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

Literatura

Osnovna:

Dušan KODEK: **Arhitektura in organizacija računalniških sistemov**, BI-TIM, Ljubljana, 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.

Vaje

Asistenta:

Dr. Miha Janež (miha.janez@fri.uni-lj.si, R3.56)

Dr. Ratko Pilipović (ratko.pilipovic@fri.uni-lj.si, R2.41)

Na vajah asistent pokaže nekaj primerov, potem pa se samostojno rešujejo naloge (se ne ocenjujejo) – asistent pomaga pri nejasnostih

Dodatne naloge z rešitvami za utrjevanje snovi bodo dostopne na učilnici

Oceno vaj pridobite z dvema kolokvijema

Ocena vaj je pozitivna (vaje opravljene), če je povprečje kolokvijev vsaj 30 %, vsak pa je vsaj 20 %

Ocena vaj velja le za tekoče šolsko leto

Obveznosti

Ocena predmeta:

- Opravljene vaje (2 kolokvija) so pogoj za pristop k izpitu
- Pisni izpit
- Teoretični izpit

Za podrobnosti glej spletno učilnico (stran Pravila).

Asistenta

Dr. Miha Janež (miha.janez@fri.uni-lj.si, R3.56)

Dr. Ratko Pilipović (ratko.pilipovic@fri.uni-lj.si, R2.41)

Opozorila

Ocena vaj, ki je pogoj za pristop k izpitu, je pozitivna (vaje opravljene), če je povprečje kolokvijev vsaj 20 % !

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 vaj (opravljeno) 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 ($\rightarrow$ pomnilnik)
- možgani ($\rightarrow$ procesor)

Papir

- ukazi (navodila)
- operandi

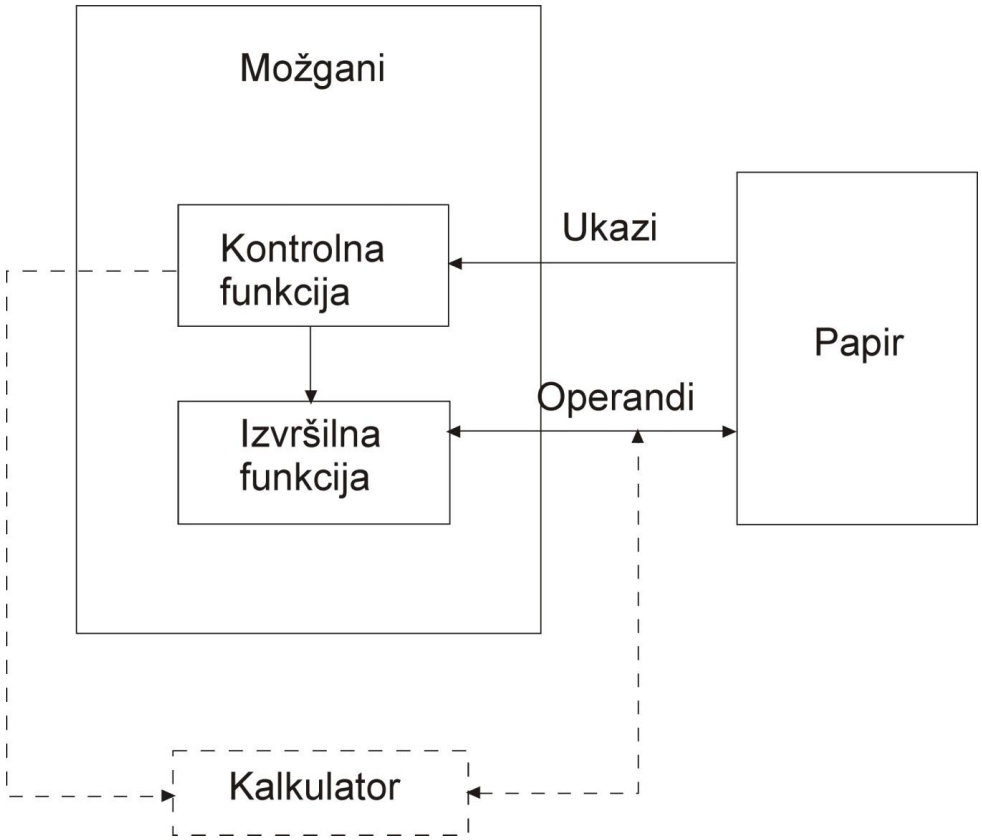
Možgani pri računanju opravljajo 2 funkciji:

- kontrolna funkcija
  - prevzema ukaze in skrbi za pravilen vrstni red izvrševanja ukazov
- izvršilna funkcija
  - npr. seštevanje, množenje, itd.

Papir lahko delimo v 2 vrsti:

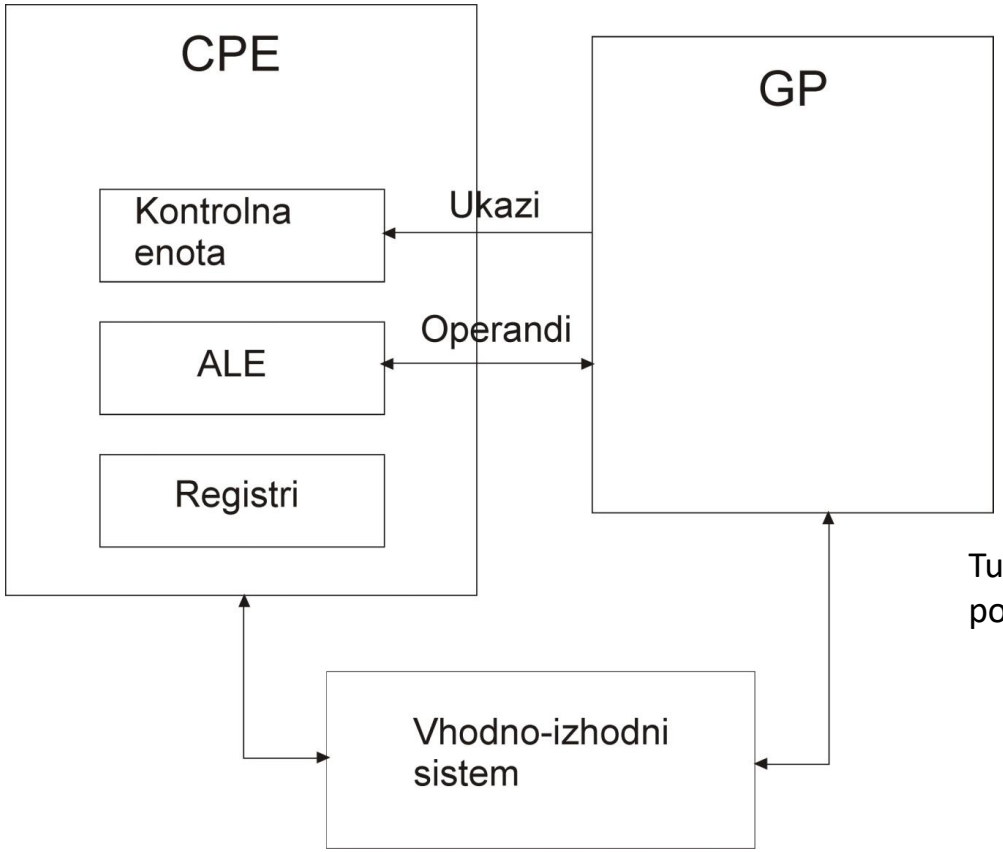
- knjiga z navodili ($\rightarrow$ ukazi)
- papir za vmesne in končne rezultate ($\rightarrow$ operandi)

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

Računanje in izračunljivost

Kakšni naj bodo stroji, ki znajo računati?

- kaj sploh je računanje?

Tudi teoretično zanimiv problem:

- Kakšen naj bo stroj, ki 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:

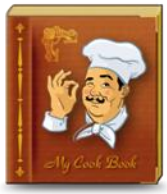
Funkcija *f(x)* je **izračunljiva**, če obstaja postopek, s katerim lahko določimo njeno vrednost (*z*) za vse možne vhodne podatke (*x*), nad katerimi je definirana.

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

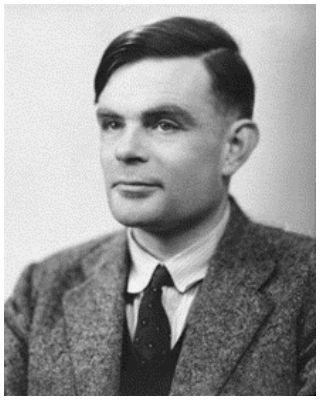


Definicija izračunljivosti je torej tudi:

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:

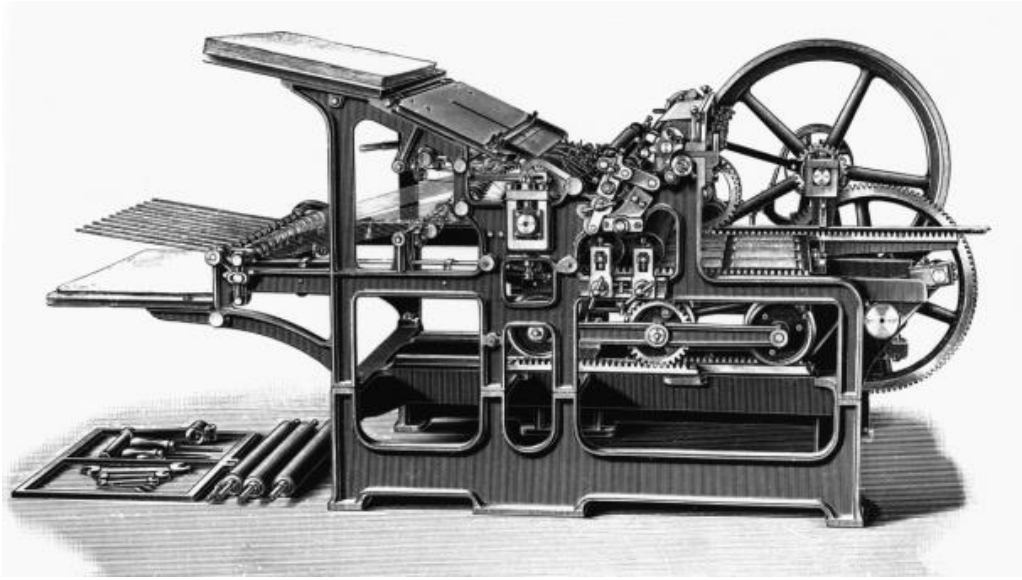
Problem je izračunljiv, če ga je možno v končnem številu korakov izračunati na Turingovem stroju

Turingovi stroji

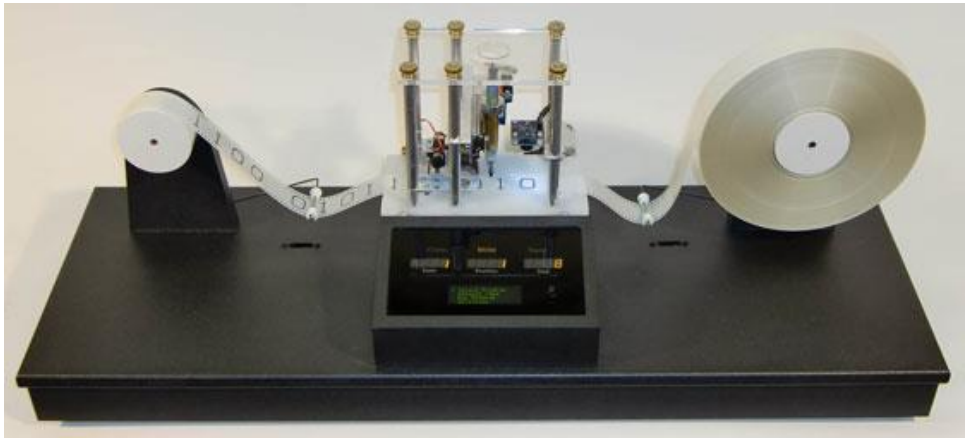
Turingov stroj (Turing machine, TM) sestavljajo:

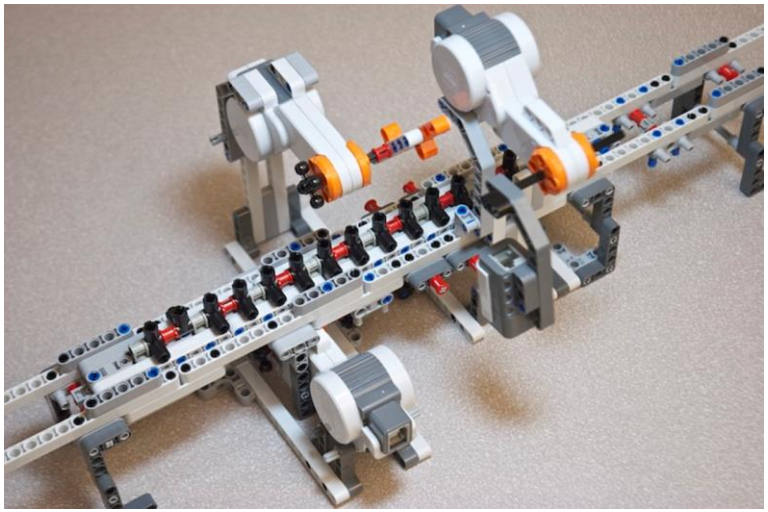
- procesor
- bralno-pisalna glava
- neskončno dolg trak
- mehanizem za pomik traku

- “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)





Delovanje Turingovega stroja (samo za ilustracijo)

Trak je razdeljen na celice

- vsaka celica je prazna (_), ali pa vsebuje enega iz končne množice znakov (tj. abecede)

Bralno-pisalna glava bere iz celice in piše v celico

Vhodni podatek *x* zapišemo v primerno kodirani obliki na trak (z znaki *abecede*)

Potrebno je definirati tudi začetno stanje

Stroj poženemo in prične se izvajanje ukazov (zaporedno)

- izvršitev enega ukaza je *korak*
- po končnem številu korakov se mora stroj ustaviti
  - na traku mora biti zapisan rezultat *z* (z znaki abecede)

Delovanje Turingovega stroja

Procesor ima (pozna) končno množico ukazov tipa:

- Če $\mathbf{s_t}$ **in** $\mathbf{m_i \rightarrow m_j, p_k, s_{t+1}}$
  - s_t je trenutno stanje (iz končne množice stanj)
  - m_i je prebrani znak
  - m_j je zapisani znak
  - p_k je pomik, ki je lahko:
    1. D ... pomik glave za 1 celico v desno
    2. L ... pomik glave za 1 celico v levo
    3. * ... ni pomika
  - S_{t+1} je naslednje stanje
- Tak model se imenuje *končni avtomat* (finite state machine, finite state automaton)

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)

Primer: Inkrement binarnega števila

- Postopek
  1. gremo na desno do števila
  2. gremo na desno do konca števila
  3. zaporedje enic pretvorimo v ničle, gremo vsakič levo
  4. ko naletimo na ničlo (ali na prazen znak), jo spremenimo v enico
  5. gremo levo na začetek števila

Program (za Turingov stroj), ki inkrementira binarno število:

stanje	prebrani znak	zapisani znak	pomik	naslednje stanje
S0	_	_	D	S0
S0	0	0	D	S1
S0	1	1	D	S1
S1	0	0	D	S1
S1	1	1	D	S1
S1	_	_	L	S2
S2	0	1	L	S3
S2	1	0	L	S2
S2	_	1	L	S3
S3	0	0	L	S3
S3	1	1	L	S3
S3	_	_	*	halt

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

Pisanje programov za TM ni enostavno

- primitivni ukazi

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: $O(2^n)$, $O(n!)$, $O(n^n)$, ...

UVOD

Arhitektura računalniških sistemov

Kako deluje računalnik (računalniški sistem)?

Kako je zgrajen?

Arhitektura računalnika

- računalnik (abstraktni), kakor ga vidi program(er) na nivoju strojnega jezika (ukazna arhitektura – ISA, Instruction Set Architecture)

Organizacija računalnika

- zgradba, sestavni deli in povezave (mikroarhitektura)

Isto arhitekturo se da realizirati z različnimi organizacijami

Kaj je računalniška arhitektura?

Rač. arhitektura določa nivoje abstrakcije/implementacije, ki omogočajo, da izvajamo aplikacije z uporabo obstoječih tehnologij.

Aplikacije zahtevajo izboljšave arhitektur

- financirajo razvoj

Tehnologije omejujejo učinkovitost,

- razvoj tehnologij omogoča nove arhitekture

Zakaj študirati računalniško arhitekturo?

Poznavanje delovanja računalnika (oz. rač. sistema) pomaga tudi pri načrtovanju, razvoju in implementaciji aplikacij,

- da lahko delujejo hitreje, ceneje, bolj učinkovito, ...

V zadnjem času postaja učinkovito izvajanje vse bolj pomembno

Vsebina

1. Uvod v arhitekturo
2. Razvoj računalnikov
3. Osnovni principi delovanja
4. Zapis informacije
5. Ukazi in ukazna arhitektura (ISA)
6. Aritmetika
7. Procedure
8. Centralna procesna enota
9. Paralelizem na nivoju ukazov
10. Predpomnilniki
11. Pomnilniki

Literatura

Osnovna:

Dušan KODEK: **Arhitektura in organizacija računalniških sistemov**, Bi-Tim, Ljubljana, 2008.

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.

Vaje

Asistenta:

Dr. Miha Janež (miha.janez@fri.uni-lj.si, R3.56)

Dr. Ratko Pilipović (ratko.pilipovic@fri.uni-lj.si, R2.41)

Na vajah asistent pokaže nekaj primerov, potem pa se samostojno rešujejo naloge (se ne ocenjujejo) – asistent pomaga pri nejasnostih

Oceno vaj pridobite z dvema kolokvijema

- Ocena vaj je pozitivna (vaje opravljene), če je povprečje kolokvijev vsaj 30 %, vsak pa je vsaj 20 %

Ocena vaj velja le za tekoče šolsko leto!

Dodatne naloge z rešitvami za utrjevanje snovi bodo dostopne na učilnici

Obveznosti

Ocena predmeta:

- Opravljene vaje (2 kolokvija) so pogoj za pristop k izpitu
- Pisni izpit (če povprečje kolokvijev < 60%)
- Teoretični izpit

Za podrobnosti glej spletno učilnico (stran Pravila).

Razlogi za strojno računanje

Čemu strojno računanje?

Ročno računanje, 2 problema:

1. počasnost
2. nezanesljivost

Povezava med ročnim in strojnim računanjem

Ročno računanje

- papir ($\rightarrow$ pomnilnik)
- možgani ($\rightarrow$ procesor)

Papir

- ukazi (navodila)
- operandi

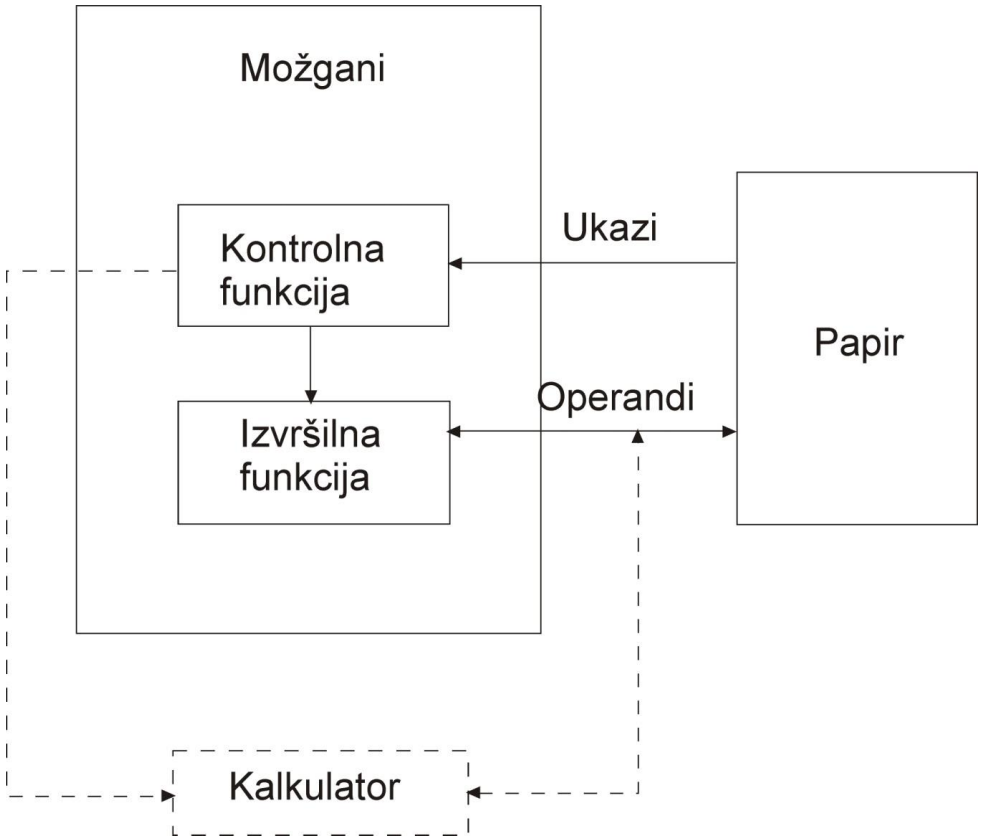
Možgani pri računanju opravljajo 2 funkciji:

- kontrolna funkcija
  - prevzema ukaze in skrbi za pravilen vrstni red izvrševanja ukazov
- izvršilna funkcija
  - npr. seštevanje, množenje, itd.

Papir lahko delimo v 2 vrsti:

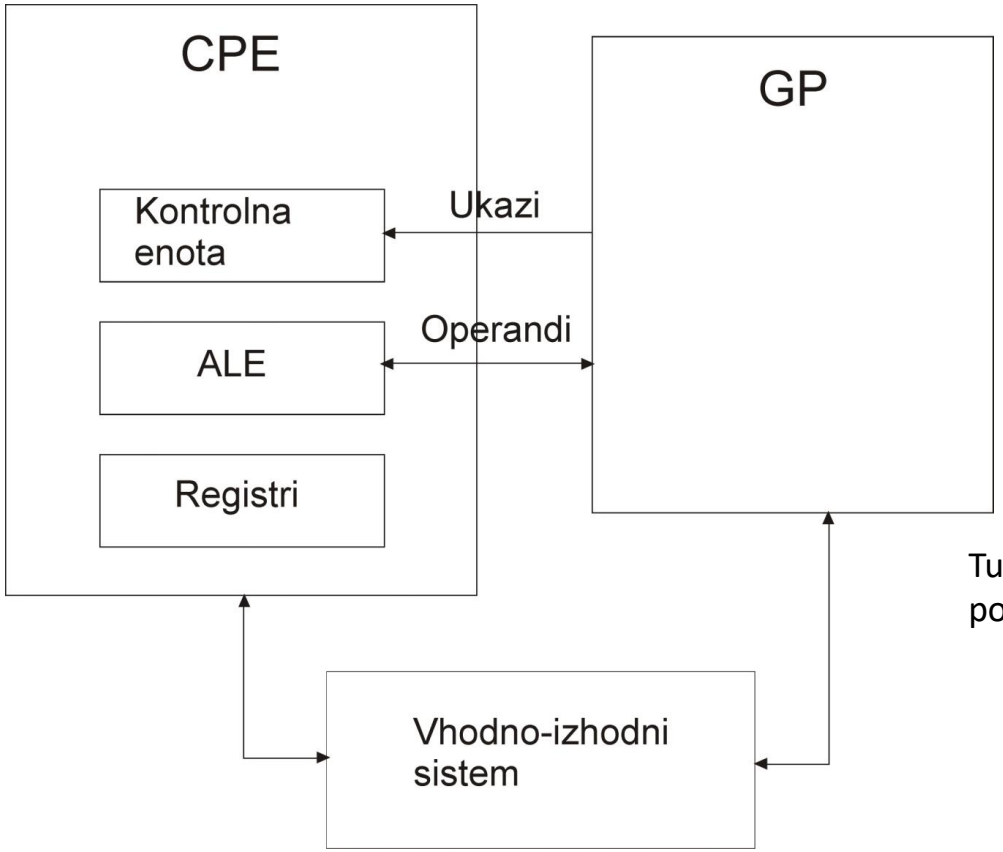
- knjiga z navodili ($\rightarrow$ ukazi)
- papir za vmesne in končne rezultate ($\rightarrow$ operandi)

Ročno računanje



Strojno računanje

Današnji računalniki računajo na podoben način kot človek



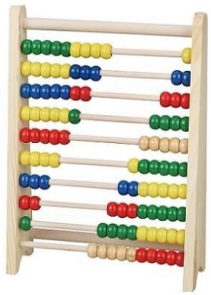
Tudi računalnik ima lahko pomnilnik ločen na 2 dela:
- del za ukaze
- del za operande

2
DOSEDANJI RAZVOJ STROJEV ZA RAČUNANJE

Digitalni princip

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

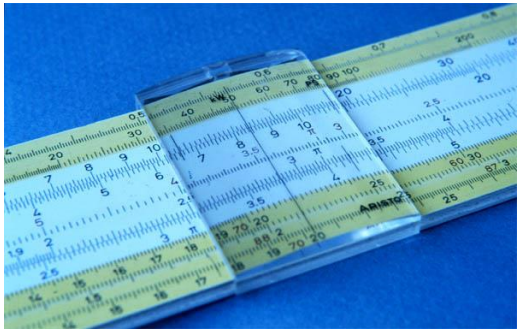
- abak



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

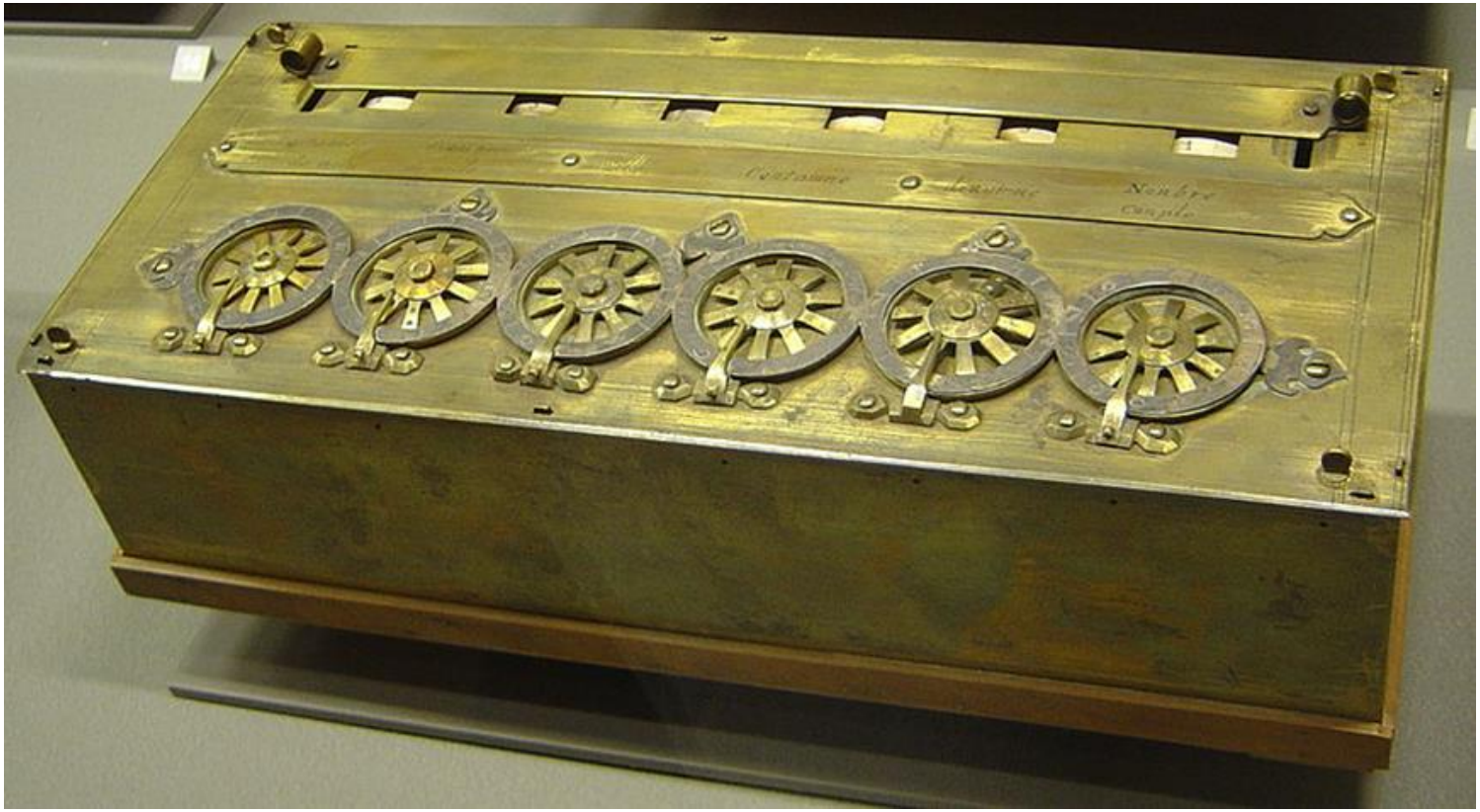
Prvi kalkulatorji

Kalkulator je naprava (stroj), ki izvaja aritmetične operacije

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

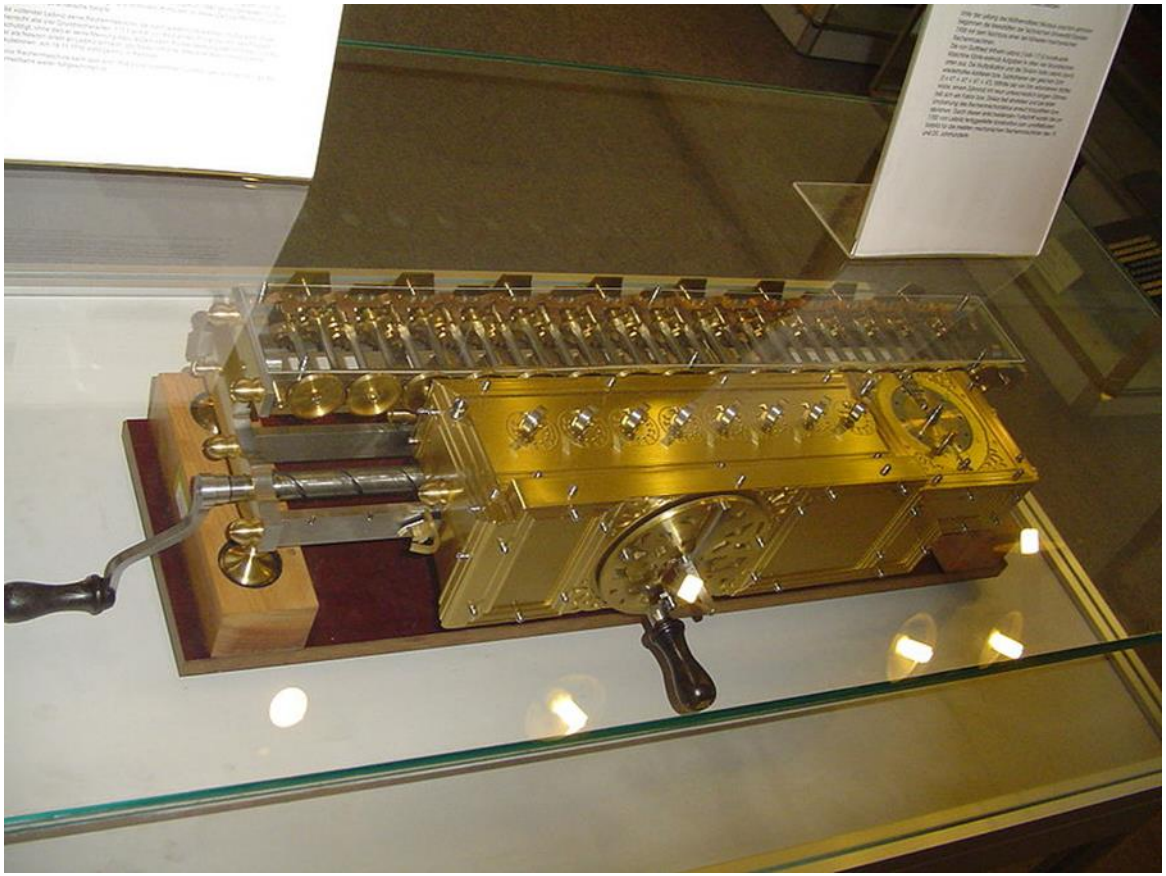
Schickard, 1623

- zobata kolesa (10 zobnikov)
- mehanizem za prenos naprej
- ročni 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

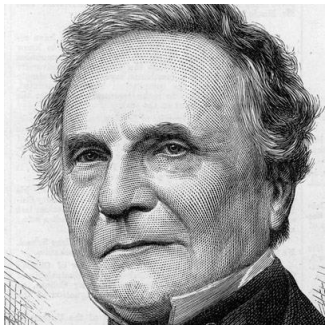
Charles Babbage

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

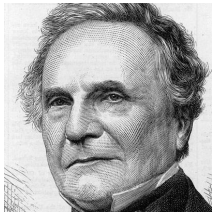


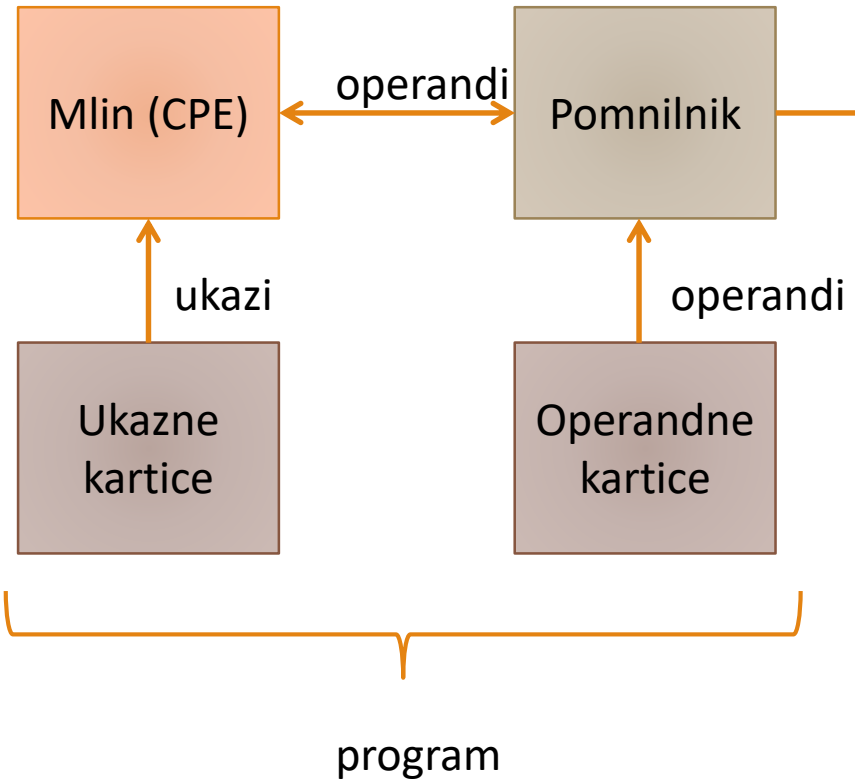
Analitični stroj (analytical engine), okrog 1835

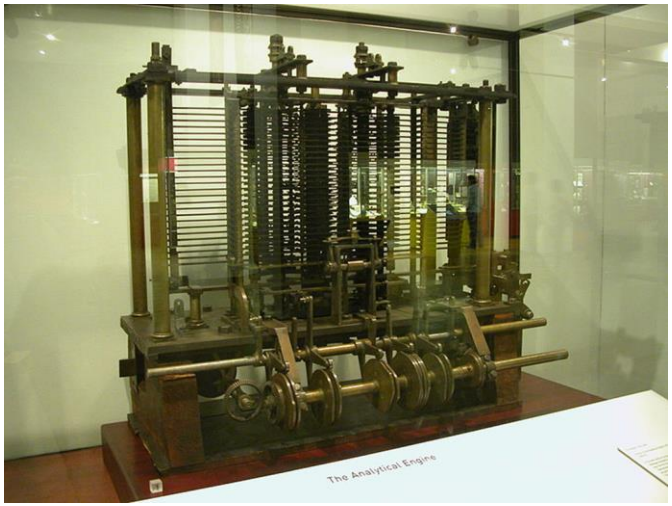
- Prvi (načrtovan) računalnik
- Ni bil realiziran zaradi velike zahtevnosti in stroškov
- Računski del
    1. Mlin (mill): izvedba operacij
    2. Pomnilnik (store): shranjuje operande
- Luknjane kartice 2 vrst
    1. Ukazne kartice (s programi)
    2. Operandne kartice

Babbage za 100 let utonil v pozabo

Zgradba analitičnega stroja







Analitični stroj (zgrajen kasneje)

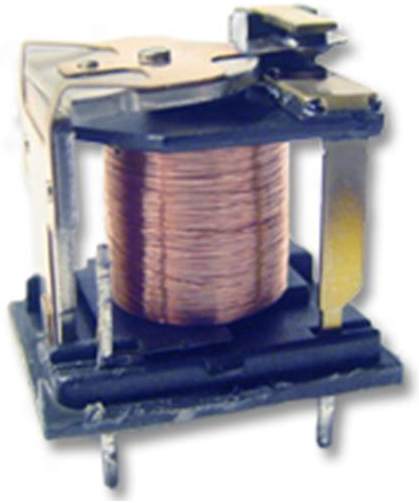
Elektromehanski stroji

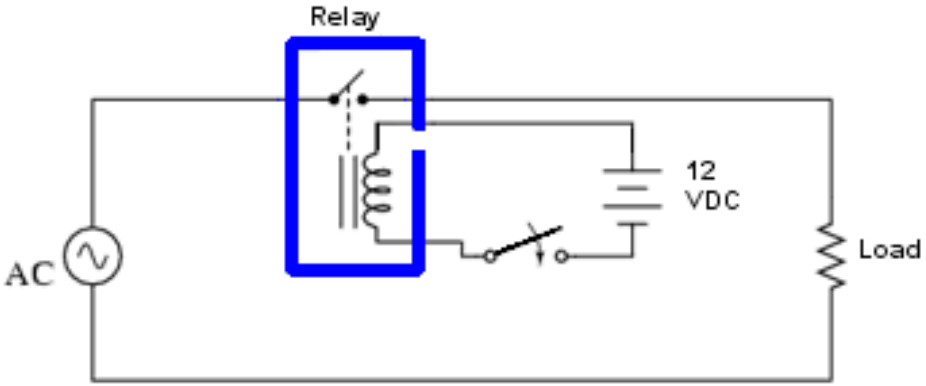
Elektrotehnika ponuja nove možnosti

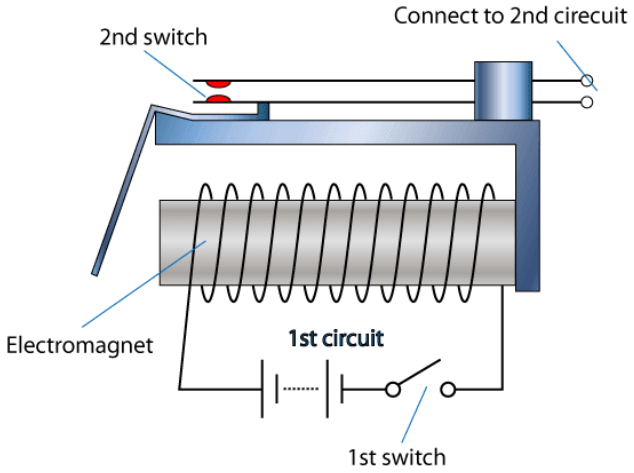
- elektromotorji za pogon mehanskih kalkulatorjev
- električno branje luknjanih kartic

Rele (relay)

električno-krmiljeno stikalo





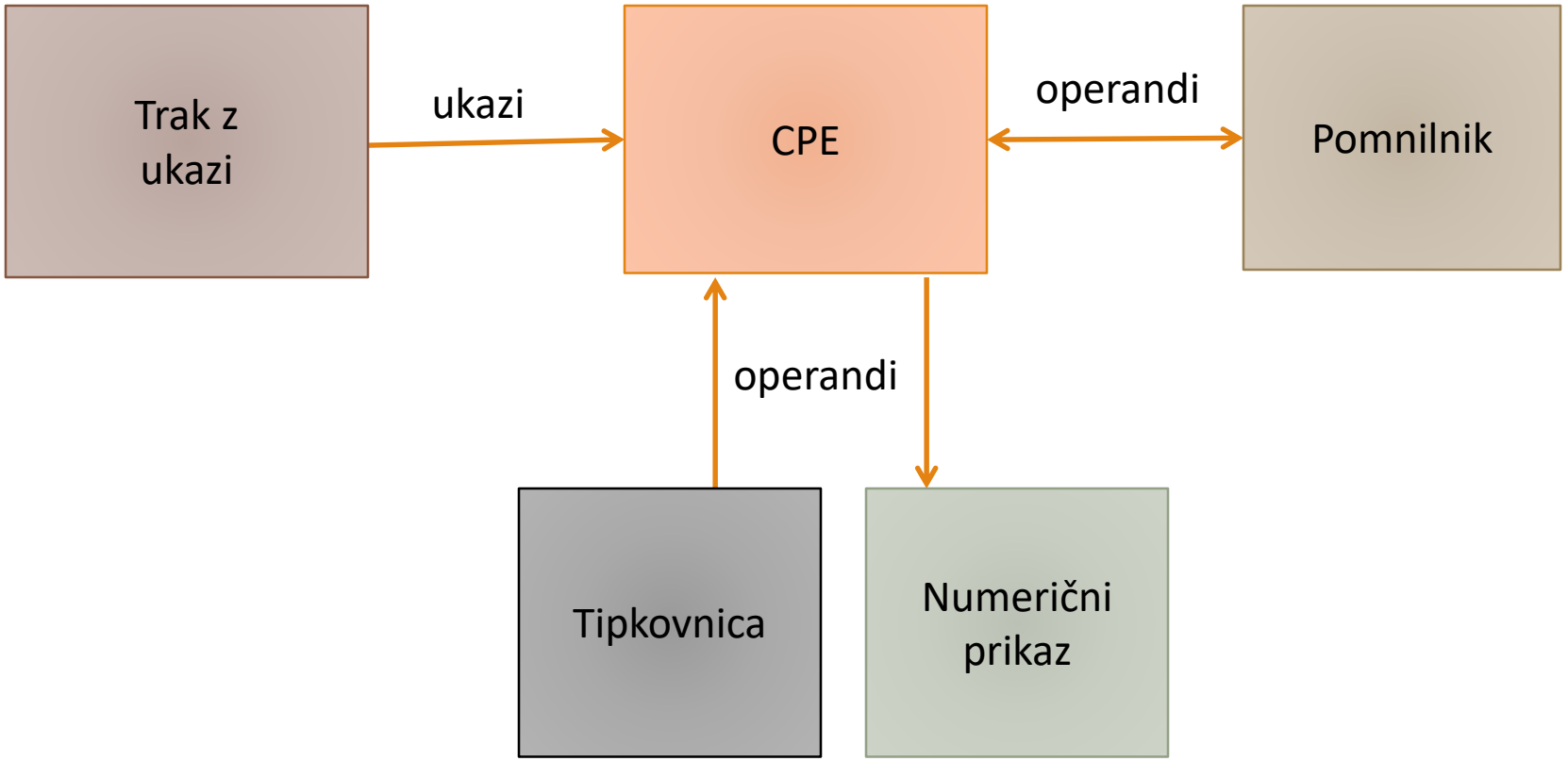


Konrad Zuse zgradi prvi delujoči računalnik

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
  - Uničen 1943 med bombardiranjem Berlina

Zgradba Z3


Trak z ukazi
ukazi
CPE
operandi
Pomnilnik
operandi
Tipkovnica
Numerični prikaz

Z4 (Deutsches Museum, Muenchen)

Harvard Mark I

- Howard Aiken, izdelava IBM 1943
- 15m v dolžino
- elektromehanska desetiška števna kolesa
- pomnilnik 72 x 23 desetiških mest
- luknjan trak (24 stolpcev - bitov)
- Ukazi oblike A1 A2 OP
  - pomnilniška naslova + operacija, 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**

- elektronka (vacuum tube), 1904
- tranzistor, 1947

Prvi elektronski računalniki

Zakaj je elektronika hitrejša?

