi z diagnostičnim sporočilom

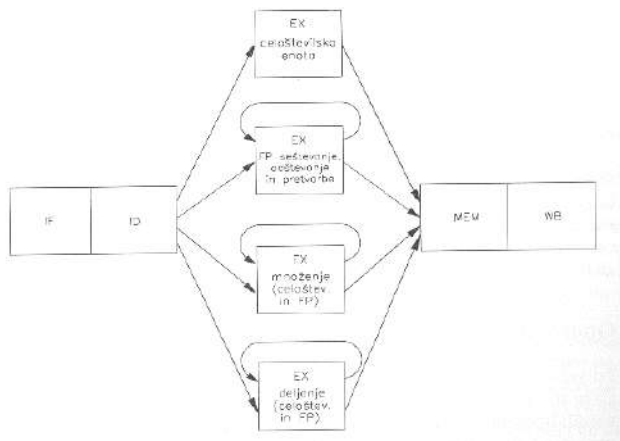
68

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 HIP 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

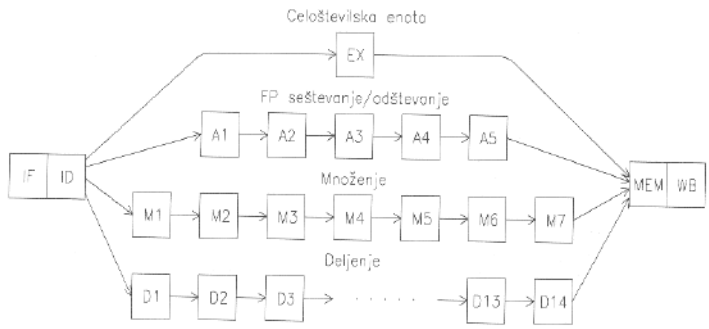
69

- 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



70

- 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



71

- 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(R2)	IF	ID	EX	MEM	WB													
FMUL F0,F4,F6		IF	ID	○	M1	M2	M3	M4	M5	M6	M7	MEM	WB					
FADD F2,F0,F8			IF	○	ID	○	○	○	○	○	○	A1	A2	A3	A4	A5	MEM	WB
FST 0(R2),F2					IF	○	○	○	○	○	○	ID	EX	○	○	○	○	MEM

72

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)

73

- 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

74

- Spremenjen vrstni red izvrševanja lahko pripelje do PN tipa WAR in WAW
- Tomasulov algoritem (1967)
- Podatkovne odvisnosti lahko delimo na
  - prave podatkovne odvisnosti
    - ukaz potrebuje kot vhodni operand rezultat enega od prejšnjih ukazov
  - imenske odvisnosti

75

```
FDIV   F0,F5,F6     F0 ← F5/F6
FADD   F4,F0,F2     F4 ← F0+F2
FST    0(R1),F4     M[R1] ← F4
FSUB   F2,F3,F7     F2 ← F3-F7
FMUL   F4,F3,F2     F4 ← F3*F2
```

  - imenske odvisnosti
    - med FADD in FSUB zaradi R2
      - nevarnost WAR (*antiodvisnost*)
    - med FADD in FMUL zaradi F4
      - nevarnost WAW (*izhodna odvisnost*)
  - prave podatkovne odvisnosti
    - med FDIV in FADD
    - med FADD in FST
    - med FSUB in FMUL

76

- 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
FST    0(R1),FT2     M[R1] ← 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

77

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

78

- 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

79

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
  - običajno se uporablja IPC = 1 / CPI
  - **večizstavitveni procesorji** (multiple issue processors)

80

- 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 $(2(n-1) + 2(n-2) ...)$

81

- Strojno ugotavljanje podatkovnih odvisnosti je zahtevno za realizacijo, zato sta se pojavili 2 rešitvi:
  1. Superskalarnost
  2. VLIW

82

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

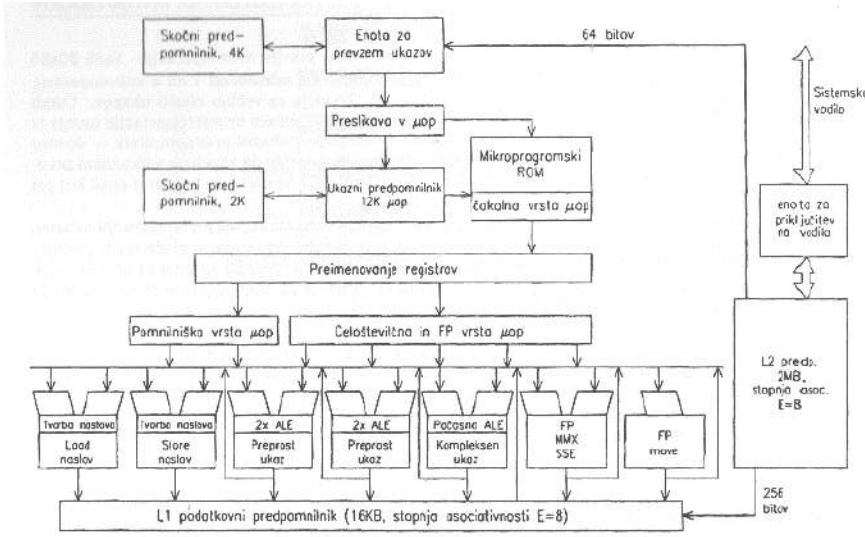
- korak izstavljanja je drugačen:
  - Iz čakalne vrste se vzame *n* ukazov
  - Izhodni register vsakega ukaza se preimenuje v enega od prostih registrov
    - S tem se odpravijo nevarnosti WAW in WAR, ki izvirajo iz imenskih odvisnosti
  - Preveri se medsebojna odvisnost operandov (kot že prej opisano)
    - Po potrebi se popravijo številke vhodnih registrov
  - Ukazi se prenesejo v ROB
    - Globina se določi na osnovi števila registrov
  - Ukazi se prenesejo v FE

85

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

86

Pentium 4



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

88

- 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)

89

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

90

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

91

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
  - problem: niti je treba definirati (paralelno programiranje)
    - eksplicitni paralelizem
    - obstoječe programe je (bi bilo) potrebno predelati!

92

- 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*

93

- 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)

94

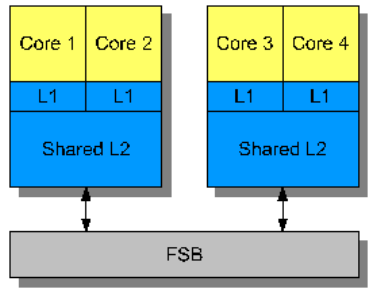
2. **Istočasna večnitnost** (simultaneous multithreading, SMT)
   - pri večizstavitvenih procesorjih
     - Intel Pentium 4: Hyper-threading (običajno 2 niti)
   - ni potrebno veliko sprememb
   - prednost: ni medsebojnih odvisnosti
   - hiba: v določenih primerih se zmogljivost tudi poslabša
     -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

95

- **Večjedrni procesorji** (multicore processors)
  - 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” ?!

96

- Primer:
  - Intel Core 2 Quad



97

8 PREDPOMNILNIK

Pomnilniška hierarhija

- Iz glavnega pomnilnika CPE jemlje ukaze in operande in vanj shranjuje rezultate
- Pomembni sta velikost in hitrost
  - velikost, da lahko rešujemo velike probleme
  - hitrost, da CPE ni treba čakati
- Oboje si nasprotuje
  - velik in hiter pomnilnik bi bil zelo drag
- GP: DRAM (dovolj poceni tehnologija za velik pomnilnik)
  - SRAM je predrag za GP
- Hitrost pomnilnikov DRAM se (tekom let) povečuje bistveno počasneje od hitrosti CPE
  - To vrzel je treba nekako premostiti, sicer CPE večino časa čaka na pomnilnik
- Zato se (poleg GP, ki je velik in relativno počasen) uporablja še majhen in hiter pomnilnik, ki mu rečemo **predpomnilnik** (cache)
  - le-ta je narejen v tehnologiji SRAM

- Če bi bili naslovi, ki jih generirajo CPE in V/I naprave, porazdeljeni naključno, ne bi pridobili ničesar
  - ker pa velja princip lokalnosti, se doseže bistveno povečanje hitrosti

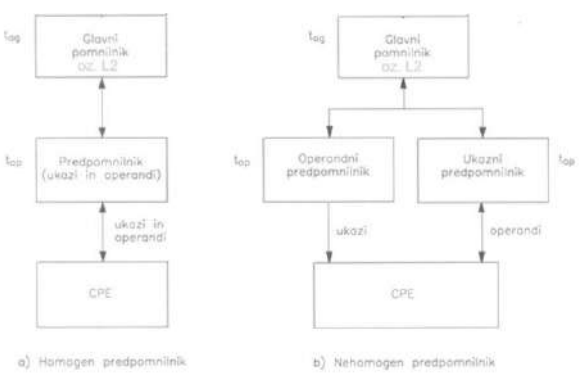
- Pomnilniška hierarhija vključuje tudi *pomožni pomnilnik* (oz. sekundarni, masovni). Kakšna je razlika?
  - Do GP ima CPE **neposreden dostop** (tako, da poda naslov pomnilniške besede)
  - pri pomožnih pomnilnikih je dostop **posreden** preko V/I ukazov, ki najprej prenesejo zahtevano besedo v GP, šele nato je možen neposreden dostop
- Zakaj je potreben pomožni pomnilnik?
  - cena enega bita na magnetnem disku je ~100x nižja kot v GP
  - vsebina je obstojna

- Bit je najmanjša enota informacije
  - shranjen je v eni pomnilniški celici, ki ima lahko 2 stanji (0, 1)
    - nekatere tehnologije sicer uporabljajo več stanj, vendar so manj zanesljive
- Danes so GP izključno elektronski, natančneje polprevodniški (iz integriranih vezij, tj. čipov)

Predpomnilnik

- PP hrani določene podatke, ki so tudi v glavnem pomnilniku
  - vsebina PP je podmnožica vsebine GP
- Pogosto imamo 2 ali 3 nivoje predpomnilnika:
  - L1 (level 1) je manjši in hitrejši in je kar na čipu CPE
  - L2 je malo večji in malo počasnejši (danes običajno tudi na CPE)
  - L3 je večji in počasnejši (običajno ni na CPE)
    - še vedno pa hitrejši od DRAMa

- Pri cevovodnih CPE (ki so danes običajne) je PP (zaradi potrebe po istočasnem dostopu do ukazov in operandov pri cevovodu) razdeljen v dva dela (nehomogeni PP):
  - ukazni in operandni (to velja za L1; L2 pa je običajno homogen)
  - podatkovna pot do PP je širša (128 ali 256 bitov)



- Zadetek: Kadar je naslov, do katerega se želi dostopiti, v PP
- Zgrešitev: sicer
  - v določenih primerih (npr. 2% dostopov) iskane besede ni v PP
  - v tem primeru je treba iz GP v PP prenesti nov blok besed (blok vsebuje iskano besedo), kar traja dolgo

- Vzemimo zaenkrat, da imamo samo L1
- Razmerje t_{ag}/t_{ap} je lahko tudi do nekaj sto
  - t_{ap} ... čas dostopa do PP
  - t_{ag} ... čas dostopa do GP
- Velikost PP je do 1% velikosti GP
  - Kako lahko sploh pričakujemo, da bo iskana informacija dovolj pogosto v PP?
  - Razlog je v lokalnosti

- Uspešnost delovanja PP merimo z **verjetnostjo zadetka** (hit ratio) H
  - Kadar je naslov, do katerega se želi dostopiti, v PP, imamo zadetek, sicer zgrešitev (verjetnost 1-H)
  - H izmerimo s štetjem pomnilniških dostopov, pri katerih pride do zadetka

$$H = N_p/N = N_p/(N_g+N_p)$$

N_p ... število zadetkov

N_g ... število zgrešitev ($=N-N_p$)

  - H je običajno celo večji od 0,95

- Čas dostopa

$$t_a = t_{ap} + (1\text{-}H)t_{ag}$$

- Treba pa je upoštevati, da se pri zgrešitvi ne prenese samo beseda, ampak celoten PP blok!
- Zato je bolje uporabiti enačbo

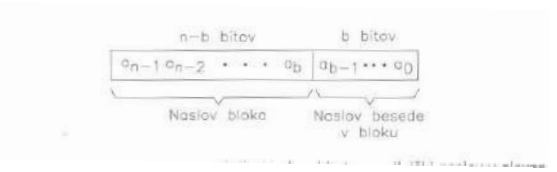
$$t_a = t_{ap} + (1\text{-}H)t_B$$

t_B ... čas za prenos bloka oz. **zgrešitvena kazen** (miss penalty) (10-100 urinih period)

Pozor: *miss penalty* ima lahko tudi druge pomene!