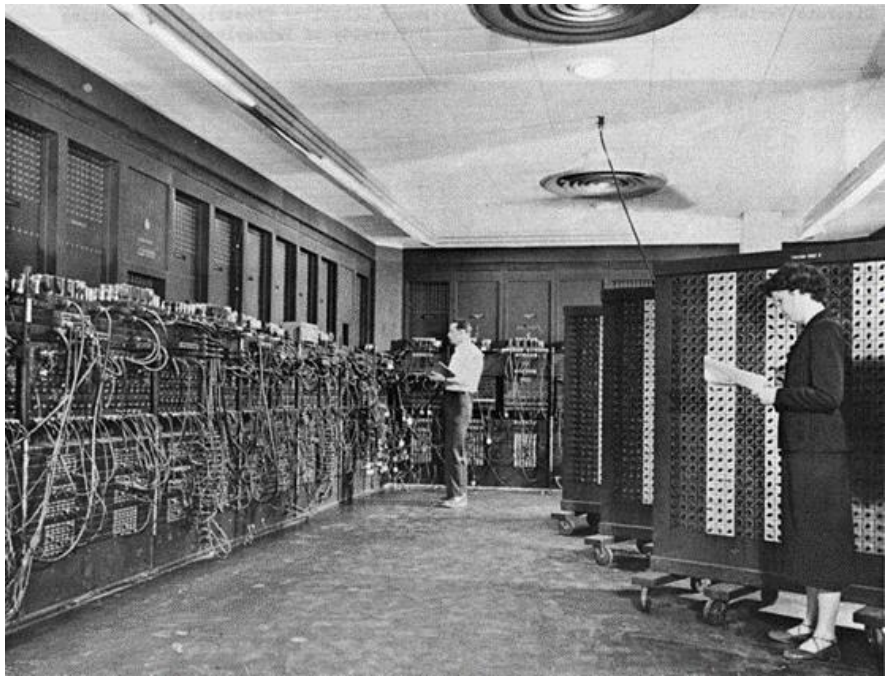
- rele potrebuje vsaj nekaj ms za preklop
- elektroni so bistveno hitrejši

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*

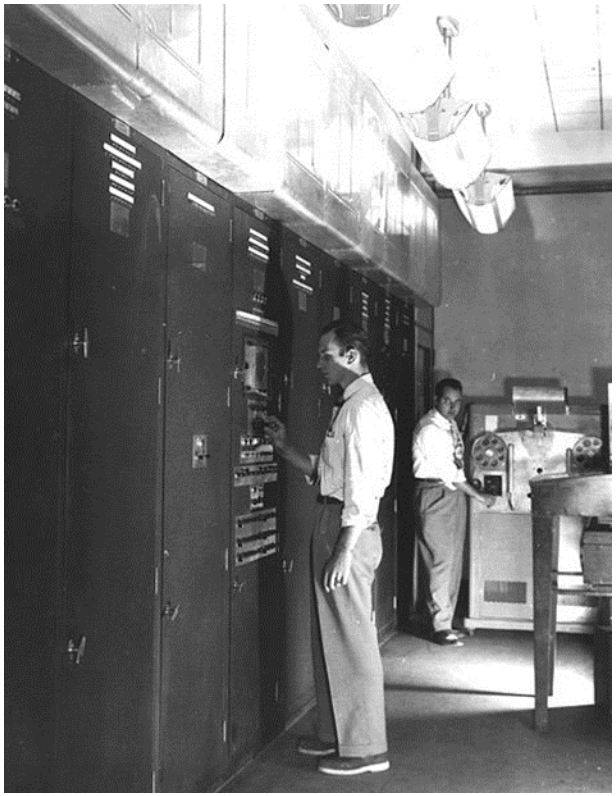
Prednosti shranjenega programa

- 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
- 3000 elektronk
- dvojiški stroj
- serijsko (bit za bitom)
- ukazi

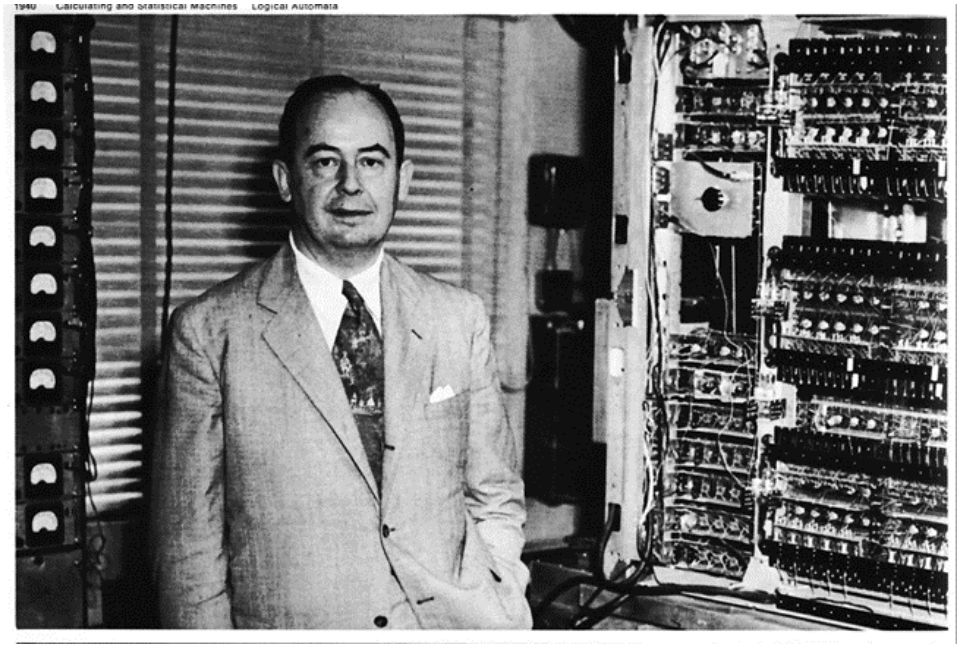
  A1 A2 A3 A4 OP
  - A1, A2: naslova vhodov
  - A3: naslov izhoda
  - A4: naslov nasl. ukaza



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

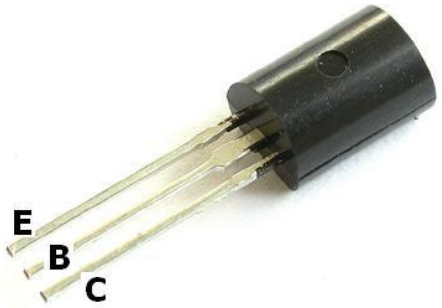
1. mehanski kalkulatorji
2. programsko voden rač. za splošne namene (Babbage, realizacija 1940. leta)
3. elektronika (ENIAC, 1945)
4. von Neumannovi rač. (shranjen program), (EDVAC, IAS, ...)

po 1951 je razvoj bolj tehnološki, manj arhitekturni

Razvoj tehnologije

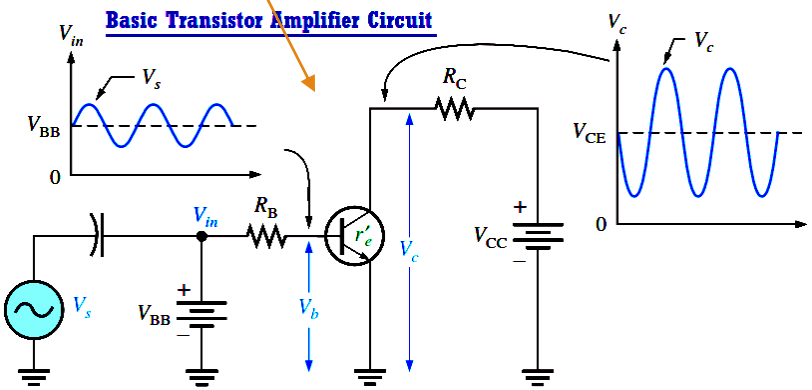
Tranzistor, 1947

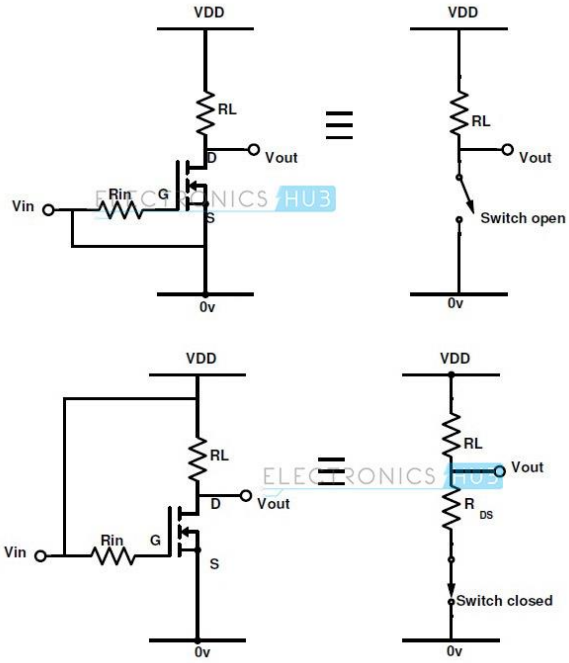
- Bell Labs (Shockley)



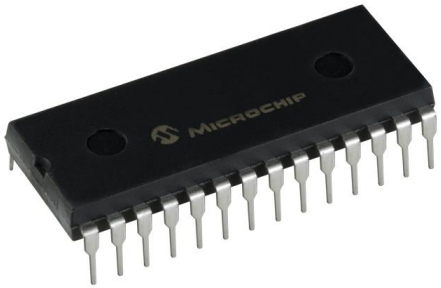
Uporaba tranzistorja

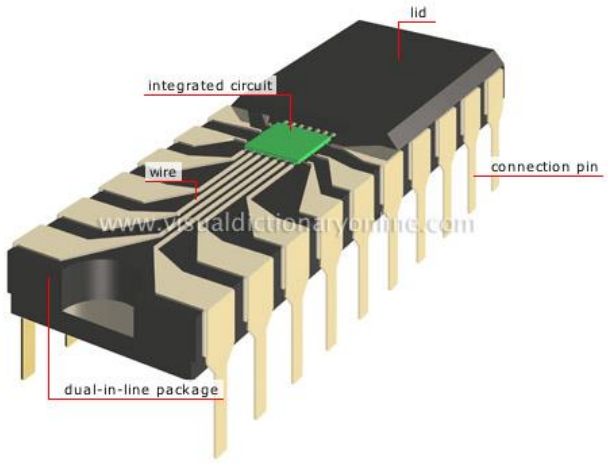
- ojačevalnik
- stikalo

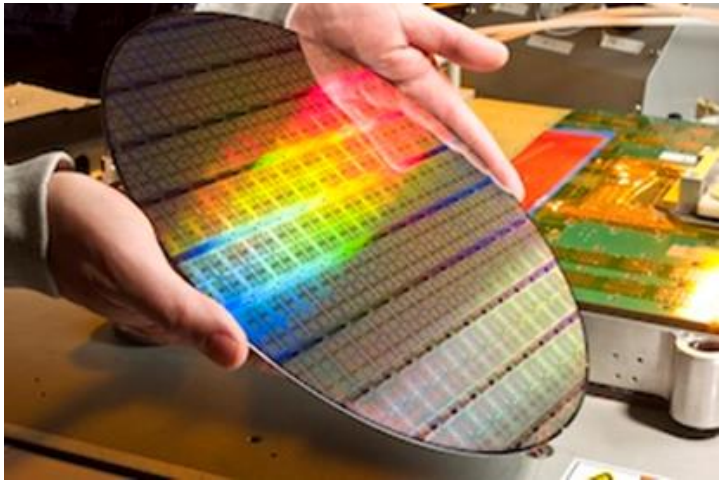




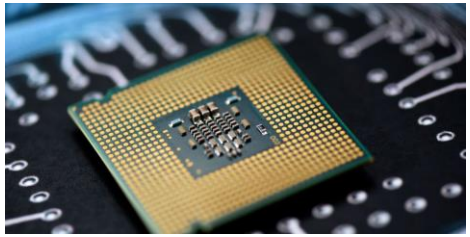
Integrirana vezja (čipi), 1958





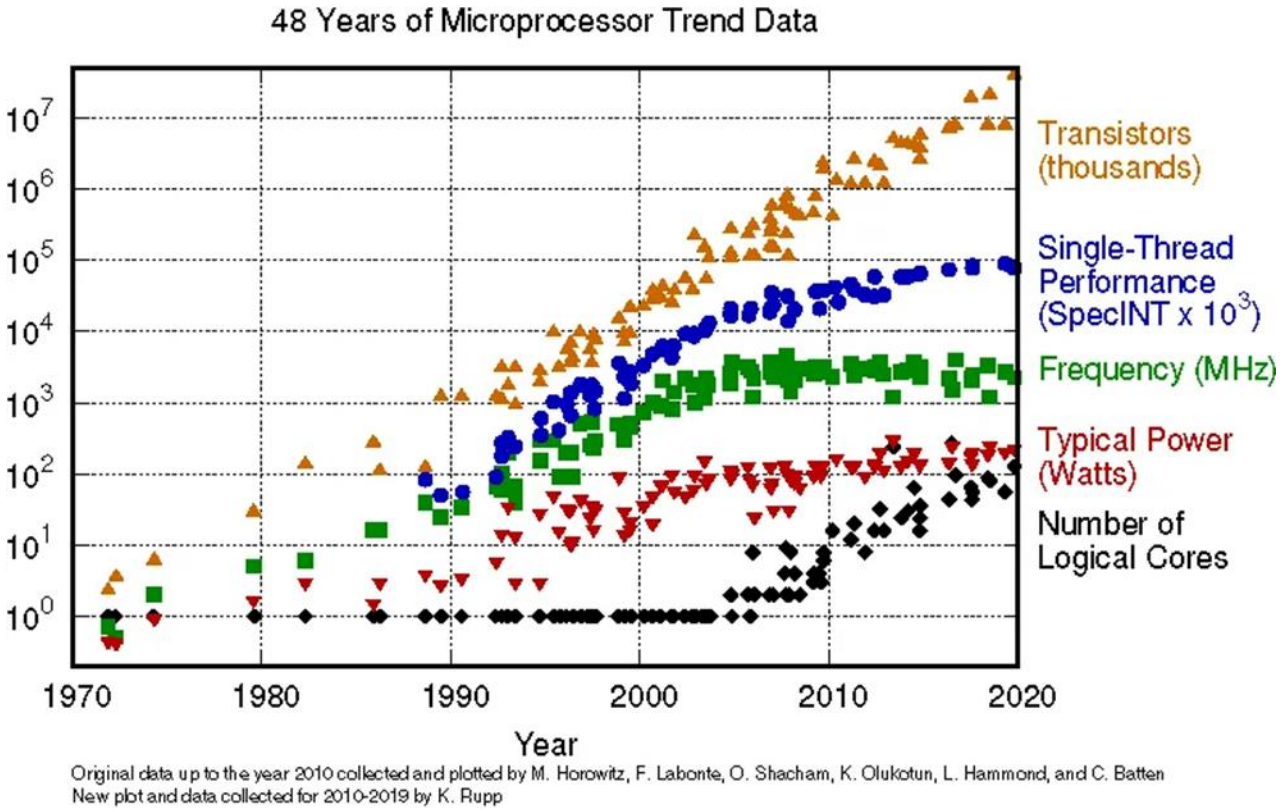


Silicijeva rezina (wafer)



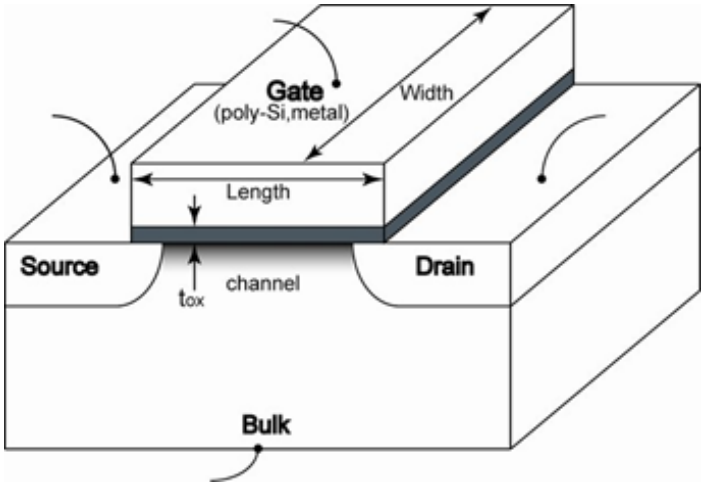
Moorov zakon

- podvojitev števila transistorjev na čipu vsakih 18 mesecev
  - tudi zmogljivost na watt
- 2000 (1971), nekaj milijard (danes)



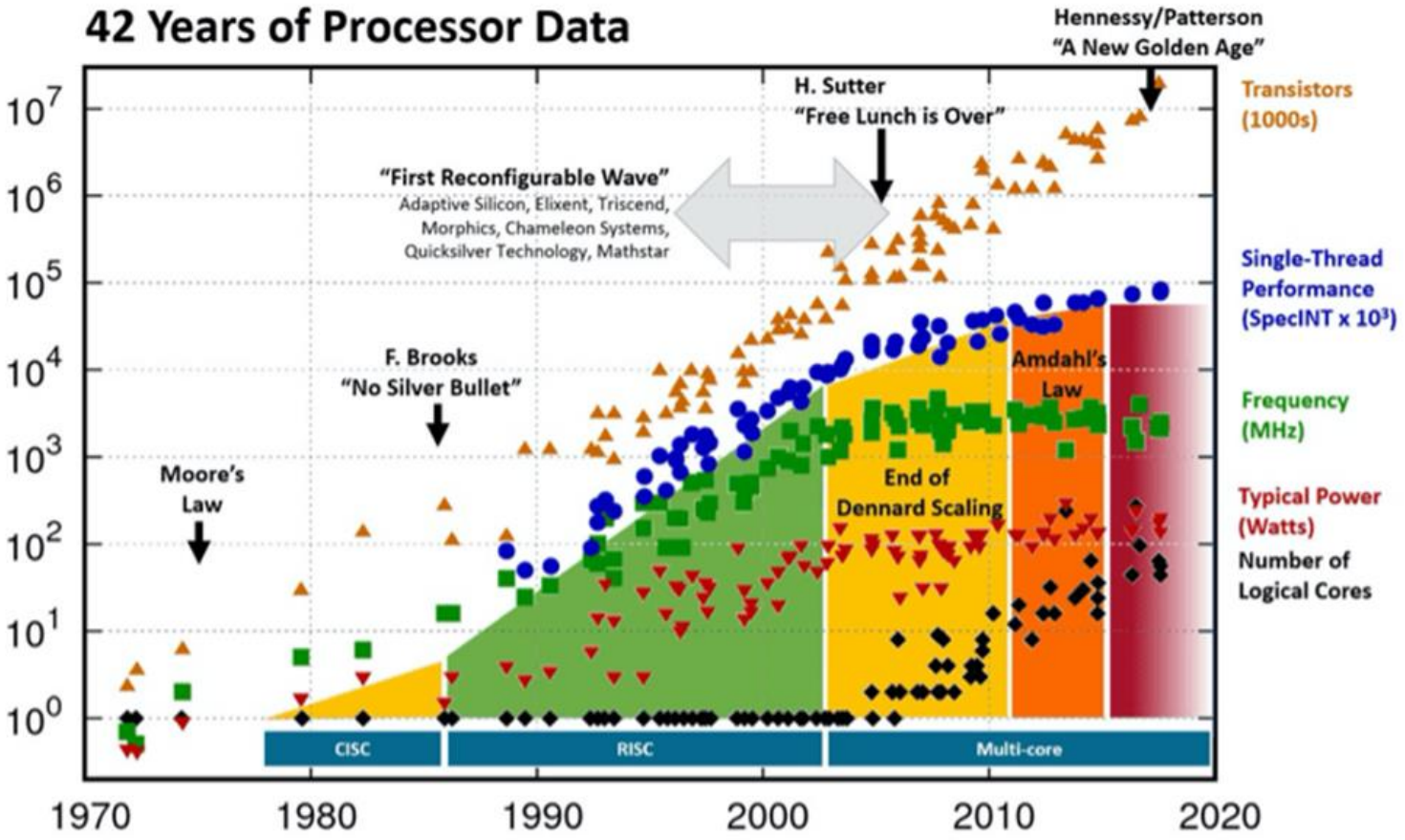
Dennardovo skaliranje

- z zmanjševanjem dimenzij tranzistorjev ostane poraba energije na površino konstantna, R. Dennard (1974)



Stagnacija pri zmogljivosti (od 2005 dalje)

- tok odtekanja (leakage current) pri majhnih dimenzijah - pregrevanje



Hennessy and Patterson, Turing Lecture 2018, overlaid over "42 Years of Processors Data"
https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/;"First Wave" added by Les Wilson, Frank Schirrmeister
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

Današnji računalniki

Namizni računalniki, Prenosniki, Tablice

Strežniki

Superračunalniki

Vgrajeni sistemi:

- telefoni, kamere
- igralne konzole, igrače
- UAV, (avtonomna) vozila
- gospodinjski aparati
- usmerjevalniki
- senzorska omrežja

...

Razvoj programiranja

Nekdaj programskih orodij ni bilo

- programiranje z vpisovanjem ničel in enic (strojni jezik)

Nalaganje programa iz zunanjega v glavni pomnilnik (50. leta)

- Bootstrap ali bootloader omogoča zagon OS

Simbolični zapis: Zbirni jezik (Assembly language)

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

Knjižnice numeričnih podprogramov (procedur)

Višji programski jeziki v 60. letih

- prvi: FORTRAN (1956), ALGOL, COBOL, Lisp, ...
- kasneje: C, Pascal, C++, Java, ...

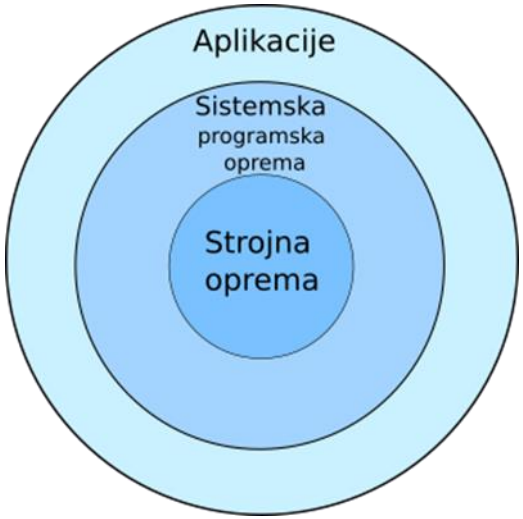
Primerjava

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

Orodja: OS, zbirniki, nalagalniki, povezovalniki, prevajalniki, urejevalniki, razhroščevalniki, programi za monitoring, ...

V zgodnji 60. letih je imel IBM 4 vrste nekompatibilnih računalnikov

- IBM 360 ISA je bil prvi prenosljivi ukazni nabor

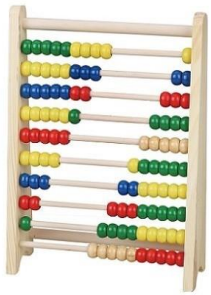


2
DOSEDANJI RAZVOJ STROJEV ZA RAČUNANJE

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

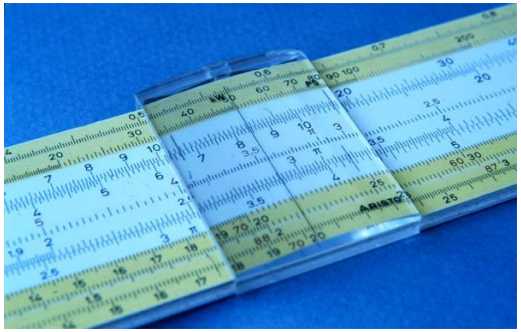
- abak



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 /

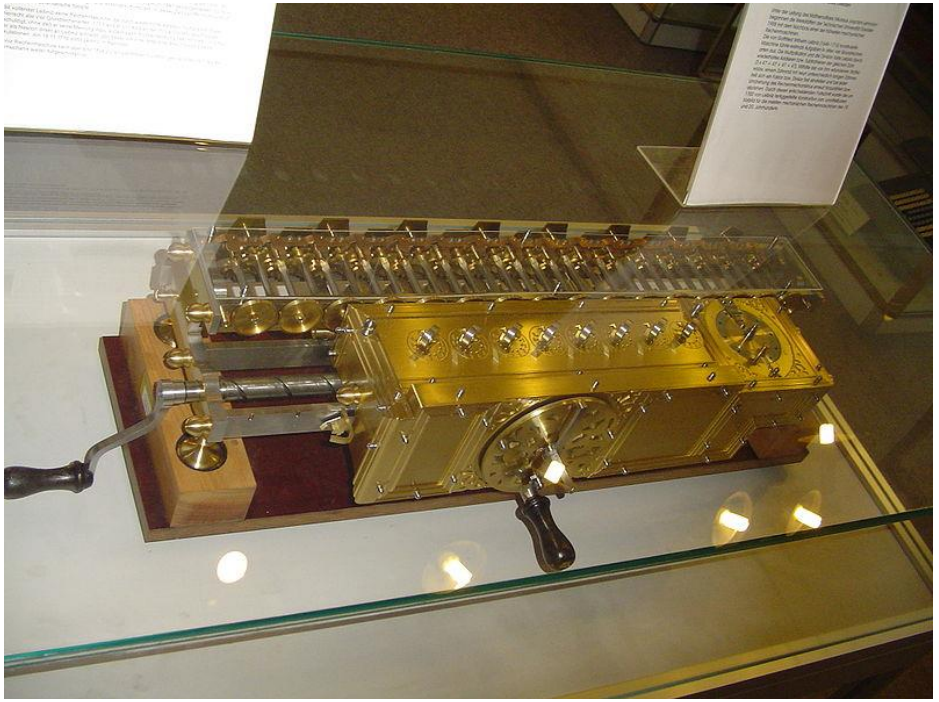
Schickard, 1623

- 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



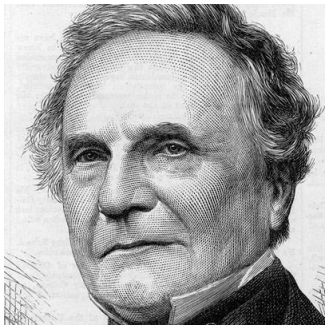
Charles Babbage

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

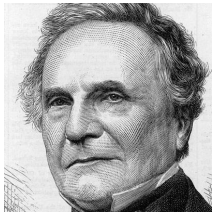


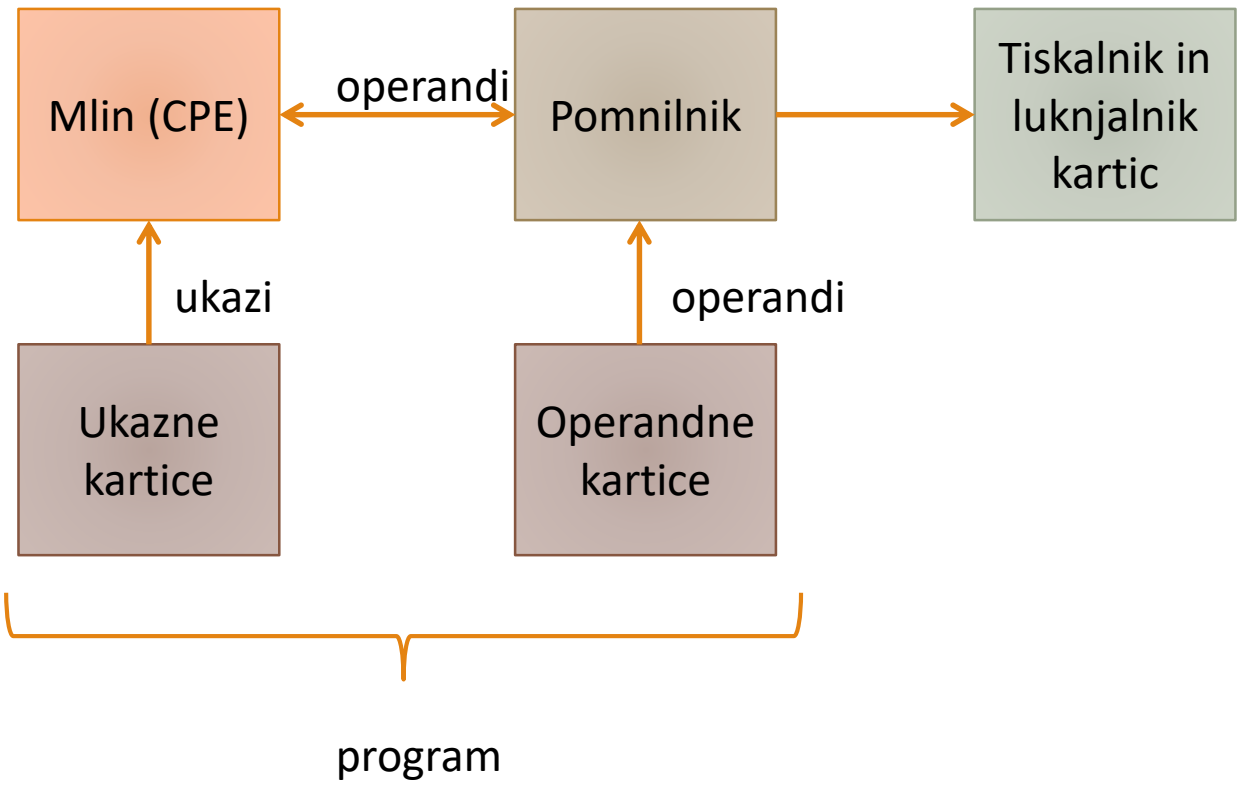
Analitični stroj (analytical engine), okrog 1835

- 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

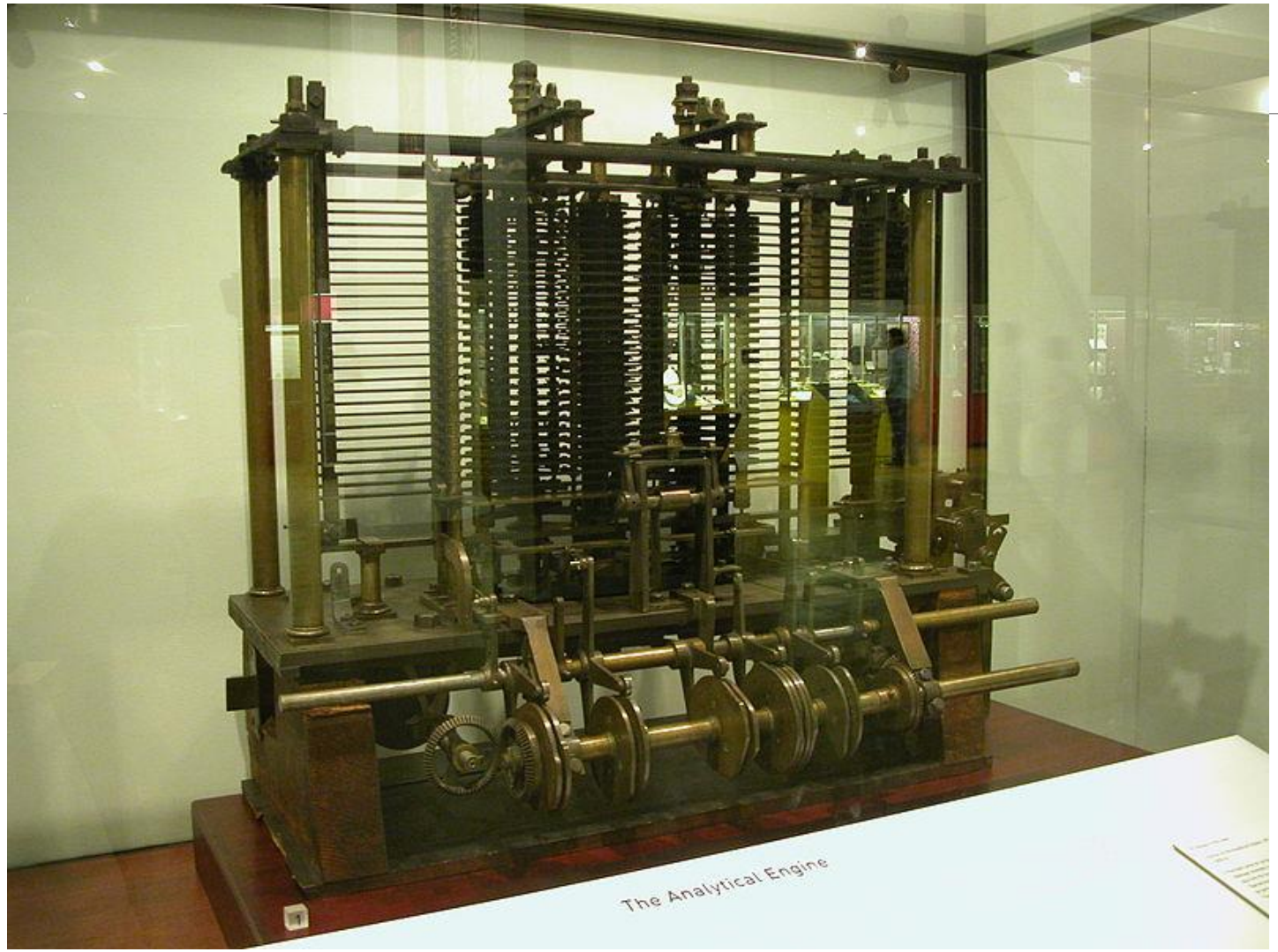
Babbage za 100 let utonil v pozabo

Zgradba analitičnega stroja





Analitični stroj (zgrajen kasneje)



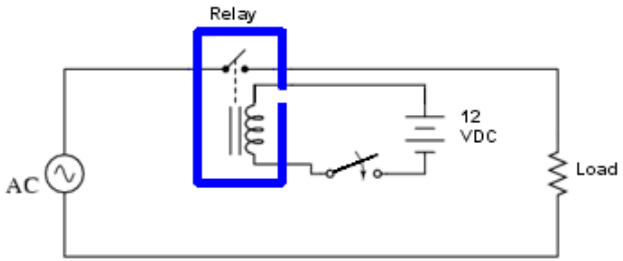
Elektromehanski stroji

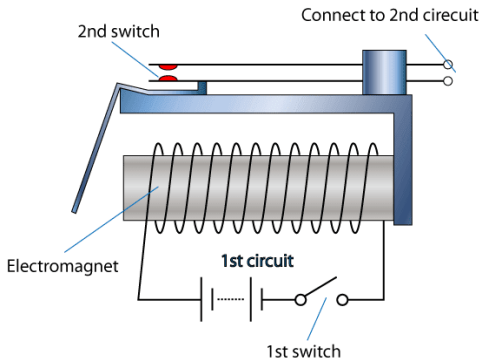
Elektrotehnika ponuja nove možnosti

- elektromotorji za pogon mehanskih kalkulatorjev
- električno branje luknjanih kartic

Rele (relay)

- električno-krmiljeno stikalo





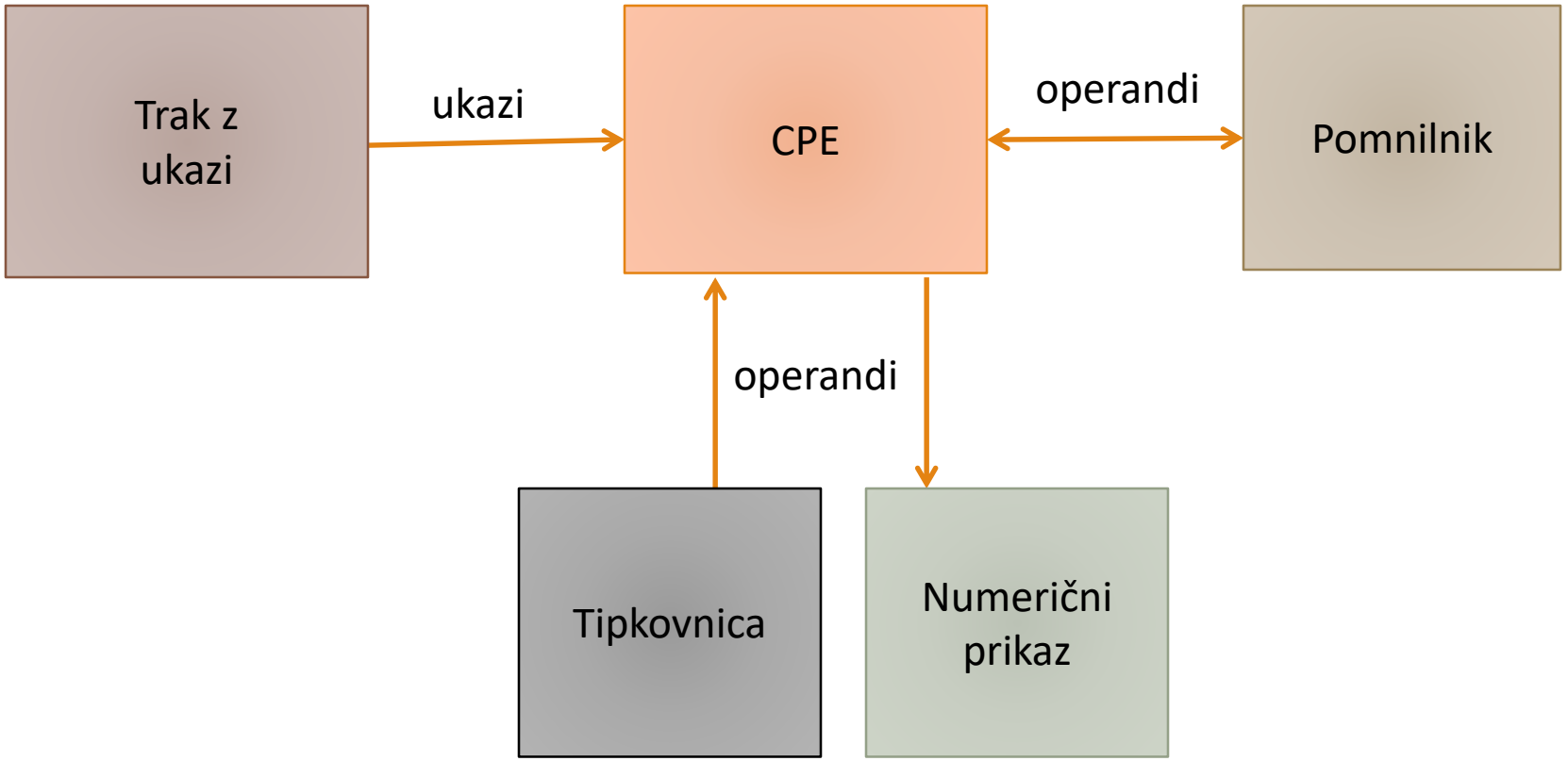
Konrad Zuse zgradil prvi delujoči računalnik

Zusejevi računalniki

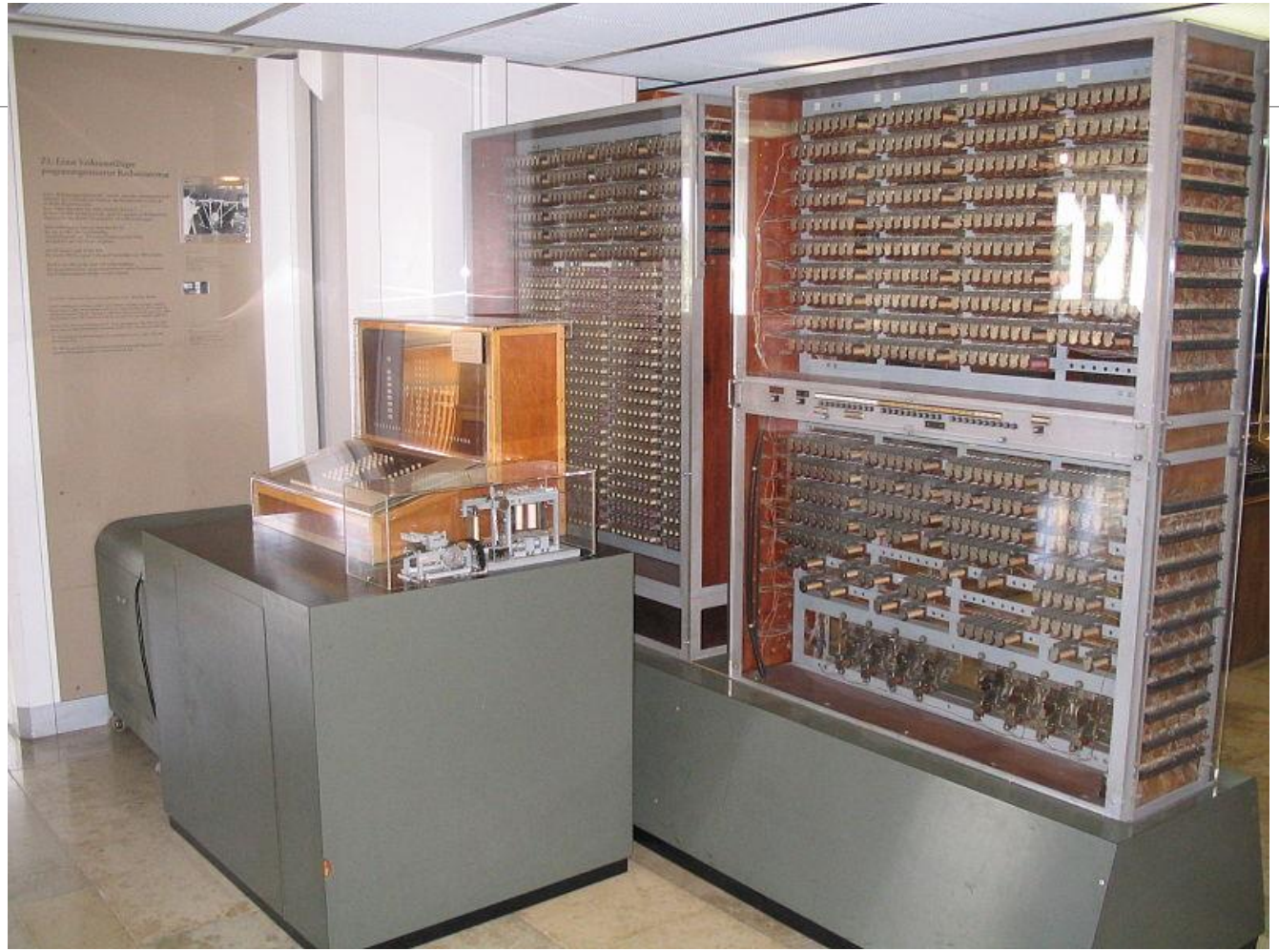
- Z1, 1938, mehanski
- Z2
- Z3, 1941, prvi delujoči (splošnonamenski) računalnik
  - 2600 relejev
  - pomnilnik 64 22-bitnih besed (releji)
  - 8-bitni ukazi
  - luknjan trak
  - plavajoča vejica: 14-bitna mantisa, 7-bitni eksp. + predznak
  - Tipkovnica
  - Hiba: ni imel pogojnih skokov
  - Frekvenca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

Z1

Zgradba Z3


Trak z ukazi
ukazi
CPE
operandi
Pomnilnik
operandi
Tipkovnica
Numerični prikaz

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?