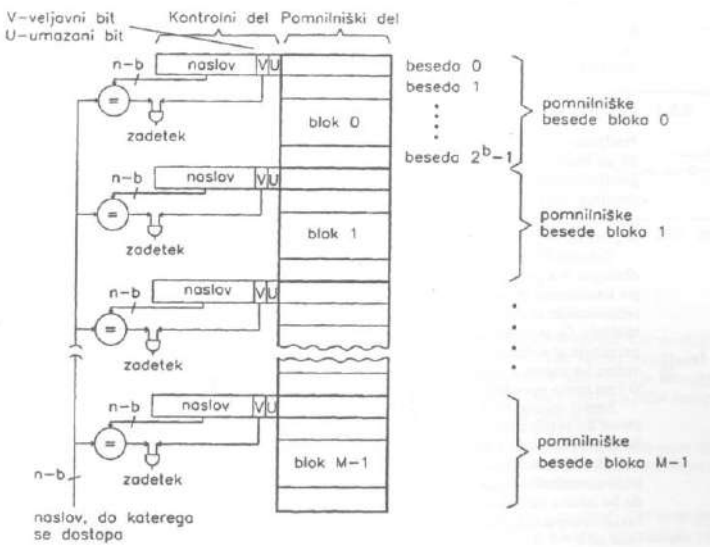
- Možno je definirati področja v GP, katerih besede se nikoli ne prenesejo v PP (*uncacheable* področja)
  - dostop do besede v takem področju je vedno zgrešitev
  - beseda se nikoli ne prenese v PP
  - npr. pri računalnikih, ki uporabljajo pomnilniško preslikan V/I, se določeni pomnilniški naslovi nanašajo na registre V/I naprav
    - pisanje v te registre povzroči odziv naprave

- Ker je vsebina PP podmnožica vsebine GP, mora predpomnilnik (poleg vsebine) vsebovati tudi naslove besed
- Zato je sestavljen iz dveh delov:
  - **kontrolni** in
  - **pomnilniški** del
- Pomnilniški del je razdeljen na **bloke** (po $B=2^b$ pomnilniških besed, b=3-8)
  - bloku se reče tudi **predpomnilniška vrstica** (cache line)
- Pomnilniški naslov:
  - Če je n-biten, rabimo v kontrolnem delu zgornjih n - b bitov naslova
    - spodnjih b bitov določa besedo v bloku, zgornjih n - b bitov pa naslov bloka



- Kontrolni del vsebuje informacijo, ki enolično opiše vsak blok:
  - vsaj naslov bloka v glavnem pomnilniku
  - običajno pa še **veljavni** in **umazani bit**
    - veljavni bit pove, ali je vsebina PP veljavna
      - V=1: je
      - V=0: ni → zgrešitev
    - umazani bit U se ob prenosu bloka v PP postavi na 0. Če pride do pisanja v blok, se postavi na 1.
- Naslov v kontrolni informaciji pove, kateri del GP je trenutno v bloku
  - rečemo, da je *preslikan* v PP

Splošna zgradba predpomnilnika



- Naslov je *n*-biten
- Velikost bloka je $B = 2^b$ besed
  - prva beseda v bloku (beseda 0) ima vedno naslov, ki je mnogokratnik dolžine bloka
- Število blokov je M_b

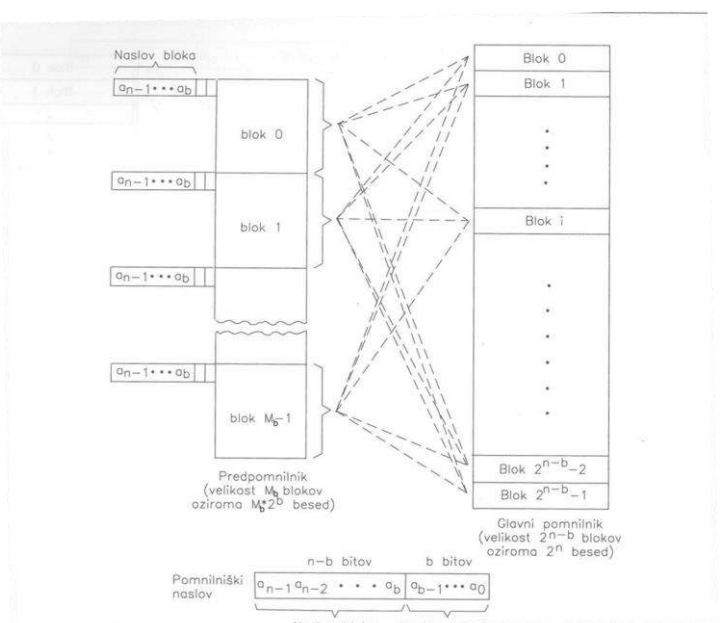
- Zgornjih *n* – *b* bitov naslova se v PP primerja z naslovi v kontrolni informaciji vseh blokov
  - če obstaja pri nekem bloku enakost, je zahtevana beseda v PP (zadetek); poleg tega mora biti še V=1
  - sicer imamo zgrešitev; potreben je dostop do GP; blok iz GP se prenese v PP
    - Če so vsi bloki zasedeni in veljavni, bo novi blok zamenjal enega od obstoječih (**zamenjava bloka**)
    - Ob zamenjavi se mora vsebina bloka, če se je spremenila, najprej prenesti v GP
    - Bit V se uporablja zato, ker včasih vsebina PP ni veljavna

- Primerjava zgornjih *n* – *b* bitov naslova z vsebinami kontrolnih delov vseh blokov mora biti zelo hitra
  - zato se uporabljajo omejitve pri preslikavi iz GP v PP: neka beseda iz GP se lahko shrani v vnaprej določeno (majhno) število blokov
  - glede na strogost te omejitve ločimo 3 vrste PP:
    - asociativni
    - set asociativni
    - direktni

Predpomnilniki glede na omejitve pri preslikavi

- Primerjavo zgornjih *n* - *b* bitov lahko izvedemo z **asociativnim pomnilnikom**
  - pri le-tem poteka dostop preko vsebine
  - če damo na vhod kombinacijo bitov, se ta primerja z delom vsebine vsake besede
  - v primeru enakosti vrne celotno besedo
  - takemu PP rečemo asociativni PP (APP)
- Vsebina AP so naslovi v kontrolnem delu
  - kontrolni del je v bistvu kar AP
- Pri zadetku se nato naredi dostop do besede v bloku (določena z *b* biti)
- To je **čisti APP** - ni omejitev:
  - vsak blok PP lahko sprejme katerokoli besedo iz GP
  - čisti APP ima največji H
- Problem pa je v tem, da so veliki AP izjemno dragi
  - prav velikih pravzaprav sploh ni

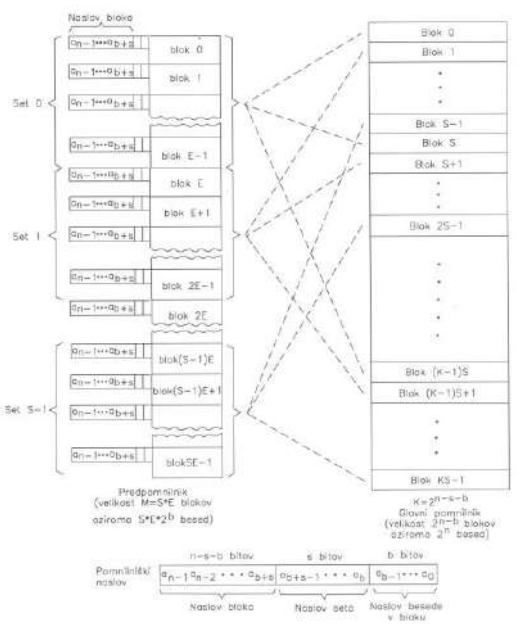
Čisti APP



- potrebno je veliko število primerjalnikov
  - npr. za PP s 100000 bloki bi potrebovali 100000 (*n*-*b*)-bitnih primerjalnikov (ogromno logike)

- Velik PP lahko naredimo le, če v preslikovanje naslovov vpeljemo omejitve
- Namesto enega velikega AP uporabimo več majhnih
- Tako dobimo **set asociativni predpomnilnik** (SAPP)

SAPP



- SAPP je razdeljen na $S = 2^s$ setov, vsak set pa je majhen AP
- Število blokov v setu $E = 2^e$ je **stopnja asociativnosti** (običajno do 16)
  - to je velikost AP v setu (= št. primerjalnikov)
- Velikost PP je

  $M_b = S*E = 2^{s+e}$ blokov oz.

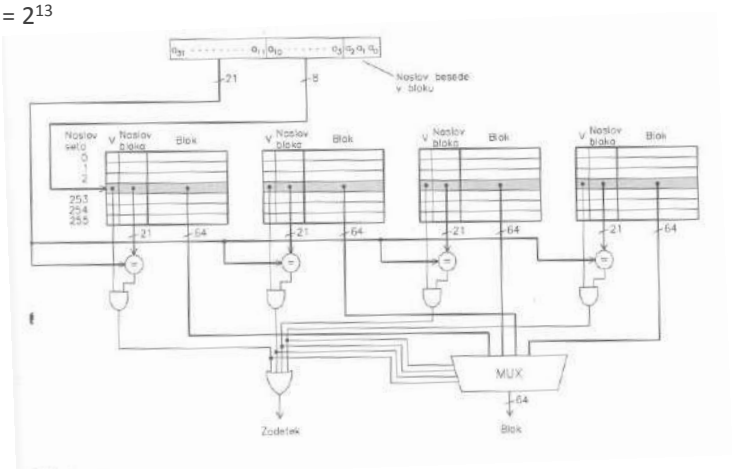
  $M = M_b*B = S*E*B = 2^{s+e+b}$ pomnilniških besed

- Pri SAPP se pojavi omejitev pri preslikovanju naslovov:
  - za vsako besedo GP je vnaprej določeno, v katerega od setov se lahko preslika
    - to določajo naslovni biti $a_{b+s-1}, a_{b+s-2}, \ldots, a_b$
    - ta naslov seta se imenuje tudi **predpomnilniški indeks** (cache index)
      - če so ti biti 0,0,...,0, se lahko preslika v set 0
      - če so ti biti 0,0,...,1, se lahko preslika v set 1
      - itd.
    - naslov A_i se torej lahko preslika le v enega od blokov seta S_i

      $S_i = A_i(n-1 : b) \bmod 2^s$

- Pri SAPP lahko s spreminjanjem *E* vplivamo na njegove lastnosti
  - pri *S* = 1 (*s* = 0) je *E* enako *M* (čisti APP)
  - pri *E* = 1 (*e* = 0) je v vsakem setu le en blok (**direktni PP**)
    - pri tem je torej za vsako besedo vnaprej določeno, v kateri blok se preslika
      - blok enak setu
    - potreben samo 1 primerjalnik

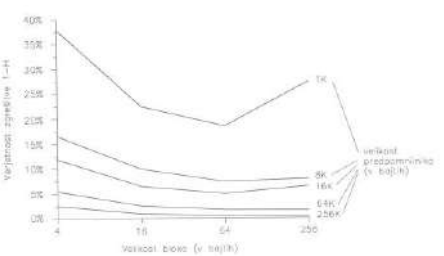
- Realizacija SAPP
  - n=32, b=3, e=2, s=8, 8-bitna pomnilniška beseda
  - Vsak izmed 256 setov ima 4 bloke, vsak blok 8 besed (=8x8=64 bitov)
  - $M = 2^{13}$



- Pri podani velikosti PP (M) se z E spreminja tudi H
  - manjši E → manjša cena in tudi manjši H
- *Predpomnilniško pravilo 2:1*
  - velja za direktne PP
  - (1-H) dir. PP velikosti M ≈ (1-H) SAPP z E=2 in velikostjo M/2
  - izkustveno

Kako velik naj bo blok?

- PP lahko povečujemo s povečevanjem E, S ali B
  - najlažje B
  - vsak blok ima eno kontrolno informacijo, kontrolni del pa je najbolj zapleten
- Pri dani velikosti PP:
  - če povečamo bloke, je boljša prostorska lokalnost, toda slabša časovna lokalnost, ker je blokov manj
  - 1-H se najprej zmanjšuje, nato pa začne naraščati



- toda 1-H ni edini parameter, ki vpliva na delovanje PP
  - pomembna je tudi zgrešitvena kazen t_B (čas prenosa bloka v PP)
  - sestavljena je iz latence in časa dejanskega prenašanja
  - pri večjem bloku je t_B večja
    - od nekod naprej lahko prevlada nad zmanjšanjem 1-H in poslabša delovanje PP
- t_B se lahko zmanjša tako, da se najprej prenese zahtevana beseda (CPE lahko takoj nadaljuje z delom), nato ostale (*requested word first*)

➢ Kateri blok naj se zamenja ob zgrešitvi?

- tudi to vpliva na 1-H
- 2 strategiji:
  1. Naključna.
     - enostavna za realizacijo
  2. LRU (Least Recently Used)
     - zamenja se blok, do katerega najdalj ni bil narejen dostop
     - izkoriščanje časovne lokalnosti
     - pri E > 4 zapletena realizacija
     - tudi pri E = 4: *psevdo* LRU

- pri večjih E naključna strategija
- pri E = 2 je 1-H pri naključni strategiji ~1,1x večja kot pri LRU

Vpliv E in zamenjevalne strategije na 1-H (pri večjem PP sta oba vpliva manjša):

M	1-H					
	E = 2		*E* = 4		*E* = 8	
	LRU	Naključno	LRU	Naključno	LRU	Naključno
16K	5,2%	5,7%	4,7%	5,3%	4,4%	5,0%
64K	1,9%	2,0%	1,5%	1,7%	1,4%	1,4%
256K	1,2%	1,2%	1,1%	1,1%	1,1%	1,1%

Pisanje

- branje je enostavnejše (in tudi bolj pogosto)
- pisanje v PP se lahko začne le, če je bil ugotovljen zadetek
- Kako se sprememba v PP odraža v GP:
  1. **Pisanje skozi** (*write through*)
     - vedno se piše v oba
  2. **Pisanje nazaj** (*write back*)
     - piše se samo v PP
     - pri zamenjavi je spremenjeni blok treba prenesti v GP
     - **umazani bit** (dirty bit) je 0 ob prenosu bloka v PP. Po pisanju v blok se postavi na 1.
     - pri zamenjavi se zapišejo v GP samo bloki z 1

- Pisanje nazaj:
  - hitrost
  - manj prometa z GP
  - ob zamenjavi bloka najhitrejši način pisanja v DRAM
- Pisanje skozi:
  - enostavno za realizacijo
  - vsebini PP in GP sta **skladni** (**koherentni**)
    - dobro za druge naprave

- **Pisalni izravnalnik** (write buffer)
  - vanj CPE shrani podatek, ki se bo (s pomočjo dodatne logike) vpisal v GP
  - s tem se odpravi čakanje zaradi hitrejšega pisanja v PP kot v GP
  - pri *pisanju skozi* je nujno potreben
- Danes se uporablja pretežno pisanje nazaj
  - uporablja se tudi pisalni izravnalnik
  - podoben je čistemu APP
  - pri pisanju CPE vedno preveri, če je beseda v enem od blokov v izravnalniku
  - umazani blok se piše v pisalni izravnalnik namesto direktno v GP
- Pri pisanju tudi pri zadetkih rabimo 2 periodi
  - najprej je potrebno branje
  - v bistvu je možna tudi le 1 perioda
    - na osnovi neke vrste cevovoda (več v knjigi)

- Zgrešitve
  - pri bralnih zgrešitvah se blok vedno prenese v PP (zamenja enega od obstoječih)
  - pri pisalnih zgrešitvah 2 možnosti:
    1. **Pisalna zamenjava** (write allocate)
       - prenos novega bloka v PP (podobno kot bri branju)
       - bolj običajno pri pisanju nazaj
       - bolj razširjena
    2. **Pisanje naokrog** (write around, no write allocate)
       - zamenjava bloka samo v GP (ne v PP)
       - bolj običajno pri pisanju skozi

Vrste zgrešitev

Vrste zgrešitev

1. **Obvezne zgrešitve** (compulsory misses)
   - reče se tudi zgrešitve prvega dostopa
2. **Velikostne zgrešitve** (capacity misses)
   - zaradi končne velikosti PP običajno ne more vsebovati vseh blokov, ki jih program potrebuje
   - zato prihaja do zamenjav blokov, ki so kmalu spet potrebni
3. **Konfliktne zgrešitve** (conflict misses)
   - zamenjava blokov, ki se preslikajo v isti set
   - pri čistem APP jih ni

Vrste zgrešitev glede na M in E (Alpha, B=64, LRU, SPEC2000):

Velikost predpomnilnika v bajtih	Stopnja asociativnosti E	Skupna verjetnost zgrešitve $1-H$	Deleži posameznih vrst (vsota = skupna verjetnost zgrešitve)					
			Obvezna zgrešitev		Velikostna zgrešitev		Konfliktna zgrešitev	
1K	1	0,191	0,009	5%	0,141	73%	0,042	22%
1K	2	0,161	0,009	6%	0,141	87%	0,012	7%
1K	4	0,152	0,009	6%	0,141	92%	0,003	2%
1K	8	0,149	0,009	6%	0,141	94%	0,000	0%
2K	1	0,148	0,009	6%	0,103	70%	0,036	24%
2K	2	0,122	0,009	7%	0,103	84%	0,010	8%
2K	4	0,115	0,009	8%	0,103	90%	0,003	2%
2K	8	0,113	0,009	8%	0,103	91%	0,001	1%
4K	1	0,109	0,009	8%	0,073	67%	0,027	25%
4K	2	0,095	0,009	9%	0,073	77%	0,013	14%
4K	4	0,087	0,009	10%	0,073	84%	0,005	6%
4K	8	0,084	0,009	11%	0,073	87%	0,002	3%
8K	1	0,087	0,009	10%	0,052	60%	0,026	30%
8K	2	0,069	0,009	13%	0,052	75%	0,008	12%
8K	4	0,065	0,009	14%	0,052	80%	0,004	6%
8K	8	0,063	0,009	14%	0,052	83%	0,002	3%
16K	1	0,066	0,009	14%	0,038	57%	0,019	29%
16K	2	0,054	0,009	17%	0,038	70%	0,007	13%
16K	4	0,049	0,009	18%	0,038	76%	0,003	6%
16K	8	0,048	0,009	19%	0,038	78%	0,001	3%
32K	1	0,050	0,009	18%	0,028	55%	0,013	27%
32K	2	0,041	0,009	22%	0,028	68%	0,004	11%
32K	4	0,038	0,009	23%	0,028	73%	0,001	4%
32K	8	0,038	0,009	24%	0,028	74%	0,001	2%
64K	1	0,039	0,009	23%	0,019	50%	0,011	27%
64K	2	0,030	0,009	30%	0,019	65%	0,002	5%
64K	4	0,028	0,009	32%	0,019	68%	0,000	0%
64K	8	0,028	0,009	32%	0,019	68%	0,000	0%
128K	1	0,026	0,009	34%	0,004	16%	0,013	50%
128K	2	0,020	0,009	46%	0,004	21%	0,006	33%
128K	4	0,016	0,009	55%	0,004	25%	0,003	20%
128K	8	0,015	0,009	59%	0,004	27%	0,002	14%

Rezultati

- Rezultati:
  - kaj opazimo za vsako od vrst zgrešitev?
    - pogostost obveznih neodvisna od M
      - delež le-teh zelo majhen, če se je program dolg
    - pogostost velikostnih pada z M
    - pogostost konfliktnih pada z E
- Kako bi zmanjšali vsako od 3 vrst zgrešitev:
  - obvezne: večji blok
    - vendar se lahko poveča zgrešitvena kazen
  - velikostne: večji PP
  - konfliktne: večji E
    - vendar se lahko poveča čas dostopa

Skladnost

- Problem **skladnosti PP** (cache coherency): vsebina bloka v PP se lahko razlikuje od vsebine v GP ali v drugih PP
  - treba je zagotoviti, da zaradi neskladnosti ne pride do napak
- En vzrok za neskladnost so prenosi med V/I napravami in GP
- Neskladnost pa se pojavlja tudi na računalnikih, ki imajo več CPE

Vpliv PP na hitrost delovanja CPE

Vpliv PP na hitrost delovanja CPE

Čas izvrševanja programa:

$CPEčas = (CPEperiode_{izvrševanje} + CPEperiode_{čakanje\ na\ pomnilnik})*t_{CPE}$

$CPEperiode_{čakanje\ na\ pomnilnik} = N_R*(1-H_R)$ * Bralna zgrešitvena kazen +
$N_W*(1-H_W)$ * Pisalna zgrešitvena kazen

$CPEperiode_{čakanje\ na\ pomnilnik} = N*(1-H)$ * Zgrešitvena kazen

$CPEčas = I*(CPI_{idealni} + M_I*(1-H)$ * Zgrešitvena kazen$)*t_{CPE}$

$CPI_{idealni} = CPEperiode_{izvrševanje} / I$

N_R ... število bralnih dostopov

N_W ... število pisalnih dostopov

N ... število vseh dostopov

I ... število ukazov

H ... povprečna verjetnost zadetka

Zgrešitvena kazen ... povprečna zgrešitvena kazen

M_I ... povprečno število pomnilniških dostopov na ukaz

$CPI_{idealni}$ predpostavi, da ni zgrešitev

- **Primer 1**
  - f_{CPE} = 300 MHz
  - ločen ukazni in operandni PP
  - $CPI_{idealni}$ = 2 (izmerjen na nekem programu)
  - 36% pomnilniških dostopov na ukaz (pri tem programu)
  - verjetnost zgrešitve v ukaznem PP = 2%
  - verjetnost zgrešitve v operandnem PP = 4%
  - DRAM: prvi dostop 60ns, naslednji trije po 10ns
  - PP: 256-bitni blok, 64-bitna podatkovna pot do DRAMa
    - prenos bloka zahteva 4 64-bitne prenose, torej 90ns
    - zgrešitvena kazen torej 27 period ure
  - Za koliko zgrešitve upočasnijo delovanje računalnika?

Čakalne CPEperiode$_{\text{ukazni PP}}$
= I*(1-H) * *Zgrešitvena kazen*
= I*0,02*27 = 0,54*I

Čakalne CPEperiode$_{\text{operandni PP}}$
= I*M_I*(1-H) * *Zgrešitvena kazen*
= I*0,36*0,04*27 = 0,39*I

Skupno je čakalnih period 0,93*I

CPI = 2,93

Upočasnitev = *CPI*/*CPI*$_{idealni}$ = 2,93/2 = 1,47

Primer2

- f_{CPE} = 600MHz
  - hitrost obeh PP ustrezno večja
- ostalo enako

Čakalne CPEperiode$_{\text{ukazni PP}}$ + *Čakalne CPEperiode*$_{\text{operandni PP}}$
= I*0,02*54 + I*0,36*0,04*54 = 1,86*I
CPI = 3,86

Upočasnitev = *CPI*/*CPI*$_{idealni}$ = 3,86/2 = 1,93
$CPI_1 * t_{CPE1}$ / $CPI_2 * t_{CPE2}$ = 2,93*2 / 3,86 = 1,52

Računalnik z 600MHz uro je (v našem primeru) le 1,52 krat hitrejši od tistega z 300MHz (zaradi PP)!