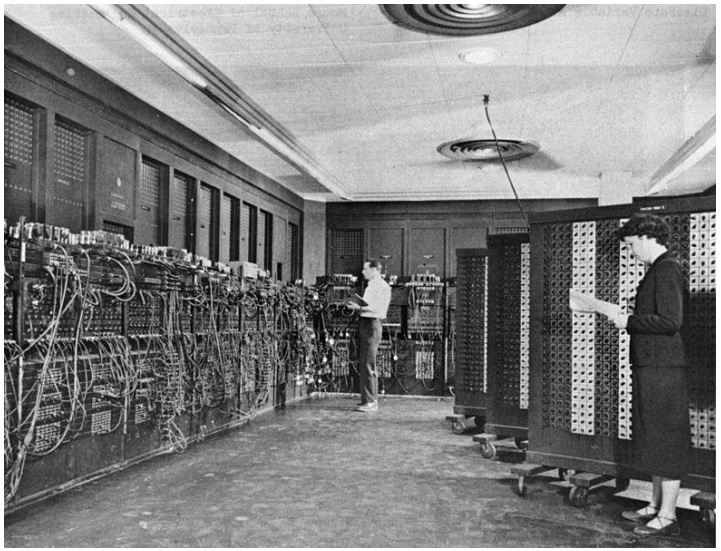
- rele potrebuje vsaj nekaj ms za preklop
- elektroni so bistveno hitrejši

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*

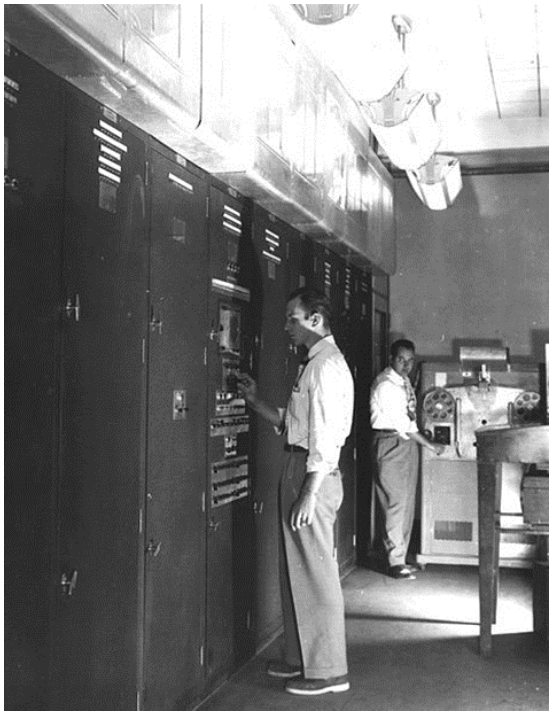
Prednosti shranjenega programa

- 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

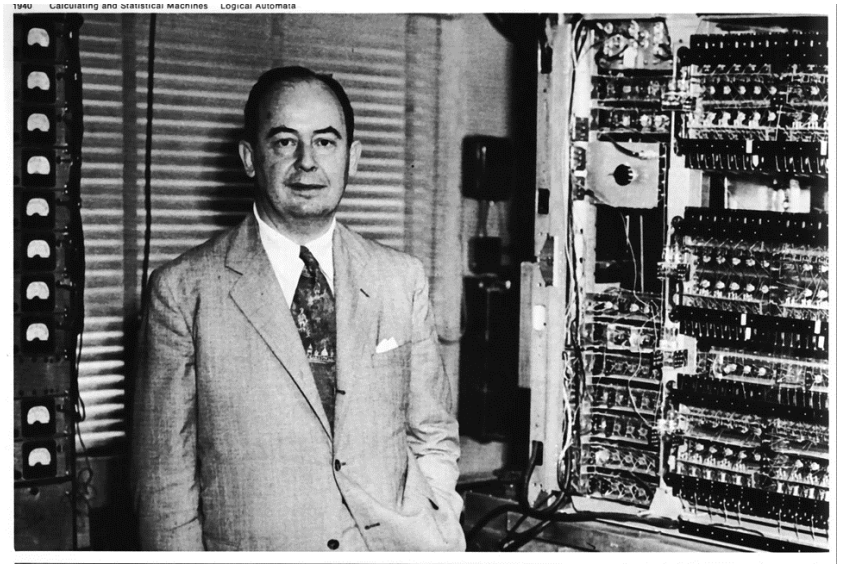


IAS, 1951

- 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

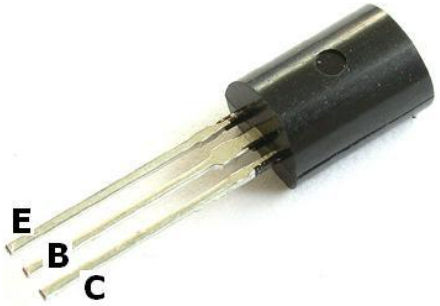
1. mehanski kalkulatorji
2. programsko voden rač. za splošne namene (Babbage, realizacija 1940. leta)
3. elektronika (ENIAC, 1945)
4. von Neumannovi rač. (shranjen program), (EDVAC, IAS, ...)

po 1951 je razvoj bolj tehnološki, ne toliko arhitekturni

Razvoj tehnologije

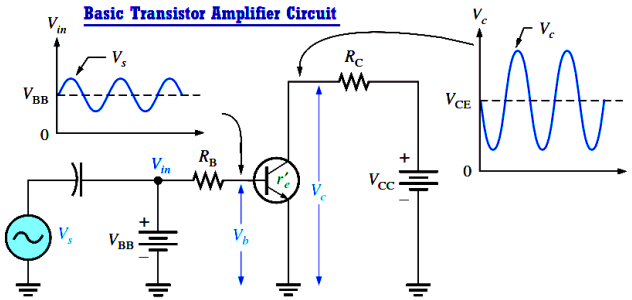
Tranzistor, 1947

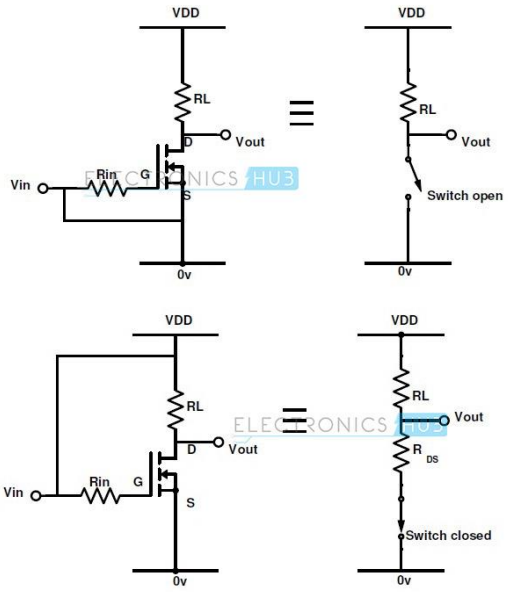
- Bell Labs (Shockley)



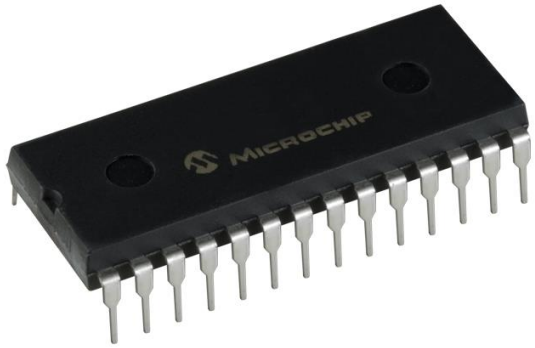
Uporaba tranzistorja

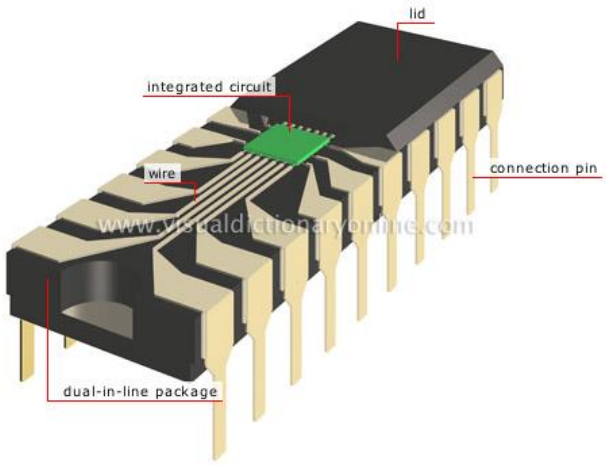
- ojačevalnik
- stikalo

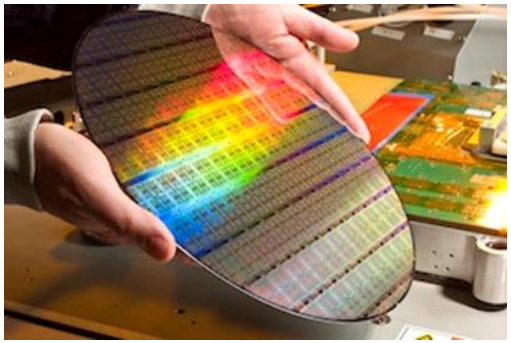




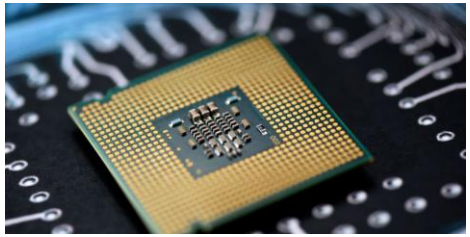
Integrirana vezja (čipi), 1958







Silicijeva rezina (wafer)

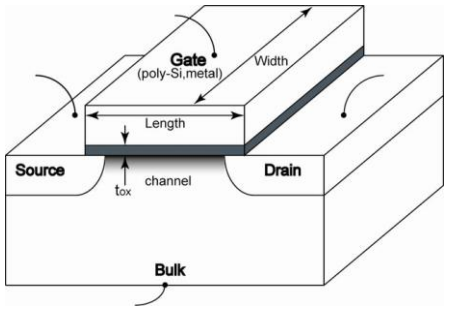


Moorov zakon

- podvojitev števila transistorjev na čipu vsakih 18 mesecev
  - tudi zmogljivost na watt
- 2000 (1971), nekaj milijard (danes)

Dennardovo skaliranje

- z zmanjševanjem dimenzij tranzistorjev je ostajala poraba energije na površino konstantna (napetost in tok sta manjša)
  - Npr. zmanjšanje širine in dolžine kanala na 70%:
    - zmanjšanje površine ~ 50%
    - zmanjšanje porabe ~ 50%
  - Poleg tega se poveča možna frekvenca in zmanjša kapacitivnost, kar se približno izravna ($P = CU^2f$)
- Stagnacija pri zmogljivosti
  - tok odtekanja pri majhnih dimenzijah - pregrevanje



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
ZAPIS INFORMACIJE IN ARITMETIKA

BRANKO ŠTER

PO KNJIGI - DUŠAN KODEK: ARHITEKTURA IN ORGANIZACIJA RAČUNALNIŠKIH SISTEMOV

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 (Unicode transformation format)
  - UTF-8
    - posamezen znak zavzame od 1 do 4 bajtov
      - kodiranje spremenljive dolžine – variable length encoding
    - 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 $r = 2$
  - $0{,}011_2$

Napaka pri rezanju decimalk

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

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

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

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

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

  $\mathbf{\log_a c = \log_a b * \log_b c}$ (pravilo)

  (na ta način se znebimo baze *b*, v našem primeru *r*,

  za a pa vzamemo kako znano bazo)

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

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

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

- 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 $r = 16$. $|N'\text{-}N| \leq 10^{-3}$

  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 ... $_3$

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:
    - $3C7_{16} = 3C7_H = 0x3C7 = \$3C7$

Sorodne baze

- 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
    - $A15_{16} = 1010\ 0001\ 0101_2$

Nepredznačena števila

- 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 ☹ (torej V)
    - Od nepredznačene vrednosti (vsota v gornji formuli) moramo odšteti 2^n
      - Npr. b=1001: V = 9 – 16 = –7
      - Npr. b=1110: V = 14 – 16 = –2

- Negativno število (zapis b pri podani vrednosti V) dobimo tako, da vrednosti V prištejemo 2^n
  - Npr.: -2 + 16 = 14, torej tak zapis kot za nepredznačeno 14
- Lahko pa tudi 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 (deset), torej je 10110 enako -10 (minus deset)

- Potrebno pa je razlikovati med pojmoma *zapis v 2'K* in *2'K nekega števila*

- Bit prenosa pri 2‘K ignoriramo!

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

$a-b = a+(-b) = a+(2^n-b) = a-b + 2^n$ (to je bit prenosa)

```
  011  (3)
 +110  (-2)      110=1000-010
 ----
(1)001
```

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

- 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

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) lahko zapišemo kot

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

  - ker pa je pri prvem produktu c_{n-1}=0 in c_n=0, pri drugem pa obratno, ga lahko zapišemo tudi kot

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

- 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)
```

- 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

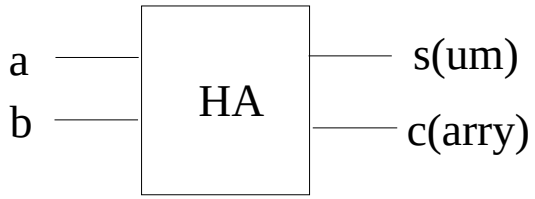
- $4102_5 - 2430_5 = 1122_5$
- $3306_7 - 0615_7 = 2361_7$
- $10110_2 - 01101_2 = 01001_2$

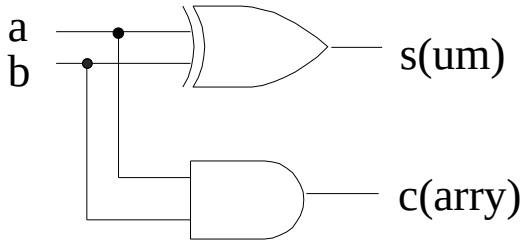
- $324_5 * 023_5 = 014112_5$
- $1101_2 * 0101_2 = 01000001_2$

ARITMETIČNA VEZJA

Polovični seštevalnik

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



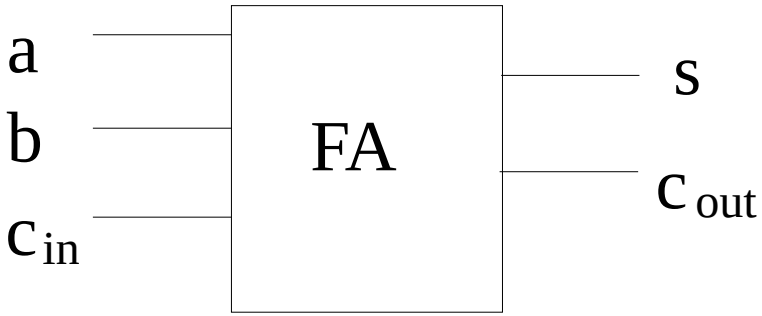


$$s = a\,b' \vee a'\,b = a \oplus b$$
$$c = a\,b \;\; (= a \,\&\, b)$$

a	*b*	*c*	*s*
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Polni seštevalnik

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



$$s = a \oplus b \oplus c_{in} \quad (= a'b'c_{in} \vee a'b\,c_{in}' \vee a\,b'c_{in}' \vee a\,b\,c_{in})$$
$$c_{out} = a\,b \vee a\,c_{in} \vee b\,c_{in}$$

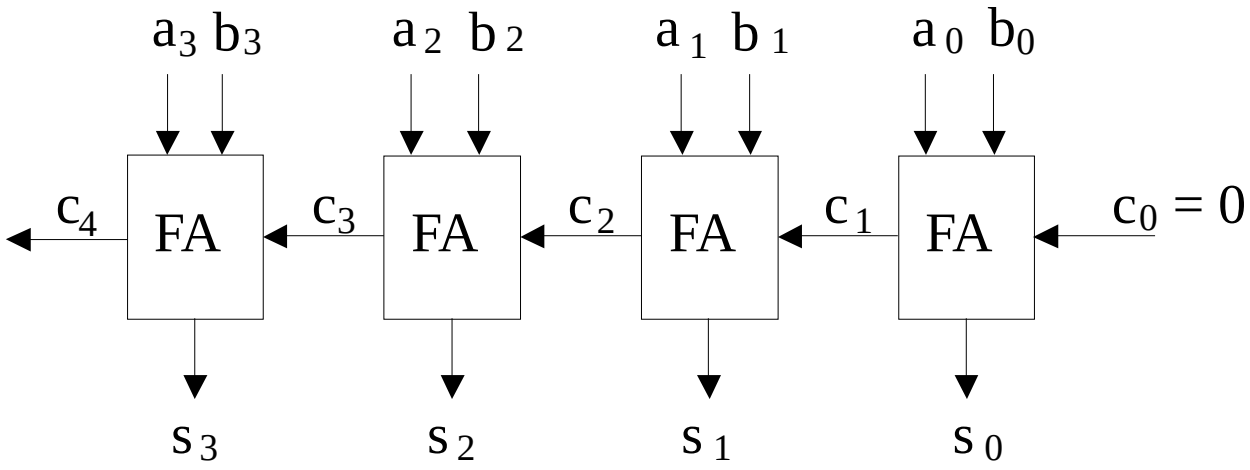
Večbitni seštevalnik

- **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})

$$s = a \oplus b \oplus c_{in}$$
$$c_{out} = a\,b \vee a\,c_{in} \vee b\,c_{in}$$

    - hiba: zakasnitev
      - Dejanska zakasnitev je odvisna od operandov
        - Npr.: pri seštevanju dveh ničel ne bo izhodnega prenosa (ne glede na vhodni prenos), pri seštevanju dveh enic pa bo vedno izhodni prenos (ne glede na vhodni prenos), zato v takih primerih ni treba čakati na vhodni prenos
      - Maksimalna zakasnitev pa narašča praktično linearno
        - V najslabšem primeru se prenos razširja čez vse FA
        - Npr.: če sta pri vseh FA na vhodu različna bita, je treba čakati ne vhodni prenos, ki določa, ali se bo pojavil tudi izhodni prenos

Večbitni seštevalnik



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

Seštevalnik / odštevalnik

- Seštevanje in odštevanje predznačenih števil v 2'K z enim vezjem
  - signal M (Add'/Sub) določa operacijo
    - 0: +
    - 1: –
  - odštevanje kot prištevanje 2'K
    - a – b = a + b' + 1
      - –b kot dvojiški komplement b
      - b' = (b_{n-1}' ... b_1' b_0') ... 1'K
      - $b_i \oplus M$
      - XOR dela kot krmiljen negator ($x \oplus 0 = x$, $x \oplus 1 = x'$)
      - +1: M vežemo na c_0

Binarno množenje

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

x_2	x_1	x_0	×	y_2	y_1	y_0
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		

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

Načini množenja

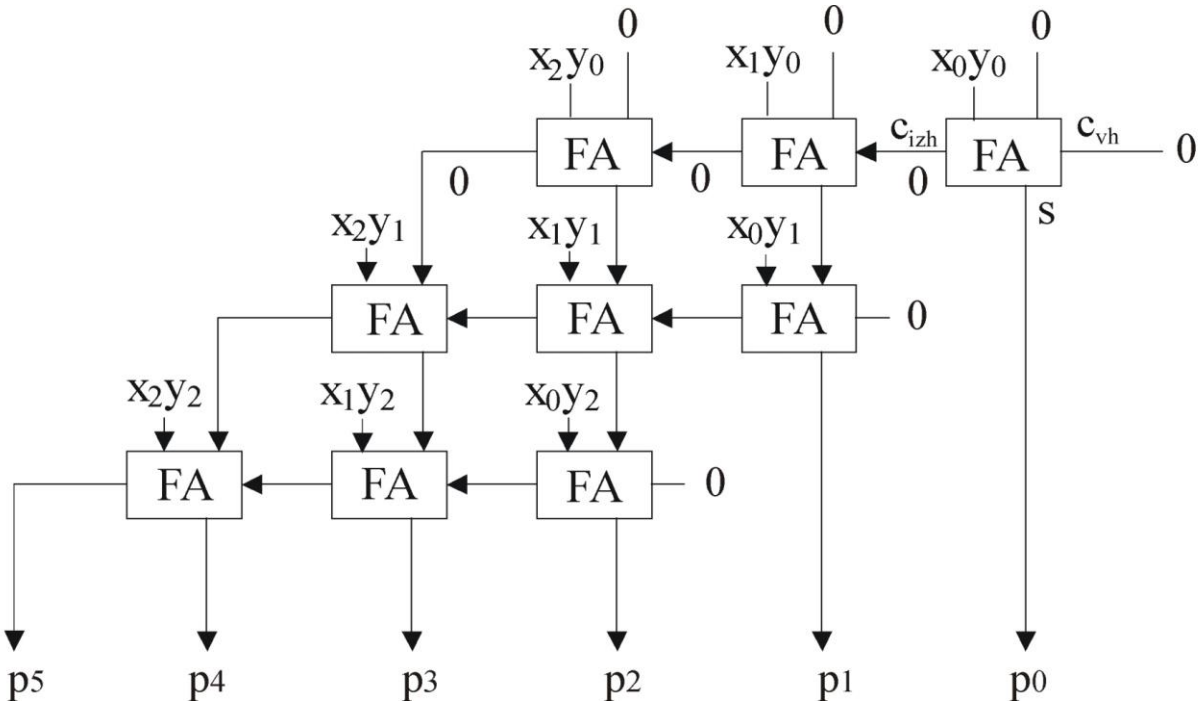
- 2 vrsti metod:
  - pomikanje in seštevanje
    - 1 bit / cikel ure
    - poceni, a ne prav hitro
    - registri
  - kombinacijski množilniki
    - brez ure
    - dragi, a hitri

- Množenje s pomiki in seštevanjem (shift-and-add multiplication)
  - 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)

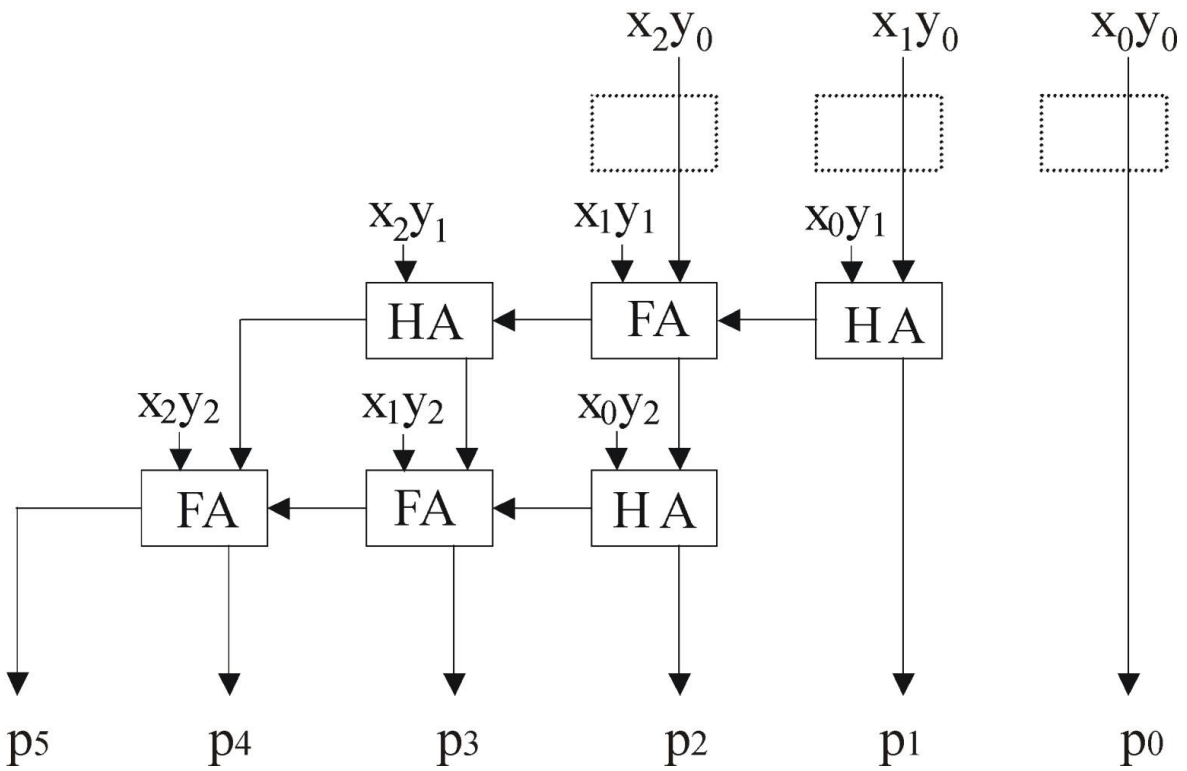
- Primer: A=5, B=6

	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)

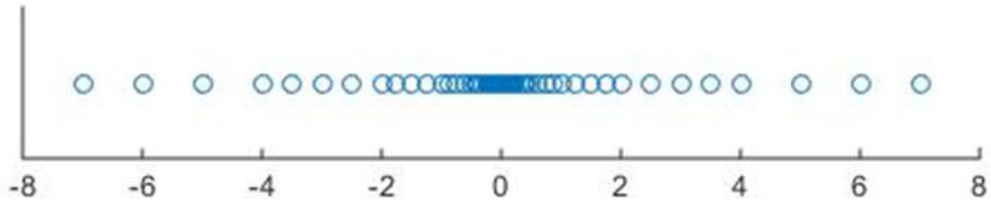
- V plavajoči vejici lahko zapišemo bistveno večja, pa tudi bistveno manjša števila kot v fiksni
  - kljub temu pa je možnih števil enako mnogo (2^n)

- Primer: plavajoča vejica v mini (7-bitnem) formatu
  - predznak: 1 bit, 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: $-111 * 2^0 = -7$

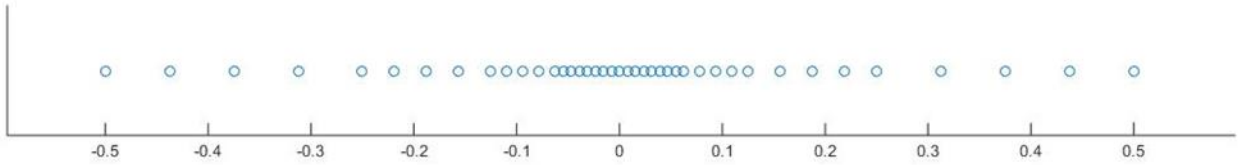
- Primer: plavajoča vejica v mini (5-bitnem) formatu
  - Predznak: 1 bit, mantisa: 2 bita, eksponent: 2 bita
    - $(-1)^S*1,m*2^{E-3}$ (normirana števila)
  - Pri m=0 so števila nenormirana (eksponent -2)
    - $(-1)^S*0,m*2^{-2}$
  - Mejne vrednosti
    - max: $1,11*2^{3-1} = 7$
    - min abs.: $0*2^{-2} = 0$
    - min abs. (razen 0): $1*2^{-4} = 0,0625$, $2*2^{-4} = 0,125$, ...
    - min: $-1,11*2^{3-1} = -7$

	Zapis v fiksni vejici	Vrednost v fiksni vejici	Zapis v plavajoči vejici	Vrednost v plavajoči vejici	
0	0 00 00	0,0	0 00 00	0,0	Nenormirana števila
Min. poz. (razen 0)	0 00 01	$2^{-2} = 0,25$	0 00 01	$0,01*2^{-2} = 2^{-4} = 0, 0625$	
	0 00 10	$2*2^{-2} = 0,50$	0 00 10	$0,10*2^{-2} = 2*2^{-4} = 0,125$	
	0 00 11	$3*2^{-2} = 0,75$	0 00 11	$0,11*2^{-2} = 3*2^{-4} = 0, 1875$	
	0 01 00	1	0 01 00	$1,00*2^{1-1} = 2^{-2} = 1$	
	0 01 01	$1 + 2^{-2} = 1,25$	0 01 01	$1,01*2^{1-1} = 5*2^{-4} = 1,25$	
	0 01 10	$1 + 2*2^{-2} = 1,50$	0 01 10	$1,10*2^{1-1} = 6*2^{-4} = 1,5$	
	0 01 11		0 01 11	$1,11*2^{1-1} = 6*2^{-4} = 1,75$	
	0 10 00		0 10 00	$1,00*2^{2-1} = 1*2^{1} = 2,0$	
	0 10 01		0 10 01	$1,01*2^{2-1} = 1,01*2^{1} = 2,5$	
	0 10 10		0 10 10	$1,10*2^{2-1} = 1,1*2^{1} = 3,0$	
	0 10 11		0 10 11	$1,11*2^{2-1} = 1,11*2^{1} = 3,5$	
	0 11 00	3,00	0 11 00	$1,00*2^{3-1} = 4$	
	0 11 01	3,25	0 11 01	$1,01*2^{3-1} = 5$	
	0 11 10	3,50	0 11 10	$1,10*2^{3-1} = 6$	
Max	0 11 11	3,75	0 11 11	$1,11*2^{3-1} = 7$	

celoten obseg števil:



del obsega:



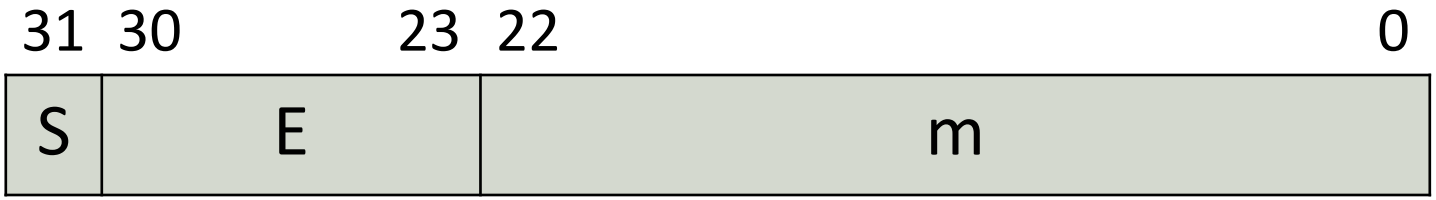
- Vsako število lahko v plavajoči vejici zapišemo na več načinov:
  - npr. $1 \times 10^{18} = 10 \times 10^{17} = 0{,}1 \times 10^{19}$ ...
  - npr. $1 \times 2^{3} = 10 \times 2^{2} = 0{,}1 \times 2^{4}$ ...
  - zato mantiso normiramo:
    - prvi bit je 1 (normalni bit), implicitno predstavljen
    - 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

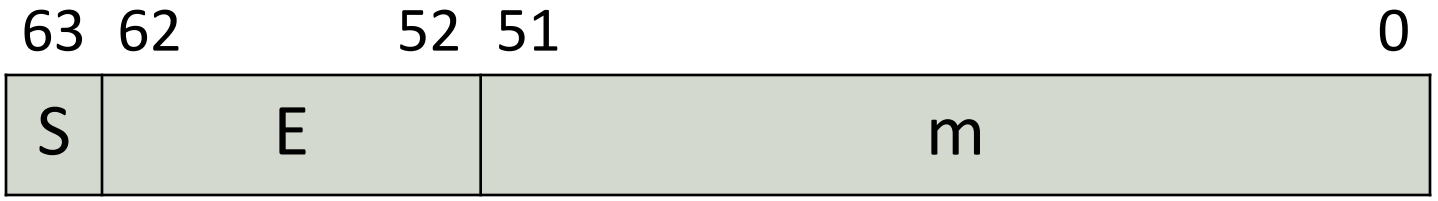
- Enojna natančnost (single precision), 32 bitov



  - 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 $\mathbf{(-1)^S \cdot 1,}\boldsymbol{m} \cdot \mathbf{2^{E-127}}$, *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	00000000000000000000000000000000 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 zaokroži k sodemu številu
    - standard IEEE 754 sicer dovoljuje tudi drugačne načine zaokroževanja, vendar so redkeje uporabljani
  - pri računanju mantiso podaljšamo za 3 dodatne bite
    - varovalni bit (guard bit)
    - zaokroževalni bit (round bit)
    - lepljivi bit (sticky bit)

Dodatni biti

- **Varovalni bit** je potreben, ker je vsota lahko za eno mesto daljša od operandov
- **Zaokroževalni bit** omogoča bolj natančno zaokroževanje
- **Lepljivi bit** se uporablja zato, da se iz izpadlih bitov vidi, ali je bil kak različen od 0 (zaradi zaokroževanja k sodemu številu)
  - 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 (če izpadejo kake enice, se shranijo v lepljivem bitu)
  - seštevanje (odštevanje) mantis
  - Če se pojavi prenos naprej, zmanjšaj mantiso (pomik za eno mesto) in povečaj začasni eksponent za 1
  - Zaokrožitev mantise
    - če grs=100 (točno polovica zadnjega mesta), zaokrožimo k sodemu številu (če je zadnji bit mantise 0, ga pustimo; če je 1, zaokrožimo navzgor)

- **Primer 1**. Seštej binarno 3,25 + 30, če je mantisa 3-bitna, imamo pa dodatne bite g, r in s.
  - $11{,}01*2^0 + 11110{,}0*2^0 = 1{,}101|000*2^1 + 1{,}111|000*2^4 = 1{,}111|000*2^4 + 0{,}001|101*2^4 = 10{,}000101*2^4 = 1{,}000|010(1)*2^5 = 1{,}000|011(grs)*2^5 = 1{,}000*2^5 = 32$

- Primer 2. Odštej binarno 30 - 4,125, če je mantisa 3-bitna, imamo pa dodatne bite g, r in s.

$30_{10} = 11110,0*2^0 = 1,11100*2^4$

$4,125_{10} = 100,001*2^0 = 1,00001*2^2$

(to število ima manjši eksponent (2^2), zato ga povečamo na 2^4, zaradi česar se pomakne mantisa za 2 mesti)

$1,00001*2^2 = 0,010|0001*2^4 = 0,010|001*2^4$

grs, s=0v1=1 grs

$1,111|000*2^4$

$-\ 0,010|001*2^4$

$1,100|111*2^4 = 1,101*2^4 = \mathbf{26}_{10}$

grs

Pravilen rezultat bi bil 25,875 (napaka 0,125 nastane zaradi pomikanja mantise manjšega števila v desno)

Množenje v plavajoči vejici

- Množenje v plavajoči vejici
  - eksponenta seštejemo (dobimo začasni eksponent)
  - mantisi zmnožimo z množilnikom (v fiksni vejici)
    - množilnik v bistvu sploh ne ve, da je nekje vmes vejica …
  - po potrebi normiramo rezultat
  - predznak produkta je XOR obeh predznakov
- Deljenje v plavajoči vejici
  - odštevanje eksponentov, deljenje mantis

Primer 1: $A \cdot B$, $A = 1{,}01 \cdot 2^2$, $B = 1{,}11 \cdot 2^0$

- začasni eksponent = 2 + 0 = 2 (ker je $2^2 * 2^0 = 2^2$)
- množimo mantisi (PAZI: Poleg mantis števili sestavljata tudi implicitni enici!)

$$\begin{array}{r} 1{,}01 \cdot 1{,}11 \\ \hline 101 \\ 101 \\ 101 \\ \hline 10{,}0011 \end{array}$$

Kako vemo, kje je vejica?

- Produkt je 6-biten (3+3), za vejico pa morajo biti 4 mesta (4 = 2+2)
  - Vsak od obeh faktorjev ima 2 mesti desno od vejice

$10{,}0011 \cdot 2^2$ normiramo: $1{,}00011 \cdot 2^3$

- predznak: $0 \oplus 0 = 0$, tj. +

$$A \cdot B = +1{,}00011 \cdot 2^3$$

- Pretvorite A, B in produkt v desetiško obliko in preverite pravilnost rezultata

➢ Primer2: Zmnoži C = A · B v enojni natančnosti (A = 0x326C8000, B = 0xBF200000). Zapiši produkt C tudi v 16-iški obliki.

0x326C8000 = 0011 0010 0110 1100 1000 0000 0000 0000
E = 01100100 = $2^6+2^5+2^2$ = 100
e = E − 127 = −27 (dejanski eksponent)
A = +1,11011001*2^{-27}

0xBF200000 = 1011 1111 0010 0000 0000 0000 0000 0000
E = 01111110 = 128−2 = 126
e = E − 127 = −1 (dejanski eksponent)
B = −1,01*2^{-1}

Zmnožimo mantisi (skupaj z normalnima enicama!):

```
  1,11011001 * 1,01  (A: 9 mest, 8 za vejico, B: 3 mesta, 2 za vejico)
  --------------
   111011001
    000000000
     111011001
  --------------
  10,0100111101  (9+3=12 mest skupno, za vejico jih mora biti 8+2=10)
```

Predznak: 0 xor 1 = 1, torej minus

C = −10,0100111101*2^{-28} (potrebno še normirati)

C = −1,00100111101 *2^{-27} (PAZI: Povečanje eksponenta za 1: -28 + 1 = -27)

E = e + 127 = 100

C = 1 01100100 00100111101000000000000 (dodamo toliko ničel, da je mantisa 23-bitna)

Združujemo v skupine po 4:

C = 1 011|0010|0 001|0011|1101|0000|0000|0000

C = $B213D000_{16}$ (oz. 0xB213D000)

3b ARITMETIKA

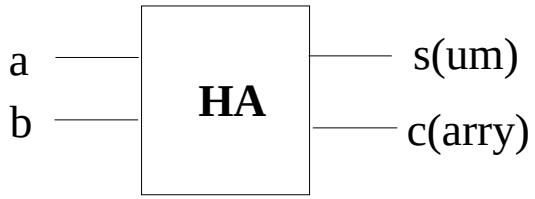
BRANKO ŠTER

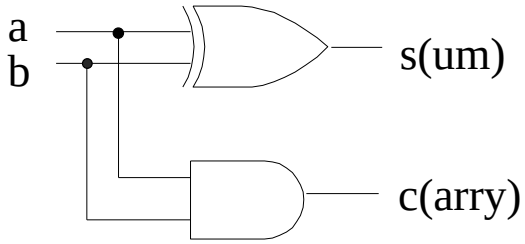
PO KNJIGI – DUŠAN KODEK: ARHITEKTURA IN ORGANIZACIJA RAČUNALNIŠKIH SISTEMOV

ARITMETIČNA VEZJA

Polovični seštevalnik

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



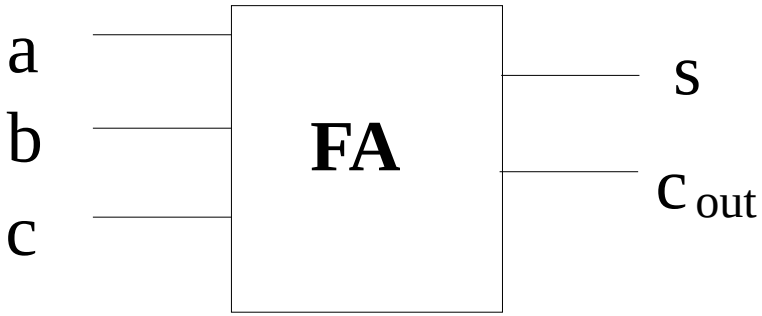


a	*b*	*c*	*s*
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$s = a\,\bar{b} \vee \bar{a}\,b = a \nabla b \,(= a \oplus b)$$
$$c = a\,b \;\; (= a \,\&\, b)$$

Polni seštevalnik

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



$$s = a \nabla b \nabla c \; (= \bar{a}\,\bar{b}\,c \vee \bar{a}\,b\,\bar{c} \vee a\,\bar{b}\,\bar{c} \vee a\,b\,c)$$
$$c_{out} = a\,b \vee a\,c \vee b\,c$$

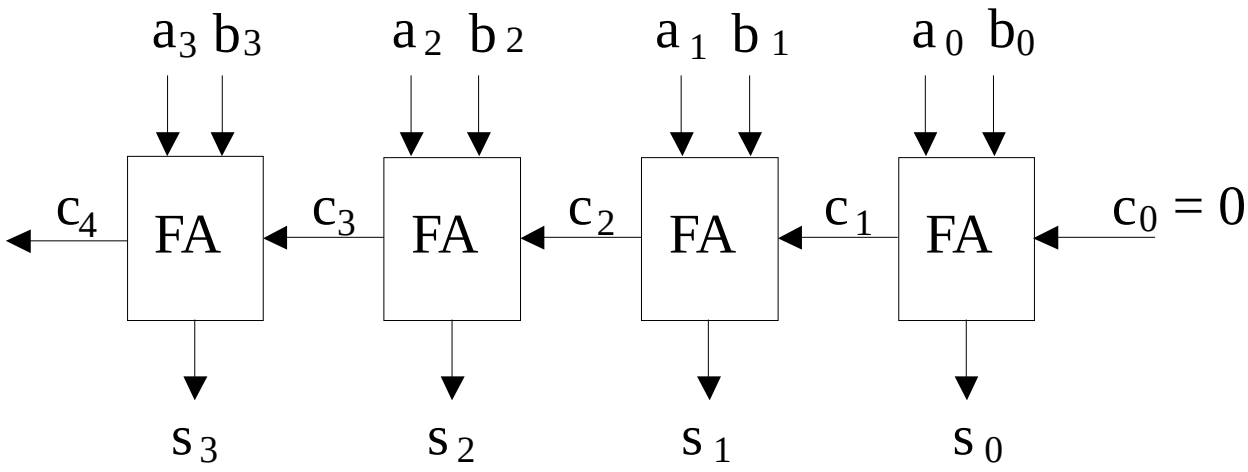
Večbitni seštevalnik

- **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})

$$s = a \nabla b \nabla c_{in}$$
$$c_{out} = a\,b \vee a\,c_{in} \vee b\,c_{in}$$

    - hiba: zakasnitev
      - Dejanska zakasnitev je odvisna od operandov
      - Maksimalna zakasnitev pa narašča praktično linearno

Večbitni seštevalnik

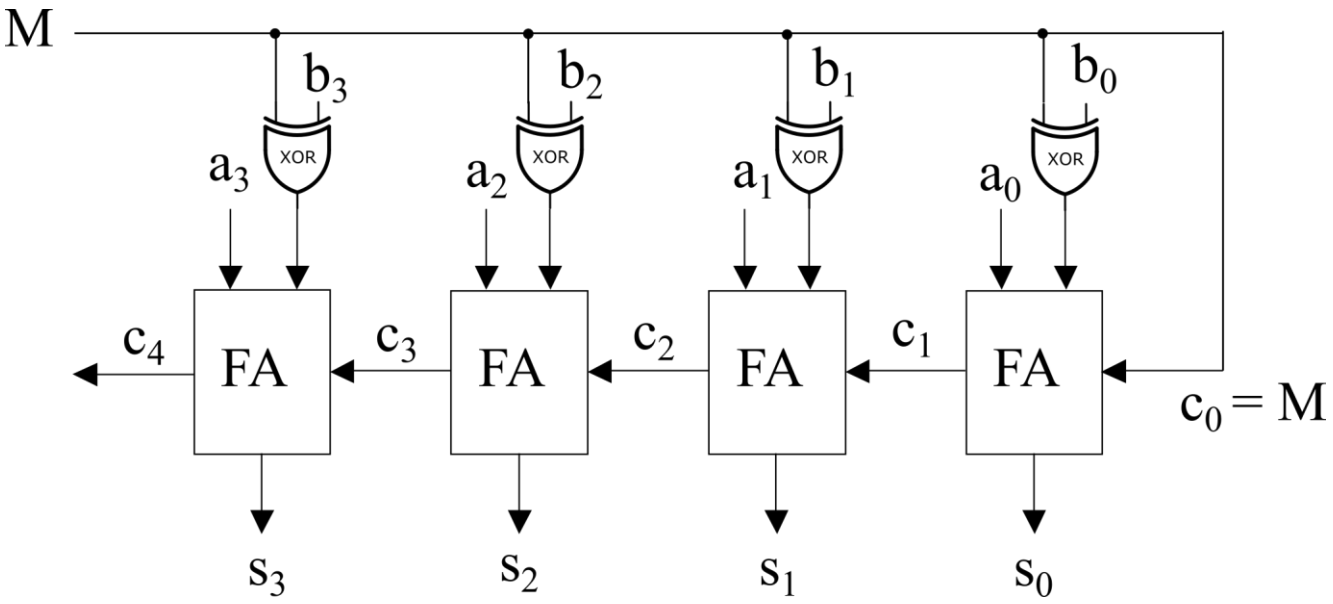


- **Seštevalnik z vnaprejšnjim prenosom** (Carry-Lookahead Adder, CLA)
  - hiter izračun vseh prenosov
    - le na osnovi vhodov a, b in c_0
  - dodatna logika
    - sprememba večnivojske oblike v dvonivojsko

Seštevalnik / odštevalnik

Seštevanje in odštevanje predznačenih števil v 2'K z enim vezjem

- signal M (Add'/Sub) določa operacijo 0: +, 1: –
- odštevanje kot prištevanje 2'K
  - a – b = a + b' + 1
    - –b kot dvojiški komplement b
    - b' = (b_{n-1}' ... b_1' b_0') ... 1'K
    - $b_i \oplus M$
    - XOR dela kot krmiljen negator ($x \oplus 0 = x$, $x \oplus 1 = x'$)
    - +1: M vežemo na c_0



Binarno množenje

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

x_2	x_1	x_0	×	y_2	y_1	y_0
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		

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

Načini množenja

- 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 (shift-and-add multiplication)

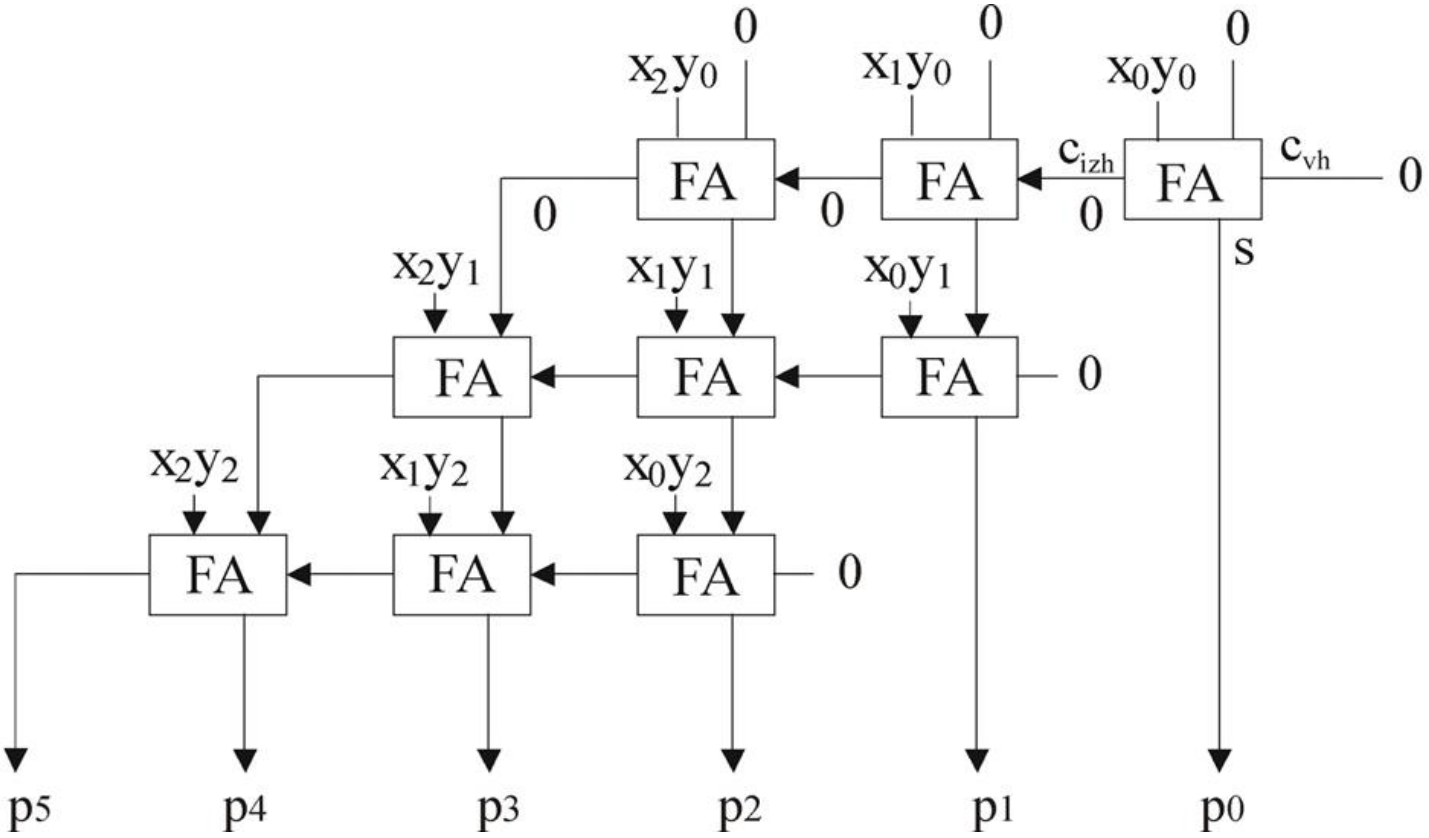
- Postopek iz n korakov:
  - Če je najnižji bit množitelja B enak 1, prištej množenec A registru P (na začetku 0)
    - sicer prištej 0
  - Pomik desno registrov P in B (kaskadno vezanih)

Primer: A=5, B=6

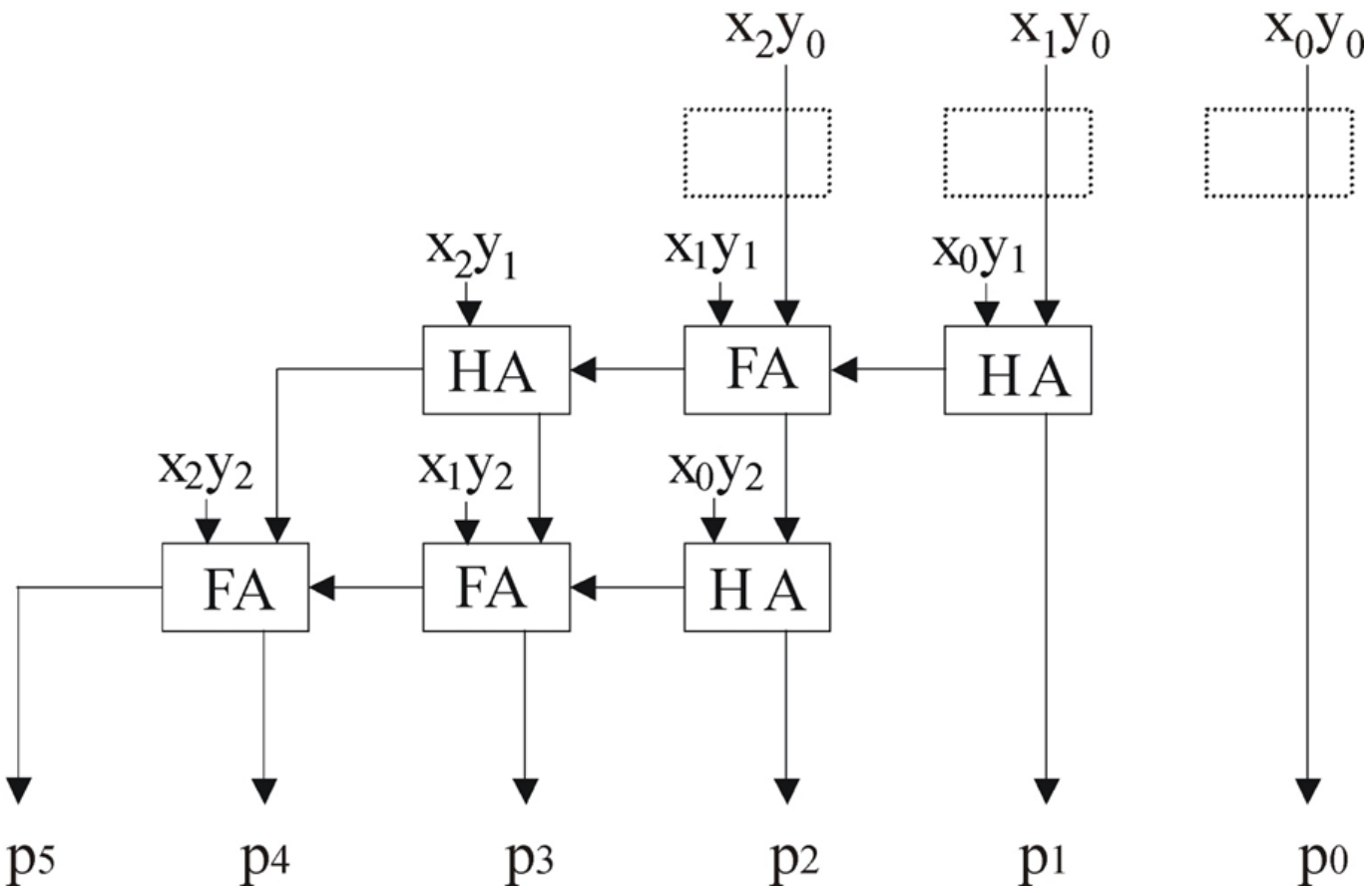
	P	B	
0	0000	0110	začetek
1	0000	0110	P ← P + 0
	0000	0011	P,B >> 1
2	0101	0011	P ← P + A
	0010	1001	P,B >> 1
3	0111	1001	P ← P + A
	0011	1100	P,B >> 1
4	0011	1100	P ← P + 0
	0001	1110	P,B >> 1

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)
      - CPE 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 (po absolutni vrednosti) števila kot v fiksni
  - kljub temu pa je možnih števil enako mnogo (2^n)

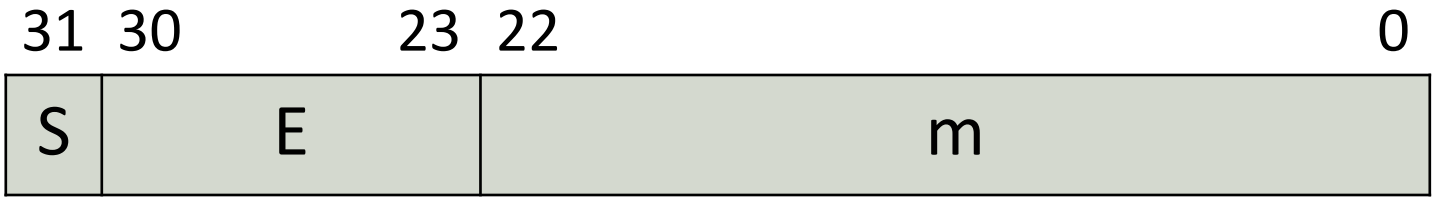
- Vsako število lahko v plavajoči vejici zapišemo na več načinov:
  - npr. $1 \times 10^{18} = 10 \times 10^{17} = 0{,}1 \times 10^{19}$ ...
  - npr. $1 \times 2^{3} = 10 \times 2^{2} = 0{,}1 \times 2^{4}$ ...
  - zato mantiso normiramo:
    - prvi bit je 1 (normalni bit), implicitno predstavljen
    - 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, 2008, 2019)
  - IEEE: Institute of Electrical and Electronics Engineers
  - 2 osnovna formata:
    - enojna natančnost (single precision), 32 bitov
    - dvojna natančnost (double precision), 64 bitov
  - Dodatni:
    - polovična (half), 16 bitov
    - 4-kratna (quadruple), 128 bitov
    - 8-kratna, 256 bitov
    - desetiški zapis, 32, 64, 128 bitov

Enojna natančnost

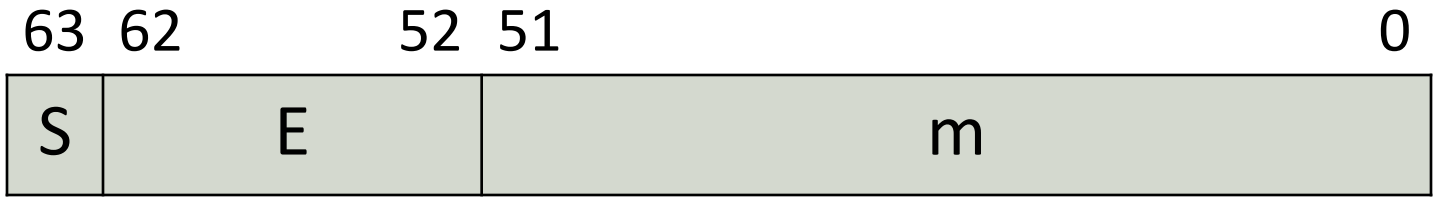
- Enojna natančnost (single precision), 32 bitov



- predznak *S* (0: +, 1: –)
- 8-bitni 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-bitni 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	00000000000000000000000000 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 zaokroži k sodemu številu
    - standard IEEE 754 sicer dovoljuje tudi drugačne načine zaokroževanja, vendar so redkeje uporabljani
  - pri računanju mantiso podaljšamo za 3 dodatne bite
    - varovalni bit (guard bit)
    - zaokroževalni bit (round bit)
    - lepljivi bit (sticky bit)

Dodatni biti

- **Varovalni bit** je potreben, ker je vsota lahko za eno mesto daljša od operandov
- **Zaokroževalni bit** omogoča bolj natančno zaokroževanje
- **Lepljivi bit** se uporablja zato, da se iz izpadlih bitov vidi, ali je bil kak različen od 0 (zaradi zaokroževanja k sodemu številu)
  - 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 (če izpadejo kake enice, se shranijo v lepljivem bitu)
  - Seštevanje (odštevanje) mantis
  - Če se pojavi prenos naprej, zmanjšaj mantiso (pomik za eno mesto) in povečaj začasni eksponent za 1
  - Zaokrožitev mantise
    - če grs=100 (točno polovica zadnjega mesta), zaokrožimo k sodemu številu (če je zadnji bit mantise 0, ga pustimo; če je 1, zaokrožimo navzgor)

- **Primer 1**. Seštej binarno 3,25 + 30, če je mantisa 3-bitna, imamo pa dodatne bite g, r in s.

$30 = 11110{,}0 * 2^0 = 1{,}1110 * 2^4$

$3{,}25 = 11{,}01 * 2^0 = 1{,}101 * 2^1$

(drugo število ima manjši eksponent (2^1), zato ga povečamo na 2^4, zaradi česar se pomakne mantisa za 3 mesta)

$1{,}101 * 2^1 = 0{,}001101 * 2^4$

$$
\begin{array}{rl}
 & \quad grs \\
 & 1{,}111\,|\,000 * 2^4 \\
+ & 0{,}001\,|\,101 * 2^4 \\
\hline
 & 10{,}000\,|\,101 * 2^4 \\
= & 1{,}000\,|\,0\underline{101} * 2^5 \qquad 0 \text{ V } 1 = 1 \text{ (sticky)} \\
= & 1{,}000\,|\,011 * 2^5 = 1{,}000 * 2^5 = \underline{\mathbf{32}}
\end{array}
$$

(napaka 1,25)

- **Primer 2.** Odštej binarno 30 - 4,125, če je mantisa 3-bitna, imamo pa dodatne bite g, r in s.

$30_{10} = 11110,0*2^0 = 1,11100*2^4$

$4,125_{10} = 100,001*2^0 = 1,00001*2^2$

(to število ima manjši eksponent (2^2), zato ga povečamo na 2^4, zaradi česar se pomakne mantisa za 2 mesti)

$1,00001*2^2 = 0,010|0001*2^4 = 0,010|001*2^4$

grs, s=0v1=1 grs

$1,111|000*2^4$

$-\ 0,010|001*2^4$

$1,100|111*2^4 = 1,101*2^4 = \mathbf{26}_{10}$

grs

Pravilen rezultat bi bil 25,875 (napaka 0,125 nastane zaradi pomikanja mantise manjšega števila v desno)

Množenje v plavajoči vejici

- Množenje v plavajoči vejici
  - eksponenta seštejemo (dobimo začasni eksponent)
  - mantisi zmnožimo z množilnikom (v fiksni vejici)
    - množilnik v bistvu sploh ne ve, da je nekje vmes vejica …
  - po potrebi normiramo rezultat
  - predznak produkta je XOR obeh predznakov
- Deljenje v plavajoči vejici
  - odštevanje eksponentov, deljenje mantis

Primer 1: $A \cdot B$, $A = 1{,}01 \cdot 2^2$, $B = 1{,}11 \cdot 2^0$

- začasni eksponent = 2 + 0 = 2 (ker je $2^2 * 2^0 = 2^2$)
- množimo mantisi (PAZI: Poleg mantis števili sestavljata tudi implicitni enici!)

$$\begin{array}{r} 1{,}01 \cdot 1{,}11 \\ 101 \\ 101 \\ \underline{101} \\ 10{,}0011 \end{array}$$

Kako vemo, kje je vejica?

- Produkt je 6-biten (3+3), za vejico pa morajo biti 4 mesta (4 = 2+2)
  - Vsak od obeh faktorjev ima 2 mesti desno od vejice

$10{,}0011 \cdot 2^2$ normiramo: $1{,}00011 \cdot 2^3$

- predznak: $0 \oplus 0 = 0$, tj. +

$$A \cdot B = +1{,}00011 \cdot 2^3$$

- Pretvorite A, B in produkt v desetiško obliko in preverite pravilnost rezultata

- Primer2: Zmnoži C = A · B v enojni natančnosti (A = 0x326C8000, B = 0xBF200000). Zapiši produkt C tudi v 16-iški obliki.

0x326C8000 = 0011 0010 0110 1100 1000 0000 0000 0000

E = 01100100 = $2^6+2^5+2^2$ = 100

e = E − 127 = -27 (dejanski eksponent)

A = +1,11011001*2^{-27}

0xBF200000 = 1011 1111 0010 0000 0000 0000 0000 0000

E = 01111110 = 128-2 = 126

e = E − 127 = -1 (dejanski eksponent)

B = -1,01*2^{-1}

Zmnožimo mantisi (skupaj z normalnima enicama!):