- Škoda zaradi zgrešitev se povečuje
  - s f_{CPE}
    - zgrešitvena kazen se meri v številu period ure
  - pa tudi z zmanjšanjem CPI
    - npr. zaradi povečane paralelnosti
- Zgrešitveno kazen lahko zmanjšamo tudi z uvedbo L2

$$CPEperiode_{čakanje\ na\ pomnilnik} = N*(1-H_{L1}) * Zgrešitvena\ kazen_{L1}$$

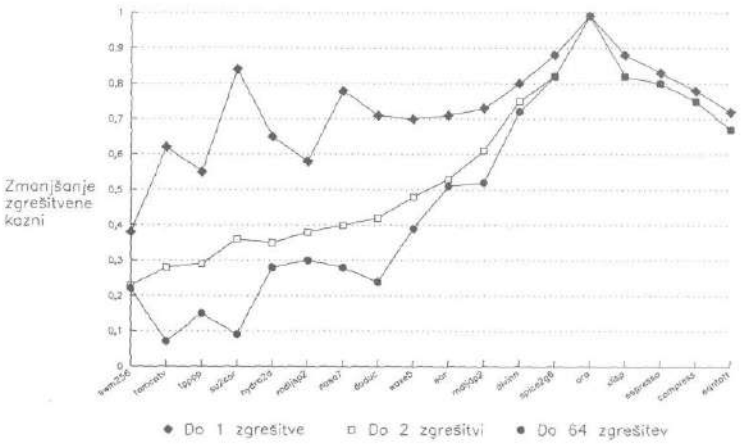
$$Zgrešitvena\ kazen_{L1} = t_{B,L2} + (1-H_{L2}) * Zgrešitvena\ kazen_{L2}$$

  - $t_{B,L2}$ ... čas prenosa bloka iz L2 v L1 pri zadetku v L2
  - H_{L2} je lokalna verjetnost zgrešitve (pogojna verjetnost, pogoj je zgrešitev v L1)
    - Do L2 se dostopa, kadar je v L1 zgrešitev
  - Globalna verjetnost zgrešitve na nivoju L2 je $(1-H_{L1})*(1-H_{L2})$

Načini za zmanjševanje zgrešitvene kazni

- Na izgubo hitrosti vplivata 1-H in t_B
  - zmanjšanje 1-H: večji PP, večji E, ustrezen B, dobra zamenjevalna strategija (prvo dvoje odvisno od stanja tehnologije, drugo dvoje pač ustrezno izberemo)
    - ni dosti manevrskega prostora
  - zmanjšanje t_B: vrstni red prenosa besed v bloku, L2
    - so pa danes tudi druge možnosti:
      1. **Vnaprejšnji prevzem bloka** (block prefetch)
         - pri prenosu bloka k v PP se prebereta še npr. bloka k+1 in k+2 in shranita v **bralni izravnalnik** (read buffer), do katerega se da hitro dostopiti
      2. **Neblokirajoči PP** (nonblocking cache)
         - med zamenjavo bloka PP deluje naprej in CPE lahko špekulativno izvršuje naslednje ukaze
         - načinu delovanja se reče *zadetek pod zgrešitvijo* (hit under miss)
         - možno je tudi *zadetek pod večkratno zgrešitvijo* (hit under multiple miss)

Razmerje zgrešitvene kazni neblokirajočega in blokirajočega PP:



9
Še nekaj principov delovanja računalnikov
(Nadaljevanje poglavja 3)

Vhod in izhod

- Osnovna naloga V/I sistema je pretvorba informacije iz ene oblike v drugo
  - izjema so naprave za shranjevanje informacije, ki tudi spadajo v to skupino
    - rečemo jim pomožni pomnilniki (npr. magnetni disk, optični disk, magnetni trak)
      - cena, obstojnost informacije
- Osnovni način delovanja V/I sistema je prenos podatkov
  - med GP in V/I napravami ali
  - med CPE in napravami
- Razlike med rač. glede izvedbe V/I so velike
  - pri znanstvenem računanju malo V/I prenosov
  - pri poslovnem veliko

- 2 skupini izvedb V/I sistema :
  1. **Programski vhod/izhod** (**programmed I/O**)
     - z V/I napravo komunicira CPE
     - vsak podatek se prenese iz GP v CPE in nato v napravo ali obratno
     - prenos je realiziran z zaporedjem ukazov
     - hiba je počasnost in zasedenost CPE
  2. **Neposredni dostop do pomnilnika** (**direct memory access - DMA**)
     - naprava komunicira neposredno z GP
     - zato rabimo **DMA krmilnik**, ki nadomesti CPE
     - posebna izvedba DMA krmilnikov so **vhodno/izhodni procesorji**

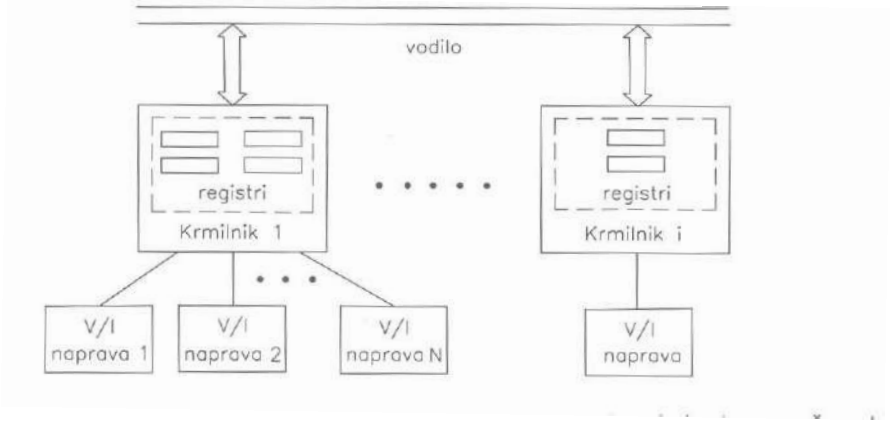
3

- Pri mnogih računalnikih srečamo oba načina dostopa
  - za počasne naprave je primeren programski vhod/izhod
  - za hitre oz. podatkovno zahtevne je nujen DMA, ker bi bil programski prepočasen

4

- Vsaka V/I naprava je priključena preko **krmilnika naprave** (device controller)
  - vezje, ki omogoča prenos podatkov v napravo in iz nje
    - lahko preprost (register), lahko kompliciran (specializiran računalnik)
  - na nekatere krmilnike je mogoče priključiti več naprav
  - s krmilnikom komuniciramo preko njegovih registrov
    - pisanje in branje pri njih sproži neko operacijo v napravi ali odraža stanje po prejšnji operaciji
      - npr. s pisanjem v ukazni register krmilnika magnetnega diska dosežemo premik bralno-pisalne glave na določeno sled
      - z branjem statusnega registra pa lahko ugotovimo, kdaj je premik končan

5



Krmilniki in vhodno/izhodne naprave

6

- Registri krmilnikov so lahko v istem naslovnem prostoru kot GP, lahko pa v posebnem
- Ločimo 3 izvedbe:
  1. **Pomnilniško preslikan vhod/izhod** (**memory mapped I/O**)
     - registri krmilnikov so v pomnilniškem naslovnem prostoru
     - iz CPE so videti kot pomnilniške lokacije
     - iz njih bere in vanje piše z ukazi za dostop do pom.
     - ni posebnih V/I ukazov

7

  2. **Ločen vhodno/izhodni prostor**
     - registri krmilnikov so v posebnem naslovnem prostoru
     - za dostop do registrov so potrebni *posebni V/I ukazi*
     - pri tem CPE aktivira tudi določen(e) signal(e), ki pove(jo), da se naslavlja V/I naslovni prostor
  3. **Posredno preko vhodno/izhodnih procesorjev**
     - tudi tu so registri krmilnikov v posebnem naslovnem prostoru, ki pa iz CPE ni neposredno dostopen
     - vmes so še vhodno/izhodni procesorji (razbremenijo CPE)
     - pri velikih računalnikih

8

Lokalnost pomnilniških dostopov

- Pojav, da programi večkrat uporabijo iste ukaze in operande in da pogosteje uporabljajo ukaze in operande, ki so v pomnilniku blizu trenutno uporabljanim
  - tipičen program 90% časa uporablja samo 10% ukazov
- Lokalnost pomnilniških dostopov močno vpliva na arhitekturo današnjih računalnikov
  - omogoča, da GP zamenjamo s **pomnilniško hierarhijo**.

9

- Štirinivojska pomnilniška hierarhija
  - M_1: predpomnilnik 1. nivoja (SRAM)
  - M_2: predpomnilnik 2. nivoja (SRAM)
  - M_3: GP (DRAM)
  - M_4: pomožni pomnilnik (magnetni disk)
- Pomnilniški prostor nivoja *i* je (v principu) podmnožica prostora na nivoju *i*+1
  - Če informacije ni v M_1, se naredi dostop do M_2; če je tudi v M_2 ni, se naredi dostop do M_3, ...
  - To se izvaja samodejno (ne da bi moral programer skrbeti za to)

10

- Zaporedje naslovov A(1), ..., A(*N*)
  - pri *N* dostopih do pom. je število različnih naslovov << *N*

- 2 vrsti lokalnosti:
  1. Prostorska
     - zaporedje ukazov je večinoma na zaporednih lokacijah
     - podatkovne strukture (npr. polja) se običajno obdeluje po zaporednih indeksih
  2. Časovna
     - zanke, začasne spremenljivke

11

Vzporedni (paralelni) računalniki

- Von Neumann: zaporedno izvajanje ukazov
- Mnogi problemi po svoji naravi dovoljujejo istočasno oz. paralelno izvajanje več operacij
- Zato so von Neumann-ov model razširili
- Flynn-ova klasifikacija (1966) uporablja 2 kriterija:
  - tok ukazov (instruction stream): koliko ukazov se izvršuje naenkrat
  - tok podatkov (data stream): koliko ponovitev operandov* en ukaz obdeluje naenkrat

12

- Npr.,

  ADD A1,A2,A3

  A1 ← A2 + A3

  N paralelnih ponovitev:

  A1(*i*) ← A2(*i*) + A3(*i*), $i = 1, ..., N$

13

- **Flynn-ova klasifikacija**:
  - **SISD** (Single Instruction stream, Single Data stream)
    - izvajajo naenkrat en ukaz na eni zbirki operandov
    - najbolj zmogljivi so vektorski računalniki
  - **SIMD** (Single Instruction stream, Multiple Data stream)
    - izvajajo en ukaz na več zbirkah operandov (*N*)
    - imajo eno kontrolno enoto in *N* ALE ter *N* množic registrov
  - **MISD** (Multiple Instruction stream, Single Data stream)
    - ne obstajajo
  - **MIMD** (Multiple Instruction stream, Multiple Data stream)
    - izvajajo več ukazov na več zbirkah operandov
    - multiprocesorji, multiračunalniki

14

- MIMD: več CPE
  - *tesno povezani* (tudi shared memory): skupen pomnilnik
  - *rahlo povezani* (tudi distributed memory): povezani preko V/I enot
- Večjedrne (multicore) računalnike (več CPE na istem čipu) lahko štejemo med tesno povezane MIMD
  - "pravi" oz. veliki MIMD pa imajo po več tisoč jeder (rekord je trenutno 3 milijone)

15

- SIMD in MIMD so **paralelni računalniki**
  - najbolj zmogljivi superračunalniki so paralelni
  - zmogljivost se običajno meri v številu operacij v plavajoči vejici na sekundo
    - GFLOPS (Giga FLOPS – Floating Point Operations Per Second) pomeni 10^9 operacij / s
      - Cray 1988, 1GFLOPS
    - TFLOPS (Tera FLOPS) pomeni 10^{12} operacij / s
    - PFLOPS (Peta FLOPS) pomeni 10^{15} operacij / s
      - trenutno je rekord 34 PFLOPS
  - od leta 1988 se povečuje zmogljivost za 2x na leto
  - današnji PCji: nekaj GFLOPS

16

Amdahlov zakon

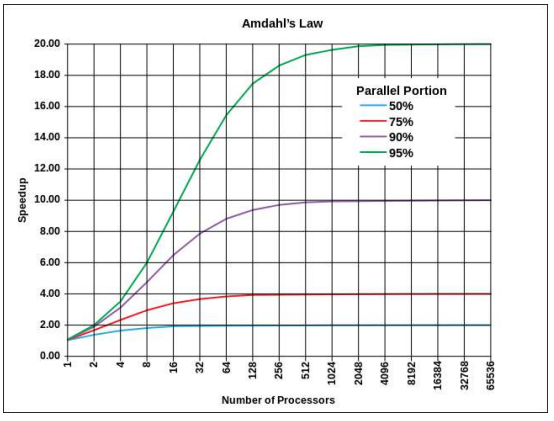
- Vzemimo, da pohitrimo delovanje določenega dela operacij
  - *f* je *zaporedni del(ež)* programa
  - *1-f* je *vzporedni del(ež)* programa
    - pri njem je delovanje *N*-krat hitrejše
      - npr. paralelno izvajanje *N* procesorjev

- Povečanje hitrosti računalnika je tedaj (Gene Amdahl, 1967):

$$S(N) = \frac{1}{f + (1-f)/N} = \frac{N}{1 + (N-1)f}$$

17

  - npr. če je *f* = 0,1, hitrosti računalnika ne moremo povečati za več kot 10-krat, tudi če preostalih 90% časa zmanjšamo na 0 (pohitrimo za faktor $N = \infty$)
  - odvisno je od problema, koliko nam paralelni računalnik koristi



18

- Gustafsonov zakon
  - lahko pa rešimo večji problem
  - če povečujemo problem, se zaporedni del f zmanjšuje in pohitritev postane skoraj linearna: $S(N) \approx N$

- Poskus, da se obide omejitve, ki jih postavlja Amdahlov zakon
  - ne morem te prepeljati hitreje, lahko pa vas gre 5
  - ni vedno možno ☹

19

Računalnik kot zaporedje navideznih računalnikov

- Večine uporabnikov arhitektura računalnika (pravzaprav) posebno ne zanima
  - programske jezike lahko implementiramo na različnih računalnikih
- Tanenbaum, 1984:
  - Računalnik kot zaporedje navideznih računalnikov
  - Vsak nivo si lahko predstavljamo kot navidezni računalnik, ki ima za “strojni” jezik kar jezik tega nivoja (večina uporabnikov se spodnjih nivojev niti ne zaveda)

20

6 nivojev:

- Nivo 5: Višji prog. jezik
  - prevajanje ali interpretiranje
- Nivo 4: Zbirni jezik
  - prevajanje
- Nivo 3: Operacijski sistem
  - interpretiranje
- Nivo 2: Strojni jezik
  - interpretiranje
- Nivo 1: Mikroprogramski jezik
  - interpretiranje
- Nivo 0: Digitalna logika

21

- 2 mehanizma za prehod med nivojema:
  - prevajanje
    - izvorni program v enem jeziku
    - ciljni program (object program) v drugem (nižjem) jeziku
      - izvornega načelno ne rabimo več
  - interpretacija
    - izvorni program se prevaja sproti
      - ukaz se prevede in izvrši
      - rabimo ga ves čas
      - bolj fleksibilno
      - večja prenosljivost
      - manjša hitrost

22

  - delno prevajanje
    - prevajanje v vmesno kodo, ki se jo interpretira
      - npr. Java

Programa:

- prevajalnik
- interpreter

23

Strojna in programska oprema računalnika

- Delitev
  - hardware
  - software
  - firmware
- Strojna in programska oprema sta funcionalno ekvivalentni
  - poljuben računalnik bi se dalo realizirati samo z elektroniko (dovolj kompleksno)

24

10 POMNILNIKI

Lastnosti pomnilnikov

1. **Cena**
   - $/GB
     - SRAM: 2000-5000 $/GB
     - DRAM: 20-80 $/GB
     - Bliskovni: nekaj $/GB
     - Magnetni disk: 0,2-2 $/GB
   - poleg pomnilniških celic je treba v ceno vključiti še vso potrebno elektroniko in/ali mehaniko

2. Hitrost dostopa

- hitrost branja in pisanja
- **čas dostopa** (access time, t_a)
  - čas od pridobitve naslova do pojavitve podatkov
  - je definiran pri branju
  - pri pisanju je podoben
- pri nekaterih pomnilnikih (DRAM) mora po vsakem dostopu preteči nek čas, preden se lahko prične naslednji dostop
  - **čas cikla** $t_c = t_a$ + čakanje
- **hitrost dostopa** (access rate) $b_a = 1/t_c$
- Gledano s strani naprave, ki bere ali piše v GP, imamo še čas t_p za prenos preko podatkovnih poti
  - prenos naslova in kontrolne informacije do GP ter prenos podatka nazaj (pri branju)
  - t_p je v rangu nekaj ns/m
- GP zaradi velikosti ne more biti na čipu CPE

- DRAM:
  - pri dostopu do poljubnega naslova: čas dostopa $t_a \sim 50$ ns, čas cikla $t_c \sim 60$ ns
  - pri dostopu do zaporednih naslovov hitreje
- SRAM:
  - čas dostopa t_a od 0,5 do 2,5 ns
- Hitrost dostopa pri magnetnem disku je približno 100.000 krat nižja kot pri polprevodniških pomnilnikih
  - v rangu več ms
  - nekje vmes so elektronski diski (EEPROM, Flash)
    - npr. USB ključki
- Razlog za uporabo pomnilniške hierarhije so velike razlike med pomnilniki v hitrosti in ceni
  - kljub zapletenosti, ki jo pomnilniška hierarhija vnaša, so prihranki v hitrosti tako veliki, da se jim ni mogoče odpovedati

3. Način dostopa

3.1 Naključni dostop (random access)

- čas dostopa t_a je konstanten in znan vnaprej ter neodvisen od prejšnjih naslovov
- RAM (random access memory)
  - DRAM (dinamični RAM) – GP
  - SRAM (statični RAM) - predpomnilnik
- načini dostopa do zaporednih bitov pri DRAM so hitrejši (vendar jih ne štejemo pod kategorijo zaporednega dostopa)
  - način strani (page mode, PM)
    - podamo NV, nato pa različne NS
    - potrebno je le, da so biti v isti vrstici (tudi, če niso zaporedni)
- rafalni ali eksplozijski način (burst mode)
  - zelo hiter, danes zelo pogosto uporabljan
  - dostop do zaporednih bitov s pomočjo majhnega internega števca, ki se prišteva k NS

3.2 Zaporedni dostop (serial access)

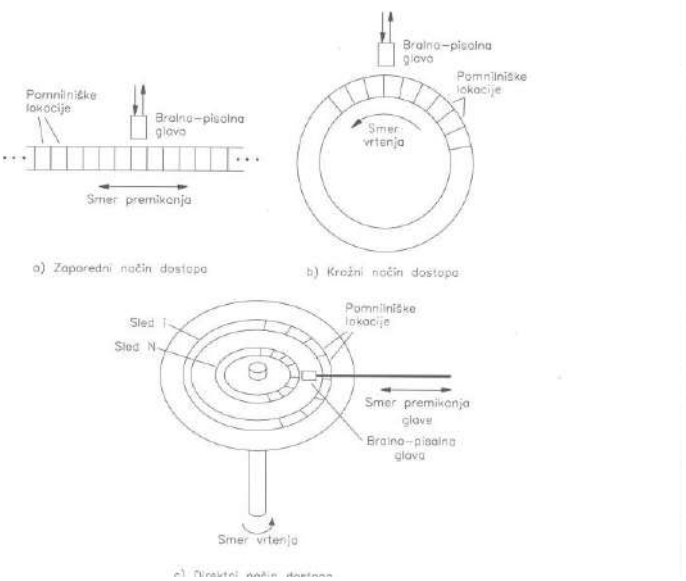
- čas dostopa je odvisen od prejšnjega naslova
  - če smo bili na naslovu A, je takoj dostopen le naslov A+1
- npr. magnetni trak

3.3 Krožni dostop (rotational access)

- posebna vrsta zaporednega dostopa
  - kot npr. magnetni trak, ki bi bil zlepljen v zanko
- npr. magnetni disk s fiksnimi glavami
- povprečen čas dostopa t_a je enak ½ periode vrtenja

3.4 Kombinacija zaporednega in krožnega dostopa

- magnetni in optični diski s premičnimi glavami
- bralno-pisalna glava se najprej premakne na ustrezno sled (zaporedni dostop), nato pa imamo krožni dostop
- hitrejši od zaporednega ali krožnega dostopa



Asociativni pomnilniki

- Pomnilniki z dostopom **preko (dela) vsebine** oz. *vsebinsko naslovljivi* (CAM, Content Addressable Memory) (ostali pomnilniki dostopajo **preko naslova**)
  - podamo del besede
  - primerja se z vsemi vpisanimi besedami (z ustreznimi biti)
  - primerjava je paralelna, zato zelo hitra
  - velika poraba logike (komparatorji), zato so AP majhni (< 1K)

4. Spremenljivost

- **Bralni pomnilniki** (ROM – Read Only Memory)
  - lahko ga beremo, vpis ni možen (vsaj za uporabnika ne)
  - luknjane kartice, tisk na papirju, CD-ROM, polprevodniški ROM
  - vsebina je obstojna (tj. tudi brez vira energije oz. napajanja)

- **Programirljivi bralni pomnilniki** (Programmable ROM - PROM)
  - lahko jih programiramo (vpišemo vsebino), sicer ne posebno hitro
  - PROM oz. OTP (One Time Programmable): na principu varovalk
  - EPROM (Erasable PROM): možen večkraten vpis in brisanje
    - programiranje z visoko napetostjo (rabimo programator), brisanje z UV-svetlobo (rabimo brisalnik) – čip ima na vrhu okence
  - EEPROM (Electrically Erasable PROM)
    - programiranje in brisanje z normalno napetostjo
  - Flash: podoben EEPROMu

  - v računalnikih so bralni pomnilniki uporabljeni za shranjevanje **zagonskih programov**, ki se vključijo ob vklopu računalnika
    - majhen del GP je torej tipa ROM

- **Bralno-pisalni pomnilniki** (Random Access Memory)
  - z enako lahkoto beremo in pišemo
  - kratica je zavajajoča: to ni pomnilnik z naključnim dostopom!

5. Obstojnost

Obstaja več razlogov za izgubo informacije:

- **Destruktivno branje**
  - pri DRAM je informacija shranjena kot naboj na (zelo) majhnih kondenzatorjih
  - pri branju se kondenzatorji v vrstici praznijo, zato jih je treba ponovno nabiti
- **Dinamično shranjevanje**
  - tudi sicer se kondenzatorji s časom praznijo (dielektrik oz. izolator ni idealen) in jih je potrebno **osveževati** (refresh) večkrat na sekundo
  - odtod ime **dinamični** RAM
    - statični RAM ne potrebuje osveževanja
  - vrstica se prebere in zapiše nazaj
- **Izpad napajanja**
  - **Obstojni** pomnilniki (nonvolatile) ohranijo vsebino tudi, ko pride do izpada napajanja (ROM, magnetni disk, optični disk, ...)
  - RAM so neobstojni (volatile)

6. Zanesljivost

- Pomnilniki brez gibljivih delov (solid state), tj. polprevodniški, so bolj zanesljivi kot magnetni diski, pri katerih je potrebno mehanično gibanje
- Tudi pri polprevodniških pa so možne napake
  - kondenzator pomnilne celice pri DRAM je tako majhen, da mu lahko stanje spremenijo že kozmični žarki
    - to je **mehka napaka**, ker se celica ne poškoduje in dela naprej
    - zaradi mehkih napak se uporabljajo **kode za detekcijo in korekcijo napak** (dodatni biti)
  - **Trda napaka** (ki je redkejša) pa povzroči trajno okvaro celice

Zaščita glavnega pomnilnika

- Operacijski sistem (OS) je program (običajno več programov), ki teče na računalniku in upravlja s programskimi in strojnimi viri, npr.
  - omogoča (lažji) dostop do V/I naprav
  - upravljanje s pomnilnikom
  - večopravilni OS omogoča, da hkrati teče več procesov, itd.
- S pojavom prvih OS se pojavi potreba po mehanizmu, ki omogoča zaščititi en program pred posegi drugega programa
- Del OS mora biti stalno v GP
- Če programer zaradi napake v svojem programu spremeni vsebino pomnilniških lokacij, kjer je OS, lahko pride tudi do **razpada sistema** (crash)
  - v tem primeru je treba ponovno prenesti programe OS s pomožnega v glavni pomnilnik (s ponovnim zagonom računalnika)