```
1,11011001 * 1,01  (A: 9 mest, 8 za vejico, B: 3 mesta, 2 za vejico)
-------------
 111011001
  000000000
   111011001
-------------
10,0100111101  (9+3=12 mest skupno, za vejico jih mora biti 8+2=10)
```

Predznak: 0 xor 1 = 1, torej minus

C = $-10,0100111101 * 2^{-28}$ (potrebno še normirati)

C = $-1,00100111101 * 2^{-27}$ (PAZI: Povečanje eksponenta za 1: -28 + 1 = -27)

E = e + 127 = 100

C = 1 **01100100** 00100111101000000000000 (dodamo toliko ničel, da je mantisa 23-bitna)

Združujemo v skupine po 4:

C = 1 **011|0010|0** 001|0011|1101|0000|0000|0000

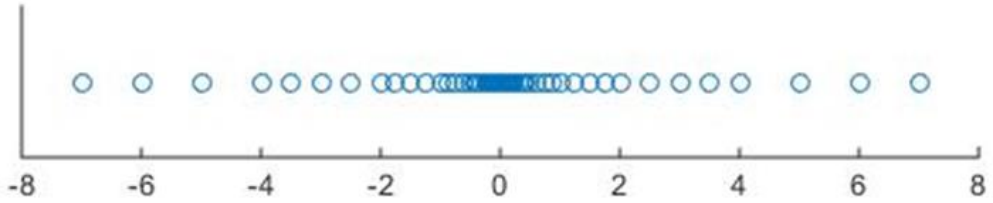
C = $B213D000_{16}$ (oz. 0xB213D000)

- Primer: plavajoča vejica v ‘mikro’ (5-bitnem) formatu
  - Predznak: 1 bit, mantisa: 2 bita, eksponent: 2 bita
    - $(-1)^S*1,m*2^{E-3}$ (normirana števila)
  - Pri m=0 so števila nenormirana (eksponent -2)
    - $(-1)^S*0,m*2^{-2}$
  - Mejne vrednosti
    - max: $1,11*2^{3-1} = 7$
    - min abs.: $0*2^{-2} = 0$
    - min abs. (razen 0): $1*2^{-4} = 0,0625$, $2*2^{-4} = 0,125$, ...
    - min: $-1,11*2^{3-1} = -7$

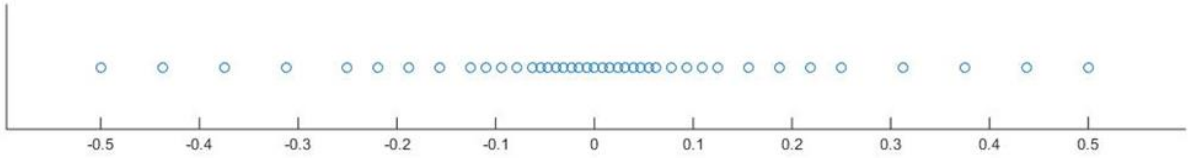
	Zapis v fiksni vejici	Vrednost v fiksni vejici	Zapis v plavajoči vejici	Vrednost v plavajoči vejici	
0	0 00 00	0,0	0 00 00	0,0	Nenormirana števila
Min. poz. (razen 0)	0 00 01	$2^{-2} = 0,25$	0 00 01	$0,01*2^{-2} = 2^{-4} = 0, 0625$	
	0 00 10	$2*2^{-2} = 0,50$	0 00 10	$0,10*2^{-2} = 2*2^{-4} = 0,125$	
	0 00 11	$3*2^{-2} = 0,75$	0 00 11	$0,11*2^{-2} = 3*2^{-4} = 0, 1875$	
	0 01 00	1	0 01 00	$1,00*2^{1-1} = 2^{-2} = 1$	
	0 01 01	$1 + 2^{-2} = 1,25$	0 01 01	$1,01*2^{1-1} = 5*2^{-4} = 1,25$	
	0 01 10	$1 + 2*2^{-2} = 1,50$	0 01 10	$1,10*2^{1-1} = 6*2^{-4} = 1,5$	
	0 01 11		0 01 11	$1,11*2^{1-1} = 6*2^{-4} = 1,75$	
	0 10 00		0 10 00	$1,00*2^{2-1} = 1*2^{1} = 2,0$	
	0 10 01		0 10 01	$1,01*2^{2-1} = 1,01*2^{1} = 2,5$	
	0 10 10		0 10 10	$1,10*2^{2-1} = 1,1*2^{1} = 3,0$	
	0 10 11		0 10 11	$1,11*2^{2-1} = 1,11*2^{1} = 3,5$	
	0 11 00	3,00	0 11 00	$1,00*2^{3-1} = 4$	
	0 11 01	3,25	0 11 01	$1,01*2^{3-1} = 5$	
	0 11 10	3,50	0 11 10	$1,10*2^{3-1} = 6$	
Max	0 11 11	3,75	0 11 11	$1,11*2^{3-1} = 7$	

- Primer: plavajoča vejica v ‘mini’ (7-bitnem) formatu
  - predznak: 1 bit, 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:



3
OSNOVNI PRINCIPI DELOVANJA RAČUNALNIKOV

BRANKO ŠTER

PO KNJIGI - DUŠAN KODEK: ARHITEKTURA IN ORGANIZACIJA RAČUNALNIŠKIH SISTEMOV

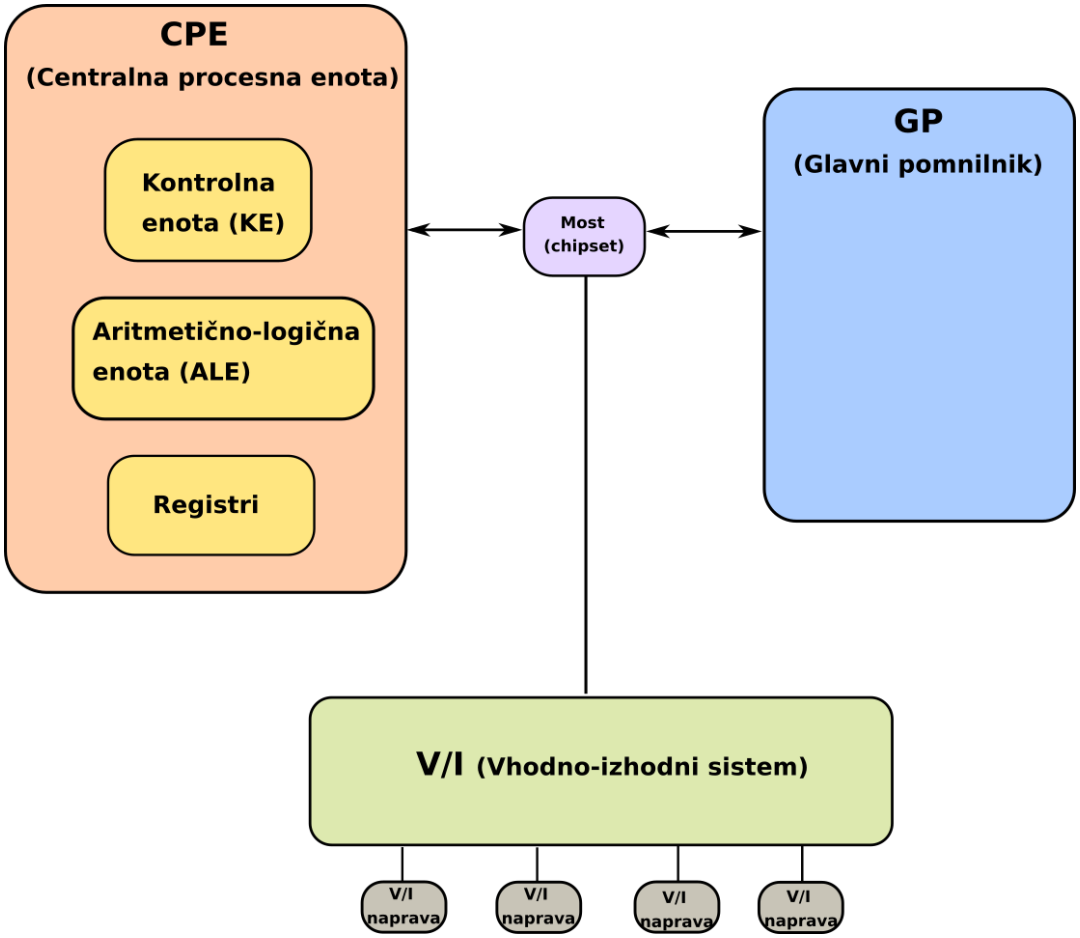
- 2 ugotovitvi iz prvih dveh poglavij:
  - Definicija izračunljivosti po Church-Turingovi hipotezi
  - lastnosti stroja, ki je zmožen izračunati vse, kar se da izračunati
- Von Neumannov računalnik
  - ekvivalenca* s TM
  - to ni edini možen tak stroj

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

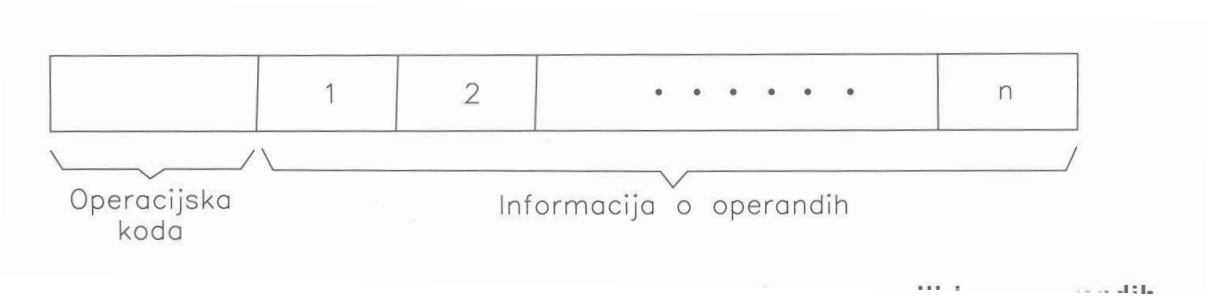
- zakaj glavni
- v njem so shranjeni ukazi in operandi
- GP sestavljajo pomnilniške besede (vsaka ima svoj naslov)
- tehnologija DRAM

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

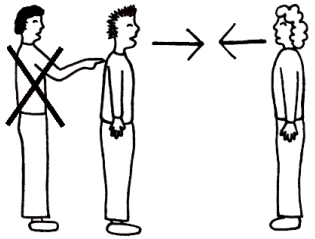
- Ukaz je shranjen v eni ali več (sosednih) pomnilniških besedah
- Vsak ukaz vsebuje
  - operacijsko kodo (katera operacija naj se izvrši)
  - informacijo o operandih, nad katerimi naj se izvrši operacija
- Format ukaza pove, kako so biti ukaza razdeljeni na operacijsko kodo in operande



- 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)

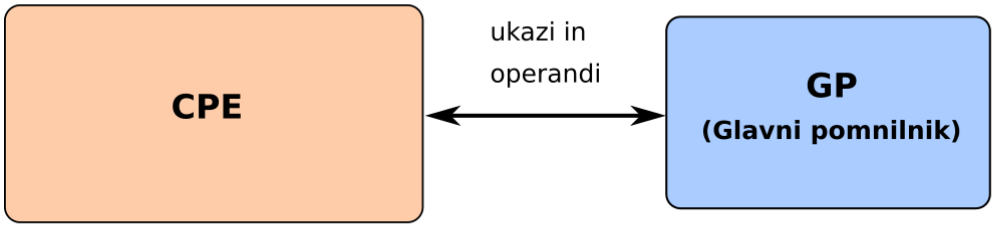


  - takrat se izvrši skok na prvi ukaz **prekinitvenega servisnega programa** (PSP)
    - pred tem se shrani vrednost PC

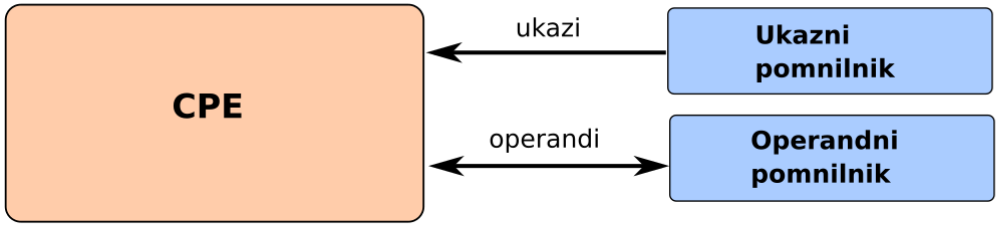
Glavni pomnilnik

- 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)

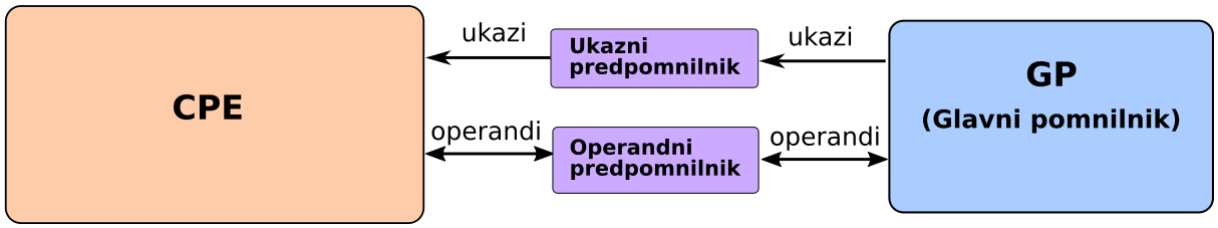
Princetonska arhitektura



Harvardska arhitektura



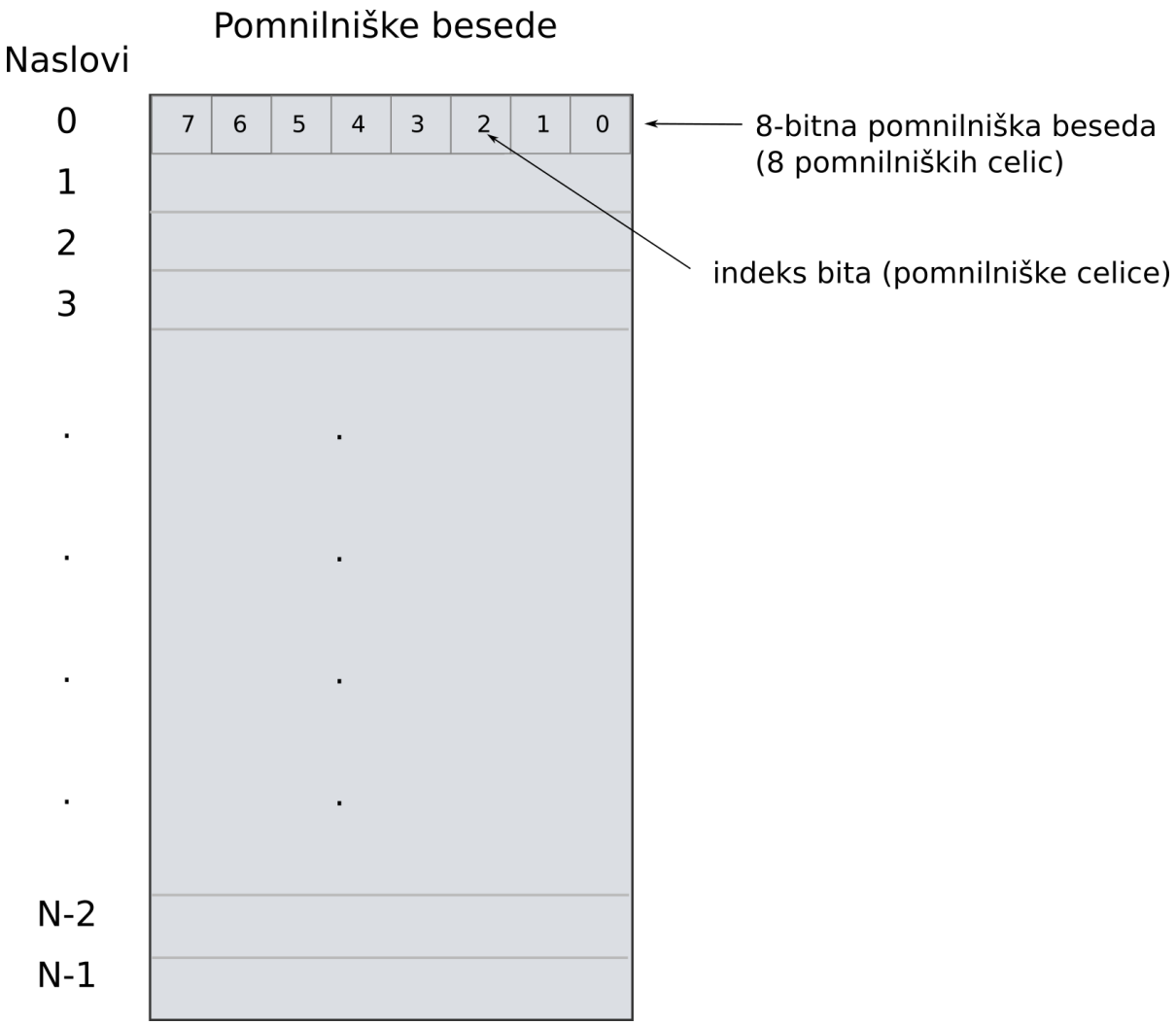
Danes prevladuje Princetonska arhitektura, vendar z ločenima *predpomnilnikoma* za ukaze in operande:



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:

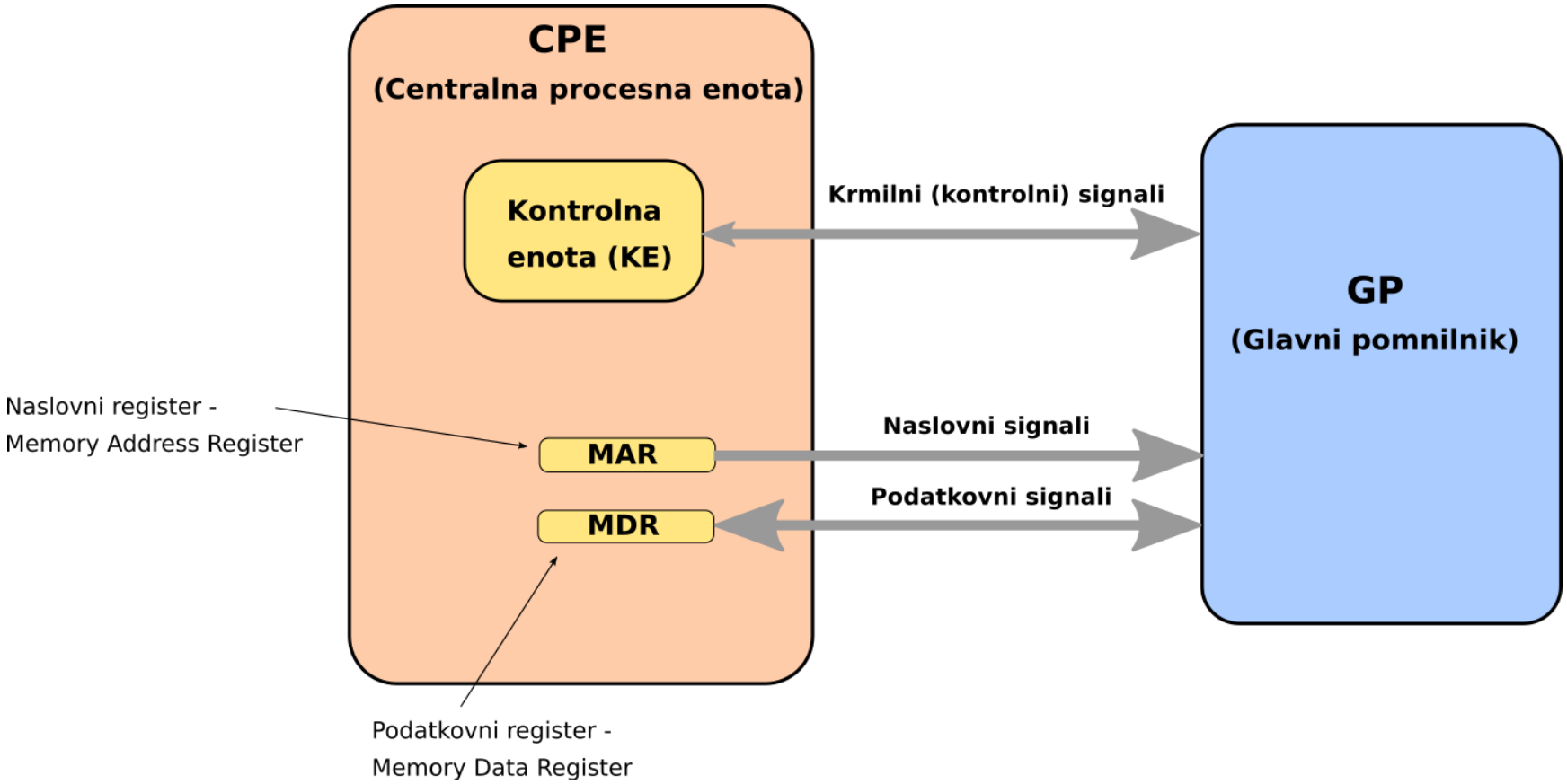


Naslovni prostor

- 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 ***krmilnimi (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
(Centralna procesna enota)
Kontrolna enota (KE)
Krmilni (kontrolni) signali
GP
(Glavni pomnilnik)
Naslovni register -
Memory Address Register
MAR
Naslovni signali
Podatkovni signali
MDR
Podatkovni register -
Memory Data Register

- 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

3
ZAPIS INFORMACIJE

BRANKO ŠTER

PO KNJIGI - DUŠAN KODEK: ARHITEKTURA IN ORGANIZACIJA RAČUNALNIŠKIH SISTEMOV

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 ASCII

- ASCII - American Standard Code for Information Interchange (1968)
- 7-bitna (128 znakov)
- 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

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

- Razširjena ‘ASCII’
  - 8-bitna
    - dodatnih 128 znakov
  - ISO/IEC 8859
    - ISO 8859-1 (Latin-1), zahodnoevropske črke
    - ISO 8859-2 (Latin-2), vzhodnoevropske črke

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 (Unicode transformation format)
  - UTF-8
    - posamezen znak zavzame od 1 do 4 bajtov
      - kodiranje spremenljive dolžine – variable length encoding
    - 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
  - $192{,}73 = 1\times10^{2} + 9\times10^{1} + 2\times10^{0} + 7\times10^{-1} + 3\times10^{-2}$

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.

  $110101{,}101_2 =$

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

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 $r = 2$
  - $0{,}011_2$

Napaka pri rezanju decimalk

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

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

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

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

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

  $\mathbf{\log_a c = \log_a b * \log_b c}$ (pravilo)

  (na ta način se znebimo baze *b*, v našem primeru *r*,

  za a pa vzamemo kako znano bazo)

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

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

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

- Primer: pretvorba $N = 0{,}8_{10}$ v bazo $r = 3$. Vzemi toliko decimalk, da napaka $|N'-N|$ 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 $r = 16$. $|N'\text{-}N| \leq 10^{-3}$

  $9(15),4(12)(12)(12)\ldots_{16}$

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

  $k = 3$

  $159{,}310 = 9(15),4(12)(12)_{16}$ oz. $9F{,}4CC_{16}$

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:
    - $3C7_{16} = 3C7_H = 0x3C7 = \$3C7$

Sorodne baze

- 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
    - $A15_{16} = 1010\ 0001\ 0101_2$

Nepredznačena števila

- 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 števk (cifer) ni pravilen

101 + 100 = (1)001

Primeri aritmetičnih operacij z nepredznačenimi števili v različnih bazah

- $0234_8 + 1525_8 = 1761_8$
- $2103_4 + 2313_4 = (1)1022_4$, pojavi se prenos
- $11001_2 + 01011_2 = (1)00100_2$, pojavi se prenos

- $3306_8 – 0615_8 = 2471_8$
- $A089_{16} – 5CED_{16} = 439C_{16}$
- $10110_2 – 01101_2 = 01001_2$

- $325_8 * 026_8 = 12016_8$
- $1101_2 * 0101_2 = 01000001_2$

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

1 Predznak-veličinski zapis (PV)

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

- 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)

2 Zapis (predstavitev) z odmikom (PO)

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

- odmik je (običajno) 2^{n-1}
- nekoč priljubljen zapis
- Hibe:
  - pri seštevanju je treba odmik odšteti
  - pri odštevanju je treba odmik prišteti

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)

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 ☹ (torej V)

- Negativno število (zapis b pri podani vrednosti V) dobimo
  - tako, da vrednosti V prištejemo 2^n
    - Npr.: -2 + 16 = 14, torej tak zapis kot za nepredznačeno 14
  - lahko pa tudi 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)
```

- Tudi obratno, če želimo ugotoviti, za katero negativno število gre:
  - nepredznačenemu zapisu odštejemo 2^n
    - Npr., 1101: 13 – 16 = -3
  - spet naredimo 2'K (1'K in prištevanje enice):
    - 1101: 1'K: 0010, +1 = 0011 (=3)

- Potrebno je razlikovati med pojmoma
  - *zapis v 2'K* in
  - *2'K nekega števila !*

- Bit prenosa pri 2‘K ignoriramo!

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

$a-b = a+(-b) = a+(2^n-b) = a-b + \mathbf{2^n}$ (to je bit prenosa)

```
  011 (3)
 +110 (-2)
 ----
(1)001
```

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

- Pri razširitvi števila na več bitov je potrebno **razširiti predznak**:
  - **0**101 → **0000 0**101
  - **1**100 → **1111 1**100
  - **0**10111 → **000**10111
  - **1**00011 → **111**00011

Primer

- Zapiši -37 kot predznačeno 8-bitno število v PV, PO, 1’K in 2’K
  - `PV:  10100101`
  - `PO:  01011011`
  - `1’K: 11011010`
  - `2’K: 11011011`

Osnovna aritmetika v 2'K

- 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

Preliv

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

- Pogoj za preliv (OF) lahko zapišemo kot

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

  - ker pa je pri prvem produktu c_{n-1}=0 in c_n=0, pri drugem pa obratno, ga lahko zapišemo tudi kot

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

- 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)
```

- Seštej 21 in -7 v 6-bitnem 2'K:

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

3
ZAPIS INFORMACIJE

BRANKO ŠTER

PO KNJIGI - DUŠAN KODEK: ARHITEKTURA IN ORGANIZACIJA RAČUNALNIŠKIH SISTEMOV

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 ASCII

- ASCII - American Standard Code for Information Interchange (1968)
- 7-bitna (128 znakov)
- 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

- Razširjena ‘ASCII’
  - 8-bitna
    - dodatnih 128 znakov
  - ISO/IEC 8859
    - ISO 8859-1 (Latin-1), zahodnoevropske črke
    - ISO 8859-2 (Latin-2), vzhodnoevropske črke

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 (Unicode transformation format)
  - UTF-8
    - posamezen znak zavzame od 1 do 4 bajtov
      - kodiranje spremenljive dolžine – variable length encoding
    - 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.

  $110101{,}101_2 =$

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

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 $r = 2$
  - $0{,}011_2$

Napaka pri rezanju decimalk

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

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

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

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

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

  $\mathbf{\log_a c = \log_a b * \log_b c}$ (pravilo)

  (na ta način se znebimo baze *b*, v našem primeru *r*,

  za a pa vzamemo kako znano bazo)

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

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

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

- Primer: pretvorba $N = 0{,}8_{10}$ v bazo $r = 3$. Vzemi toliko decimalk, da napaka $|N'-N|$ 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 $r = 16$. $|N'\text{-}N| \leq 10^{-3}$

  $9(15),4(12)(12)(12)\ldots_{16}$

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

  $k = 3$

  $159{,}310 = 9(15),4(12)(12)_{16}$ oz. $9F{,}4CC_{16}$

Pretvorba med poljubnima bazama

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

- Npr. $26{,}5_8$ v *r*=3
  - 211,12 12 ... $_3$

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:
    - $3C7_{16} = 3C7_H = 0x3C7 = \$3C7$

Sorodne baze

- 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
    - $A15_{16} = 1010\ 0001\ 0101_2$

Nepredznačena števila

- 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 števk (cifer) ni pravilen

101 + 100 = (1)001

Primeri aritmetičnih operacij z nepredznačenimi števili v različnih bazah

- $0234_8 + 1525_8 = 1761_8$
- $2103_4 + 2313_4 = (1)1022_4$, pojavi se prenos
- $11001_2 + 01011_2 = (1)00100_2$, pojavi se prenos

- $3306_8 – 0615_8 = 2471_8$
- $A089_{16} – 5CED_{16} = 439C_{16}$
- $10110_2 – 01101_2 = 01001_2$

- $325_8 * 026_8 = 12016_8$
- $1101_2 * 0101_2 = 01000001_2$

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

1 Predznak-veličinski zapis (PV)

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

- 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)

2 Zapis (predstavitev) z odmikom (PO)

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

- odmik je (običajno) 2^{n-1}
- nekoč priljubljen zapis
- Hibe:
  - pri seštevanju je treba odmik odšteti
  - pri odštevanju je treba odmik prišteti

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)

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 ☹ (torej V)

- Negativno število (zapis b pri podani vrednosti V) dobimo
  - tako, da vrednosti V prištejemo 2^n
    - Npr.: -2 + 16 = 14, torej tak zapis kot za nepredznačeno 14
  - lahko pa tudi 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)
```

- Tudi obratno, če želimo ugotoviti, za katero negativno število gre:
  - nepredznačenemu zapisu odštejemo 2^n
    - Npr., 1101: 13 – 16 = -3
  - spet naredimo 2'K (1'K in prištevanje enice):
    - 1101: 1'K: 0010, +1 = 0011 (=3)

- Potrebno je razlikovati med pojmoma
  - *zapis v 2'K* in
  - *2'K nekega števila !*

- Bit prenosa pri 2‘K ignoriramo!

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

$a-b = a+(-b) = a+(2^n-b) = a-b + \mathbf{2^n}$ (to je bit prenosa)

```
  011 (3)
 +110 (-2)
 ----
(1)001
```

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

- Pri razširitvi števila na več bitov je potrebno **razširiti predznak**:
  - **0**101 → **0000 0**101
  - **1**100 → **1111 1**100
  - **0**10111 → **000**10111
  - **1**00011 → **11 1**00011

Primer

- Zapiši -37 kot predznačeno 8-bitno število v PV, PO, 1’K in 2’K
  - `PV:  10100101`
  - `PO:  01011011`
  - `1’K: 11011010`
  - `2’K: 11011011`

Osnovna aritmetika v 2'K

- 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

Preliv

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

- Pogoj za preliv (OF) lahko zapišemo kot

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

  - ker pa je pri prvem produktu c_{n-1}=0 in c_n=0, pri drugem pa obratno, ga lahko zapišemo tudi kot