- Problem se je še povečal s pojavom večuporabniških (multiuser) in večopravilnih (multitasking) OS
  - istočasno se izvaja le en program (če imamo eno CPE), vendar si programi delijo isti pomnilniški prostor
  - treba je poskrbeti, da en program ne posega v prostor drugega (namenoma ali nehote, vseeno)
    - predvsem pisanje (spreminjanje), pa tudi branje, če gre za tajne informacije
- Nekateri programi OS so v bralnem pomnilniku in so s tem zaščiteni proti pisanju
  - ostali del OS se prenese z diska v GP
  - če bi bil ves OS v ROMu, bi bilo treba pri novejši verziji spremeniti čipe (oz. vsaj firmware)
    - nerodno, poleg tega to ne ščiti uporabnikov

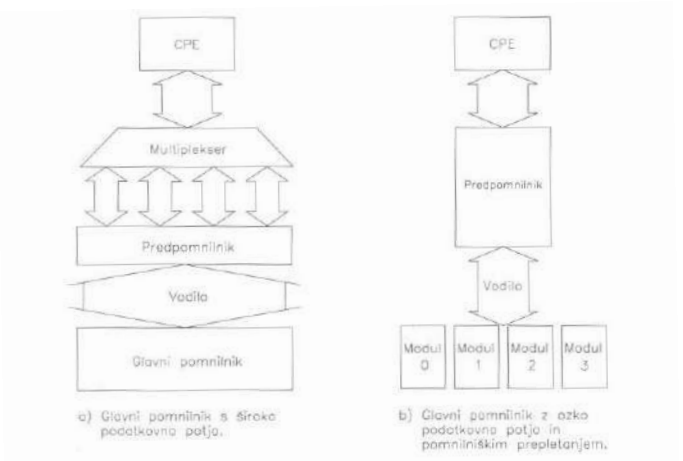
- Najpreprostejši zaščitni mehanizem je par registrov, ki vsebuje spodnjo in zgornjo mejo naslova, ki pripada programu
  - vsak pomnilniški naslov *A* se pred dostopom do pomnilnika preveri
  - naslov je veljaven, če velja

    spodnja meja <= A <= *zgornja meja*
  - slabosti:
    - programi morajo zasedati zvezen prostor v pomnilniku
    - vse besede so zaščitene na enak način
      - raje bi imeli "samo branje", "branje ali pisanje", ...

- Boljše rešitve uporabljajo **bloke** ali **strani** (pages) velikosti 1024, 2048 ali 4096 besed, ki so zaščiteni vsak zase
  - vsak program zaseda določeno število strani
  - vsaka stran ima svoj **zaščitni ključ** (protection key), ki je neko zaporedje bitov
    - shranjeno v tabeli strani za navidezni pomnilnik

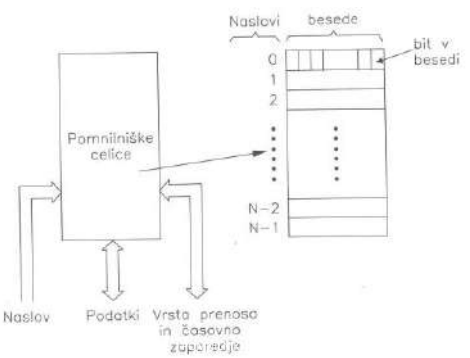
- Cilj zaščite je običajnim uporabnikom preprečiti dostop do *privilegiranega načina* delovanja (privileged mode)
  - v določenih primerih uporabnik potrebuje storitve, ki so dovoljene samo v privilegiranem načinu
    - mnogi OS imajo za ta namen *sistemske klice* (system calls)
  - preprosti sistemi (npr. vgrajeni - embedded) imajo običajno samo en način (privilegiran)
    - gonilnike naprav (device drivers) lahko programira običajen uporabnik
- Poleg strojne zaščite je možna tudi programska

- Kljub PP je potrebno eventualno še vedno dostopati do GP
  - Hitrost pomnilnikov DRAM se povečuje bistveno počasneje od hitrosti CPE
- Ena od možnosti pohitritve je povečanje števila naenkrat prenešenih bitov. 2 načina:
  1. **Širše podatkovne poti do GP**.
     - dostop do sestavljenih pomnilniških besed
  2. **Pomnilniško prepletanje** (memory interleaving).
     - GP je razdeljen na *m* samostojnih delov M_0, M_1, ..., M_{m-1}
     - to so **moduli** oz. **banke**
     - ***m*-kratno prepletanje** (*m*-way interleaving)
     - širina podatkovnih poti se ne poveča (vsaj v osnovni izvedbi)
     - vsak modul je samostojen pomnilnik, ki deluje neodvisno od ostalih
     - z dekodiranjem določenih bitov naslova se izbere enega od modulov
     - možnih je *m* istočasnih dostopov
     - po začetni zakasnitvi je možen po en prenos na urino periodo



Organizacija glavnega pomnilnika

- Pove, kako so biti sestavljeni v pomnilniške besede in kakšen je dostop do njih
- GP je videti kot enodimenzionalno zaporedje pomnilniških besed; vsaka ima svoj enoličen naslov



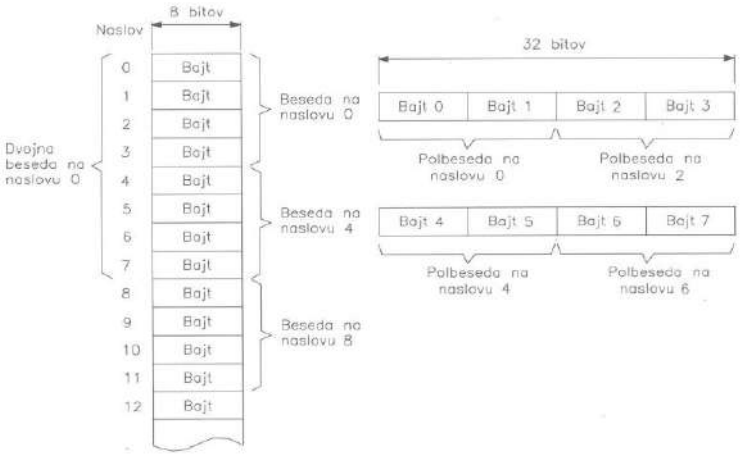
- Osnovna parametra pomnilnika sta:
  1. **Pomnilniška beseda**
     - to je najmanjše število bitov s svojim naslovom
       - **dolžina besede** (običajno 1B oz. 8 bitov)
     - običajno je možen dostop do več besed
  2. **Pomnilniški naslov**
     - binarno število
     - **dolžina naslova** določa velikost pomnilniškega prostora
       - pri *m*-bitnem naslovu $a_{m-1}...a_1a_0$ je lahko največ 2^m besed

- 3 vrste signalov
  - naslovni
  - podatkovni
  - kontrolni
- Dolžina registrov CPE je enaka mnogokratniku dolžine pomnilniške besede
- Pomnilniški prostor vsako leto naraste s faktorjem med 1,5 in 2 (torej eksponentno)
- Velikost naslova določa širino vsega, kar lahko vsebuje naslov:
  - ukazov
  - registrov
  - aritmetike za računanje naslova
- Zato je povečati dolžino naslova izjemno težko
  - premajhna dolžina naslova je največja možna napaka pri razvoju novega računalnika, ker jo je kasneje skoraj nemogoče popraviti

- GP, ki omogoča dostop do **sestavljenih pomnilniških besed**, je možno narediti na 2 načina:
  1. več paralelnih pomnilnikov
     - spodnji biti naslova določajo, za katerega gre
       - npr. pri dostopu do 8 besed naenkrat je 8 pomnilnikov, spodnji 3 biti določajo pomnilnik
  2. vedno se naredi dostop do vseh (npr. 8) besed

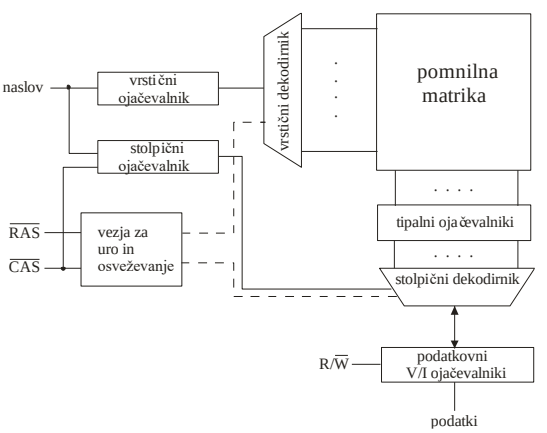
  - Kjer je možen dostop do sestavljenih besed, je dobro, če je podatkovno vodilo temu ustrezno široko, sicer je potrebnih več prenosov
    - tudi, če je več prenosov, programer tega ne vidi

- Primer:
  - dolžina pomnilniške besede 1B
  - dva sosedna bajta tvorita polbesedo (halfword, 16 bitov)
  - štirje sosedni bajti tvorijo besedo (word, 32 bitov)
  - osem sosednih bajtov tvorijo dvojno besedo (doubleword, 64 bitov)
  - npr. pravilo debelega konca
    - naslov vsake od sestavljenih besed je enak naslovu bajta z največjo težo
  - pri večini računalnikov je potrebna **poravnanost**
    - sestavljene besede morajo biti na naslovih, ki so večkratniki 2, 4, oz. 8
    - sicer je potrebnih več dostopov!
    - npr. če je polbeseda na 24-bitnem naslovu 10FFFF
      - prvi bajt ima naslov 10FFFF, drugi pa 10FFFF+1 = 110000
      - razlikujeta se v 17 bitih!

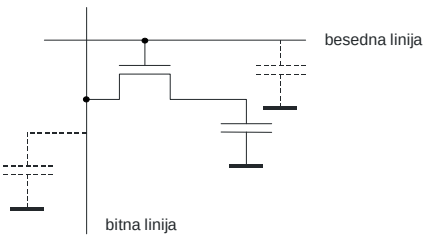


Tehnologija polprevodniških pomnilnikov

- **DRAM** (Dinamični RAM)
  - zgradba
    - izhodi vrstičnega dekodirnika so *besedne linije*
    - na stolpični dekodirnik so vezane *bitne linije*
    - naslov je razdeljen na 2 dela:
      - vrstični
      - stolpični



- Pomnilna celica DRAM
  - kondenzator
    - nabit: eno logično stanje (npr. “1”); prazen: drugo logično stanje (npr. “0”)
    - C_s ~ 20fF (s ... storage)
  - stikalni transistor (MOS)

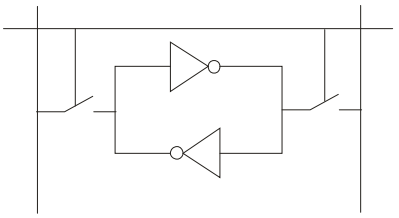


- DRAM vsebuje bitno ravnino oz. matriko ALI
  - v njej so besedne in bitne linije, na presečiščih pa so pomnilne celice
  - razlog za 2D organizacijo je velikost dekodirnika in število ter dolžina linij
    - npr. 1Mb pri 1D: dekodirnik 20/1M, 1M besednih linij, zelo dolga bitna linija (z 1M celicami! – ogromna kapacitivnost)
    - 2D: 2 dekodirnika 10/1024, 1024 besednih linij, 1024 bitnih linij, 1024 celic na bitni liniji
- Primer: DRAM 32Mb x 1
  - 25-bitni naslov: 15 (vrstični del) + 10 (stolpični del)
    - torej 2^{15} besednih linij, 2^{10} bitnih linij
    - običajno je besednih linij več kot bitnih
    - zato so lahko krajše (hitrejši dostop zaradi manjše kapacitivnosti)
- Primer: DRAM 32Mb x 8
  - podobno, vendar 8 bitnih ravnin