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

- 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)
```

- Seštej 21 in -7 v 6-bitnem 2'K:

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

4
OSNOVNI PRINCIPI DELOVANJA RAČUNALNIKOV

BRANKO ŠTER

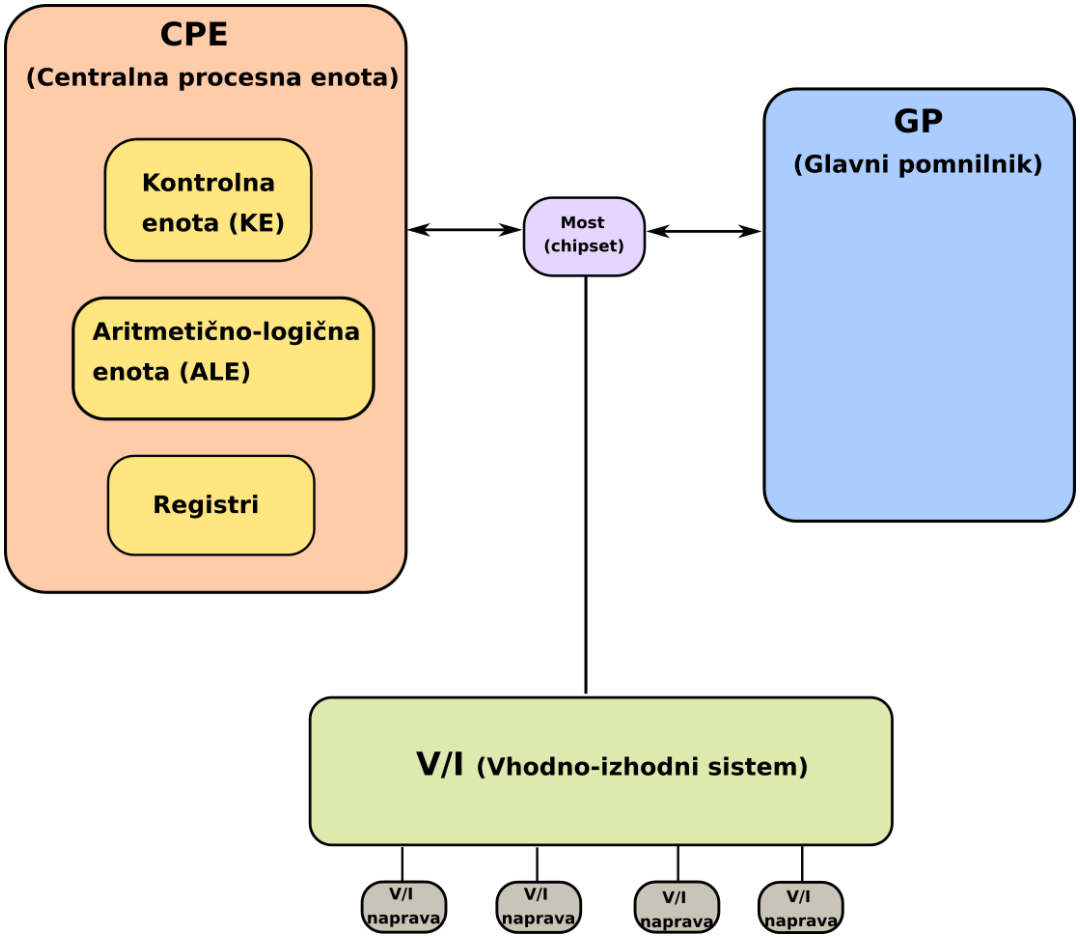
PO KNJIGI - DUŠAN KODEK: ARHITEKTURA IN ORGANIZACIJA RAČUNALNIŠKIH SISTEMOV

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

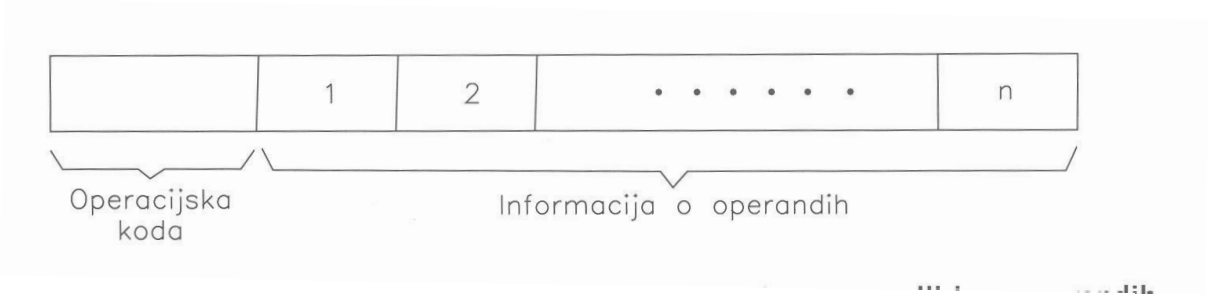
- zakaj glavni
- v njem so shranjeni ukazi in operandi
- GP sestavljajo pomnilniške besede (vsaka ima svoj naslov)
- tehnologija DRAM

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

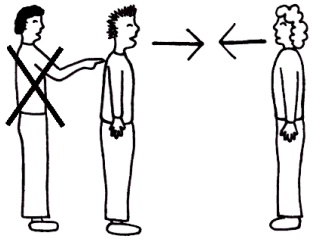
- Ukaz je shranjen v eni ali več (sosednih) pomnilniških besedah
- Vsak ukaz vsebuje
  - operacijsko kodo (katera operacija naj se izvrši)
  - informacijo o operandih, nad katerimi naj se izvrši operacija
- Format ukaza pove, kako so biti ukaza razdeljeni na operacijsko kodo in operande



- 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)

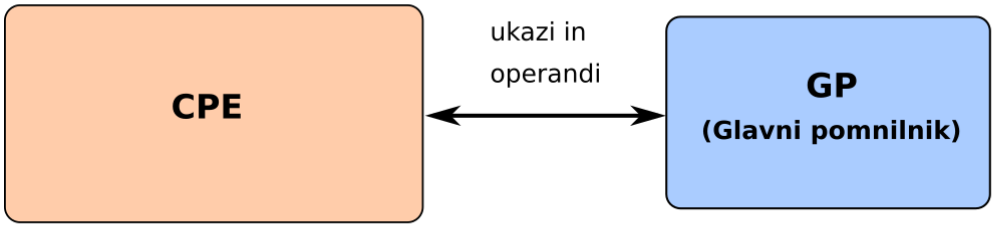


  - takrat se izvrši skok na prvi ukaz **prekinitvenega servisnega programa** (PSP)
    - pred tem se shrani vrednost PC

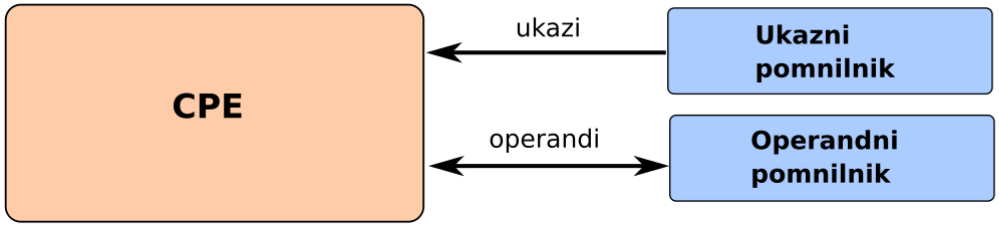
Glavni pomnilnik

- 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)

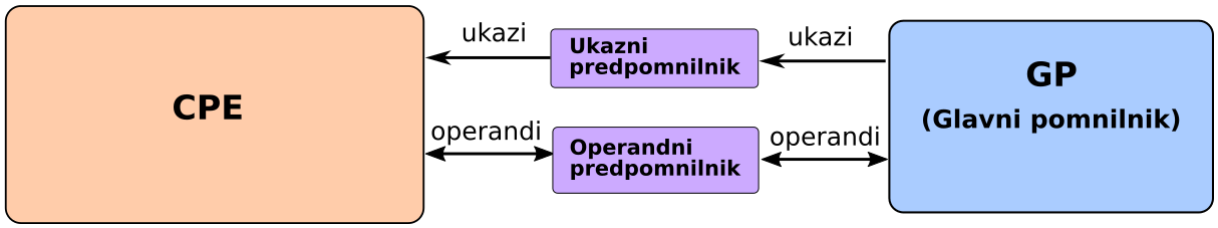
Princetonska arhitektura



Harvardska arhitektura



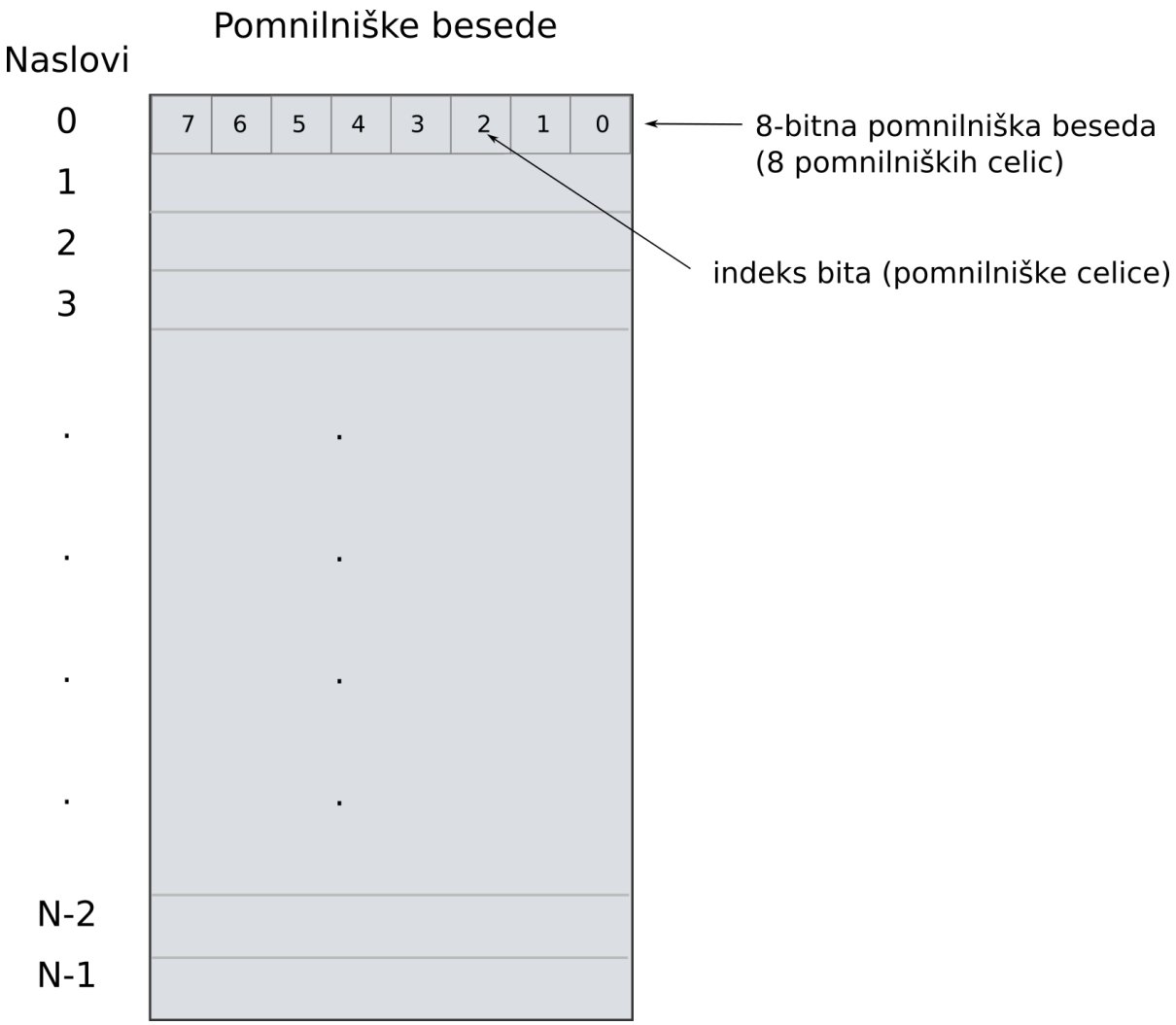
Danes prevladuje Princetonska arhitektura, vendar z ločenima *predpomnilnikoma* za ukaze in operande:



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
    - pomnilniška beseda je definirana 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:

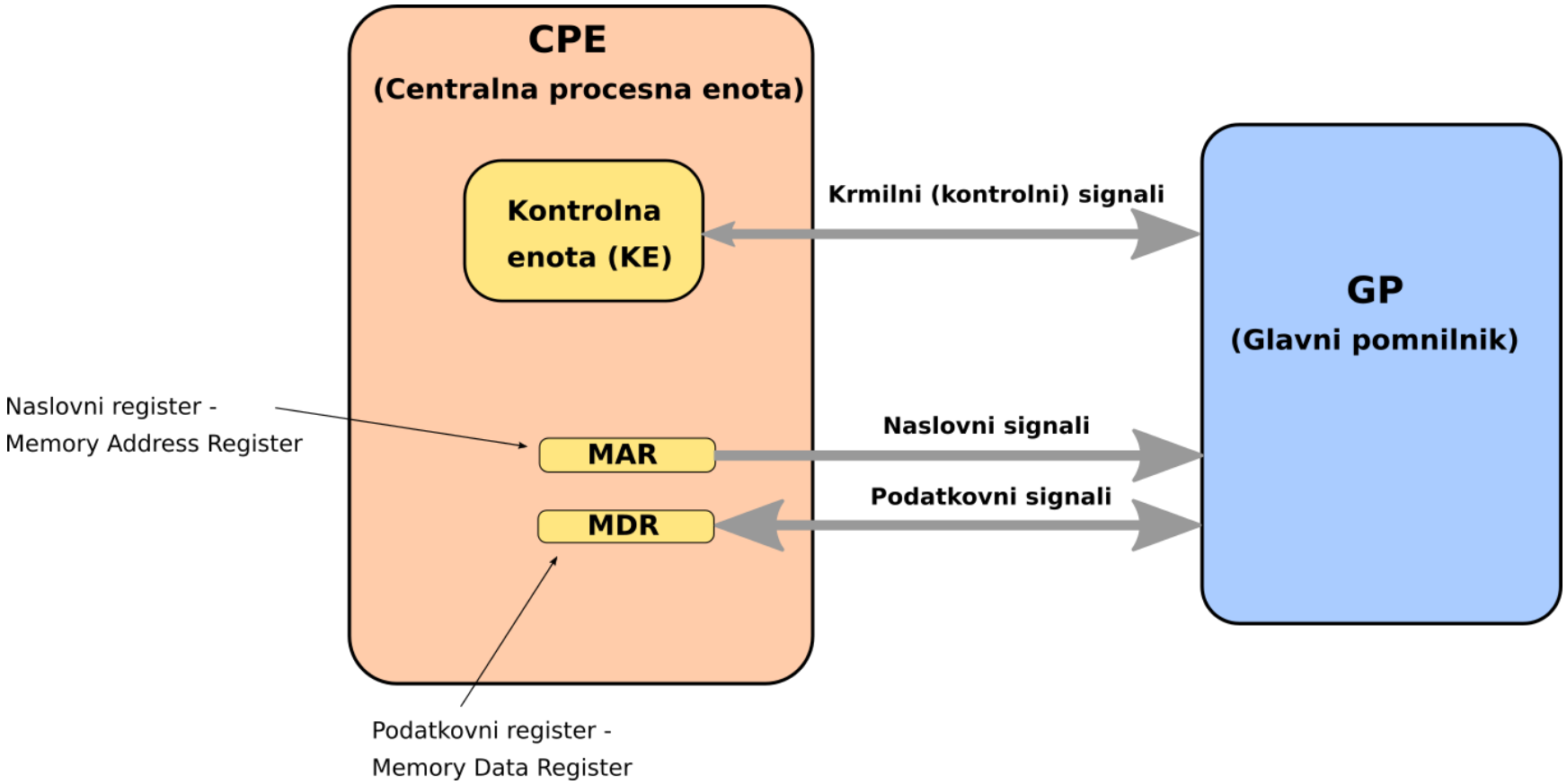


Naslovni prostor

- 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)
    - 2^{40} = 1T (tera)
- 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 ***krmilnimi (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
(Centralna procesna enota)
Kontrolna enota (KE)
Krmilni (kontrolni) signali
GP
(Glavni pomnilnik)
Naslovni register -
Memory Address Register
MAR
Naslovni signali
Podatkovni signali
MDR
Podatkovni register -
Memory Data Register

MAR in MDR

- 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

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

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

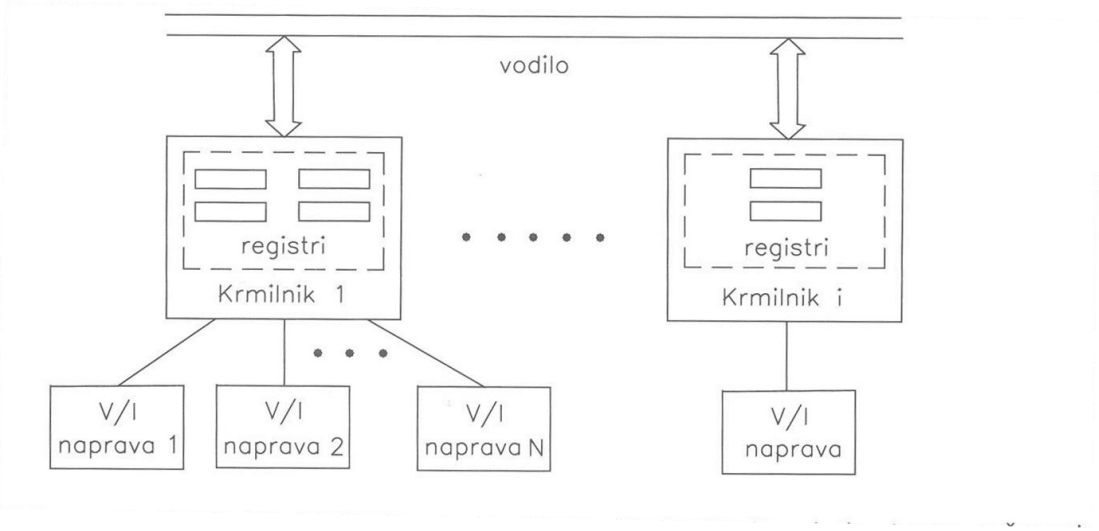
2 skupini izvedb V/I sistema :

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**

- 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

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



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
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

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)

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

2 mehanizma za prehod med nivojema:

- Prevajanje (prevajalnik)
  - izvorni program v enem jeziku
  - ciljni program (object program) v drugem (nižjem) jeziku
    - izvornega načelno ne rabimo več
- Interpretacija (interpreter)
  - izvorni program se prevaja sproti
    - ukaz se prevede in izvrši
    - rabimo ga ves čas
    - bolj fleksibilno
    - večja prenosljivost
    - manjša hitrost
- delno prevajanje
  - prevajanje v vmesno kodo, ki se jo interpretira
    - npr. Java

Strojna in programska oprema računalnika

- Delitev
  - hardware
  - software
  - firmware
    - program, ki je vgrajen v HW napravo (kot ROM ali bliskovni pomnilnik) in skrbi za njeno osnovno funkcionalnost
- Strojna in programska oprema sta funcionalno ekvivalentni
  - poljuben računalnik bi se načeloma dalo realizirati samo z elektroniko (dovolj kompleksno)

5a
RISC-V in ukazi load/store

BRANKO ŠTER

Prevajanje ukazov

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

- Primer 1 (iz jezika C v zbirni jezik):
  - Predpostavimo zaenkrat, da se vrednosti spremenljivk a, b in c že nahajajo v registrih x5, x6 in x7

```
a = b + c;              // v jeziku C
add x5, x6, x7          ; Pomen: x5 ← x6 + x7
```

- Primer 2:

```
a = b + c + d + e;      // x1: a, x2: b, x3: c, x4: d, x5: e
add x1, x2, x3          ; 3 ukazi
add x1, x1, x4          ;   v zbirnem
add x1, x1, x5          ;     jeziku
```

- Primer 3:

```
A[12] = h + A[8];          // x1: A(=naslov), x3: h

lw  x2, 32(x1)    ; x2 ← M[x1+32]      (32=8*4)
add x2, x2, x3    ; x2 ← x2 + x3
sw  x2, 48(x1)    ; M[x1+48]← x2       (48=12*4)
```

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

```
addi  x1, x2, 5          ; x1 ← x2 + 5
```

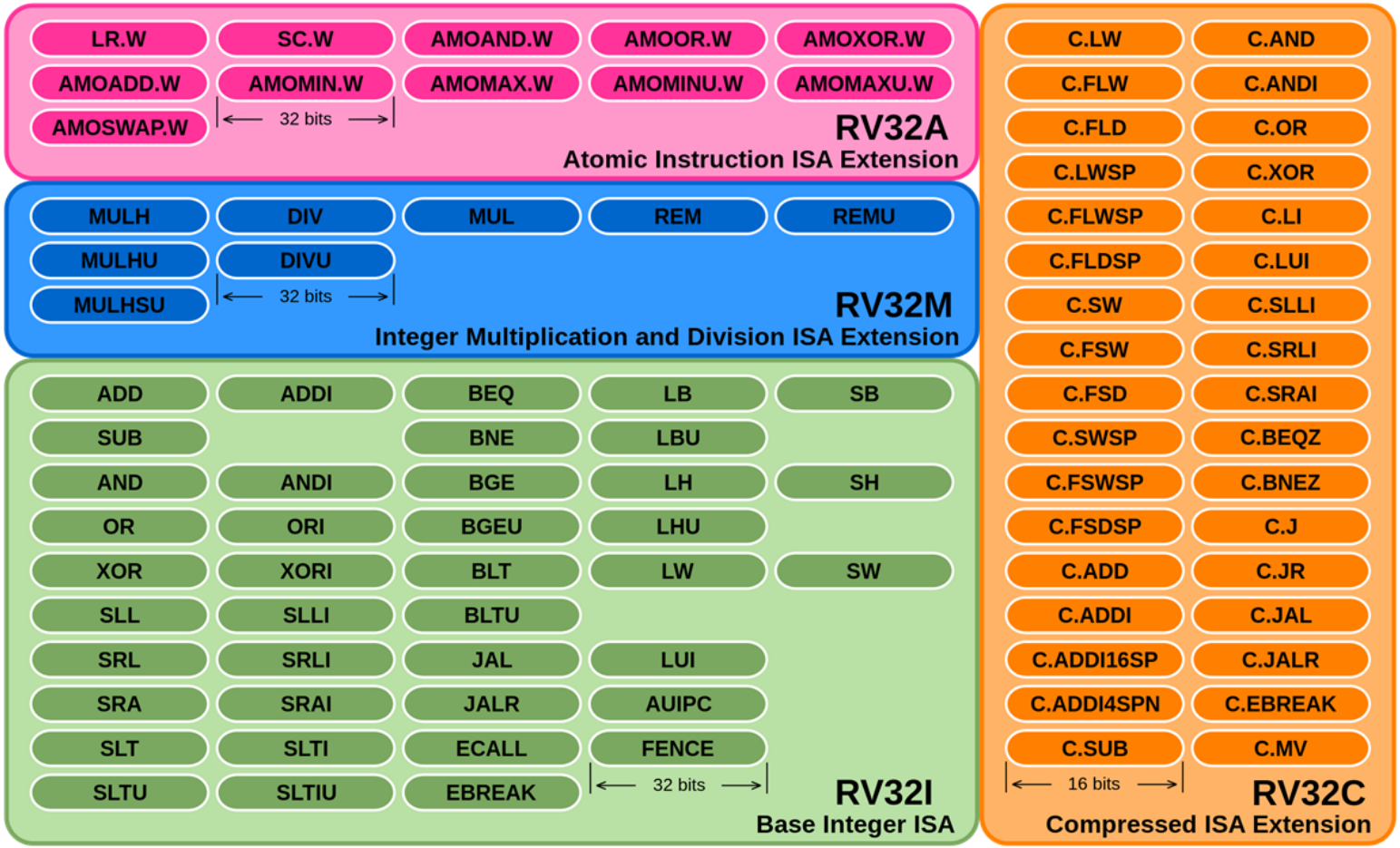
(add immediate)

Ukazna arhitektura (ISA)

- **Ukazna arhitektura** (**Instruction Set Architecture**, **ISA**)
  - natančno definira vse ukaze (nabor ukazov) nekega procesorja
  - ne govori pa o implementaciji
- RISC-V se vedno bolj uveljavlja kot odprta RISC arhitektura
  - Druge RISC arhitekture: ARM, MIPS, ...

- RISC-V je ukazna arhitektura (instruction-set architecture, ISA), ki je bila prvotno razvita za raziskave in poučevanje računalniških arhitektur
  - vse bolj pa postaja tudi standard na področju odprtih računalniških arhitektur za industrijske implementacije
  - RISC-V ISA je definirana brez detajlov implementacije
  - RISC-V je dejansko družina
    - **RV32I - celoštevilska 32-bitna** (XLEN=32)
    - RV64I - celoštevilska 64-bitna (XLEN=64)
    - RV128I - celoštevilska 128-bitna (XLEN=128)
    - RV32E - za majhne mikrokrmilnike (Embedded)
    - Razširitve (extensions):
      - M (množenje/deljenje),
      - F (plavajoča vejica, enojna natančnost),
      - D (plavajoča vejica, dvojna natančnost),
      - A (atomski ukazi),
      - ...

RV32IMAC



By Eduardo Corpeño - Own work using: Inkscape. This is the first illustration of a series on the RISC-V ISAs, which is available on risc-v-diagrams on GitHub, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=118039169

Lastnosti RISC-V

- 8-bitna pomnilniška beseda
- 32-bitni pomnilniški naslov
- Način shranjevanja operandov v CPE
  - 32 32-bitnih splošnonamenskih registrov x0, x1, …, x31
    - Vsebina x0 je vedno 0 (pri pisanju vanj se ne zgodi nič)
- Število eksplicitnih operandov v ukazu
  - vsi ALE ukazi imajo 3 eksplicitne operande
    - Tip R ima dva izvorna (source) registra *rs1* in *rs2* ter en ciljni destination register *rd*
    - Tip I ima en izvorni (source) register *rs1*, 12-bitni takojšnji (immediate) operand *imm* ter en ciljni destination register *rd*

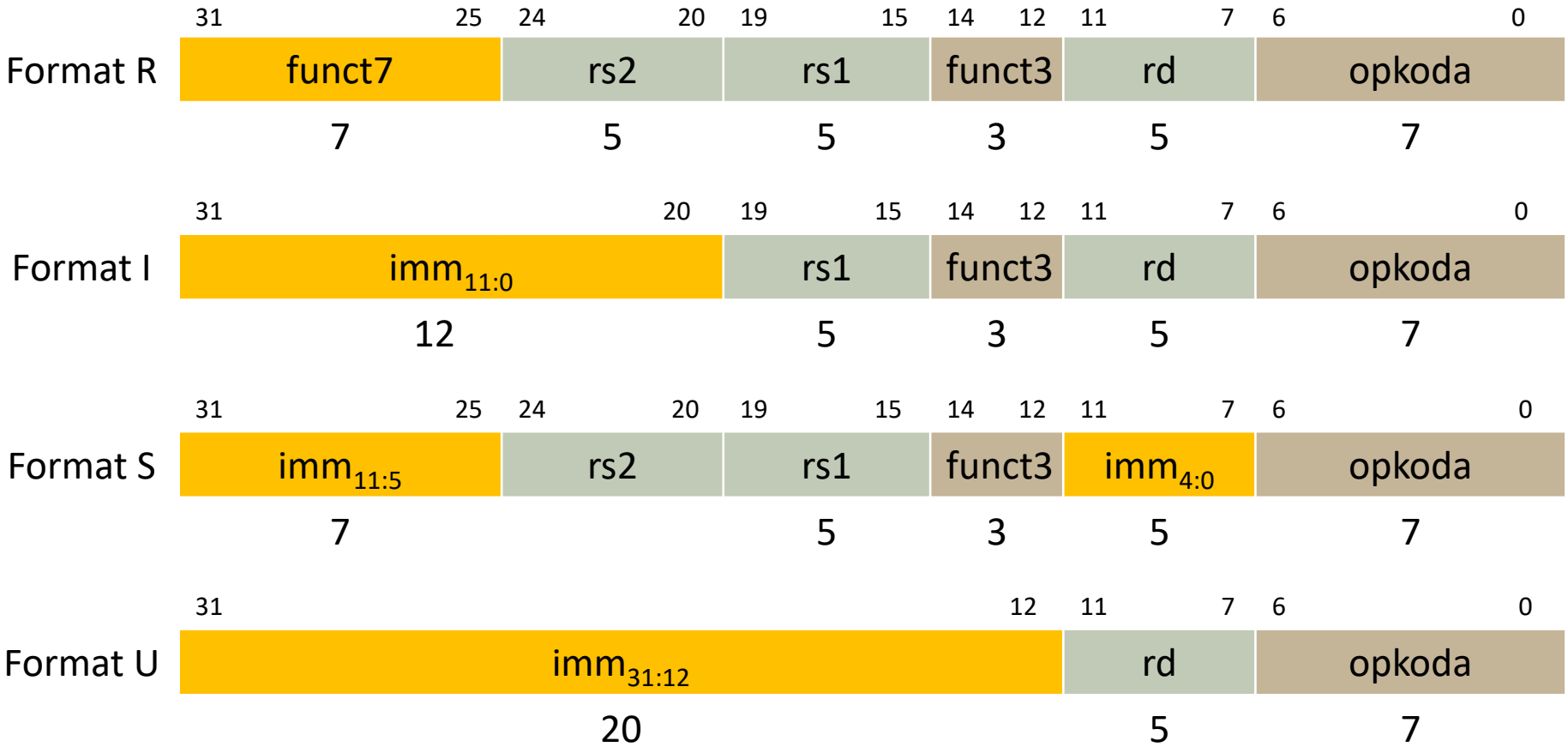
Lokacija operandov in načini naslavljanja

- 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

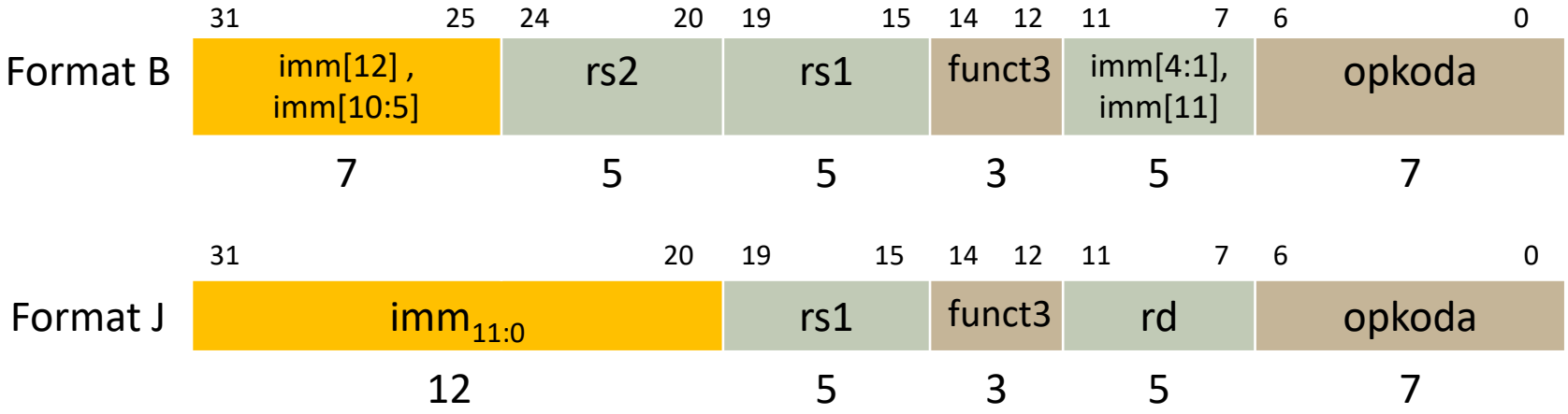
- Operacije in operandi
  - pomnilniški operandi so lahko 8-, 16- ali 32-bitni (bajt, polbeseda, beseda)
  - 16- in 32-bitni operandi so v pomnilniku shranjeni po **pravilu tankega konca** (**little endian rule**)
    - dobro je, da so **poravnani** (**aligned**), kar pomeni, da je operand, ki je sestavljen iz več bajtov, na naslovu, ki je deljiv s številom bajtov
      - 16-biten operand je na naslovu, deljivem z 2
      - 32-biten operand na naslovu, deljivem s 4
    - sicer je potreben prenos operanda v dveh kosih
  - 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)

Format ukazov pri RV32I:

- vsi ukazi so 32-bitni in poravnani
- 4 osnovni formati (R, I, S, U)



- 2 dodatna formata (B, J):



Format ukaza nam pove, kaj pomenijo posamezni biti v strojni kodi ukaza

- rs (source register): iz njega se bere,
- rd (destination register): vanj se piše

Število ukazov nabora RV32I:

- 40
- ni ukazov za množenje, deljenje
- ni ukazov v plavajoči vejici

Vrste ukazov

- Ukaze RISC-V 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

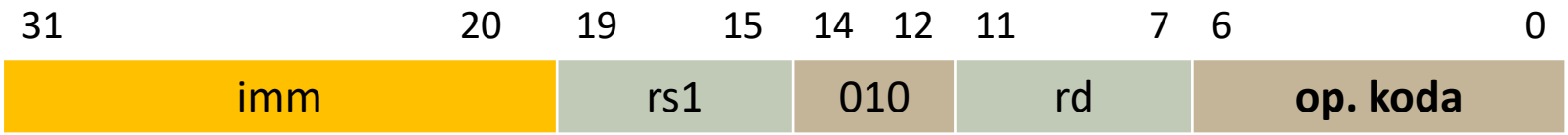
Ukazi za prenos podatkov (load/store)

- Uporabljajo format I z baznim naslavljanjem (bazni register je rs1)

Load word: `lw rd, imm(rs1)` ; rd $\leftarrow_{32}$ M[rs1 + se(imm)]

Npr.: `lw x5, 30(x6)` ; x5 $\leftarrow_{32}$ M[30 + x6]

Format I:



12 bitov	5 bitov	3 biti	5 bitov	7 bitov
	x6		x5	lw
000000011110	00110	010	00101	0000011
30	6	2	5	3

Celoten ukaz v strojni kodi: 0x01E32283

Torej: ukaz v zbirnem jeziku lw x5,30(x6) zbirnik 'prevede' v strojni ukaz 00000001111000110010001010000011

- 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

Razširitev operanda

Kaj, 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))
    - Npr.: 0x6F (01101111) se razširi v 0x0000006F (00000000 00000000 00000000 01101111),

      0x94 (10010100) se razširi v 0xFFFFFF94 (11111111 11111111 11111111 10010100)
  - lh (load halfword (signed))
    - Npr.: 0x73A1 (01110011 10100001) se razširi v 0x000073A1 (00000000 00000000 01110011 10100001),

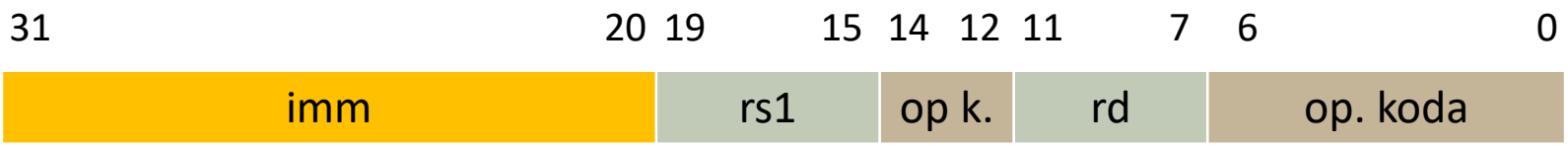
      0xC40A (11000100 00001010) se razširi v 0xFFFFC40A (11111111 11111111 11000100 00001010)
- razširitev ničle
  - lbu (load byte unsigned)
    - Npr., 0x94 (10010100) se razširi v 0x00000094 (00000000 00000000 00000000 10010100)
  - lhu (load halfword unsigned)
    - Npr., 0xC40A (11000100 00001010) se razširi v 0x0000C40A (00000000 00000000 11000100 00001010)

Load byte:

`lb x5,80(x6)` ; $x5_{31..8} \leftarrow_{raz} M[80 + x6]_7$, $x5_{7..0} \leftarrow_8 M[80 + x6]$

$x5_{31..8}$ so biti od 31 do 8 (najvišjih 24 bitov),

$\leftarrow_{raz}$ pomeni razširitev predznaka



12 bitov	5 bitov	3biti	5 bitov	7 bitov
	x6	lb*	x5	lb*
000001010000	00110	000	00101	0000011
80	6	0	5	3

** lb sta oba dela skupaj (3+7 bitov)*

Celoten ukaz v strojni kodi: 0x05010083

Load byte unsigned

```
lbu  x5,80(x6)
```

pomen: $x5_{31..8} \leftarrow_{raz} 0$, $x5_{7..0} \leftarrow_{8} M[80 + x6]$

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
I	000 0000011	LB	Load byte
I	001 0000011	LH	Load halfword
I	010 0000011	LW	Load word
I	100 0000011	LBU	Load byte unsigned
I	101 0000011	LHU	Load halfword unsigned
S	000 0100011	SB	Store byte
S	001 0100011	SH	Store halfword
S	010 0100011	SW	Store word

- Odmik je 12-biten z razširitvijo predznaka (torej je lahko tudi negativen)
- 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)

Osnovne direktive zbirnika

`.data`	– začetek podatkovnega segmenta
`.text`	– začetek ukaznega segmenta
`.byte <n1>(,<n2>..)`	– določi zaporedna 8-bitna števila
`.half <n1>(,<n2>..)`	– določi zaporedna 16-bitna števila
`.word <n1>(,<n2>..)`	– določi zaporedna 32-bitna števila
`.dword <n1>(,<n2>.. )`	– določi zaporedna 32-bitna števila
`.align <n>`	– poravnava

Direktive so namenjene zbirniku (programu), ne procesorju!

Primer programa v zbirnem jeziku za RISC-V

```
        .data   (podatkovni segment; vzemimo, da se začne na naslovu 0x400 = 1024_10)
A:      .byte 5         (bajt z vrednostjo 5 na naslovu A = 1024)
B:      .byte 6         (bajt z vrednostjo 6 na naslovu B = 1025)
C:      .byte 0         (bajt z vrednostjo 0 na naslovu C = 1026)

        .text                   (kodni segment; običajno bo kar na naslovu 0)
        lb x2, A(x0)            (x2 ← M[1024 + 0])
        lb x3, B(x0)            (x3 ← M[1025 + 0])
        add x4, x2, x3          (x4 ← x2 + x3)
        addi x5, x0, C          (x5 ← 1026)
        sb x4, 0(x5)            (x4 → M[1026+0])
```

Uporaba oznak

Namesto oznake (labele) A lahko pišemo tudi kar neposredno naslov 0x400 (vsaka labela vsebuje pomnilniški naslov – bodisi ukaza, bodisi operanda)

```
lb x5, 0x400(x0)
```

(Če je bazni register različen od 0, ga je prej seveda treba naložiti)

Toda, pozor pri operacijah store!

- Npr. **`sb x4,C(x0)`** ne deluje, kot bi pričakovali
- Tak ukaz Ripes tolmači kot psevdoukaz in pred njim doda ukaz auipc (to bomo obravnavali kasneje), ki baznemu registru da vrednost PC.
- Ker pa se x0 ne more spremeniti, ne dobi te vrednosti
- Rešitev je, da namesto x0 uporabimo katerikoli register drug register, za katerega nam ni pomembno, ali se njegova vrednost spremeni (služi le kot začasni register), npr. **`sb x4,C(x8)`**
- Deluje pa tudi **`sb x4,0x402(x0)`**, vendar pri 'ročnem' pisanju v zbirnem jeziku to ni prikladno, ker programmer običajno ne pozna naslova (razen v takem preprostem primeru, kot je naš)

5b
Aritmetični ukazi

BRANKO ŠTER

- Osnovni nabor ukazov RV32I

RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
fm | pred | succ		rs1	000	rd	0001111	FENCE
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK

Registri RISC-V

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5–7	t0–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

ABI

- Application binary interface
  - vmesnik med program v strojni kodi (tudi programi iz knjižnic in programi OS)
  - Določa način zapisa podatkovnih struktur in način klicanja podprogramov na nizkem nivoju
    - API določa podobne stvari v izvorni kodi
  - ABI določa:
    - strukturo registrov, organizacijo sklada, vrste pomnilniških dostopov
    - podatkovni tipi (velikost, poravnanost, endian)
    - način klica sistemskih klicev
    - dogovor o klicih podprogramov (Calling convention)

ALE ukazi

1. **aritmetične operacije** (+, –)
   - ADD, ADDI,
   - SUB
   - LUI, AUIPC
2. **logične bitne operacije** (&, V, ∇)
   - AND, ANDI
   - OR, ORI
   - XOR, XORI
3. **pomiki** (shift) (levi, desni; logični, aritmetični)
   - SLL, SLLI
   - SRL, SRLI
   - SRA, SRAI
4. **ukazi za primerjavo oz. set operacije** (pogoj: <)
   - SLT, SLTI, SLTU, SLTIU

Logične bitne operacije

- Logične bitne operacije delujejo po istoležnih bitih (bitwise operations):
  - IN (AND), &

```
  00110010
& 01010110
----------
  00010010
```

  - ALI (OR), V, |

```
  00110010
| 01010110
----------
  01110110
```

  - Ekskluzivni ALI (XOR), ∇, ^

```
  00110010
^ 01101001
----------
  01011011
```

  - NE (NOT) – tega RISCV sicer nima, ker se da to narediti z XOR z enicami

```
~00011011
----------
 11100100
```

Uporaba bitnih operacij

Bitne operacije se uporabljajo tudi za branje in vpisovanje posameznih bitov v besedo

- Nastavljanje bita:
  - Kako nastavimo nek bit na 1 (ostale pa pustimo pri miru):

```
    xxxxxxxx
or  00010000
    xxx1xxxx
```

- Brisanje bita:
  - Kako postavimo nek bit na 0 (ostale pa pustimo pri miru):

```
    xxxxxxxx
and 11101111
    xxx0xxxx
```

- Branje bita:
  - Kako samo pogledamo vrednost določenega bita:

```
    xxxxxxxx
and 00010000
    000x0000
```

Če je iskani bit 1, je dobljeni izraz od 0 različen, sicer je 0.

Pomiki

- Pomiki:
  - levi
    - SLL - Shift Left (Logical) in SLLI (SLL immediate)
      - 0110 $\rightarrow$ 1100
  - desni
    - **logični** (0110 $\rightarrow$ 0011)
      - v izpraznjena mesta gredo ničle
    - **aritmetični** (0110 $\rightarrow$ 0011, 1011 $\rightarrow$ 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
- 2 operanda sta v registrih
  - tretji je lahko v registru ali takojšnji (immediate)

rd ← rs1 op rs2
dd ← rs1 op Takojšnji operand (immediate)

ALE ukazi (1): aritmetične in logične operacije

Tip operacije	Ukaz	Opis	Format	Polja	Opkoda
Aritmetične	ADD	Add	R	0000000 rs2 rs1 000 rd	0110011
	SUB	Subtract	R	0100000 rs2 rs1 000 rd	0110011
	ADDI	Add imm.	I	imm12 rs1 000 rd	0010011
	LUI	Load upper imm.	U	imm20 rd	0110111
	AUIPC	Add upper imm. PC	U	imm20 rd	0010111
Tip operacije	**Ukaz**	**Opis**	**Format**	**funct7, funct3**	**opkoda**
Logične	AND	And	R	0000000 rs2 rs1 111 rd	0110011
	OR	Or	R	0000000 rs2 rs1 110 rd	0110011
	XOR	Exclusive or	R	0000000 rs2 rs1 100 rd	0110011
	ANDI	And imm.	I	imm12 rs1 111 rd	0010011
	ORI	Or imm.	I	imm12 rs1 110 rd	0010011
	XORI	Excl.-or imm.	I	imm12 rs1 100 rd	0010011

ALE ukazi (2): Pomiki

Tip opera-cije	Ukaz	Opis	Format	Polja	Opkoda
shift	SLL	Shift left logical	R	0000000 rs2 rs1 001 rd	0110011
	SRL	Shift right logical	R	0000000 rs2 rs1 101 rd	0110011
	SRA	Shift right arithmetic	R	0100000 rs2 rs1 101 rd	0110011
	SLLI	Shift left logical imm.	I	0000000 shamt* rs1 001 rd	0010011
	SRLI	Shift right logical immediate	I	0000000 shamt* rs1 101 rd	0110011
	SRAI	Shift right arithmetic imm.	I	0000000 shamt* rs1 101 rd	0110011

*shamt ... shift amount

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

ALE ukazi (3): Ukazi za primerjavo

Tip operacije	Ukaz	Opis	Format	Polja	Opkoda
set	SLT	Set if less than	R	0000000 rs2 rs1 010 rd	0110011
	SLTU	Set if less than unsigned	R	0000000 rs2 rs1 011 rd	0110011
	SLTI	Set if less than immediate	I	imm12 rs1 010 rd	0010011
	SLTUI	Set if less than unsig. imm.	I	imm12 rs1 011 rd	0010011

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

ADD:

```
add  x3, x5, x6          ; x3 ← x5 + x6
```

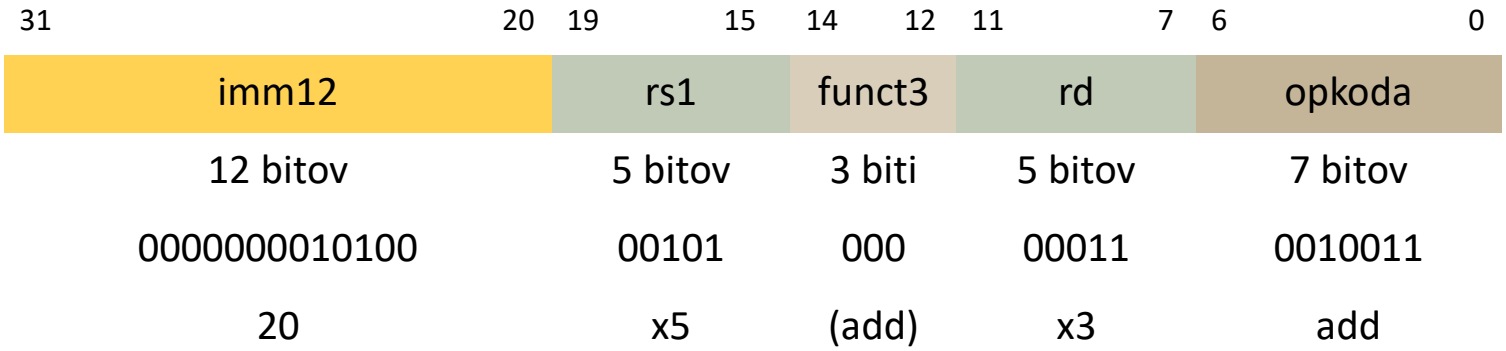
Format R:

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opkoda
7 bitov	5 bitov	5 bitov	3 biti	5 bitov	7 bitov
0000000	00110	00101	000	00011	0110011
(add)	x6	x5	(add)	x3	add

ADDI (Add immediate)

addi x3, x5, 20 ; x3 ← x5 + 20

Format I:

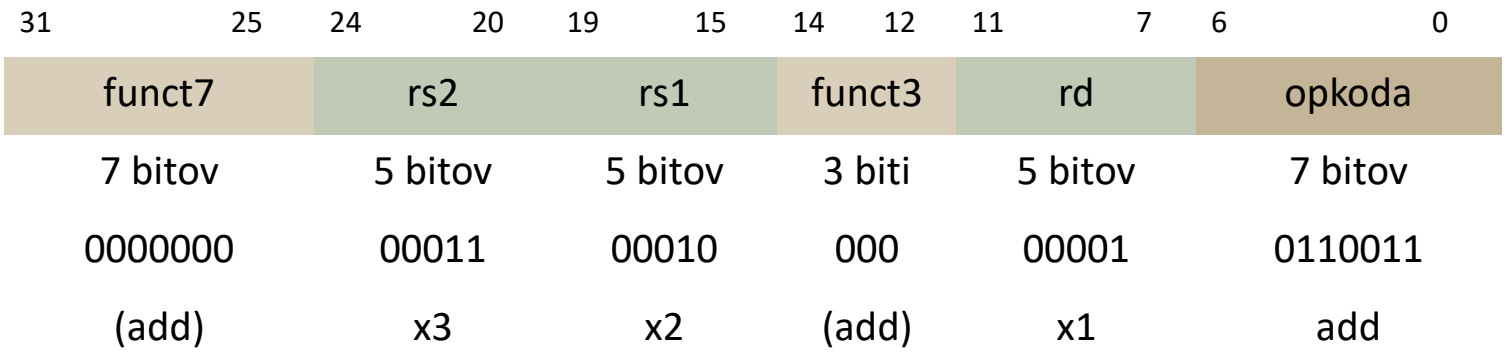


- Takojšnjemu (12-bitnemu) operandu se razširi predznak (na 32 bitov).
- A pozor: če pišemo v šestnajstiškem zapisu, ne sme imeti na začetku enice (zbirnik javi napako)!
  - Če želimo, da je negativen, moramo dodati predznak (npr. -0x800)
  - V desetiškem zapisu ima negativno število itak predznak (npr. imm. -1 se razširi na same enice v registru)

AND

and x1, x2, x3 ; x1 ← x2 & x3

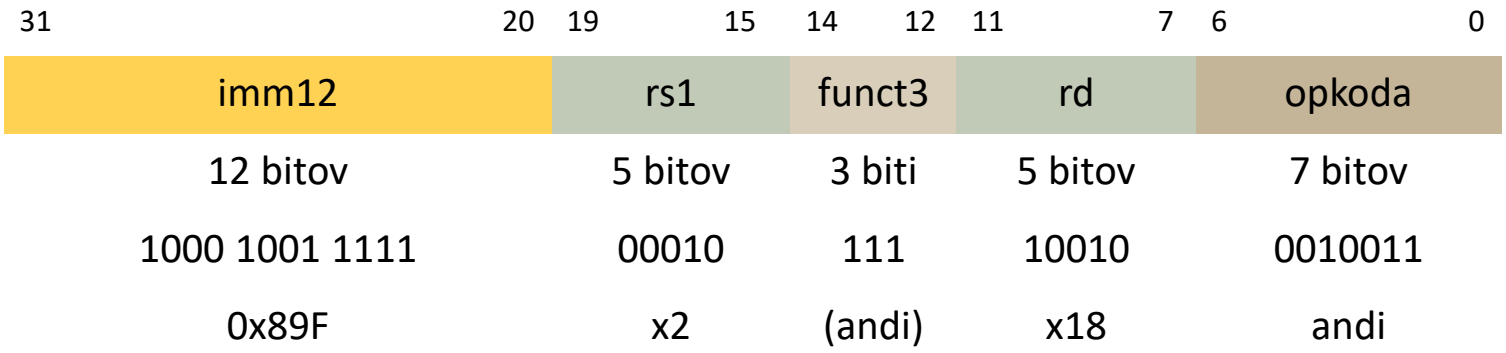
Format R:



ANDI (and immediate)

andi x18, x2, 0x49F ; x18 ← x2 & 0x49F

Format I:

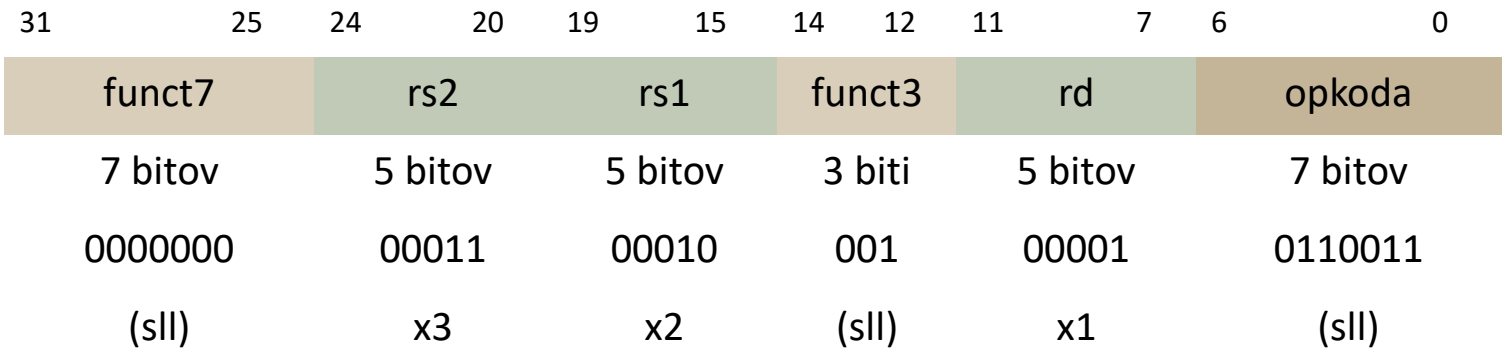


- Takojšnjemu (12-bitnemu) operandu se razširi predznak (na 32 bitov).
- A pozor: če pišemo v šestnajstiškem zapisu, ne sme imeti na začetku enice (zbirnik javi napako)!
  - Če želimo, da je negativen, moramo dodati predznak

SLL (shift left logical)

sll x1, x2, x3 ; $x1 \leftarrow x2 << x3$ (oz. $r2 \times 2^{r3}$)

Format R:

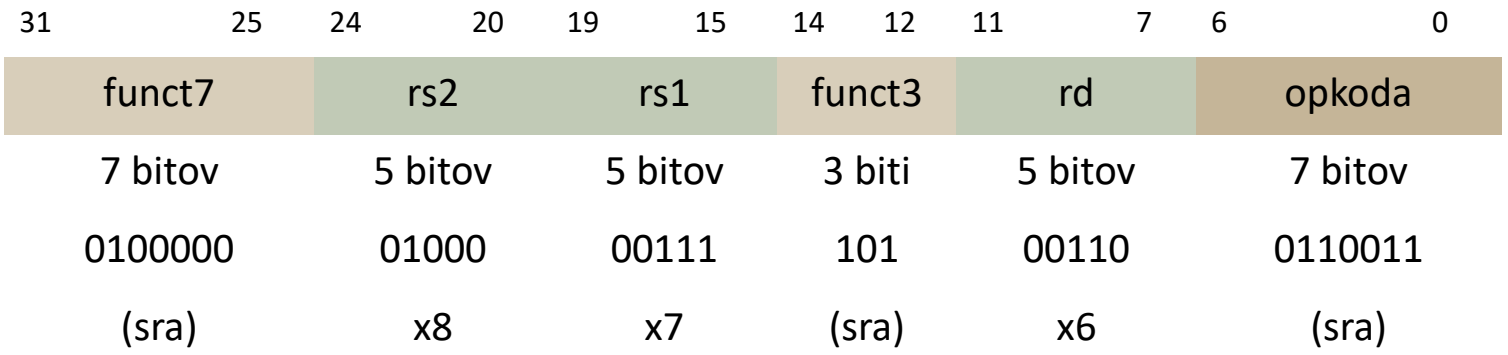


SRA (shift right arithmetic)

sra x6, x7, x8 ; x6 ← x7 >> x8

; $x6_{31} \leftarrow x7_{31}$

Format R:



LUI (Load upper immediate)

- poseben ukaz, ki 20-bitno (konstantno) vrednost naloži v gornjih 20 bitov registra, spodnjih 12 bitov pa je 0
- Zakaj sploh potrebujemo tak ukaz?
  - Problem je, kako naložiti 32-bitno konstanto v register
  - z enim 32-bitnim ukazom ni možno
  - zato to lahko storimo v 2 korakih:
    1. naložimo zgornjih 20 bitov
    2. naložimo spodnjih 12 bitov

  - Npr.: 0x12345678

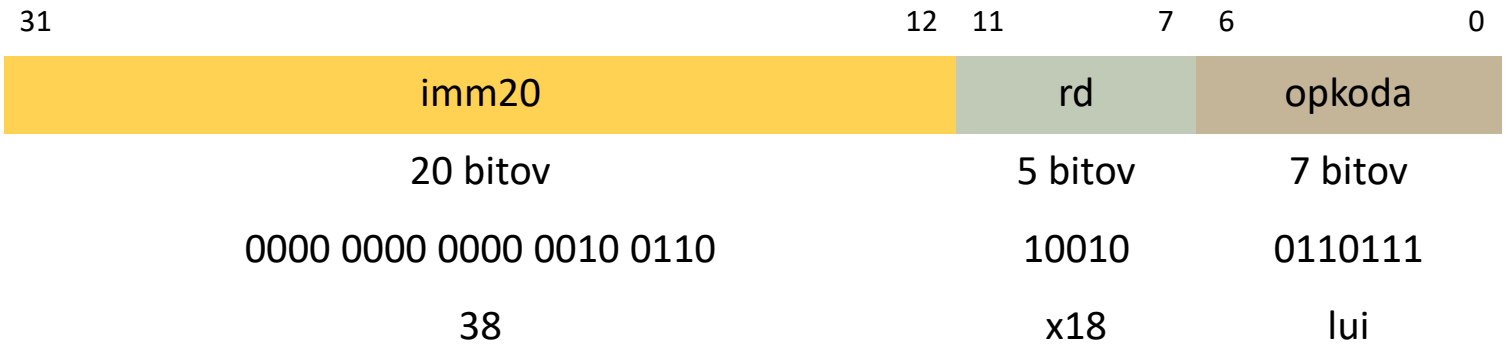
```
lui   x5, 0x12345
addui   x5, x5, 0x678
```

(lahko tudi z `ori`)

LUI (load upper immediate)

lui x18, 38 # $x18_{31..12} \leftarrow 38,\ x18_{11..0} \leftarrow 0$

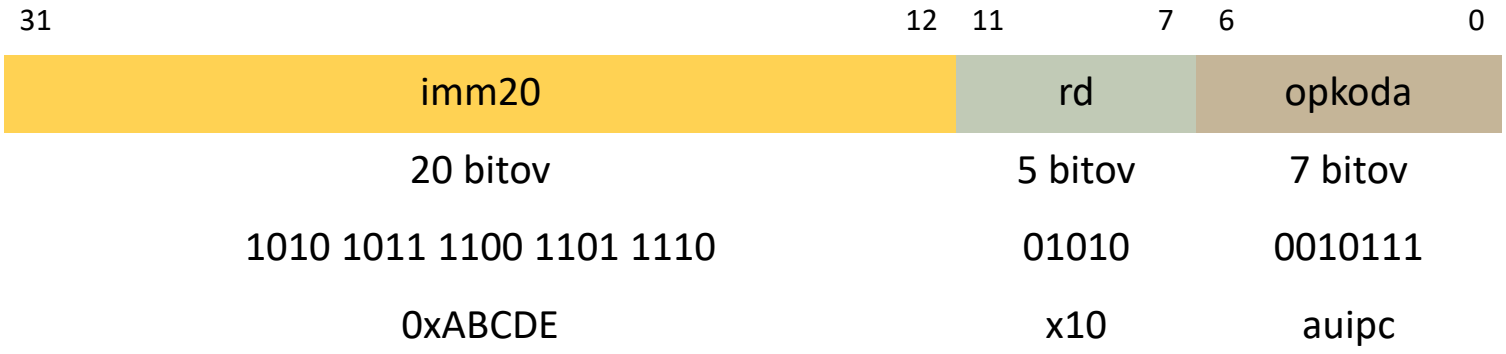
Format U:



AUIPC (add upper immediate to PC)