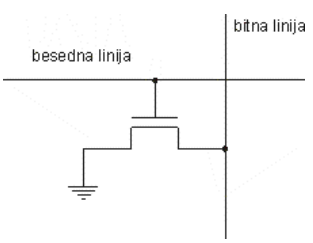
- Pomnilniški dostop:
  - bitne linije *prednabijemo* (precharge) na polovično napetost
    - se ne izpraznijo prav hitro zaradi relativno velike kapacitivnosti (C_b), ki je posledica parazitnih kapacitivnosti velikega števila celic na liniji
  - podamo naslov vrstice (NV)
  - aktiviramo signal RAS' (row address strobe), ki je aktivno nizek
  - vsebina vrstice (naboj na kondenzatorjih) gre preko bitnih linij na *tipalne ojačevalnike* (sense amplifier, SA)
    - v resnici ne čakamo, da se kondenzator popolnoma izprazni, ampak le delno (zaradi hitrosti)
    - ker je $C_b > C_s$, se napetost bitne linije le malo spremeni - običajno nekaj sto mV
    - SA zazna to razliko in vrne logično vrednost (0 ali 1)
  - vrednosti se shranijo v *register vrstice* (oz. *buffer*)
  - podamo naslov stolpca (NS)
  - aktiviramo signal CAS' (column address strobe), ki je tudi aktivno nizek
  - pri bralnem dostopu (WE'(write enable) = 1) dobimo na izhodu iskani bit
    - pri pisalnem dostopu (WE' = 0) se bit vpiše v register vrstice
  - register vrstice se vpiše nazaj v celice

- DRAMi uporabljajo *naslovno multipleksiranje*
  - naslov vrstice in naslov stolpca sta na istih pinih
    - s tem se zmanjša število priključkov (pinov) za bite naslova
      - naslovi so pri DRAMih seveda dolgi (npr. 30 bitov pri 1Gb)
      - priključki so glavni dejavnik pri ceni čipa
    - ne izgubimo kaj dosti na času, saj potrebujemo NV prej kot NS
- Današnji DRAMi so sinhronski (SDRAM)
  - sinhronizirani so s sistemsko uro
  - imajo 3-stopenjski cevovod
    - register na vhodu
    - DRAM (asinhronski)
    - register na izhodu
  - najpogostejši so DDR (1,2,3)
    - double data rate

- **SRAM** (Statični RAM)
  - zgradba je v osnovi podobna kot pri DRAM
  - pomnilna celica
    - zapah (podoben RS-zapahu, le način vpisovanja je drugačen)
    - informacija se ne izgublja (vkolikor ne izključimo napajanja)
      - zato se celica imenuje statična



- Pomnilna celica pri **ROM**:
  - bitna linija je vnaprej nabita (prednabita)
  - signal na besedni liniji povzroči, da transistor prične prevajati
  - tok teče iz bitne linije proti masi, zato se zmanjša naboj na bitni liniji
  - posledično upade napetost bitne linije, kar zazna posebno vezje (v izhodni stopnji), ki to tolmači kot “0”
  - če transistorja ni, napetost ne upade (“1”)



- **Bliskovni** pomnilnik (**Flash** memory) je vrsta programirljivega pomnilnika ROM (programmable ROM), za katerega lahko uporabnik določi oz. vpiše vsebino, ta pa je potem obstojna (z izklopom napajanja se ne izgubi)
  - V Flash celici je izpeljanka običajnega MOS tranzistorja, ki ima znotraj oksidne plasti dodatno (t.i. plavajočo) plast – kadar je ta nabita z elektroni, tranzistor efektivno ne prevaja (kakor da ga v celici ne bi bilo)

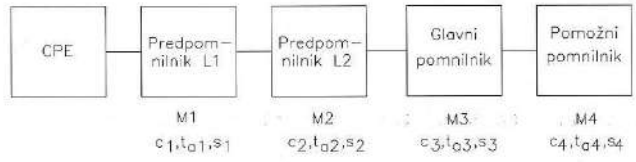
11
NAVIDEZNI POMNILNIK

Navidezni pomnilnik

- Razlog za pojav NP je želja po povečanju GP
- Navidezni pomnilnik (virtual memory) je prostor v pomožnem pomnilniku (magnetni disk), ki je za uporabnika videti kot GP
  - to zahteva dodatno logiko v CPE
    - običajen dostop do pomožnega pomnilnika poteka sicer preko V/I ukazov (tj. bistveno drugače od dostopa do GP)
    - mehanizem naredi te razlike nevidne
- Vse skupaj je del pomnilniške hierarhije

Pomnilniška hierarhija

- PH je zaporedje pomnilnikov (M_1, M_2, ..., M_n), pri katerem vsak pomnilnik komunicira samo s svojima sosedoma



- Vsebina M_{i-1} je podmnožica vsebine M_i
- Celotna informacija je le na M_n
- CPE vidi PH kot en sam pomnilnik
  - naslov, ki ga da CPE, je naslov besede na najvišjem nivoju (M_n)
  - vendar hitrejši

- PH funkcionira le zaradi lokalnosti pomnilniških dostopov
- PH prinese tudi probleme: čas dostopa spremenljiv in težko napovedljiv (običajno le statistično)

cena / bit: $c_i > c_{i+1}$

dostopni čas: $t_{ai} > t_{a,i+1}$

velikost: $s_i < s_{i+1}$

N dostopov:
- v N_1 primerih je inf. v M_1,
- v N_2 primerih je inf. v M_2 in ne v M_1,
- v N_3 primerih je inf. v M_3 in ne v M_1 ali M_2,
- itd.

$$N = N_1 + N_2 + \ldots N_n$$

Globalna verjetnost zadetka

$H_1 = N_1/N$
$H_2 = (N_1 + N_2)/N$
...
$H_n = (N_1 + N_2 + \ldots + N_n)/N = 1$

Ekskluzivna verjetnost zadetka

$h_i = N_i/N$

to ni lokalna verj. zadetka (le-ta je $N_i/(N - N_1 - N_2 - \ldots - N_{i-1})$)

$h_1 = H_1$
$h_i = H_i - H_{i-1}$, i=2, 3, ..., n

torej:

$h_2 = H_2 - H_1$,
$h_3 = H_3 - H_2$, ...

Cena bita

$$c = (c_1s_1 + c_2s_2 + \ldots + c_ns_n) / (s_1 + s_2 + \ldots + s_n)$$

Če želimo, da bo c blizu c_n, mora biti s_n veliko večji od vseh ostalih skupaj.

Povprečen čas dostopa, kot ga vidi CPE:
Če informacije ni na nivojih pod i, se mora prenesti z nivoja i na nižje nivoje.
Čas dostopa do nivoja i je

$$T_i = \sum_{k=1}^{i} t_{ak} = t_{a1} + \cdots + t_{ai}$$

Verjetnost za to je h_i.

Povpr. čas dostopa n-nivojske PH:

$$\begin{aligned} t_a = \textstyle\sum_{i=1}^{n} h_i T_i &= h_1T_1 + h_2T_2 + \cdots + h_nT_n \\ &= h_1t_{a1} + h_2(t_{a1} + t_{a2}) + \cdots + h_n(t_{a1} + \cdots + t_{an}) \\ &= t_{a1} \cdot 1 + t_{a2} \cdot (1 - H_1) + \cdots + t_{an} \cdot (1 - H_{n-1}) \\ &= t_{a1} + \textstyle\sum_{i=2}^{n} (1 - H_{i-1}) \cdot t_{ai} \end{aligned}$$

Če imamo samo 2 nivoja:

$$t_a = t_{a1} + (1-H_1)*t_{a2}$$

Bolj pravilno je

$$t_a = t_{a1} + (1-H_1)*t_B \qquad (t_B \text{ je čas prenosa bloka})$$

Lahko sta M_1 PP in M_2 GP, lahko pa sta M_1 GP in M_2 HD.
Zgrešitvena kazen pri PP je 10-100, pri HD pa 10-100 milijonov!

Pri NP se zato uporabljajo drugačne rešitve kot pri PP:

- dovolj veliki bloki, da ni preveč prenosov
- čista asoc. preslikava (da je čimmanj zgrešitev)
- zamenjava blokov se opravlja programsko (ne strojno)
  - Večja počasnost prog. rešitve je zanemarljiva v primerjavi s časom dostopa do bloka na disku.
  - Poleg tega je možno uporabiti bolj zahtevne algoritme za izbiro bloka, ki naj se zamenja
    - že z majhnim zmanjšanjem zgr. kazni se počasnost prog. rešitve več kot odtehta

NP uporabljajo le pisanje nazaj (pisanje skozi ni uporabno)

Zaradi zgr. je povp. hitrost dostopa manjša, kot če NP ne bi imeli.

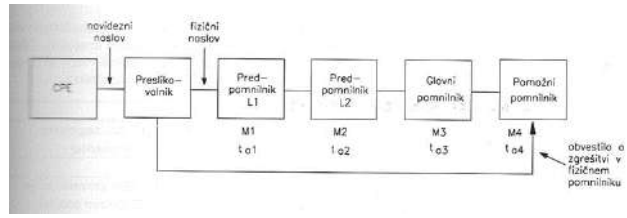
- Npr., da dovolimo 10% povečanje časa dostopa. Kolikšna sme biti največja verjetnost zgr. 1-H?
  - t_{a1} = 40ns (DRAM), t_B = 15ms (mag. disk)

$$t_a = t_{a1} + (1-H)*t_{a2} = 1{,}1*t_{a1}$$
$$(1-H)*t_{a2} = 0{,}1*t_{a1}$$
$$(1-H) = 0{,}1*t_{a1}/t_{a2} = 4\text{ns}/15\text{ms}$$
$$= 0{,}27*10^{-6} = 2{,}7*10^{-5}\,\%$$

Verj. zgr. je lahko največ $2{,}7*10^{-5}$ %, torej zelo majhna!

Preslikovanje navideznih naslovov

- Naslovi, ki jih daje CPE, se pri vsaki PH vedno nanašajo na najvišji nivo.
- Pri rač. z NP se imenujejo **navidezni naslovi**
- Pri vsakem pomnilniškem dostopu je **navidezni naslov** potrebno preslikati v **fizični naslov** (v GP). Le-ta gre v PP.



Preslikovanje:

$A_f = f(A_n)$

A_f ... fizični naslov

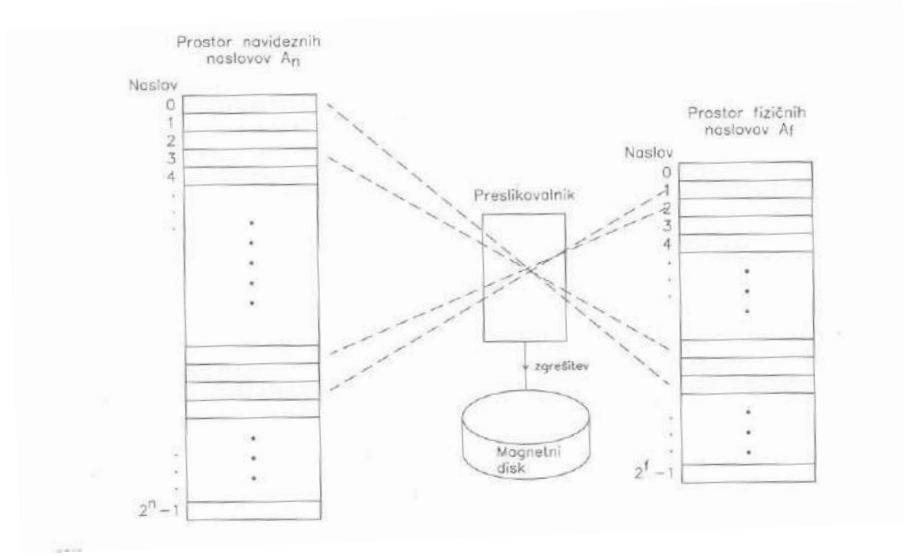
A_n ... navidezni naslov

f ... preslikovalna funkcija

- f se sproti prilagaja, da je verj. zadetka čimvečja

Če je naslov A_n prisoten v GP, ga preslikovalnik preslika v A_f

- če ni, preslikovalnik ugotovi zgrešitev in blok se prenese z diska v GP, nato se A_n preslika v A_f in CPE izvede dostop



- Preslikovanje izvaja **enota za upravljanje s pomnilnikom** (MMU, memory management unit)

- Pri NP je potrebna samo logika za preslikovanje, ne pa tudi za zamenjavo (kot pri PP)
  - zamenjevanje blokov je realizirano programsko
    - prekinitev, PSP prenese blok z diska

Vrste navideznih pomnilnikov

2 vrsti:

- NP z bloki fiksne velikosti (**strani**)
  - Strani: 4, 8, 16KB (podobne so blokom v PP)
    - naslov je enodimenzionalen
    - zamenjava bloka preprosta (ker so vsi bloki enako veliki)
    - velikost strani lahko tudi prilagodimo lastnostim diska
- NP z bloki spremenljive velikosti (**segmenti**)
  - Segmenti so lahko različnih velikosti.