```
auipc x10, 0xABCDE    # x10 ← (0xABCDE << 12) + PC
```

Format U:



- Tudi to je poseben ukaz, ki 20-bitno (konstantno) vrednost naloži v gornjih 20 bitov registra (spodnjih 12 bitov je 0), temu pa prišteje vrednost programskega števca PC
- Ta ukaz omogoča PC-relativno naslavljanje
  - Tako se da celoten program linearno premakniti v drug del pomnilnika

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

```
        .data           # (na naslovu 0x400)
A:      .word 5
B:      .word 0

        .text
        lw x1, A(x0)
        slli x2, x1, 3
        slli x3, x1, 1
        add x4, x2, x3
        sw x4, B(x5)  # x5 je začasni register
```

Set-ukazi (oz. ukazi za primerjavo)

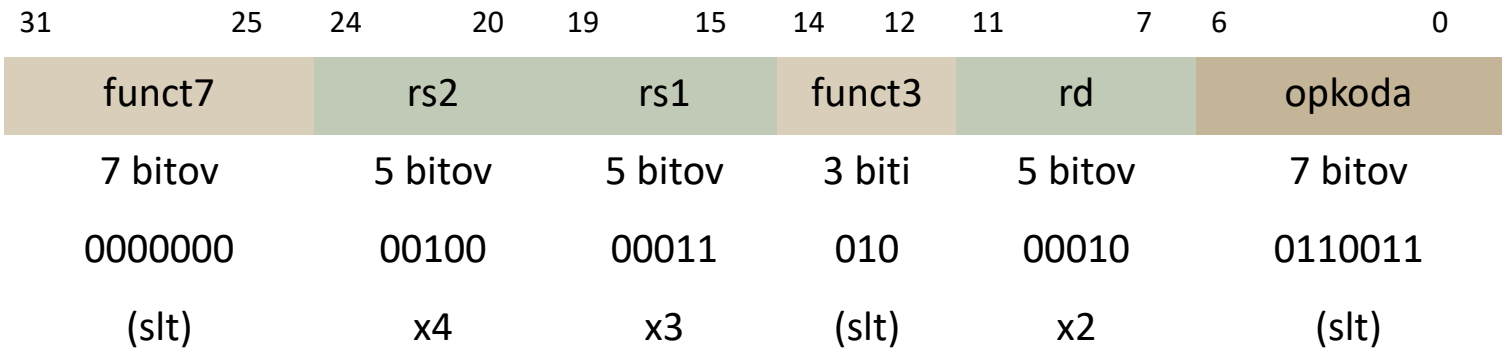
Če je podani pogoj izpolnjen, postavijo v ciljni register 1 (...0001), sicer 0 (...0000)

- SLT (Set if Less Than), format R
  - slt rd, rs1, rs2 ; rd ← (rs1 < rs2) ? 1 : 0
- SLTI (Set if Less Than Immediate), format I
  - slti rd, rs1, imm ; rd ← (rs1 < imm i) ? 1 : 0
- SLTIU (Set if Less Than Immediate Unsigned), format I
  - sltiu rd, rs1, imm ; rd ← (rs1 < imm i) ? 1 : 0
- SLTU (Set if Less Than Unsigned), format R
  - sltu rd, rs1, rs2 ; rd ← (rs1 < rs2) ? 1 : 0

SLT (set if less than)

slt x2, x3, x4 ; x2 ← (x3 < x4)

Format R:



Psevdo-ukazi

- Poleg direktiv obstajajo tudi psevdo-ukazi, ki niso dejanski ukazi procesorja, ampak so namenjeni zbirniku, ki jih prevede v dejanske ukaze
- Primeri:

nop	(addi x0, x0, 0)	no operation
la rd, sym	(auipc+addi)	nalaganje naslova (load address)
li rd, imm		nalaganje konstante (load imm.)
mv rd, rs	(addi rd, rs, 0)	vsebina rs se kopira v rd
not rd, rs	(xori rd, rs, -1)	eniški komplement
neg rd, rs	(sub rd, x0, rs)	dvojiški komplement
seqz rd, rs	(sltiu rd, rs, 1)	set if equal (to) zero
snez rd, rs	(sltu rd, x0, rs)	set if not equal (to) zero
sltz rd, rs	(slt rd, rs, x0)	set if less than zero
sgtz rd, rs	(slt rd, x0, rs)	set if greater than zero
...		

Pseudoinstruction	Base Instruction(s)	Meaning
`la rd, symbol`	`auipc rd, symbol[31:12]` `addi rd, rd, symbol[11:0]`	Load address
`l{b|h|w|d} rd, symbol`	`auipc rd, symbol[31:12]` `l{b|h|w|d} rd, symbol[11:0](rd)`	Load global
`s{b|h|w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `s{b|h|w|d} rd, symbol[11:0](rt)`	Store global
`fl{w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `fl{w|d} rd, symbol[11:0](rt)`	Floating-point load global
`fs{w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `fs{w|d} rd, symbol[11:0](rt)`	Floating-point store global
`nop`	`addi x0, x0, 0`	No operation
`li rd, immediate`	*Myriad sequences*	Load immediate
`mv rd, rs`	`addi rd, rs, 0`	Copy register
`not rd, rs`	`xori rd, rs, -1`	One's complement
`neg rd, rs`	`sub rd, x0, rs`	Two's complement
`negw rd, rs`	`subw rd, x0, rs`	Two's complement word
`sext.w rd, rs`	`addiw rd, rs, 0`	Sign extend word
`seqz rd, rs`	`sltiu rd, rs, 1`	Set if = zero
`snez rd, rs`	`sltu rd, x0, rs`	Set if ≠ zero
`sltz rd, rs`	`slt rd, rs, x0`	Set if < zero
`sgtz rd, rs`	`slt rd, x0, rs`	Set if > zero
`beqz rs, offset`	`beq rs, x0, offset`	Branch if = zero
`bnez rs, offset`	`bne rs, x0, offset`	Branch if ≠ zero
`blez rs, offset`	`bge x0, rs, offset`	Branch if ≤ zero
`bgez rs, offset`	`bge rs, x0, offset`	Branch if ≥ zero
`bltz rs, offset`	`blt rs, x0, offset`	Branch if < zero
`bgtz rs, offset`	`blt x0, rs, offset`	Branch if > zero
`bgt rs, rt, offset`	`blt rt, rs, offset`	Branch if >
`ble rs, rt, offset`	`bge rt, rs, offset`	Branch if ≤
`bgtu rs, rt, offset`	`bltu rt, rs, offset`	Branch if >, unsigned
`bleu rs, rt, offset`	`bgeu rt, rs, offset`	Branch if ≤, unsigned
`j offset`	`jal x0, offset`	Jump
`jal offset`	`jal x1, offset`	Jump and link
`jr rs`	`jalr x0, rs, 0`	Jump register
`jalr rs`	`jalr x1, rs, 0`	Jump and link register
`ret`	`jalr x0, x1, 0`	Return from subroutine
`call offset`	`auipc x6, offset[31:12]` `jalr x1, x6, offset[11:0]`	Call far-away subroutine
`tail offset`	`auipc x6, offset[31:12]` `jalr x0, x6, offset[11:0]`	Tail call far-away subroutine

Psevdoukaz la je koristen za nalaganje naslova

- Če je naslov nizek, lahko naložimo vrednost v register takole:
  - lw rd, offset(rs), ali
  - lw rd, label(rs)

- Ne moremo pa naložiti naslova, večjega od 2^{11}-1 (2047)
- Poljuben naslov lahko naložimo v register s psevdoukazom la:

```
la rd, symbol
```

  - prevede se v:

```
auipc rd, symbol[31:12]
addi rd, rd, symbol[11:0]
```

Isti program za poljubne naslove:

```
        .data           # (na visokem naslovu, npr. 0x10000400)
A:      .word 5
B:      .word 0

        .text
        la x10, A
        lw x1, 0(x10)
        slli x2, x1, 3
        slli x3, x1, 1
        add x4, x2, x3
        la x10, B
        sw x4, 0(x10)
```

5c
Kontrolni ukazi

Kontrolni ukazi

- Kontrolni ukazi omogočajo spremembo vrstnega reda izvajanja ukazov
  - takim ukazom rečemo skoki
  - Zmožnost odločitev razlikuje računalnik od kalkulatorja!
- 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
  - seveda pa tudi klice podprogramov ter poljubne skoke

- Skoki pri RISC-V:
  - Brezpogojni skok:
    - JAL (jump and link) – format J
    - JALR (jump and link register) – format I
  - Pogojni skoki (format B):
    - BEQ (branch if equal to), če rs1==rs2
    - BNE (branch if not equal zero), če rs1!=rs2
    - BLT, BLTU (branch if less than (unsigned)), če rs1 < rs2
    - BGE, BGEU (branch if greater or equal (unsigned)), če rs1 >= rs2
  - Pogojni skoki uporabljajo format B in *PC-relativno naslavljanje*
    - za bazni register je uporabljen PC

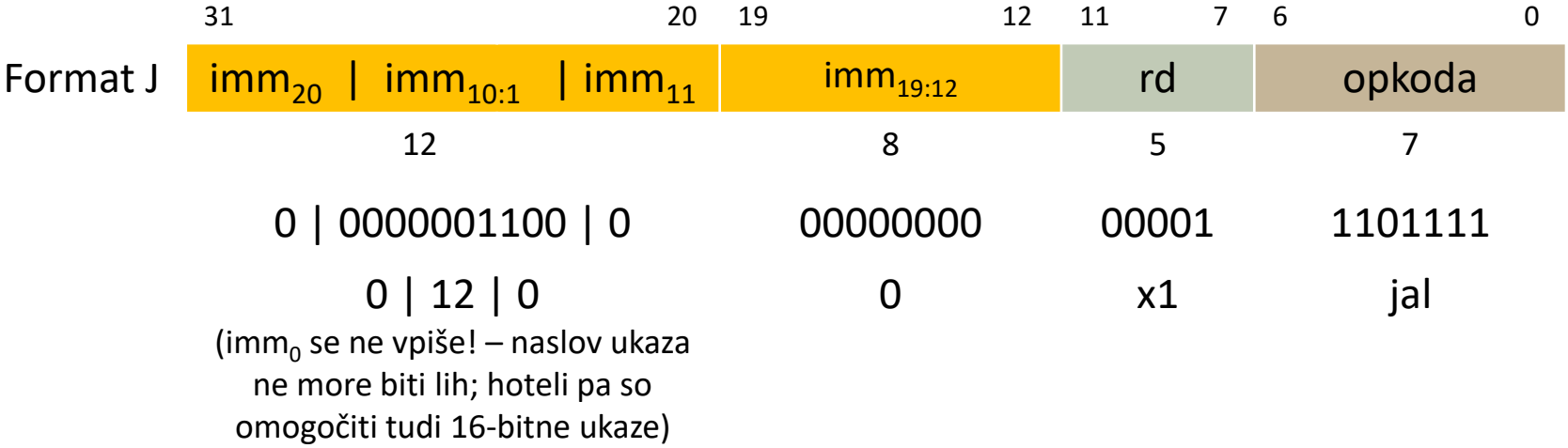
JAL (Jump and link):

jal rd, target # rd ← PC+4,
PC ← target = PC + se(2*imm20)

Zbirnik gornji ukaz prevede v `jal rd,imm20`

Pazi: target in imm ni ista stvar!

```
Npr.:  0x10          jal  x1, nekam   # x1 ← PC+4 (= 0x14 =20)
                      . . .           # PC ← nekam (= 0x28)
                      . . .           # imm20 = (0x28-0x10)/2 =
       0x28  nekam:   . . .           #       = (40-16)/2 = 12
```



- Če pišemo takojšnji operand:

```
jal rd, imm21 ,
```

je imm21 dejanska razlika, imm20 pa je imm21/2

  - Npr., jal x5, 20 skoči za 5 ukazov naprej,
    - imm21 = 20, imm20 = 10

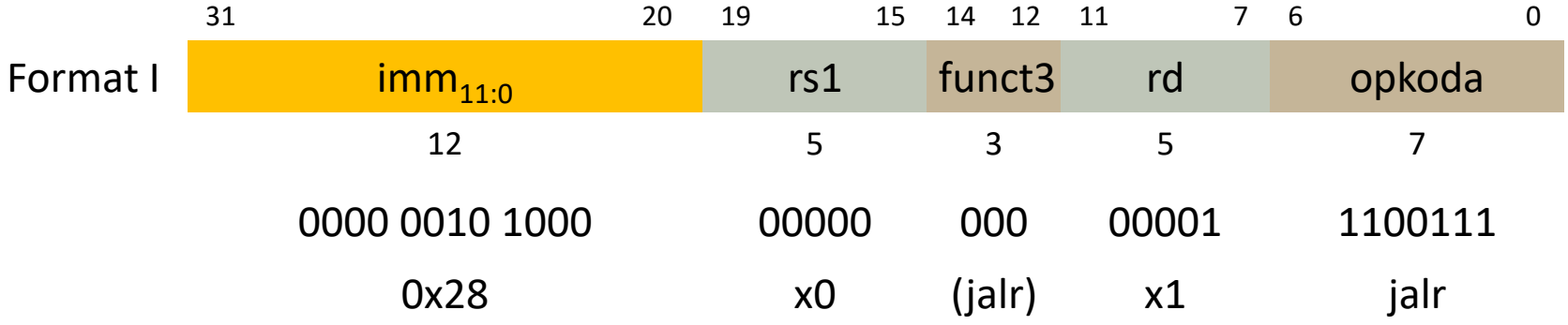
JALR (Jump and link register):

jalr rd, imm12(rs1)

rd ← PC+4,
PC ← target
= (rs1 + se(imm12)) & (-2)
zadnji bit je 0 (-2 = ..11110 = ~1)
(~ je 1'K)

rs1 omogoča skoke na zelo oddaljene procedure (naloži se ga predhodno z lui)

```
Npr.: 0x10          jalr  x1, nekam(x0)   # x1 ← PC+4 (= 0x14 =20)
                    . . .                 # PC ← nekam + x0 (= 0x28)
                    . . .                 #          imm12 = (0x28-0) = 40
      0x28  nekam:  . . .
```



Opomba: v simulatorju Ripes se ukaz JALR piše malo drugače!

jalr rd, rs, imm12

- Ukaz JAL lahko uporabimo tudi kot brezpogojni skok brez shranjevanja v rd (torej ‘linka’):
  - `jal x0, Oznaka`, kar dela psevdoukaz `j Oznaka`
- Ukaz JALR je koristen tudi kot ‘indirektni skok’
  - skočni naslov se da spreminjati s spreminjanjem vsebine registra
  - uporabno npr. pri stavku switch v jeziku C

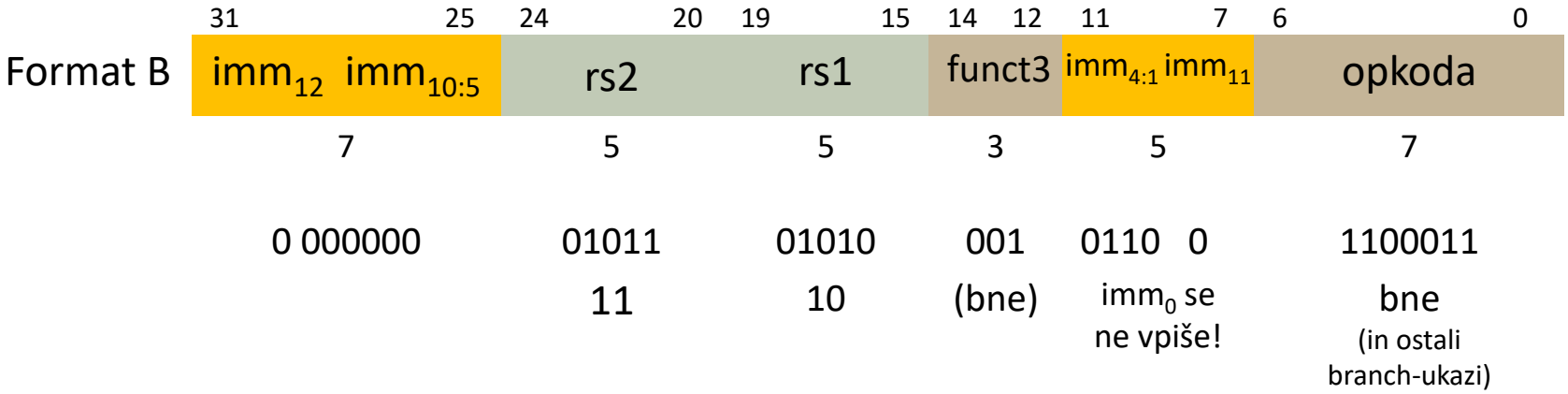
BNE (Branch if Not Equal to):

`bne rs1,rs2,imm13` (PC-relativno)

Pozor: imm_0 se ne vpiše v strojni ukaz
imm12 = $imm_{12:1}$ = (ciljni naslov – PC (ukaza bne))/2
Npr., imm12 = 10 pomeni imm13=20 in skoči za 5 ukazov naprej
Če v ukazu zapišemo oznako (labelo), zbirnik izračuna imm12 ← (label – PC)/2

Npr.:

```
        bne  x10, x11, 20          # če x10 != x11, potem PC ← PC + 12,
        . . .                      #       sicer PC ← PC + 4
        . . .
lab1:   . . .
```



Vejitve

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

- Če pogoj ni izpolnjen, je treba skočiti na blok2

- Ukaz beq izvaja pogojni skok

```
beq rs1,rs2,LABEL
```

  - Če rs1 == rs2, CPE skoči na naslov LABEL
- Podoben ukaz je

```
bne rs1,rs2,LABEL
```

  - Če rs1 != rs2, CPE skoči na naslov LABEL

- Primer:

```
if (c < 5)
        a = b + 1;
else
        a = 2;
```

Predpostavimo x1: c, x2: a, x3: b

- Več možnosti:

```
        addi  t0, x0, 5     # t0 = 5
        bge   x1, t0, Else  # if (c >= 5) goto Else;
        addi  x2, x3, 1     # a = b + 1;
        jal    x0, Ven      # goto Ven;
Else:   addi  x2, x0, 2     # a = 2;
Ven:    naslednji ukaz      # naslednji ukaz
```

```
        slti  t0, x1, 5     # t0 = (c < 5)?
        beq   t0, x0, Blk2  # if (t0==0) goto Else;
        addi  x2, x3, 1     # a = b + 1;
        jal   x0, Ven       # goto Ven;
Else:   addi  x2, x0, 2     # a = 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 RISC-V

`sum = 0;`
`i = 5;`
`while ( i > 2)`
`{`
`  sum = sum + i;`
`  i--;`
`}`
	`addi  x1, x0, 0  # sum`
	`addi  x2, x0, 2  # 2`
	`addi  x3, x0, 5  # i`
	`Loop:  bge  x2, x3, Ven  # if(2>=i) Ven`
	`add  x1, x1, x3`
	`addi  x3, x3, -1`
	`jal x0, Loop`
	`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  x1, x0, 0  # sum`
	`addi  x2, x0, 2  # 2`
	`addi  x3, x0, 5  # i`
	`Loop:  add  x1, x1, x3`
	`addi  x3, x3, -1`
	`blt  x2, x3, 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

```
                //int a[10] = {5, 5, 5, 8, 2};
                //int ax = 5;
                //int i = 0;
                while (a[i] == ax)
                   i++;

                (x1: i, x2: ax, x3: bazni naslov a, tj. naslov od a[0], &a[0])

Loop:           slli    x4, x1, 2       # 4*i
                add     x4, x4, x3      # a + 4*i
                lw      x5, 0(x4)       # v x4 je naslov a[i]
                bne     x5,x2,Exit      # (a[i]==ax)? Exit
                addi    x1, x1, 1
                jal     x0, Loop
Exit:           ...
```

- Nekateri procesorji imajo samo pogojne skoke tipa ‘branch if equal zero’ in ‘branch if not equal zero’ (npr. MIPS) – za druge primerjave je potrebno nastaviti nek register (npr. s SLT) na 1 oz. 0 in potem izvesti pogojni skok – a pri tem sta potrebna 2 ukaza
- Drugi (npr. ARM) imajo zastavice, ki povedo, ali je bil rezultat neke operacije Z (zero) ali N (negative), tudi, ali je prišlo do preliva (V - overflow). Pogojni skok nato pogleda vrednost teh zastavic.
  - To pa vnaša podatkovne odvisnosti, kar ni dobro za realizacijo cevovoda.

Primeri kontrolnih ukazov

	Primer ukaza	Ime ukaza	Opis
JAL	x9, 84(x8)	Jump and link	x9 ← PC + 4, PC ← x8 + 84 (če je rd==x0, je jal navaden brezpogojni skok)
JALR	x2, 84(x8)	Jump and link register	x9 ← PC + 4, PC ← x8 + 84
BEQ	x7, x8, 0x8C	Branch if EQual to	če x7 == x8, potem PC ← PC + 0x8C, sicer PC ← PC + 4
BNE	x7, x8, 0x8C	Branch if Not Equal to	če x7 != x8, potem PC ← PC + 0x8C, sicer PC ← PC + 4
BLT	x5, x6, 24	Branch if Less Than	če x5 < x6, potem PC ← PC + 24, sicer PC ← PC + 4
BGE	x5, x6, 24	Branch if Greater or Equal than	če x5 >= x6, potem PC ← PC + 24, sicer PC ← PC + 4
BLTU	x5, x6, 24	Branch if Less Than, Unsigned	če x5 < x6 **(nepredznačeno)**, potem PC ← PC + 24, sicer PC ← PC + 4
BGEU	x5, x6, 24	Branch if Greater or Equal than, Unsigned	če x5 >= x6 **(nepredznačeno)**, potem PC ← PC + 24, sicer PC ← PC + 4

Sistemski ukazi

CSR – Control and Status Register

- 12-bitni takojšnji operand določa enega od možnih 4096 registrov CSR

Format	Opkoda	funct3	Ukaz		Opis (ze ... zero extended)
I	1110011	001	CSRRW	Atomic Read/Write CSR	rd ← ze(CSR), razen če rd==x0. CSR ← rs1
I	1110011	010	CSRRS	Atomic Read and Set bits in CSR	rd ← ze(CSR). Biti CSR, ki imajo v rs1 (=maska) 1, se postavijo na 1, razen če rs1==x0
I	1110011	011	CSRRC	Atomic Read and Clear bits in CSR	rd ← ze(CSR). Biti CSR, ki imajo v rs1 (=maska) 1, se brišejo (0), razen če rs1==x0
I	1110011	101	CSRRWI	CSRRW immediate	rd ← ze(CSR), razen če rd==x0. CSR ← ze($uimm_{4:0}$) (v polju rs1)
I	1110011	110	CSRRSI	CSRRS imm.	rd ← ze(CSR). Biti CSR, ki imajo v rs1 (=maska) 1, se postavijo na 1, razen če $uimm_{4:0}$==0
I	1110011	111	CSRRCI	CSRRC imm.	rd ← ze(CSR). Biti CSR, ki imajo v rs1 (=maska) 1, se brišejo (0), razen če $uimm_{4:0}$==0

- Sistemski ukazi shranjujejo podani register CSR v podani splošnonamenski register rd, v CSR pa naložijo novo vrednost (nove bite)
  - pogosto pa ne potrebujemo obeh ‘storitev’, ampak le eno
- Psevdoukazi za enostavnejše primere:

Psevdoukaz	Ukaz	Opis
CSRR rd, csr	CSRRS rd, csr, x0	samo branje CSR
CSRW csr, rs1	CSRRW x0, csr, rs1	samo pisanje CSR
CSRWI csr, uimm	CSRRWI x0, csr, uimm	samo pisanje CSR iz immed.
CSRS csr, rs1	CSRRS x0, csr, rs1	nastavljanje (set) bitov v CSR, kadar stare vrednosti ne rabimo
CSRC csr, rs1	CSRRC x0, csr, rs1	brisanje (clear) bitov v CSR, kadar stare vrednosti ne rabimo
CSRSI csr, uimm	CSRRSI x0, csr, imm	nastavljanje (set) bitov v CSR iz imm., kadar stare vrednosti ne rabimo
CSRCI csr, uimm	CSRRCI x0, csr, imm	brisanje (clear) bitov v CSR iz imm., kadar stare vrednosti ne rabimo

Preostali sistemski ukazi

- FENCE
  - pomnilniška pregrada zagotavlja, da se pred pomnilniškim dostopom dokončajo vsi morebitni prejšnji dostopi
  - to je pomembno predvsem v kontekstu večnitenja in spremenjenega vrstnega reda izvajanja ukazov (out-of-order)
- FENCE.I
  - pregrada za ukaze zagotavlja, da se pred branjem ukaza izvedejo vsa morebitna prejšnja pisanja
- ECALL (environment call)
  - implementacija sistemskih klicev
  - sistemski klici omogočajo uporabniku, do dobi usluge od jedra OS (privilegiran način delovanja), tipično dostop do HW (pomnilnik, disk, terminal, ...)
- EBREAK
  - med izvajanjem programa vrne kontrolo razhroščevalniku

Kaj mi bo zbirni jezik?

Za prevedbo iz višjenivojskega ali ‘srednjenivojskega’ jezika (C) v zbirni jezik poskrbi prevajalnik

- npr.: gcc, clang, lcc, IAR, Visual C, Watcom, ...
- danes so prevajalniki že zelo dobri

Kljub temu pa je včasih potrebno napisati kako zbirniško kodo – v takem primeru ni potrebno prevajati konstruktov višjega jezika v zbirni jezik

- Torej, ni treba začeti z višjenivojsko kodo in jo prevajati, temveč lahko neposredno pišemo v zbirnem jeziku, saj lahko kaj naredimo tudi bolj učinkovito

Primeri uporabe programiranja v zbirnem jeziku

- **Zagonski programi** - nizkonivojska koda v bralnem oz. bliskovnem pomnilniku za inicializacijo in testiranje strojne opreme pred zagonom operacijskega sistema, npr. BIOS
- **Deli jedra OS**, sistemski klici za določeno arhitekturo
- Nekateri jeziki in prevajalniki omogočajo vključevanje delov zbirniške kode (**inline** assembly), npr. za specifično CPE
- **Disassembly** – koda v zbirnem jeziku, ki jo je ustvaril prevajalnik ob prevajanju iz višjega jezika – lahko se uporabi za razhroščevanje in/ali optimizacijo
- V zgodnjih računalnikih je bilo možno v zbirnem jeziku napisati **bolj učinkovito** kodo.
- **Vzvratno inženirstvo** (Reverse engineering) - strojne kode ni težko prevesti (disassembler) v zbirni jezik. Na ta način je *v principu* možno rekonstruirati izvorno kodo.

- Zanimiva uporaba zbirnika je tudi preverjanje, ali je indeks polja znotraj obsega (bounds check)
  - Če želimo preveriti na čimkrajši način, ali je neka spremenljivka x v obsegu $0 \leq x < y$, lahko predznačeno število obravnavamo kot nepredznačeno.
  - Negativna števila v 2'K izgledajo kot velika števila v nepredznačenem formatu!
  - Tako nam nepredznačena primerjava x < y preverja tako tudi, če je x < 0
  - Npr.: če x20 >= x11 ali x20 < 0, potem skoči na oznako IndexOutOfBounds :

```
bgeu x20, x11, IndexOutOfBounds
```

5d
Splošne lastnosti ukazov

PO KNJIGI - DUŠAN KODEK: ARHITEKTURA IN ORGANIZACIJA RAČUNALNIŠKIH SISTEMOV

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 pomnilnih 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 operande
  - 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**

$$OP3 \leftarrow OP2 + OP1$$

    - operandi so običajno v registrih

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

$$OP2 \leftarrow OP2 + OP1$$

- **1-operandni**
  - akumulator

$$AC \leftarrow AC + OP1$$

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

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

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

  - 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?

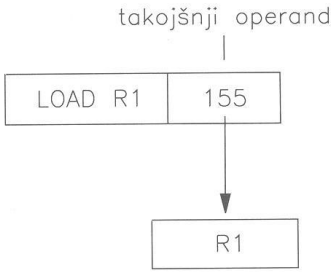
- **Lokacija operandov**
  - registri CPE
  - GP
  - (registri krmilnika V/I naprave)

- 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**
    - en operand v registru, drugi *lahko* v pomnilniku
  - **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)

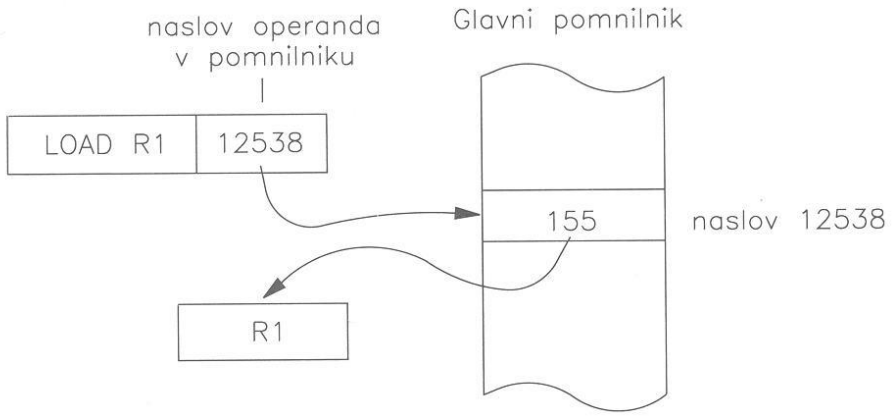


2. **Neposredno naslavljanje** (direct addressing)

- 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)



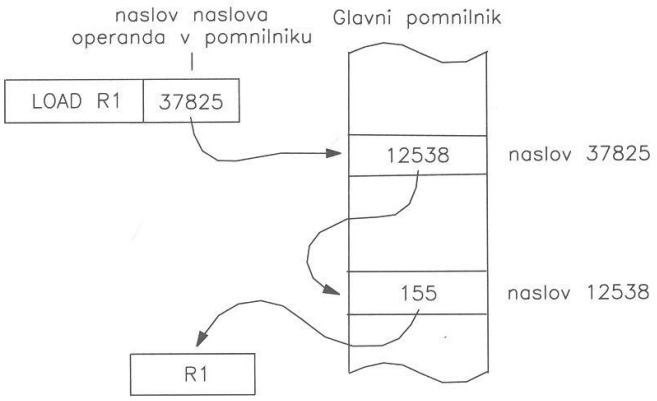
- Težave:
  - velik naslovni prostor → dolg naslov → dolgi ukazi
  - povečanje pom. prostora → drugačni ukazi → nezdružljivost za nazaj
  - primeri, ko operand ni na stalnem naslovu

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

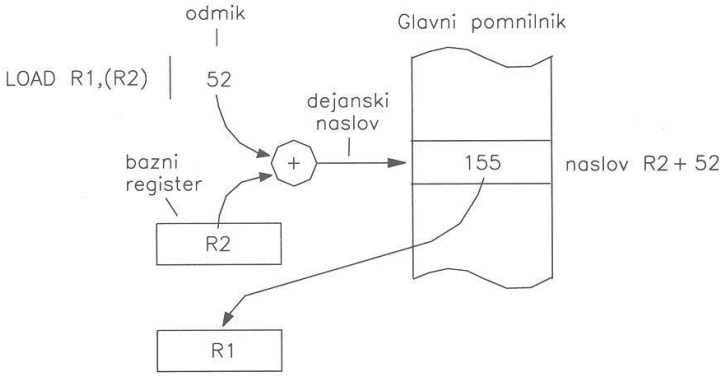
Posredno naslavljanje:

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 $\leftarrow$ 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. **Prenosi podatkov** (data transfer)
   - izvor, ponor
   - v resnici gre za kopiranje
   - Običajni **mnemoniki**:
     - LOAD: GP $\rightarrow$ R
     - STORE: R $\rightarrow$ GP
     - MOVE: R $\rightarrow$ R ali GP $\rightarrow$ GP
     - PUSH: GP ali R $\rightarrow$ Sklad
     - POP (PULL): Sklad $\rightarrow$ GP ali R
   - tudi CLEAR in SET

2. 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

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 (unconditional 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

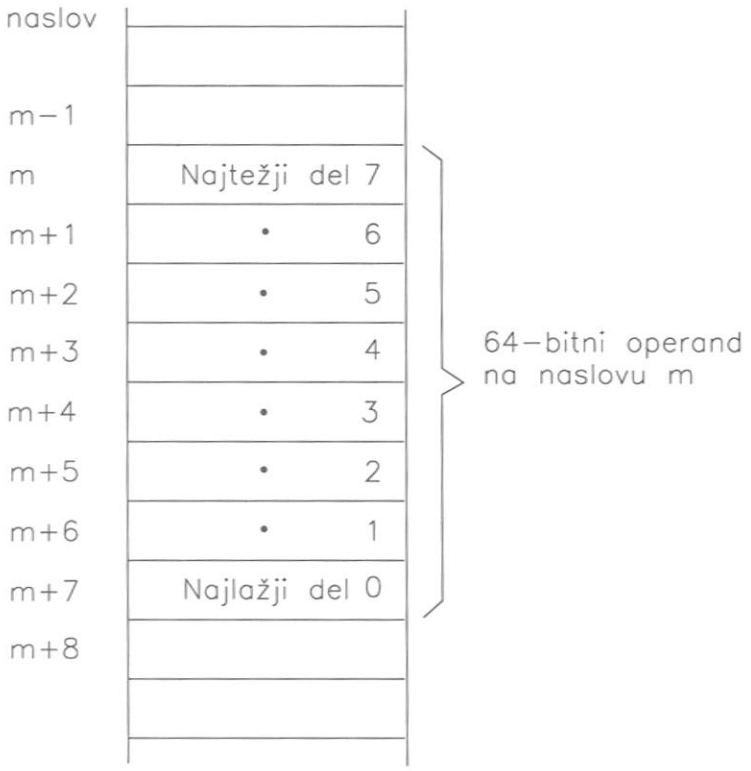
- Vrste operandov:

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

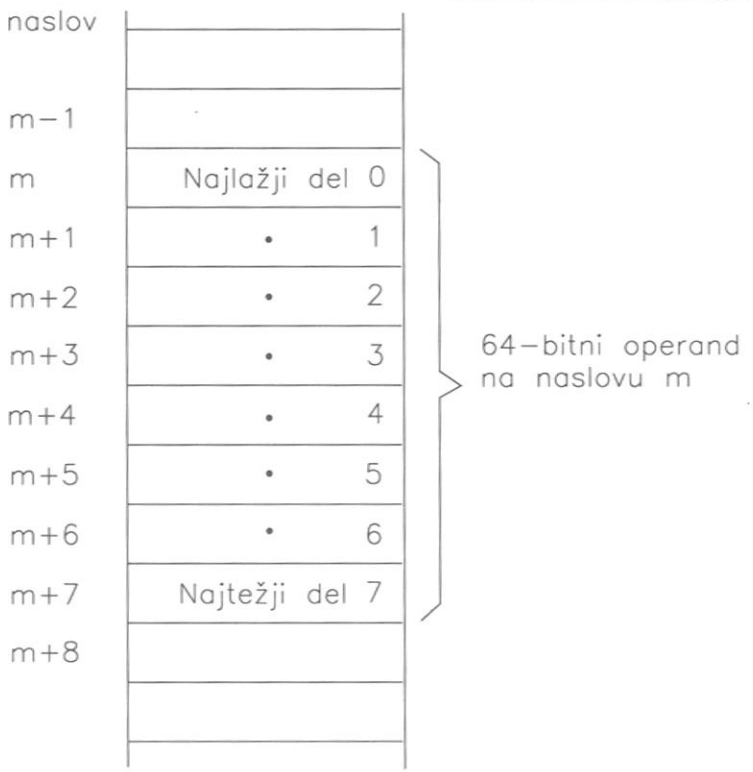
- Operandi dolžin večkratnikov 2 imajo posebna imena:
  - 8 Bajt (byte)
  - 16 Polovična beseda (halfword)
  - 32 Beseda (word)
  - 64 Dvojna beseda (double word)
  - 128 Štirikratna beseda (quad word)

    - 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



a) Pravilo debelega konca (Big Endian)



b) Pravilo tankega konca (Little Endian)

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

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

5
UKAZI

Prevajanje ukazov

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

- Primer 1 (iz jezika C v zbirni jezik):
  - Predpostavimo zaenkrat, da se vrednosti spremenljivk a, b in c že nahajajo v registrih x5, x6 in x7

```
a = b + c;              // v jeziku C
add x5, x6, x7          ; Pomen: x5 ← x6 + x7
```

- Primer 2:

```
a = b + c + d + e;   // x1: a, x2: b, x3: c, x4: d, x5: e
add x1, x2, x3       ; 3 ukazi
add x1, x1, x4       ;   v zbirnem
add x1, x1, x5       ;     jeziku
```

- Primer 3:

```
A[12] = h + A[8];          // x1: A(=naslov), x3: h

lw  x2, 32(x1)    ; x2 ← M[x1+32]      (32=8*4)
add x2, x2, x3    ; x2 ← x2 + x3
sw  x2, 48(x1)    ; M[x1+48]← x2       (48=12*4)
```

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

```
addi  x1, x2, 5          ; x1 ← x2 + 5
```

(add immediate)

Ukazna arhitektura (ISA)

- **Ukazna arhitektura** (**Instruction Set Architecture**, **ISA**)
  - natančno definira vse ukaze (nabor ukazov) nekega procesorja
  - ne govori pa o implementaciji
- RISC-V se vedno bolj uveljavlja kot odprta RISC arhitektura
  - Druge RISC arhitekture: ARM, MIPS, ...

RISC-V

- RISC-V je ukazna arhitektura (instruction-set architecture, ISA), ki je bila prvotno razvita za raziskave in poučevanje računalniških arhitektur
  - vse bolj pa postaja tudi standard na področju odprtih računalniških arhitektur za industrijske implementacije
  - RISC-V ISA je definirana brez detajlov implementacije
  - RISC-V je dejansko družina
    - RV32I - celoštevilska 32-bitna (XLEN=32)
    - RV64I - celoštevilska 64-bitna (XLEN=64)
    - RV128I - celoštevilska 128-bitna (XLEN=128)
    - RV32E - za majhne mikrokrmilnike (Embedded)

Lastnosti RISC-V

- 8-bitna pomnilniška beseda
- 32-bitni pomnilniški naslov
- Način shranjevanja operandov v CPE
  - 32 32-bitnih splošnonamenskih registrov x0, x1, …, x31
    - Vsebina x0 je vedno 0 (pri pisanju vanj se ne zgodi nič)
- Število eksplicitnih operandov v ukazu
  - vsi ALE ukazi imajo 3 eksplicitne operande
    - Tip R ima dva izvorna (source) registra *rs1* in *rs2* ter en ciljni destination register *rd*
    - Tip I ima en izvorni (source) register *rs1*, 12-bitni takojšnji (immediate) operand *imm* ter en ciljni destination register *rd*

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

Operacije in operandi

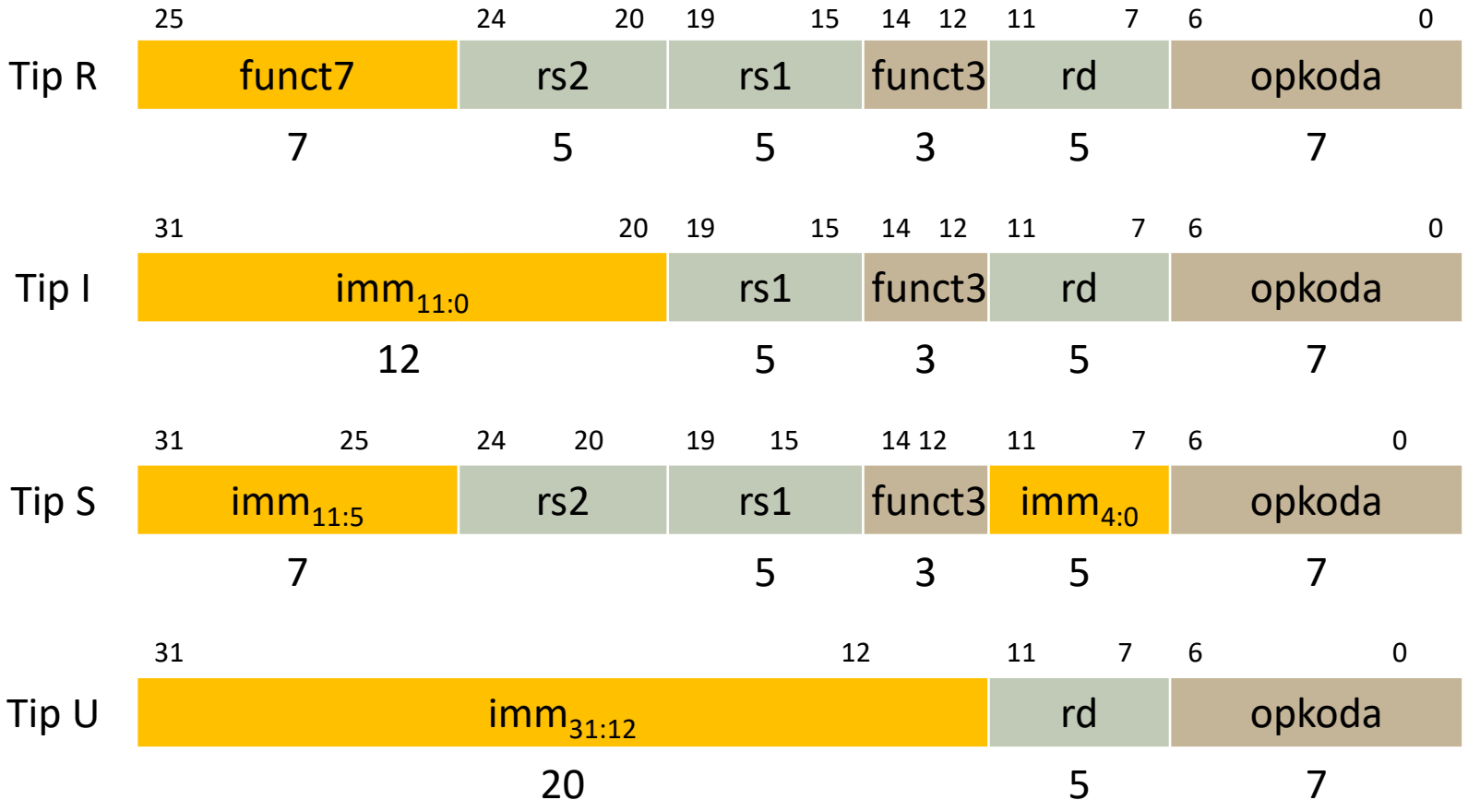
- pomnilniški operandi so lahko 8-, 16- ali 32-bitni (bajt, polbeseda, beseda)
- 16- in 32-bitni operandi so v pomnilniku shranjeni po **pravilu tankega konca** (**little endian rule**)
  - dobro je, da so poravnani (aligned), kar pomeni, da je operand, ki je sestavljen iz več bajtov, na naslovu, ki je deljiv s številom bajtov
    - 16-biten operand je na naslovu, deljivem z 2 (torej sodem) (zadnji bit enak 0)
    - 32-biten operand na naslovu, deljivem s 4 (zadnja dva bita enaka 0)
  - sicer je potreben prenos operanda v dveh kosih

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 tankega konca** (**little endian rule**)
- 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)

Format ukazov pri RV32I:

- vsi ukazi so 32-bitni in poravnani
- 4 osnovni formati (R, I, S, U)



- Format ukaza nam pove, kaj pomenijo posamezni biti v strojni kodi ukaza
  - rs (source register): iz njega se bere,
  - rd (destination register): vanj se piše
- Število ukazov nabora RV32I
  - vseh ukazov v naboru RV32I je 40
  - ni ukazov za množenje, deljenje
  - ni ukazov v plavajoči vejici

Vrste ukazov

- Ukaze RISC-V 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

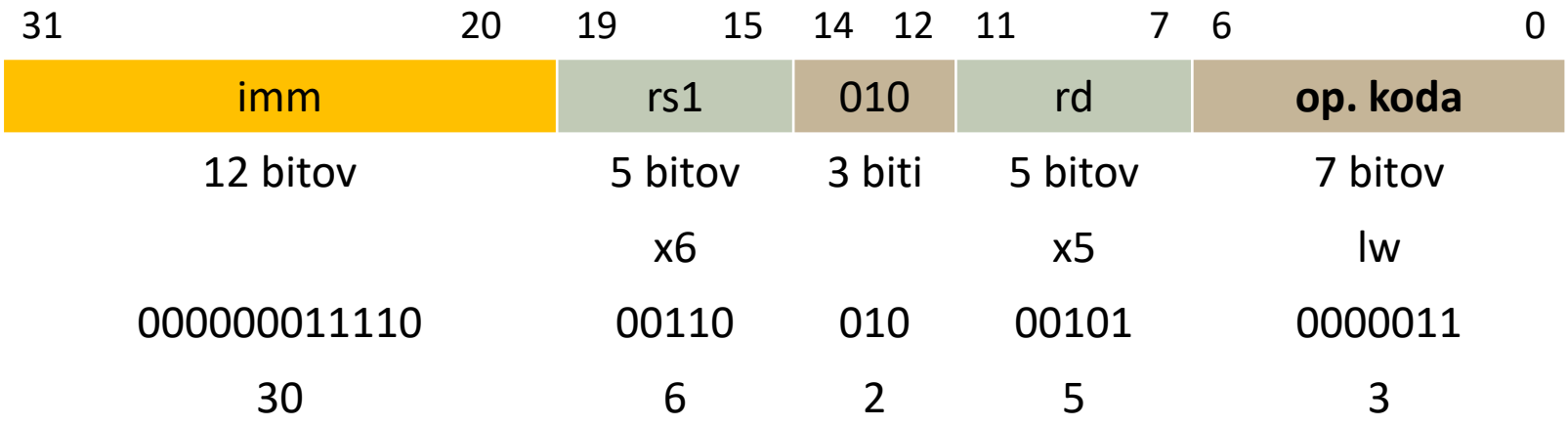
Ukazi za prenos podatkov (load/store)

- Uporabljajo format I z baznim naslavljanjem (bazni register je rs1)

Load word: `lw rd, imm(rs1)` ; rd $\leftarrow_{32}$ M[rs1 + se(imm)]

Npr.: `lw x5, 30(x6)` ; x5 $\leftarrow_{32}$ M[30 + x6]

Format I:



Celoten ukaz v strojni kodi: 0x01E32283

Torej: ukaz v zbirnem jeziku lw x5,30(x6) zbirnik 'prevede' v strojni ukaz 00000001111000110010001010000011

- 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

Razširitev operanda

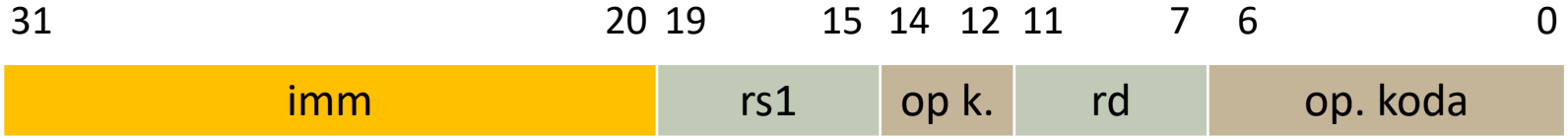
- Kaj pa, če je vrednost, ki jo želimo naložiti v (32-bitni) register, krajša od njegove dolžine?
  - Na katero vrednost postavimo preostale bite?
- 2 možnosti:
  - razširitev predznaka
    - lb (load byte (signed))
      - Npr.: 0x6F (01101111) se razširi v 0x0000006F (00000000 00000000 00000000 01101111),
        0x94 (10010100) pa se razširi v 0xFFFFFF94 (11111111 11111111 11111111 10010100)
    - lh (load halfword (signed))
      - Npr.: 0x73A1 (01110011 10100001) se razširi v 0x000073A1 (00000000 00000000 01110011 10100001),
        0xC40A (11000100 00001010) pa se razširi v 0xFFFFC40A (11111111 11111111 11000100 00001010)
  - razširitev ničle
    - lbu (load byte unsigned)
      - Npr., tudi 0x94 (10010100) se razširi v 0x00000094 (00000000 00000000 10010100)
    - lhu (load halfword unsigned)
      - Npr., tudi 0xC40A (11000100 00001010) se razširi v 0x0000C40A (00000000 00000000 11000100 00001010)

Load byte:

lb x5, 80(x6) ; $x5_{31..8} \leftarrow_{raz} M[80 + x6]_7$, $x5_{7..0} \leftarrow_8 M[80 + x6]$

$x5_{31..8}$ so biti od 31 do 8 (najvišjih 24 bitov),

$\leftarrow_{raz}$ pomeni razširitev predznaka



12 bitov	5 bitov	3biti	5 bitov	7 bitov
	x6	lb*	x5	lb*
000001010000	00110	000	00101	0000011
80	6	0	5	3

** lb sta oba dela skupaj (3+7 bitov)*

Celoten ukaz v strojni kodi: 0x05010083

Load byte unsigned

lbu x5, 80(x6)

pomen: $x5_{31..8} \leftarrow_{raz} 0$, $x5_{7..0} \leftarrow_{8} M[80 + x6]$

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
I	000 0000011	LB	Load byte
I	001 0000011	LH	Load halfword
I	010 0000011	LW	Load word
I	100 0000011	LBU	Load byte unsigned
I	101 0000011	LHU	Load halfword unsigned
S	000 0100011	SB	Store byte
S	001 0100011	SH	Store halfword
S	010 0100011	SW	Store word

- Odmik je 12-biten z razširitvijo predznaka (torej je lahko tudi negativen)
- 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)

Osnovne direktive zbirnika

.data – začetek podatkovnega segmenta

.text – začetek ukaznega segmenta

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

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

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

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

Direktive so namenjene zbirniku (programu), ne procesorju!

Primer programa v zbirnem jeziku za RISC-V

```
        .data   (podatkovni segment; vzemimo, da se začne na naslovu 0x400 = 1024_10)
var1:   .byte 5         (bajt z vrednostjo 5 na naslovu var1 = 1024)
var2:   .byte 6         (bajt z vrednostjo 6 na naslovu var2 = 1025)
sum:    .byte 0         (bajt z vrednostjo 0 na naslovu sum = 1026)

        .text           (kodni segment; običajno bo kar na naslovu 0)
        lb x5, 0x400(x0)        (load byte z naslova 0x400 v register x5)
        lb x6, 0x401(x0)        (x6 ← M[1025 + 0])
        add x7, x5, x6          (x7 ← x5 + x6)
        sb x7, 0x402(x0)        (x7 → M[1026+0])
```

Uporaba oznak

- Namesto naslova 0x400 lahko pišemo oznako labelo var1, katere vrednost je 0x400 (vsaka labela vsebuje pomnilniški naslov – bodisi ukaza, bodisi operanda)

  `lb x5, 0x400(x0) -> lb x5, var1(x0)`

  (Če je bazni register različen od 0, ga je prej seveda treba naložiti)

- Toda, pozor pri operacijah store v simulatorju Ripes!
  - Npr. **`sb x7,sum(x0)`** ne deluje, kot bi pričakovali!
  - Tak ukaz Ripes tolmači kot psevdoukaz in pred njim doda ukaz auipc (to bomo obravnavali kasneje).
  - Rešitev je, da namesto x0 uporabimo katerikoli register drug register, za katerega nam ni pomembno, ali se njegova vrednost spremeni (služi le kot začasni register), npr. **`sb x7,sum(x8)`**

Ukazi RISC-V (2. del)

- Osnovni nabor ukazov RV32I

RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
fm | pred | succ		rs1	000	rd	0001111	FENCE
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK

Registri

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5–7	t0–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

ALE ukazi

1. **aritmetične operacije** (+, –)
   - ADD, ADDI,
   - SUB
   - LUI, AUIPC
2. **logične bitne operacije** (&, V, ∇)
   - AND, ANDI
   - OR, ORI
   - XOR, XORI
3. **pomiki** (shift) (levi, desni; logični, aritmetični)
   - SLL, SLLI
   - SRL, SRLI
   - SRA, SRAI
4. **ukazi za primerjavo oz. set operacije** (pogoj: <)
   - SLT, SLTI, SLTU, SLTIU

Logične bitne operacije

- Logične bitne operacije delujejo po istoležnih bitih (bitwise operations):
  - IN (AND), &

```
   00110010
& 01010110
 ----------
   00010010
```

  - ALI (OR), V

```
   00110010
V 01010110
 ----------
   01110110
```

  - Ekskluzivni ALI (XOR), ∇

```
   00110010
^ 01101001
 ----------
   01011011
```

  - NE (NOT) – tega RISCV sicer nima, ker se da to narediti z XOR z enicami

```
  ~00011011
 ----------
  11100100
```

Uporaba bitnih operacij

Bitne operacije se uporabljajo tudi za branje in vpisovanje posameznih bitov v besedo

- Nastavljanje bita:
  - Kako nastavimo nek bit na 1 (ostale pa pustimo pri miru):

```
    xxxxxxxx
or  00010000
    xxx1xxxx
```

- Brisanje bita:
  - Kako postavimo nek bit na 0 (ostale pa pustimo pri miru):

```
    xxxxxxxx
and 11101111
    xxx0xxxx
```

- Branje bita:
  - Kako samo pogledamo vrednost določenega bita:

```
    xxxxxxxx
and 00010000
    000x0000
```

    Če je iskani bit 1, je dobljeni izraz od 0 različen, sicer je 0.

Pomiki

- Pomiki:
  - levi
    - SLL - Shift Left (Logical) in SLLI (SLL immediate)
      - 0110 $\rightarrow$ 1100
  - desni
    - **logični** (0110 $\rightarrow$ 0011)
      - v izpraznjena mesta gredo ničle
    - **aritmetični** (0110 $\rightarrow$ 0011, 1011 $\rightarrow$ 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
- 2 operanda sta v registrih
  - tretji je lahko v registru ali takojšnji (immediate)

rd ← rs1 op rs2
dd ← rs1 op Takojšnji operand (immediate)

ALE ukazi (1): aritmetične in logične operacije

Tip operacije	Ukaz	Opis	Format	Polja	Opkoda
Aritmetične	ADD	Add	R	0000000 rs2 rs1 000 rd	0110011
	SUB	Subtract	R	0100000 rs2 rs1 000 rd	0110011
	ADDI	Add imm.	I	imm12 rs1 000 rd	0010011
	LUI	Load upper imm.	U	imm20 rd	0110111
	AUIPC	Add upper imm. PC	U	imm20 rd	0010111
Tip operacije	**Ukaz**	**Opis**	**Format**	**funct7, funct3**	**opkoda**
Logične	AND	And	R	0000000 rs2 rs1 111 rd	0110011
	OR	Or	R	0000000 rs2 rs1 110 rd	0110011
	XOR	Exclusive or	R	0000000 rs2 rs1 100 rd	0110011
	ANDI	And imm.	I	imm12 rs1 111 rd	0010011
	ORI	Or imm.	I	imm12 rs1 110 rd	0010011
	XORI	Excl.-or imm.	I	imm12 rs1 100 rd	0010011

ALE ukazi (2): Pomiki

Tip opera-cije	Ukaz	Opis	Format	Polja	Opkoda
shift	SLL	Shift left logical	R	0000000 rs2 rs1 001 rd	0110011
	SRL	Shift right logical	R	0000000 rs2 rs1 101 rd	0110011
	SRA	Shift right arithmetic	R	0100000 rs2 rs1 101 rd	0110011
	SLLI	Shift left logical imm.	I	0000000 shamt* rs1 001 rd	0010011
	SRLI	Shift right logical immediate	I	0000000 shamt* rs1 101 rd	0110011
	SRAI	Shift right arithmetic imm.	I	0000000 shamt* rs1 101 rd	0110011

*shamt ... shift amount

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

ALE ukazi (3): Ukazi za primerjavo

Tip operacije	Ukaz	Opis	Format	Polja	Opkoda
set	SLT	Set if less than	R	0000000 rs2 rs1 010 rd	0110011
	SLTU	Set if less than unsigned	R	0000000 rs2 rs1 011 rd	0110011
	SLTI	Set if less than immediate	I	imm12 rs1 010 rd	0010011
	SLTUI	Set if less than unsig. imm.	I	imm12 rs1 011 rd	0010011

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

ADD:

```
add  x3, x5, x6          ; x3 ← x5 + x6
```

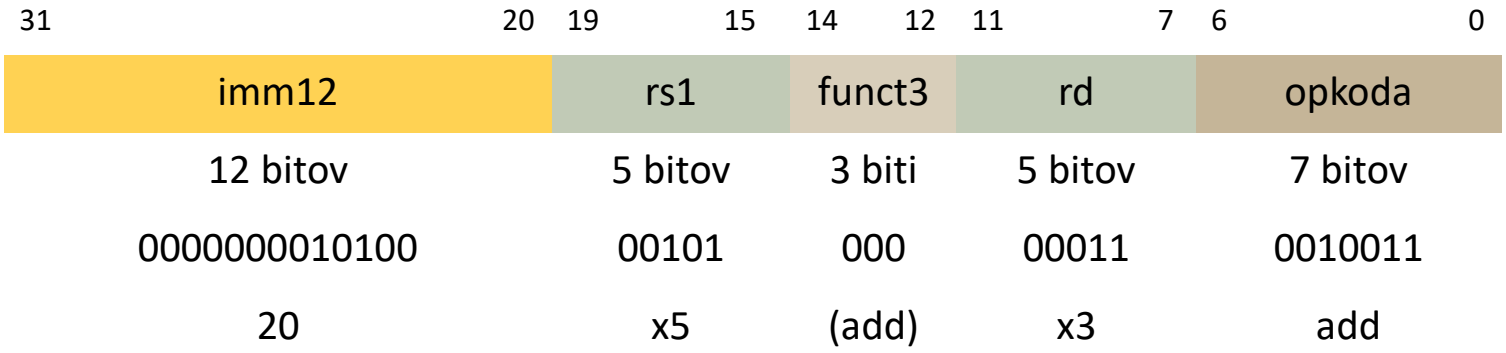
Format R:

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opkoda
7 bitov	5 bitov	5 bitov	3 biti	5 bitov	7 bitov
0000000	00110	00101	000	00011	0110011
(add)	x6	x5	(add)	x3	add

ADDI (Add immediate)

addi x3, x5, 20 ; x3 ← x5 + 20

Format I:

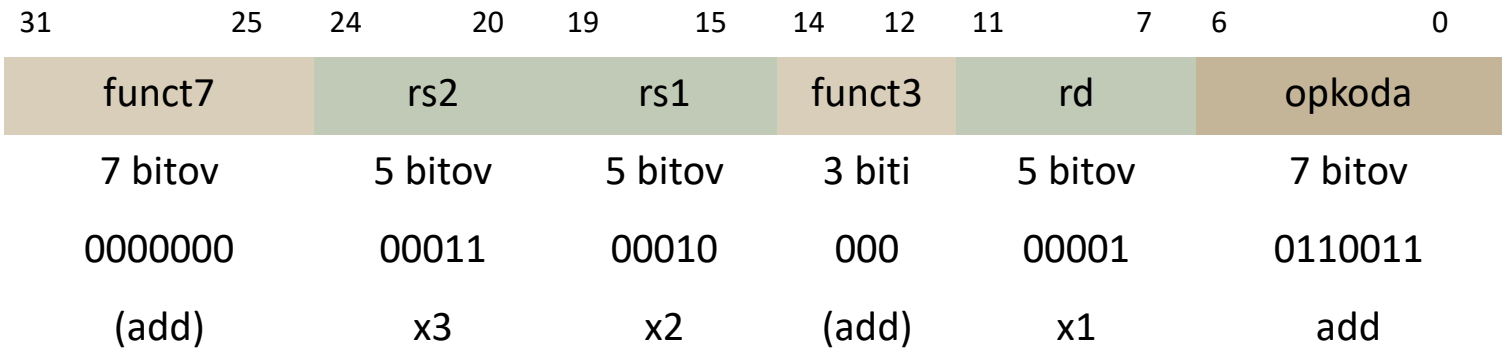


- Takojšnjemu (12-bitnemu) operandu se razširi predznak (na 32 bitov).
- A pozor: če pišemo v šestnajstiškem zapisu, ne sme imeti na začetku enice (zbirnik javi napako)!
  - Če želimo, da je negativen, moramo dodati predznak (npr. -0x800)
  - V desetiškem zapisu ima negativno število itak predznak (npr. imm. -1 se razširi na same enice v registru)

AND

and x1, x2, x3 ; x1 ← x2 & x3

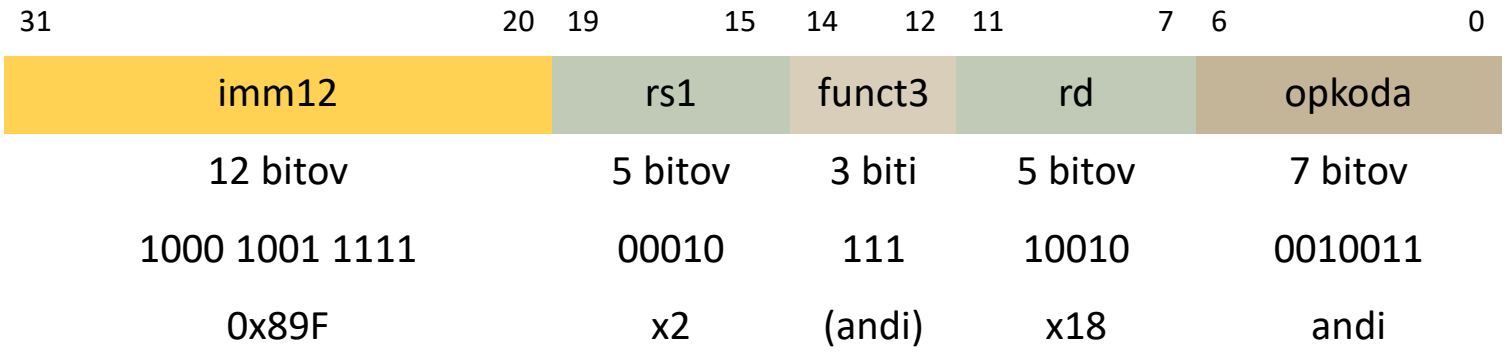
Format R:



ANDI (and immediate)

andi x18, x2, 0x49F ; x18 ← x2 & 0x49F

Format I:

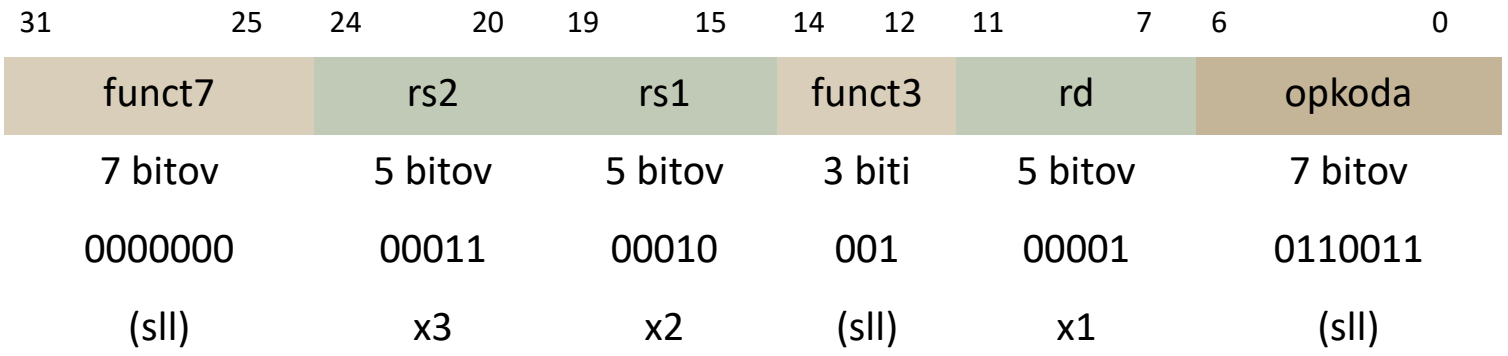


- Takojšnjemu (12-bitnemu) operandu se razširi predznak (na 32 bitov).
- A pozor: če pišemo v šestnajstiškem zapisu, ne sme imeti na začetku enice (zbirnik javi napako)!
  - Če želimo, da je negativen, moramo dodati predznak

SLL (shift left logical)

sll x1, x2, x3 ; x1 ← x2 << x3 (oz. r2 × 2^{r3})

Format R:

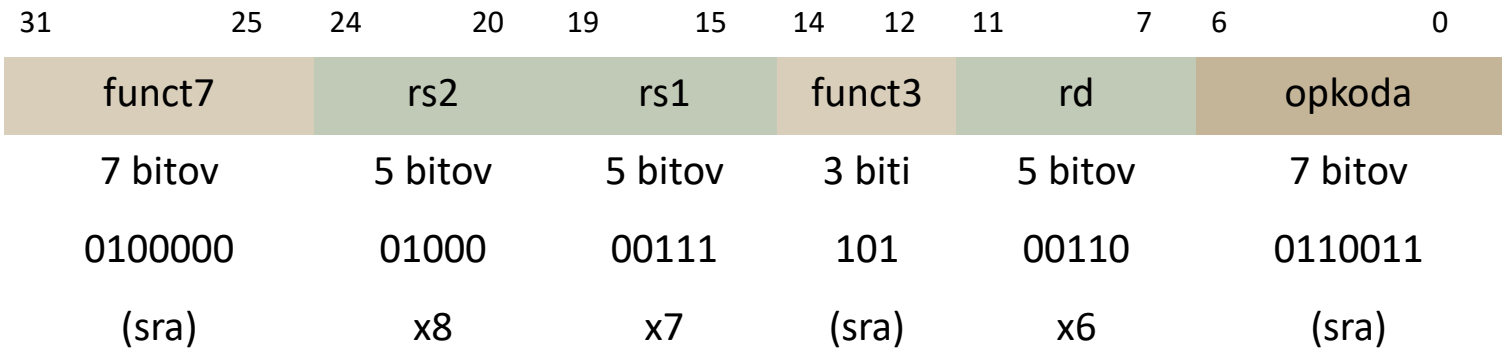


SRA (shift right arithmetic)

sra x6, x7, x8 ; x6 ← x7 >> x8

; $x6_{31} \leftarrow x7_{31}$

Format R:



LUI (Load upper immediate)

- poseben ukaz, ki 20-bitno (konstantno) vrednost naloži v gornjih 20 bitov registra, spodnjih 12 bitov pa je 0
- Zakaj sploh potrebujemo tak ukaz?
  - Problem je, kako naložiti 32-bitno konstanto v register
  - z enim 32-bitnim ukazom ni možno
  - zato to lahko storimo v 2 korakih:
    1. naložimo zgornjih 20 bitov
    2. naložimo spodnjih 12 bitov

  - Npr.: 0x12345678

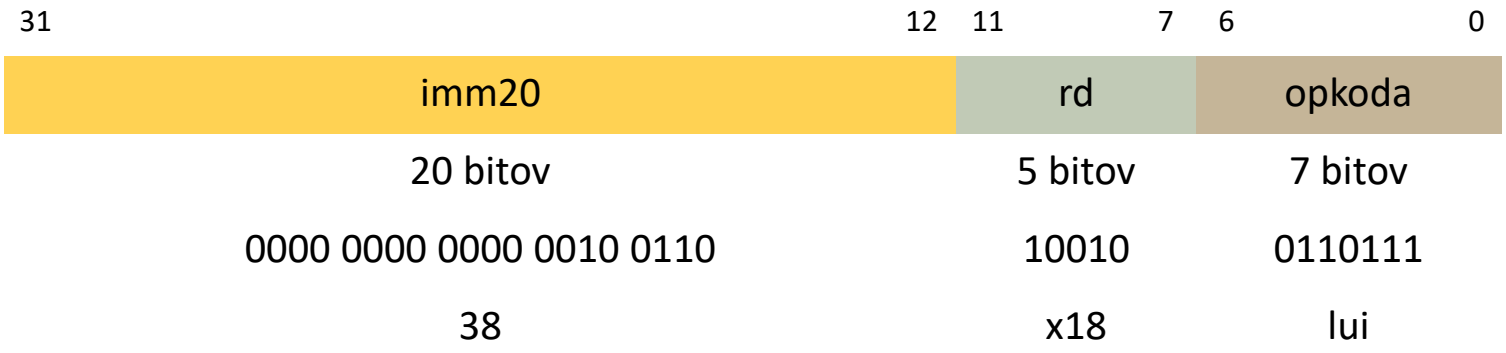
```
lui   x5, 0x12345
addui   x5, x5, 0x678
```

(lahko tudi z `ori`)

LUI (load upper immediate)

lui x18, 38 ; $x18_{31..12} \leftarrow 38$, $x18_{11..0} \leftarrow 0$

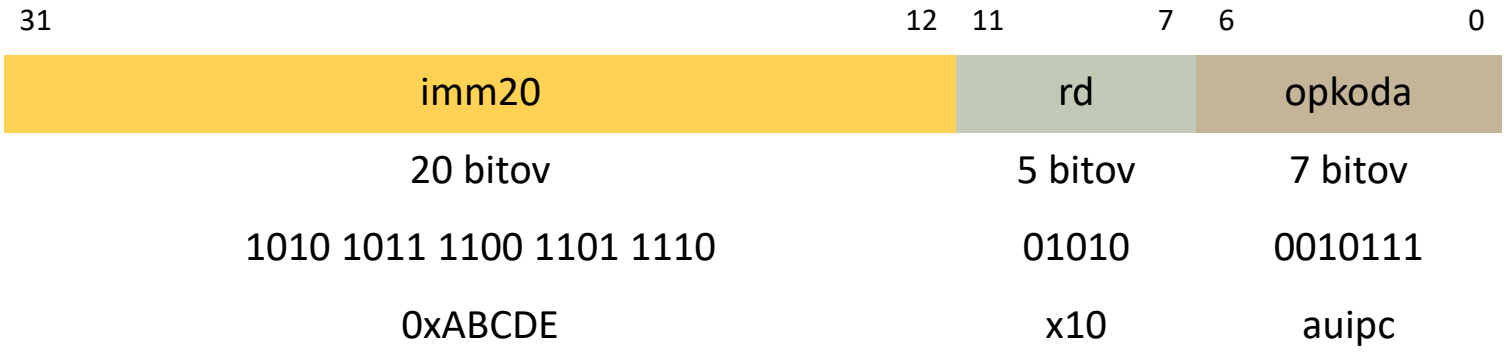
Format U:



AUIPC (add upper immediate to PC)

auipc x10, 0xABCDE ; x10 ← (0xABCDE << 12) + PC

Format U:



- Tudi to je poseben ukaz, ki 20-bitno (konstantno) vrednost naloži v gornjih 20 bitov registra (spodnjih 12 bitov je 0), temu pa prišteje vrednost programskega števca PC
- Ta ukaz omogoča PC-relativno naslavljanje
  - Tako se da celoten program linearno premakniti v drug del pomnilnika

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

```
        .data           ; (na naslovu 0x400)
A:      .word 5
B:      .word 0

        .text
        lw x1, A(x0)
        slli x2, x1, 3
        slli x3, x1, 1
        add x4, x2, x3
        sw x4, B(x5)  ; x5 je začasni register
```

Set-ukazi (oz. ukazi za primerjavo)

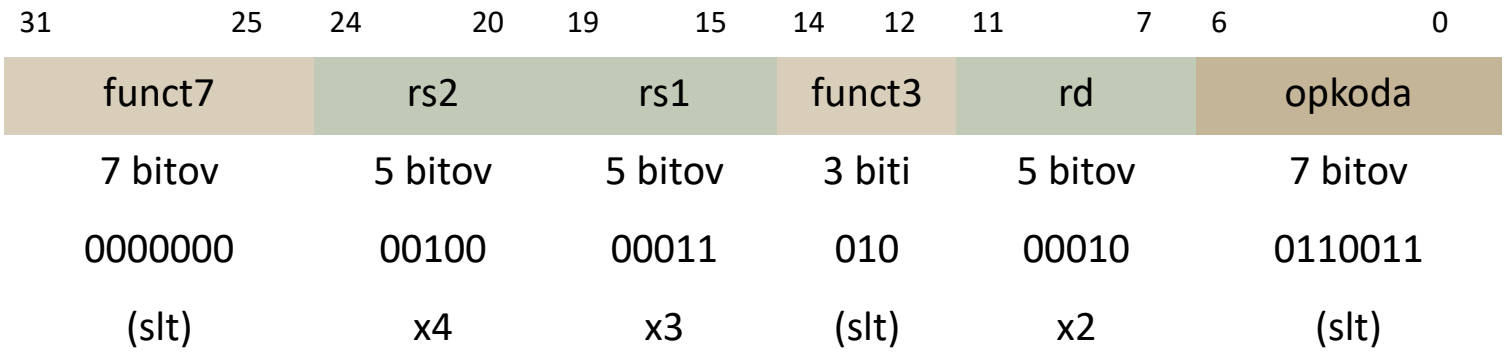
Če je podani pogoj izpolnjen, postavijo v ciljni register 1 (...0001), sicer 0 (...0000)

- SLT (Set if Less Than), format R
  - slt rd, rs1, rs2 ; rd ← (rs1 < rs2) ? 1 : 0
- SLTI (Set if Less Than Immediate), format I
  - slti rd, rs1, imm ; rd ← (rs1 < imm i) ? 1 : 0
- SLTIU (Set if Less Than Immediate Unsigned), format I
  - sltiu rd, rs1, imm ; rd ← (rs1 < imm i) ? 1 : 0
- SLTU (Set if Less Than Unsigned), format R
  - sltu rd, rs1, rs2 ; rd ← (rs1 < rs2) ? 1 : 0

SLT (set if less than)

slt x2, x3, x4 ; x2 ← (x3 < x4)

Format R:



Psevdo-ukazi

- Poleg direktiv obstajajo tudi psevdo-ukazi, ki niso dejanski ukazi procesorja, ampak so namenjeni zbirniku
- Primeri:
  - **nop** (addi x0, x0, 0) no operation
  - **mv rd, rs** (addi rd, rs, 0) vsebina rs se kopira v rd
  - **not rd, rs** (xori rd, rs, -1) eniški komplement
  - **neg rd, rs** (sub rd, x0, rs) dvojiški komplement
  - **seqz rd, rs** (sltiu rd, rs, 1) set if equal (to) zero
  - **snez rd, rs** (sltu rd, x0, rs) set if not equal (to) zero
  - **sltz rd, rs** (slt rd, rs, x0) set if less than zero
  - **sgtz rd, rs** (slt rd, x0, rs) set if greater than zero
  - ...

Ukazi RISC-V (3. del)

Kontrolni ukazi

- Kontrolni ukazi omogočajo spremembo vrstnega reda izvajanja ukazov
  - takim ukazom rečemo skoki
  - Zmožnost odločitev razlikuje računalnik od kalkulatorja!
- 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
  - seveda pa tudi klice podprogramov ter poljubne skoke

- Skoki pri RISC-V:
  - Brezpogojni skok:
    - JAL (jump and link) – format J
    - JALR (jump and link register) – format I
  - Pogojni skoki (format B):
    - BEQ (branch if equal to), če rs1==rs2
    - BNE (branch if not equal zero), če rs1!=rs2
    - BLT, BLTU (branch if less than (unsigned)), če rs1 < rs2
    - BGE, BGEU (branch if greater or equal (unsigned)), če rs1 >= rs2
  - Pogojni skoki uporabljajo format B in *PC-relativno naslavljanje*
    - za bazni register je uporabljen PC

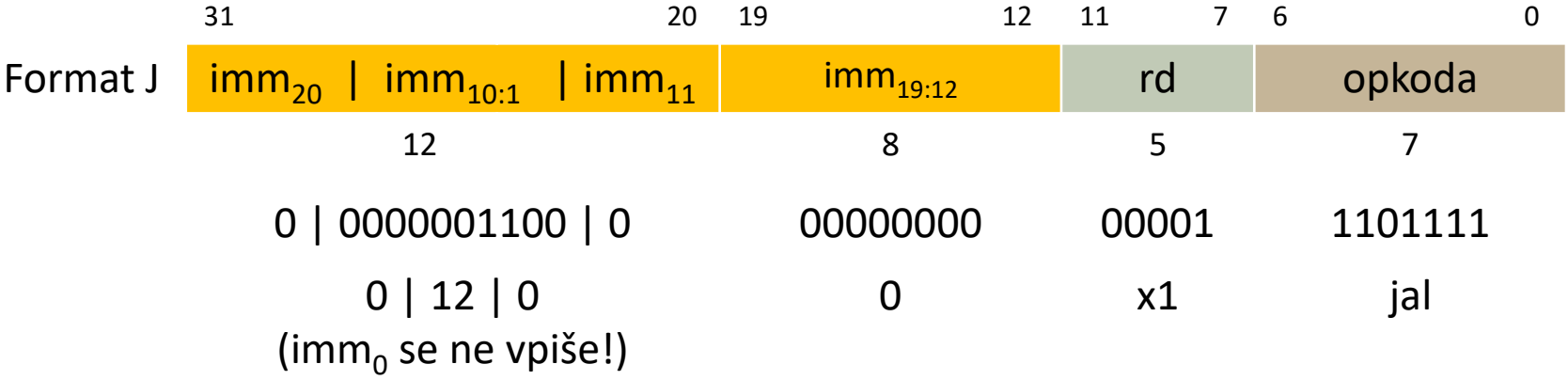
JAL (Jump and link):

jal rd, target # rd ← PC+4,
PC ← target = PC + se(2*imm20)

Zbirnik gornji ukaz prevede v `jal rd,imm20` (lahko pa ga sami zapišemo tako)

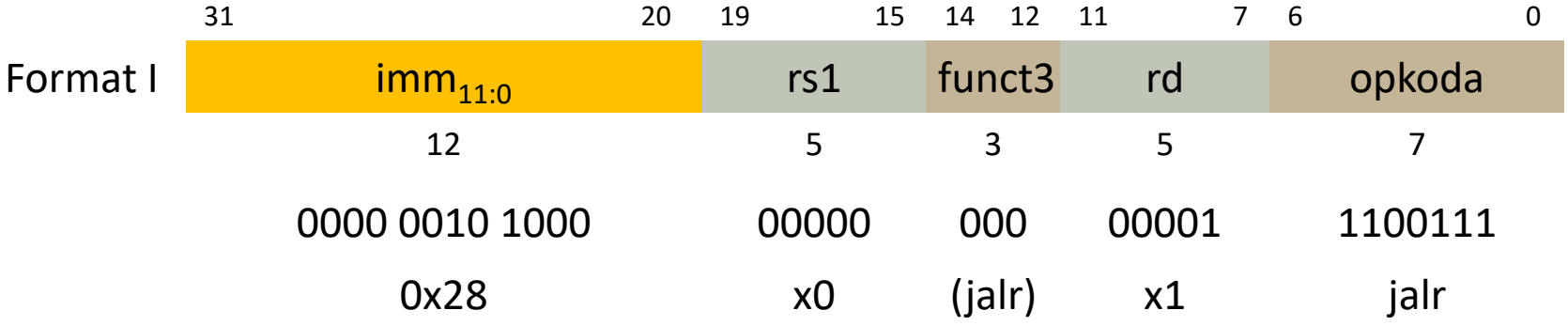
Pazi: target in imm ni ista stvar!

```
Npr.: 0x10          jal  x1, nekam   # x1 ← PC+4 (= 0x14 =20)
                     . . .           # PC ← nekam (= 0x28)
                     . . .           # imm20 = (0x28-0x10) =
      0x28  nekam:   . . .           #       = (40-16) = 24
```



JALR (Jump and link register):

```
        jalr rd, imm12(rs1)          # rd ← PC+4,
                                     # PC ← target
                                     #      = (rs1 + se(imm12)) & (-2)
                                     #      zadnji bit je 0 (-2 = ..11110 = ~1)
                                                          (~ je 1'K)
Npr.: 0x10      jalr  x1, nekam(x0)  # x1 ← PC+4 (= 0x14 =20)
                . . .                # PC ← nekam + x0 (= 0x28)
                . . .                #           imm12 = (0x28-0) = 40
      0x28 nekam:  . . .
```



Opomba: v simulatorju Ripes se ukaz JALR piše malo drugače:
jalr x1, x2, 28 ali jalr x1, x2, nekam

BNE (Branch if Not Equal to):

bne rs1, rs2, imm12 (PC-relativno)

Pozor: imm_0 se ne vpiše v strojni ukaz, ampak biti $imm_{12:1}$ in s tem pol manjši

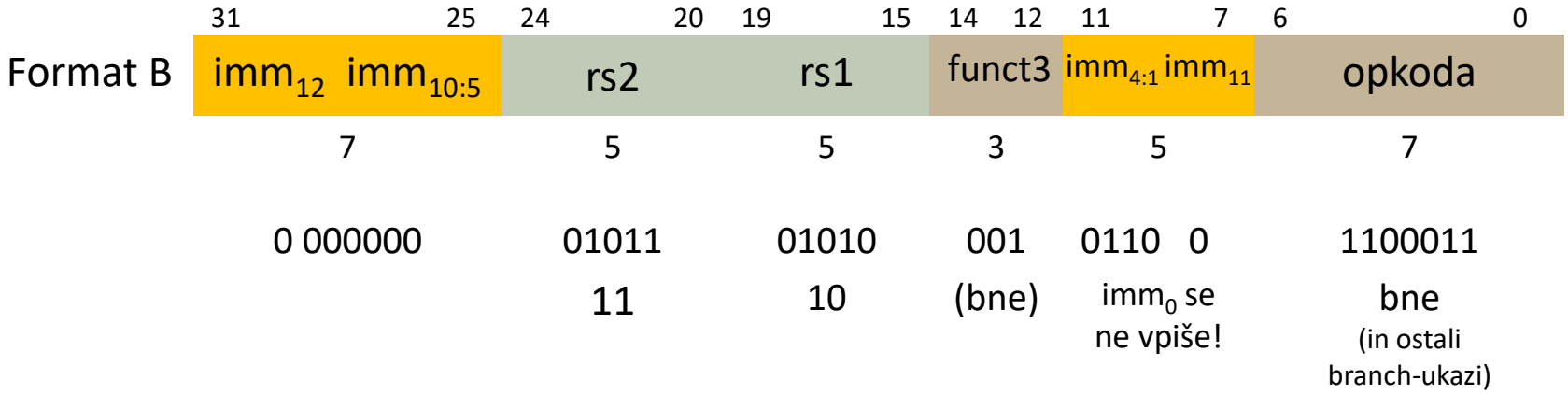
imm12 = ciljni naslov – PC (ukaza bne)

Npr., imm12 = 20 skoči za 5 ukazov naprej

Če v ukazu zapišemo oznako (labelo), zbirnik izračuna imm12 ← label – PC !