Ostranjevanje

paging

- Pomožni pom. je razdeljen na bloke enake velikosti (**strani**, pages). Vse strani skupaj sestavljajo NP.
- GP je razdeljen na enako velike **okvire strani** (page frames).
- Vsako stran NP lahko prenesemo v poljuben okvir

Naj bo

- NP velikosti 2^n besed (npr. 8-bitnih)
- stran velikosti 2^p besed
- GP velikosti 2^f besed

število strani v NP = $2^n / 2^p = 2^{n-p}$

število okvirov v GP = $2^f / 2^p = 2^{f-p}$

Navidezni naslov	Stran
0 ... 2^p-1	0
2^p ... $2*2^p$-1	1
$2*2^p$ ... $3*2^p$-1	2
. . .	
$(2^{n-p}-2)*2^p$ ... $(2^{n-p}-1)*2^p$ -1	2^{n-p}-2
$(2^{n-p}-1)*2^p$ ... $2^{n-p}*2^p$ -1 = 2^n - 1	2^{n-p}-1

Fizični naslov	Okvir strani
0 ... 2^p-1	0
2^p ... $2*2^p$-1	1
	. . .
$(2^{f-p}-1)*2^p$ ... $2^{f-p}*2^p$ - 1 = 2^f - 1	2^{f-p}-1

Preslikovalno funkcijo definira **tabela strani** (**page table**).

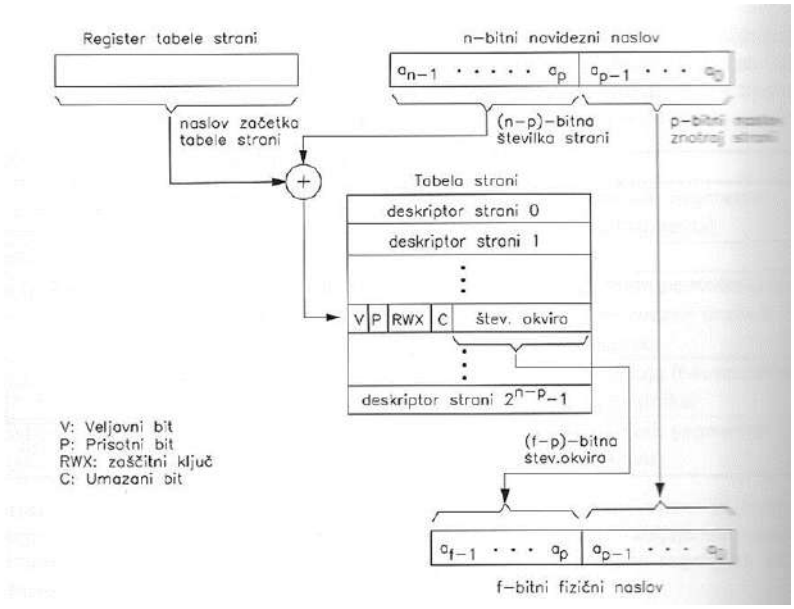
- Za vsako stran NP ima tabela eno polje (**opisnik strani**, **page descriptor**).

Notranja fragmentacija: vsak program zaseda določeno št. strani, v povp. pa ½ strani ostane neizkoriščena

Navidezni naslov ima 2 dela:

- zgornjih n-p bitov je **številka strani**
- spodnjih p bitov je **naslov besede znotraj strani** (ta po preslikavi ostane enak)

Številka strani se preko tabele strani preslika v številko okvira strani.



Opisnik strani ima 5 parametrov:

- **Veljavni bit V** (valid)
  - na začetku je 0
- **Prisotni bit P** (present) pove, ali je stran v GP
  - P=1: 'zadetek' (take strani se imenujejo **aktivne strani**)
    - FN pove, kje v GPju je stran
  - P=0: 'zgrešitev' (**napaka strani** (**page fault**))
    - FN ni veljaven
    - stran je treba prenesti z diska v GP
      - to naredi PSP, ko se sproži past
- **Umazani bit C** (change)
  - Ko se stran prenese v GP, se C postavi na 0
  - Če pride do pisanja na vsaj en naslov na tej strani, gre C na 1
  - Ko se stran zamenja z neko drugo, je treba stran prenesti na disk samo, če je C=1 (sicer je itak enaka kot na disku)

- **Zaščitni ključ RWX** pove vrsto dostopa do strani
  - read, write, execute
  - Namenjeno zaščiti pomnilnika
  - Vsak program ima svojo tabelo strani
  - Spreminjanje vsebine tabel strani je dovoljeno samo v privilegiranem načinu delovanja OS (le v takem načinu je dovoljeno pisati v del pom., kjer je tab. strani)
  - Kadar iz zašč. ključa sledi, da dostop ni dovoljen, se sproži past zaščite
    - ta ima višjo prioriteto kot napaka strani (bit P se ignorira)
- **Številka okvira FN** (Frame number) podaja številko okvira, v katerem je stran
  - naslov okvira je FN $\times$ 2^p (če je seveda P=1, sicer stran ni v GPju)
  - k FN $\times$ 2^p se prišteje naslov znotraj strani in dobimo fizični naslov besede, do katere se dostopa
  - tudi naslov okvira ima spodnjih p bitov enakih 0 (tako kot naslov strani v NP)

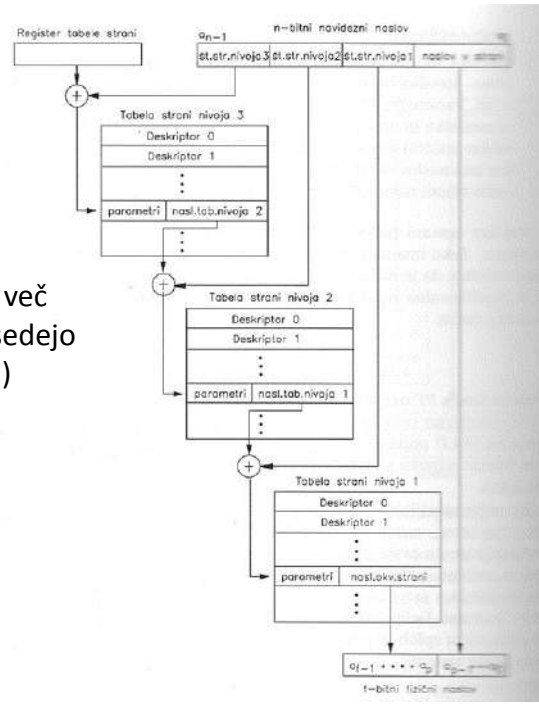
Preslikovalna funkcija *f*:

$A_f = FN * 2^p + A_n(p-1 : 0)$ PT ... tabela strani

$FN = PT[A_n(n-1 : p)]$ FN ... številka okvira iz PT

- Preslikovalna funkcija velja pri pogoju, da je v opisniku strani bit P enak 1. Sicer imamo napako strani in stran se najprej prenese v GP, šele nato se izvede preslikava.
- To je **linearno preslikovanje**.

Danes so PT v več nivojih (da zasedejo manj prostora)

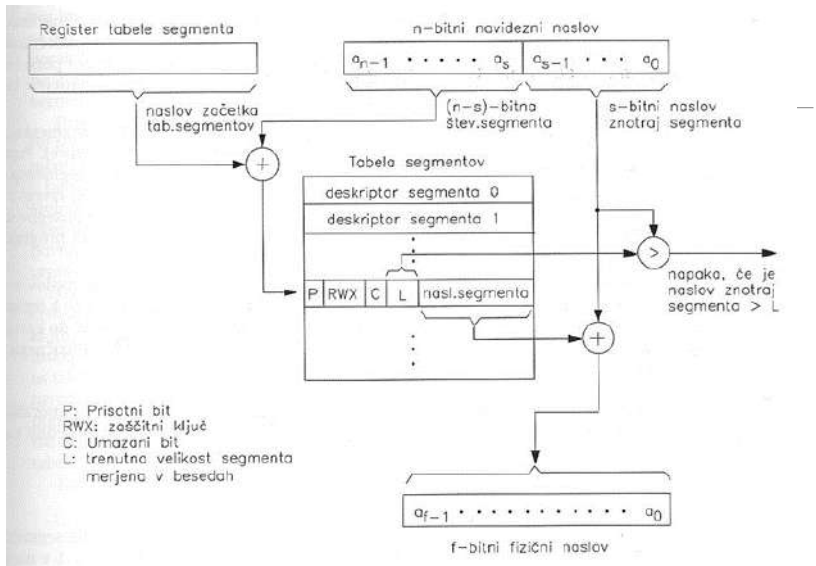


- **Premetavanje** (thrashing): Kadar je napak strani veliko, se lahko večji del časa prenašajo strani med GP in diskom, uporabniški programi pa čakajo.
- Ostranjevanje je enostavno, ne omogoča pa uporabe boljših načinov za zaščito pomnilnika, pa tudi več programov si ne more deliti istega fizičnega prostora (pri segmentaciji si lahko).

Segmentacija

- Segmenti so različnih dolžin
- Vsebina segmenta ima za program svoj pomen
- Število in velikost segmentov se lahko med izvajanjem programa spreminjata
- Segment se lahko prenese na poljuben naslov v GP

- Zgornjih *n-s* bitov navideznega naslova določa **številko segmenta**, spodnjih *s* bitov pa **naslov besede v segmentu** (odmik, segment offset).
- V **tabeli segmentov** ima vsak segment svoj **opisnik segmenta** (ki ga naslovimo s številko segmenta + začetni naslov tabele), ki vsebuje tudi naslov segmenta.
- Fizični naslov dobimo tako, da seštejemo naslov segmenta in naslov besede v segmentu.



- Hiba je v tem, da je odmik treba prištevati – zato za prevajalnike oz. programerje ni neviden.

Segmentacija z ostranjevanjem

- Segmenti so razdeljeni na strani. Potrebna je tabela segmentov in tabele strani (po ena za vsak segment).
- Namen je izkoriščanje prednosti segmentacije, vendar brez problemov, ki se pojavijo pri čisti segmentaciji.

Problemi pri realizaciji navideznega pomnilnika

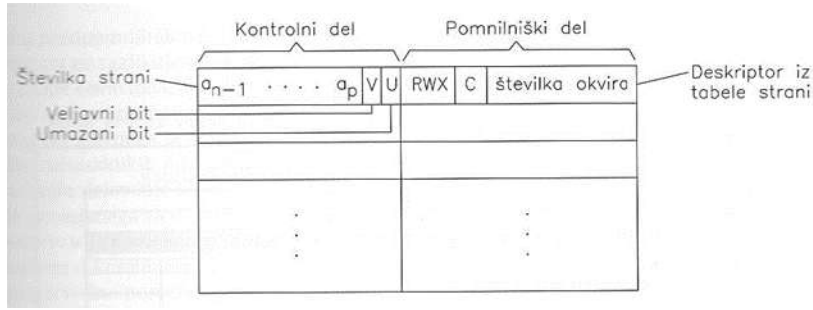
Problemov je več, najpomembnejši pa je, kako pospešiti preslikovanje.

Pri vsakem pom. dostopu sta potrebna 2 dostopa (pri več nivojih še več):

- Dostop do tabele strani v GP
- Dostop do fizičnega naslova

Računalnik bi postal prepočasen.

Preslikovalni predpomnilnik (translation cache) ali **TLB** (translation lookaside buffer) vsebuje podmnožico opisnikov, ki so bili nedavno uporabljeni.


Kontrolni del
Pomnilniški del
Številka strani
a_{n-1} · · · · a_p
V
U
RWX
C
številka okvira
Deskriptor iz tabele strani
Veljavni bit
Umazani bit