Npr.:

```
        bne  x10, x11, 12        # če x10 != x11, potem PC ← PC + 12,
        . . .                    #       sicer PC ← PC + 4
        . . .
lab1:   . . .
```



Vejitve

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

- Če pogoj ni izpolnjen, je treba skočiti na blok2

- Ukaz beq izvaja pogojni skok

```
beq rs1,rs2,LABEL
```

  - Če rs1 == rs2, CPE skoči na naslov LABEL
- Podoben ukaz je

```
bne rs1,rs2,LABEL
```

  - Če rs1 != rs2, CPE skoči na naslov LABEL

- Primer:

```
if (c < 5)
        a = b + 1;
else
        a = 2;
```

Predpostavimo x1: c, x2: a, x3: b

- Več možnosti:

```
        addi  t0, x0, 5     # t0 = 5
        bge   x1, t0, Blk2  # if (c >= 5) goto Blk2;
Blk1:   addi  x2, x3, 1     # a = b + 1;
        jal    x0, Ven      # goto Ven;
Blk2:   addi  x2, x0, 2     # Blk2: a = 2;
Ven:    naslednji ukaz      # Ven: naslednji ukaz
```

```
        slti  t0, x1, 5     # t0 = (c < 5)?
        beq   t0, x0, Blk2  # if (t0==0) goto Blk2;
Blk1:   addi  x2, x3, 1     # a = b + 1;
        jal   x0, Ven       # goto Ven;
Blk2:   addi  x2, x0, 2     # Blk2: a = 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 RISC-V

`sum = 0;`
`i = 5;`
`while ( i > 2)`
`{`
`  sum = sum + i;`
`  i--;`
`}`
	`addi  x1, x0, 0  # sum`
	`addi  x2, x0, 2  # 2`
	`addi  x3, x0, 5  # i`
	`Loop:  bge  x2, x3, Ven  # if(2>=i) Ven`
	`add  x1, x1, x3`
	`addi  x3, x3, -1`
	`jal x0, Loop`
	`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  x1, x0, 0  # sum`
	`addi  x2, x0, 2  # 2`
	`addi  x3, x0, 5  # i`
	`Loop:  add  x1, x1, x3`
	`addi  x3, x3, -1`
	`blt  x2, x3, 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

```
                //int a[10] = {5, 5, 5, 8, 2};
                //int ax = 5;
                //int i = 0;
                while (a[i] == ax)
                   i++;
```

(x1: i, x2: ax, x3: bazni naslov a, tj. naslov od a[0], &a[0])

```
Loop:           slli    x4, x1, 2       # 4*i
                add     x4, x4, x3      # a + 4*i
                lw      x5, 0(x4)       # v x4 je naslov a[i]
                bne     x5,x2,Exit      # (a[i]==ax)? Exit
                addi    x1, x1, 1
                jal     x0, Loop
Exit:           ...
```

Primeri kontrolnih ukazov

	Primer ukaza	Ime ukaza	Opis
JAL	x9, 84(x8)	Jump and link	x9 ← PC + 4, PC ← x8 + 84 (če je rd==x0, je jal navaden brezpogojni skok)
JALR	x2, 84(x8)	Jump and link register	x9 ← PC + 4, PC ← x8 + 84
BEQ	x7, x8, 0x8C	Branch if EQual to	če x7 == x8, potem PC ← PC + 0x8C, sicer PC ← PC + 4
BNE	x7, x8, 0x8C	Branch if Not Equal to	če x7 != x8, potem PC ← PC + 0x8C, sicer PC ← PC + 4
BLT	x5, x6, 24	Branch if Less Than	če x5 < x6, potem PC ← PC + 24, sicer PC ← PC + 4
BGE	x5, x6, 24	Branch if Greater or Equal than	če x5 >= x6, potem PC ← PC + 24, sicer PC ← PC + 4
BLTU	x5, x6, 24	Branch if Less Than, Unsigned	če x5 < x6 (nepredznačeno), potem PC ← PC + 24, sicer PC ← PC + 4
BGEU	x5, x6, 24	Branch if Greater or Equal than, Unsigned	če x5 >= x6 (nepredznačeno), potem PC ← PC + 24, sicer PC ← PC + 4

Sistemski ukazi

CSR – Control and Status Register

- 12-bitni takojšnji operand določa enega od možnih 4096 registrov CSR

Format	Opkoda	funct3	Ukaz		Opis (ze ... zero extended)
I	1110011	001	CSRRW	Atomic Read/Write CSR	rd ← ze(CSR), razen če rd==x0. CSR ← rs1
I	1110011	010	CSRRS	Atomic Read and Set bits in CSR	rd ← ze(CSR). Biti CSR, ki imajo v rs1 (=maska) 1, se postavijo na 1, razen če rs1==x0
I	1110011	011	CSRRC	Atomic Read and Clear bits in CSR	rd ← ze(CSR). Biti CSR, ki imajo v rs1 (=maska) 1, se brišejo (0), razen če rs1==x0
I	1110011	101	CSRRWI	CSRRW immed.	rd ← ze(CSR), razen če rd==x0. CSR ← ze($uimm_{4:0}$) (v polju rs1)
I	1110011	110	CSRRSI	CSRRS immed.	rd ← ze(CSR). Biti CSR, ki imajo v rs1 (=maska) 1, se postavijo na 1, razen če $uimm_{4:0}$==0
I	1110011	111	CSRRCI	CSRRC immed.	rd ← ze(CSR). Biti CSR, ki imajo v rs1 (=maska) 1, se brišejo (0), razen če $uimm_{4:0}$==0

- Sistemski ukazi shranjujejo podani register CSR v podani splošnonamenski register rd, v CSR pa naložijo novo vrednost (nove bite)
  - pogosto pa ne potrebujemo obeh ‘storitev’, ampak le eno
- Psevdoukazi za enostavnejše primere:

Psevdoukaz	Ukaz	Opis
CSRR rd, csr	CSRRS rd, csr, x0	samo branje CSR
CSRW csr, rs1	CSRRW x0, csr, rs1	samo pisanje CSR
CSRWI csr, uimm	CSRRWI x0, csr, uimm	samo pisanje CSR iz immed.
CSRS csr, rs1	CSRRS x0, csr, rs1	nastavljanje (set) bitov v CSR, kadar stare vrednosti ne rabimo
CSRC csr, rs1	CSRRC x0, csr, rs1	brisanje (clear) bitov v CSR, kadar stare vrednosti ne rabimo
CSRSI csr, uimm	CSRRSI x0, csr, imm	nastavljanje (set) bitov v CSR iz imm., kadar stare vrednosti ne rabimo
CSRCI csr, uimm	CSRRCI x0, csr, imm	brisanje (clear) bitov v CSR iz imm., kadar stare vrednosti ne rabimo

- Za prevedbo iz višjenivojskega ali ‘srednjenivojskega’ jezika (C) v zbirni jezik poskrbi prevajalnik (npr. clang, gcc C, lcc, IAR, Visual C, Watcom, ...)
  - danes so prevajalniki že zelo dobri
- Kljub temu pa je včasih potrebno napisati kako zbirniško kodo – v takem primeru ni potrebno prevajati konstruktov višjega jezika v zbirni jezik
  - Torej, ni treba začeti z višjenivojsko kodo in jo prevajati, temveč lahko neposredno pišemo v zbirnem jeziku, saj lahko kaj naredimo tudi bolj učinkovito
- Primeri uporabe programiranja v zbirnem jeziku:
  - Zagonski program - nizkonivojska koda v bralnem oz. bliskovnem pomnilniku za inicializacijo in testiranje strojne opreme pred zagonom operacijskega sistema, npr. BIOS
  - Deli jedra OS, sistemski klici za določeno arhitekturo
  - Nekateri jeziki in prevajalniki omogočajo vključevanje delov zbirniške kode (inline assembly), npr. za specifično CPE
  - Disassembly – koda v zbirnem jeziku, ki jo je ustvaril prevajalnik ob prevajanju iz višjega jezika – lahko se uporabi za razhroščevanje in/ali optimizacijo
  - V zgodnjih računalnikih je bilo možno v zbirnem jeziku napisati kodo, bližjo optimalni.
  - Vzvratno inženirstvo (Reverse engineering) - strojne kode ni težko prevesti (disassembler) v zbirni jezik. Na ta način je *v principu* možno rekonstruirati izvorno kodo, drugo pa je vprašanje legalnosti takega početja

Ukazi RISC-V (4. del)

PROCEDURE

Procedure

Procedura je podprogram, ki opravlja specifično nalogo na osnovi parametrov oz. argumentov, ki mu jih poda klicoči program oz. klicatelj.

- Proceduri lahko rečemo tudi *podprogram*, v jeziku C pa se imenuje *funkcija*

Pri klicu procedure so potrebni naslednji koraki:

1. Klicatelj postavi parametre nekam, kjer procedura lahko dostopa do njih.
2. Klicatelj program prenese kontrolo na proceduro.
3. Pridobi pomnilniške vire, potrebne za proceduro.
4. Procedura Izvedite želeno nalogo.
5. Procedura da rezultat nekam, kjer ga lahko klicatelj dobi.
6. Vrnitev na naslednji ukaz klicatelja.

- Ker so registri najhitrejša vrsta pomnilnika, jih uporabimo, kjer se le da.
- Programi za RISC-V morajo uporabljati dogovor o klicu podprogramov (calling convention), ki 8 registrov (x10–x17) uporablja za parametre oz. argumente procedur:
  - x10, x11 (a0, a1) za argumente ali za vrnjene vrednosti
  - x12-x17 (a2-a7) za argumente
- Dogovor uporablja register x1 (ra) za povratni naslov (return address)

- Zbirni jezik RISC-V vključuje ukaz JAL, ki je zelo primeren za podprograme. Shrani povratni naslov v register rd in skoči na podani naslov

```
jal x1, NaslovProcedure
```

- Za vrnitev se uporablja indirektni skok JALR:

```
jalr x0, 0(x1)      # skoči na naslov v x1 (ra)
```

- Kaj pa, če prevajalnik potrebuje več kot 8 registrov za argumente?
  - Vsak register, ki ga klicatelj uporabi, mora na koncu vrniti v začetno stanje.

Sklad

- Struktura, ki je najbolj primerna za ‘razlivanje’ (spill) registrov v pomnilnik, je sklad.
  - Sklad je podatkovna struktura (linearni seznam), organizirana v smislu LIFO.
  - Za delo s skladom potrebujemo **skladovni kazalec** (SP): kazalec, ki kaže na vrh sklada.
  - Pri RISC-V ima vlogo skladovnega kazalca register x2 - njegovo ABI ime je sp.

- Sklad največ uporabljamo v podprogramih za:
  - shranjevanje povratnega naslova
  - prenos parametrov v podprograme
  - shranjevanje začasnih spremenljivk (npr. lokalnih v podprogramih)
  - shranjevanje registrov v podprogramih

- Operaciji PUSH in POP:
  - PUSH ‘porine’ register na sklad
  - POP ‘pobere’ register s sklada

Možne so 4 variante (pri vseh predpostavimo, da registrski operand zaseda *n* pomnilniških besed):

1. Sklad narašča v smeri naraščajočih naslovov in SP kaže na prvo prosto mesto na skladu:
   - PUSH reg: M[SP] ← reg; SP ← SP + n;
   - POP reg: SP ← SP - n; reg ← M[SP];
2. Sklad narašča v smeri naraščajočih naslovov in SP kaže na zadnji podatek na skladu:
   - PUSH reg: SP ← SP + n; M[SP] ← reg;
   - POP reg: reg ← M[SP]; SP ← SP - n;
3. Sklad narašča v smeri padajočih naslovov in SP kaže na prvo prosto mesto na skladu:
   - PUSH reg: M[SP] ← reg; SP ← SP - n;
   - POP reg: SP ← SP + n; reg ← M[SP];
4. Sklad narašča v smeri padajočih naslovov in SP kaže na zadnji podatek na skladu:
   - PUSH reg: SP ← SP - n; M[SP] ← reg;
   - POP reg: reg ← M[SP]; SP ← SP + n;

- Običajno sklad raste iz višjih naslovov proti nižjim (ni pa nujno - lahko gre k višjim in kopica k nižjim).
- RISC-V uporablja varianto 4

Dogovor o klicu procedur (calling convention)

- Dogovor o klicu procedur (calling convention) za RISC-V pravi, naj
  - sklad narašča v smeri padajočih naslovov in naj
  - skladovni kazalec kaže na zadnji podatek na skladu.
- Sklad se torej začne na dnu podatkovnega segmenta – na naslovu, ki sicer ni del sklada.
- Na začetku vsakega programa moramo nastaviti skladovni kazalec
  - npr.: `addi sp, x0, 0x500`, če sami pišemo program
  - Bolj običajno pa je, da začetna koda iz skripte povezovalnika (linker) prebere, kje je konec podatkovnega segmenta in inicializira skladovni kazalec tam
- RISC- V 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:

```
PUSH reg:  addi sp, sp, -4
           sw reg, 0(sp)

POP reg:   lw reg, 0(sp)
           addi sp, sp, 4
```

- V splošnem je treba za sabo pospraviti – torej, če uporabimo znotraj procedure neke registre, je treba njihove stare vrednosti shraniti in na pred izstopom iz procedure obnoviti, ker jih klicatelj morda uporablja.
- Da pa se lahko izognemo shranjevanju in obnavljanju registrov, katerih vrednosti ne bodo nikoli uporabljene, kar se tipično zgodi z začasnimi registri, RISC-V loči 19 registrov v dve skupini:
  - x5−x7 (t0-t2) in x28−x31 (t3-t6): začasni registri, ki jih pri klicu procedure klicani podprogram ne ohrani,
  - x8 (s0/fp), x9 (s1) in x18−x27 (s2-s11): registri, ki jih je treba shraniti pri klicu podprograma (če se uporablja, jih klicani program shrani in obnovi).
- S tem preprostim dogovorom se zmanjša ‘razlivanje’ registrov.
  - V primeru zgoraj, ker klicatelj ne pričakuje, da bosta registra x5 in x6 ohranjena po klicu procedure, lahko prištedimo dva ukaza store in dva ukaza load. Še vedno pa moramo shraniti in obnoviti x20 - klicani program mora domnevati, da klicatelj potrebuje njegovo vrednost.

Procedure-listi in gnezdene procedure

- Procedura, ki ne kliče nobene druge (niti printf), se imenuje list (leaf)
  - Taki načelno ni treba shraniti povratnega naslova, ki je v x1(ra), na sklad, saj ga ne bo prepisal povratni naslov kake druge funkcije
- Kadar podprogram kliče drug podprogram, ali celo samega sebe (rekurzija), so stvari malo bolj zapletene.
  - Npr.: glavni program kliče proceduro A z argumentom 3, (x10 ← 3) in z uporabo jal x1, A.
  - Predpostavimo, da procedura A kliče proceduro B prek jal x1, B z argumentom 7, prav tako postavljenim v x10.
  - Ker A še ni končal svoje naloge, pride do konflikta glede uporaba registra x10.
  - Podobno obstaja konflikt glede povratnega naslova v registru x1, saj je v x1 zdaj povratni naslov za B.
  - Če ničesar ne storimo, se zaradi tega procedura A ne bo mogla vrniti k klicatelju.

```
        addi x10, x0, 3         # x10 = 3
        jal x1, A   #

A:      ...
        addi x10, x0, 7
        jal x1, B
```

- Rešitev je, da potisnemo vse druge registre, ki morajo biti ohranjeni, na sklad, tako kot smo storili s shranjenimi registri.
  - Klicatelj potisne vse registre argumentov (x10–x17) ali začasne registre (x5-x7 in x28-x31), ki so potrebni po klicu.
  - Klicani potisne register x1 s povratnim naslovom in vse 'saved' registre (x8-x9 in x18-x27), ki jih uporablja.
  - Skladovni kazalec sp se prilagodi tako, da upošteva število registrov, postavljenih na sklad.
  - Po vrnitvi se registri obnovijo iz pomnilnika in skladovni kazalec je ustrezno postavi.

Okvir procedure (frame)

- Kazalec na okvir (frame pointer, fp) si zapomni stanje kazalca na sklad (sp) pred shranjevanjem argumentnih registrov
  - sp se bo kasneje spreminjal
  - zato je spremenljivke lažje referencirati preko fp (ki se tekom procedure ne spreminja)
  - pri RISC-V temu služi x8 (fp)
    - a C-prevajalnik ga uporablja le, če se sp spreminja znotraj procedure
    - Po dogovoru o klicih je velikost okvira poravnana na 16 bajtov

Okvir znotraj sklada →

SP →	Lokalna polja in strukture (če jih je kaj)
	Shranjeni 'shranjeni' (saved) registri (če jih je kaj)
	Shranjen povratni naslov
FP →	Shranjeni argumentni registri (če jih je kaj)

- Spremenljivka v jeziku C je na splošno lokacija v pomnilniku, njena interpretacija pa je odvisna od njene vrste in razreda shranjevanja (storage class).
  - Primeri tipov vključujejo cela števila in znake.
  - C ima dva razreda shranjevanja: automatic in static.
  - Avtomatske spremenljivke so lokalne za proceduro in se zavržejo, ko se procedura konča.
  - Statične spremenljivke pa se ohranijo.
  - Spremenljivke, deklarirane zunaj vseh funkcij (torej globalne), veljajo za statične, tako kot vse spremenljivke, eksplicitno deklarirane kot static. Ostale so avtomatske.
  - Za poenostavitev dostopa do statičnih podatkov nekateri prevajalniki RISC-V rezervirajo register x3 za uporabo kot globalni kazalec ali gp (global pointer).

Globalni kazalec gp je torej register, ki kaže na statično območje.

Kateri registri se ohranijo pri klicu procedure?

Se ohranijo	Se ne ohranijo
Shranjeni registri: x8-x9, x18-x27	Začasni registri: x5-x7, x28-x31
Kazalec na sklad: x2 (sp)	Argumentni registri: x10-x17
Kazalec na okvir: x8 (fp)	
Povratni naslov: x1 (ra)	

Poleg tega se na sklad shranjuje tudi lokalne spremenljivke, ki so prevelike za v registre, npr. lokalna polja in strukture. Del sklada, ki hrani shranjene registre in lokalne spremenljivke procedure, se imenuje okvir procedure.

- RISC-V teži k temu, da pogoste primere skuša izvesti hitro, zato je možno veliko funkcij izvesti z 8 argumentnimi registri, 12 shranjenimi reg. in 7 začasnimi, torej brez uporabe pomnilnika.
  - Kadar pa je parametrov več kot 8, jih klicatelj porine na sklad tik nad kazalcem na okvir (fp).
- Pri rekurzivnih funkcijah pa je sicer vedno tudi možnost iterativne izvedbe

Uporaba registrov pri klicih procedur:

Register	Uporaba	Ali se mora ob klicu ohraniti?
x0	0	
x1 (ra)	povratni naslov (return address)	da
x2 (sp)	kazalec na sklad (stack pointer)	da
x3 (gp)	global pointer	da
x4 (tp)	thread pointer	da
x5-x7 (t0-t2)	začasni (temporary) registri	ne
x8 (fp/s0), x9 (s1)	shranjeni* (saved) registri	da
x10, x11 (a0, a1)	argumenti/rezultati	ne
x12-x17 (a2-a7)	argumenti	ne
x18-x27 (s2-s11)	shranjeni* (saved) registri	da
x28-x31 (t3-t6)	začasni (temporary) registri	ne

* v smislu, da morajo biti shranjeni, če se jih uporablja, in nato obnovljeni

Primer 1: Procedura-list

```
fun1(int a, int b);
int main() {   // funkcija, ki kliče drugo funkcijo
   fun1( 5, 6);
   return 0;
}
fun1(int a, int b) {  // ta funkcija je procedura-list
   return(a+b);
}
```

Če program prevedemo z *riscv32-unknown-elf-gcc -static -o prog1 prog1.c* in pogledamo disassembly, ki ga izpiše *riscv32-unknown-objdump -d prog1* (-d ... disassembly), vidimo:

```
<main>:
   1018c:       ff010113      addi    sp,sp,-16       # Kazalec na sklad se zmanjša za 4 besede (za manj se ne sme)
   10190:       00112623      sw      ra,12(sp)       # Tudi funkcija main shrani povratni naslov na sklad,
   10194:       00812423      sw      s0,8(sp)        #   shrani staro vrednost x8(s0 oz. fp)
   10198:       01010413      addi    s0,sp,16        #   in nastavi fp (frame pointer) na začetek sklada
   1019c:       00600593      li      a1,6            # 2. argument <- 6 (x11)
   101a0:       00500513      li      a0,5            # 1. argument <- 5 (x10)
   101a4:       01c000ef      jal     ra,101c0 <fun1>    # Klic funkcije fun1, v x1(ra) se shrani povratni naslov 101a8
   101a8:       00000793      li      a5,0            # a5 <- 0 (main vrne 0)
   101ac:       00078513      mv      a0,a5           # a0 <- a5 (=0)
   101b0:       00c12083      lw      ra,12(sp)       # x1(ra) <- stara vrednost
   101b4:       00812403      lw      s0,8(sp)        # x8(s0/fp) <- stara vrednost
   101b8:       01010113      addi    sp,sp,16        # Kazalec na sklad se vrne v prejšnje stanje
   101bc:       00008067      ret

<fun1>:
   101c0:       fe010113      addi    sp,sp,-32       # Kazalec na sklad se zmanjša za 8 besed
   101c4:       00812e23      sw      s0,28(sp)       # fp <- sp + 28 (1. lokacija (slot) sklada)
   101c8:       02010413      addi    s0,sp,32        # fp <- sp + 32 (začetna vrednost sp)
   101cc:       fea42623      sw      a0,-20(s0)      # shrani 1. argument (x10(a0)=5) na sklad (lokacija 5)
   101d0:       feb42423      sw      a1,-24(s0)      # shrani 2. argument (x11(a1)=6) na sklad (lokacija 6)
   101d4:       fec42703      lw      a4,-20(s0)      # a4 <- 5
   101d8:       fe842783      lw      a5,-24(s0)      # a5 <- 6
   101dc:       00f707b3      add     a5,a4,a5        # a5 <- 5 + 6
   101e0:       00078513      mv      a0,a5           # x(10)a0 <- a5 (=11)
   101e4:       01c12403      lw      s0,28(sp)       # fp <- stara vrednost registra (iz 1. slota sklada)
   101e8:       02010113      addi    sp,sp,32        # Kazalec na sklad se vrne v prejšnje stanje
   101ec:       00008067      ret                     # = jalr x0, 0(x1), vrnitev v klicoči program
```

Primer 2: Procedura, ki kliče drugo proceduro

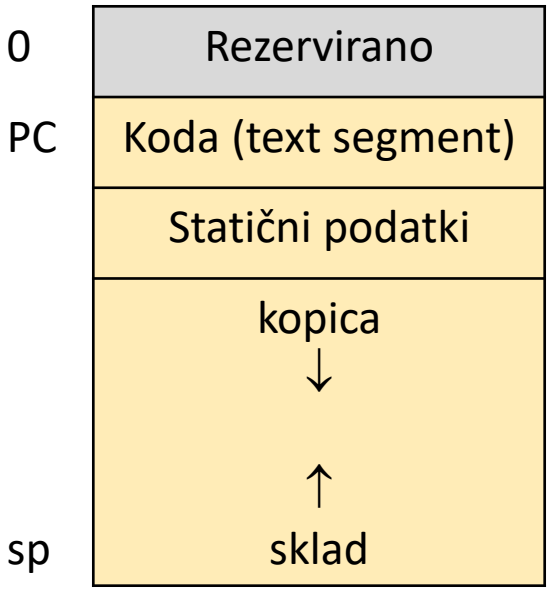
```
#include <stdio.h>
int fun1(int a, int b);
int main() {
    fun1( 5, 6);
    return 0;
}
int fun1(int a, int b) {
    int c;
    c = a+b;
    printf("%d", c);
    return(c);
}
```

```
<main>:
...
   101a4:      01c000ef          jal         ra,101c0 <fun1>  # povratni naslov ra <- 101a8
...

<fun1>:
   101c0:      fd010113          addi        sp,sp,-48      # malo večji okvir (3*16B)
   101c4:      02112623          sw          ra,44(sp)      # povratni naslov (101a8) se tu shrani na sklad, ker ga JAL prepiše!
   101c8:      02812423          sw          s0,40(sp)
   101cc:      03010413          addi        s0,sp,48
   101d0:      fca42e23          sw          a0,-36(s0)
   101d4:      fcb42c23          sw          a1,-40(s0)
   101d8:      fdc42703          lw          a4,-36(s0)
   101dc:      fd842783          lw          a5,-40(s0)
   101e0:      00f707b3          add         a5,a4,a5
   101e4:      fef42623          sw          a5,-20(s0)
   101e8:      fec42583          lw          a1,-20(s0)
   101ec:      000257b7          lui         a5,0x25
   101f0:      07878513          addi        a0,a5,120 # 25078 <__clzsi2+0x4c>
   101f4:      1d4000ef          jal         ra,103c8 <printf>    # ra je zdaj 101f8!
   101f8:      fec42783          lw          a5,-20(s0)
   101fc:      00078513          mv          a0,a5
   10200:      02c12083          lw          ra,44(sp)      # obnovi s sklada povratni naslov 101a8
   10204:      02812403          lw          s0,40(sp)
   10208:      03010113          addi        sp,sp,48
   1020c:      00008067          ret
```

Kopica

- Nekatere podatkovne strukture (npr. povezani seznami ali drevesa) se med tekom programa glede na podatke lahko zelo spreminjajo
  - zato jih je težko umestiti med statične podatke, saj ni jasno vnaprej, koliko prostora bodo zasedli v (glavnem) pomnilniku
  - segment za dinamično alokacijo takih podatkov se imenuje *kopica* (*heap*)
    - običajno raste iz druge smeri kot sklad, tako da se medsebojno približujeta
    - malloc() je C-funkcija za dinamično alokacijo (zadaj je sistemski klic, npr. sbrk na Linuxu), free() pa za sprostitev zaseženega pomnilnika
    - C-program sam nadzoruje dodeljevanje pomnilnika, kar lahko vodi tudi do problemov (npr. odtekanje ('memory leak'), ali pa uporaba kazalca po sprostitvi ('dangling pointer'))
    - Java se temu izogne tako, da uporablja samodejno alokacijo pomnilnika in 'garbage collection'



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 operande
  - 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**

$$OP3 \leftarrow OP2 + OP1$$

    - operandi so običajno v registrih

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

$$OP2 \leftarrow OP2 + OP1$$

- **1-operandni**
  - akumulator

$$AC \leftarrow AC + OP1$$

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

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

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

  - 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?

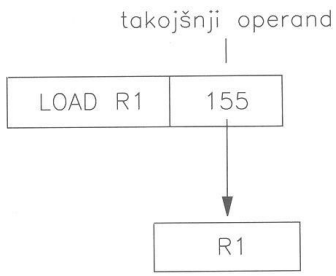
- **Lokacija operandov**
  - registri CPE
  - GP
  - (registri krmilnika V/I naprave)

- 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**
    - en operand v registru, drugi *lahko* v pomnilniku
  - **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)

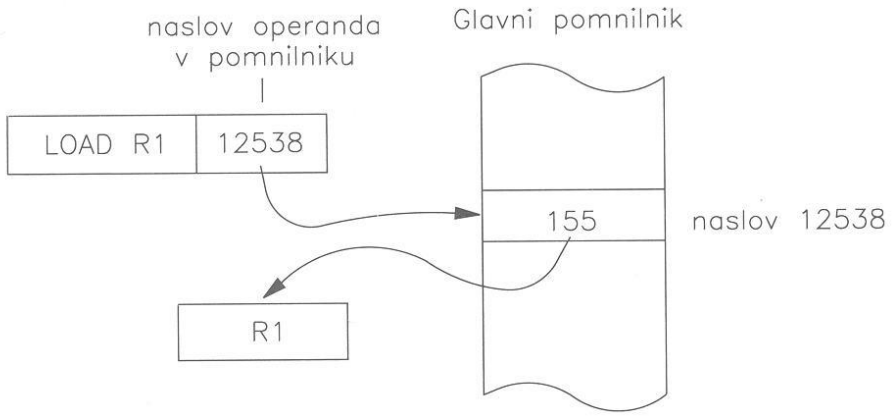


2. **Neposredno naslavljanje** (direct addressing)

- 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)



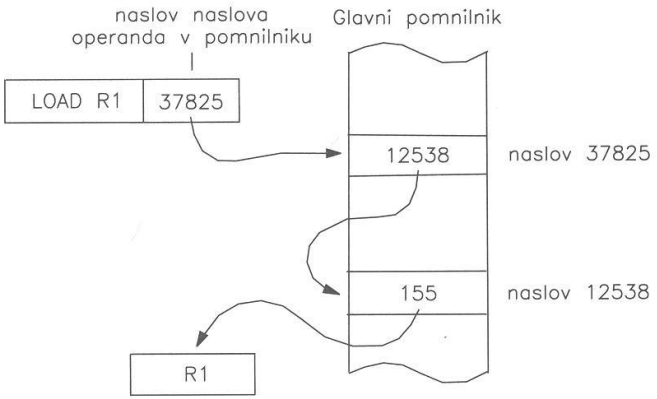
- Težave:
  - velik naslovni prostor → dolg naslov → dolgi ukazi
  - povečanje pom. prostora → drugačni ukazi → nezdružljivost za nazaj
  - primeri, ko operand ni na stalnem naslovu

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

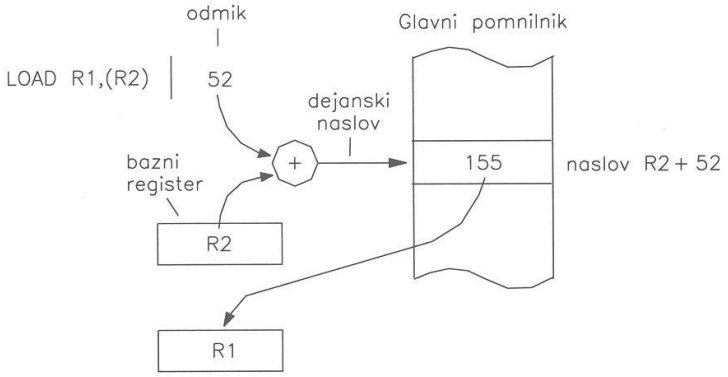
Posredno naslavljanje:

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 $\leftarrow$ 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. **Prenosi podatkov** (data transfer)
   - izvor, ponor
   - v resnici gre za kopiranje
   - Običajni **mnemoniki**:
     - LOAD: GP $\rightarrow$ R
     - STORE: R $\rightarrow$ GP
     - MOVE: R $\rightarrow$ R ali GP $\rightarrow$ GP
     - PUSH: GP ali R $\rightarrow$ Sklad
     - POP (PULL): Sklad $\rightarrow$ GP ali R
   - tudi CLEAR in SET

2. 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

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 (unconditional 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:

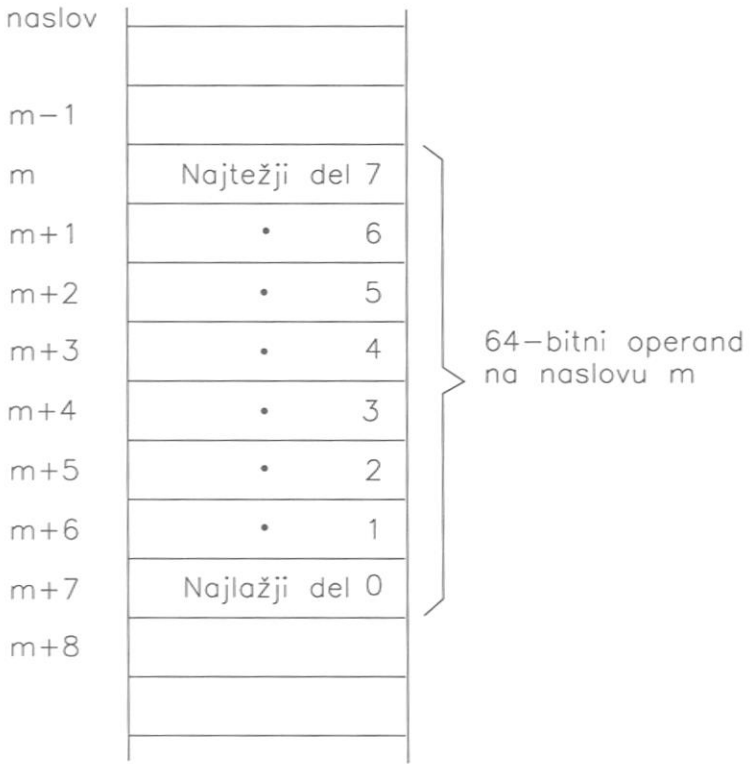
1. **bit**
   - v višjih jezikih jih običajno ni
   - koristno pri sistemskih operacijah
2. **znak**
   - običajno 8-bitni ASCII
   - več znakov tvori **niz** (string)
3. **celo število**
   - predznačeno ali nepredznačeno
   - dolžine 8, 16, 32, 64 bitov
4. **realno število**
   - št. v plavajoči vejici (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:

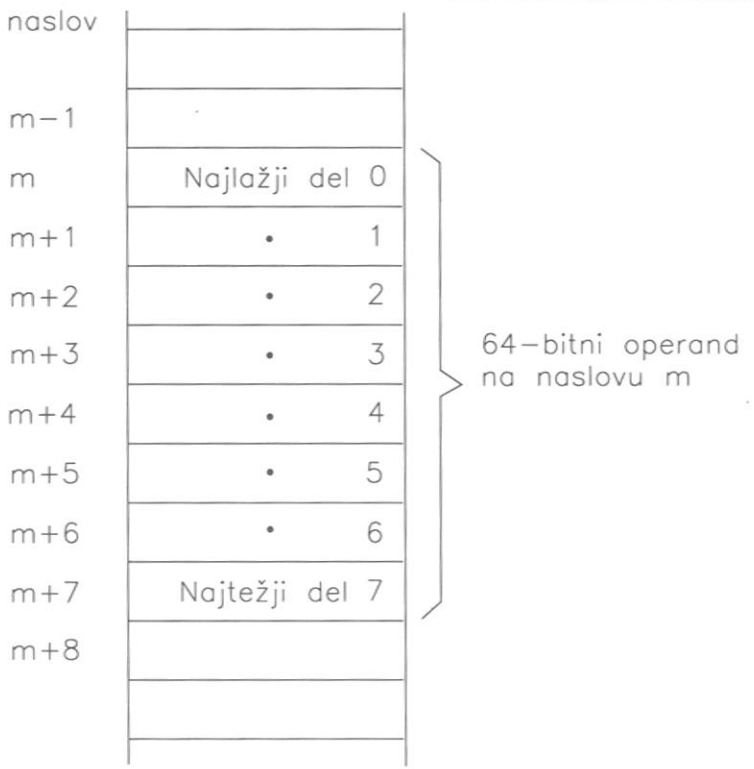
  8 Bajt (byte)
  16 Polovična beseda (halfword)
  32 Beseda (word)
  64 Dvojna beseda (double word)
  128 Štirikratna beseda (quad word)

  - 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



a) Pravilo debelega konca (Big Endian)



b) Pravilo tankega konca (Little Endian)

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

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

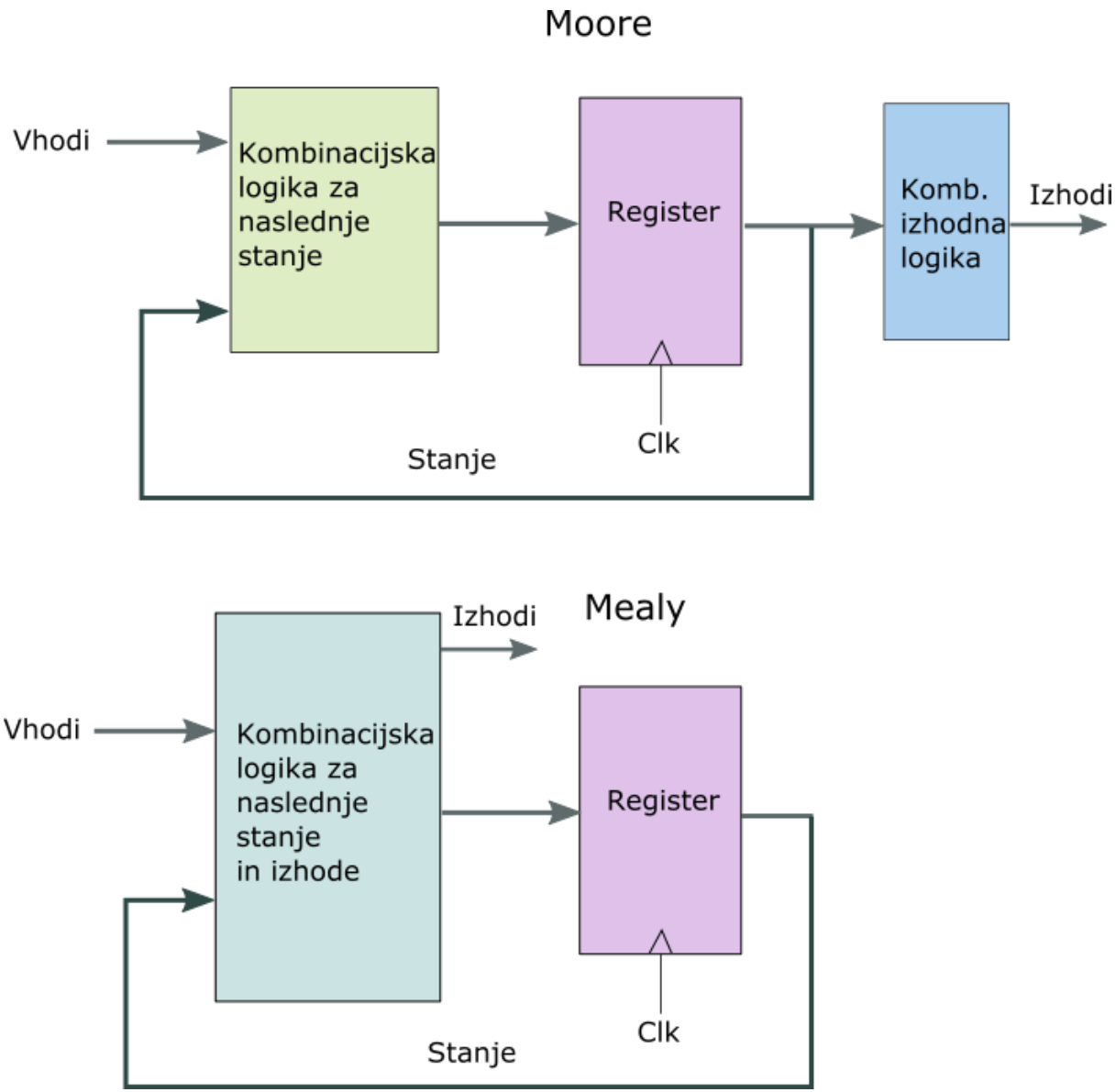
- Večina ukazov se izvrši v enem ciklu CPE
  - lažja real. cevovoda
- Registrsko-registrska zasnova (load/store)
  - zaradi zahteve 1
- Ukazi realizirani s trdo ožičeno logiko
  - ne mikroprogramsko
- Malo ukazov in načinov naslavljanja
  - hitrejše in enostavnejše dekodiranje in izvrševanje
- Enaka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



- Vzemimo primer dvobitnega števca, ki periodično šteje od 0 do 3.
  - Če vzamemo 2 D-flip-flopa, bo nižji imel na vhodu $D_0 = Q_0'$, višji pa $D_1 = Q_0$ xor Q_1.
  - Imamo torej dvobitni register in kombinacijsko logiko iz enega invertorja in vrat xor.
  - Zakasnitev obojih vrat določa najmanjšo možno periodo
    - Seveda je treba upoštevati še vzpostavitveni in držalni čas
  - Tako je maksimalna frekvenca navzgor omejena

Delovanje CPE

1. **Dobava ukaza iz pomnilnika** (fetch)
2. **Izvrševanje ukaza**
   a) dekodiranje ukaza (in branje registrov)
   b) izvedba ALE operacije (ali izračun naslova)
   c) prenos operandov v CPE iz pomnilnika (po potrebi)
   d) shranjevanje rezultata (po potrebi) – pisanje v register
   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 (načelno) 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

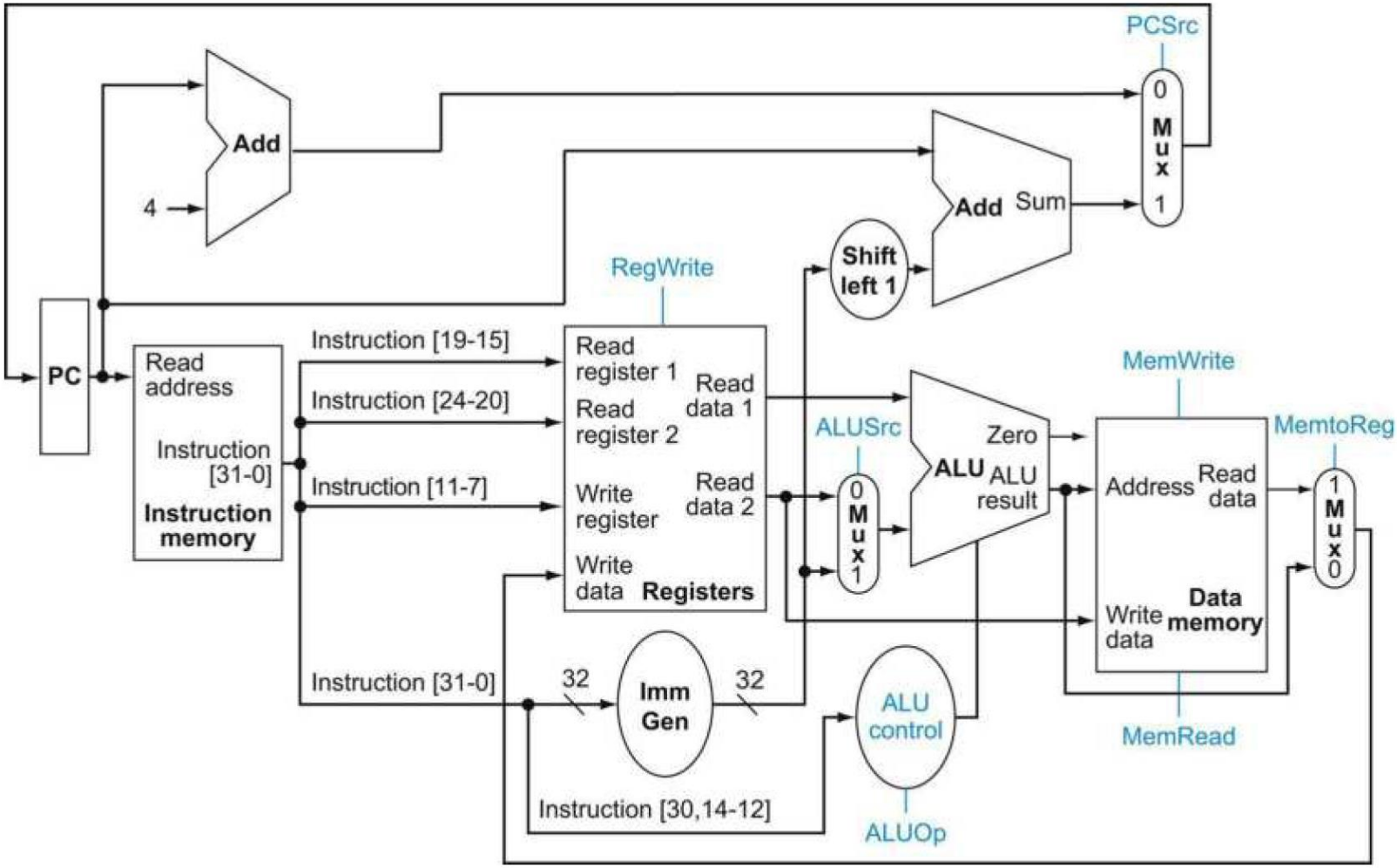
Načini implementacije CPE

- Različni načini (sinhronske) implementacije:
  1. Celoten ukaz v 1 periodi ure (‘single-cycle’) – ***enocikelni procesor***
     - dolga perioda!
  2. Ukaz v več periodah ure (‘multi-cycle’) – ***večcikelni procesor***, ukazi pa se ne prekrivajo
     - Kontrolna enota vsebuje *končni avtomat* (FSM – Finite State Machine), ki nadzoruje dogajanje
     - Avtomat prehaja med stanji ob fronti ure
  3. Ukaz v več periodah ure, a se ukazi časovno delno prekrivajo (cevovod - ‘pipeline’) – ***cevovodni procesor***

Implementacija enocikelnega procesorja

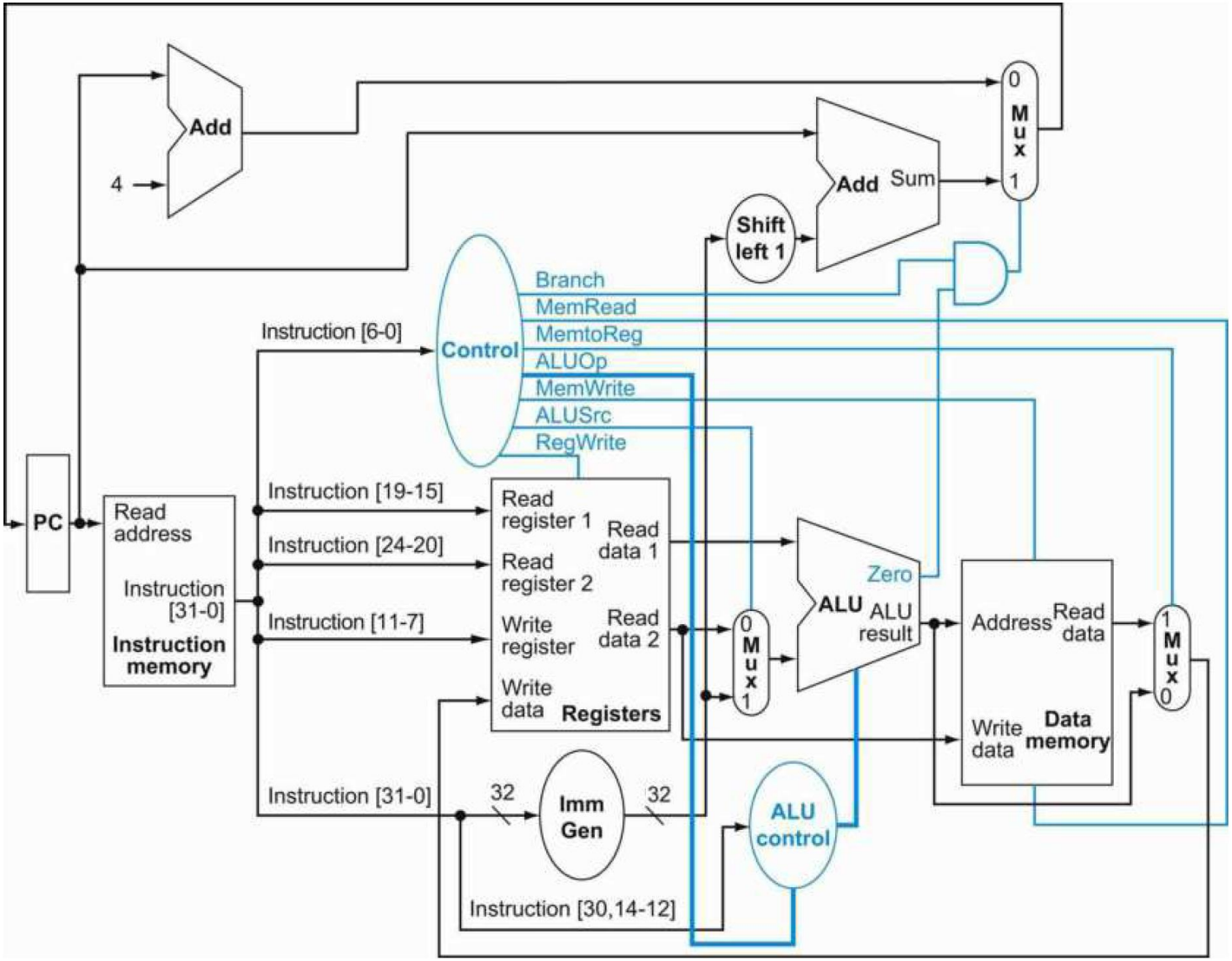
- Preprost primer:
  - implementacija ukazov:
    - lw,
    - sw,
    - beq,
    - add, sub, and, or (format R)
- ALE operacija se izvrši nad dvema 32-bitnima vhodnima operandoma *x* in *y*
  - Rezultat je 32-bitni izhod in bit *Zero*
  - Signali ALEop (4 biti) pridejo iz kontrolne enote
    - ta jih tvori iz bitov operacijske kode (in podaljškov *funct*) ukaza

Podatkovna enota enocikelnega procesorja



Patterson, Hennessy: Computer Organization and Design RISC-V Edition: The Hardware Software Interface (Morgan Kaufmann)

Enocikelni procesor (z dodano kontrolno enoto)



ALE s 4-bitno kontrolno kodo

- majhno kontrolno vezje na osnovi funct7, funct3 in 2-bitne kode ALUOp:
  - 00: add (za load in store)
  - 01: sub and test if zero (za beq)
  - 10: določena glede na funct

Opkoda	ALUOp	Operacija	funct7	funct3	ALE operacija	ALU control
lw	00	load word	xxxxxxx	xxx	+	0010
sw	00	store word	xxxxxxx	xxx	+	0010
beq	01	branch if equal	xxxxxxx	xxx	-	0110
(format R)	10	add	0000000	000	+	0010
(format R)	10	sub	0100000	000	-	0110
(format R)	10	and	0000000	111	and	0000
(format R)	10	or	0000000	110	or	0001

- Kakšen je namen vrat IN (AND)?
- Kontrolna enota mora za vsak ukaz določiti operacije, označeno na sliki z modro:
  - ALUsrc, Branch, MemRead, MemWrite, ...
- Kontrolna enota je v primeru enocikelnega procesorja kar preprosta tabela
  - zato jo lahko realiziramo s kombinacijskim vezjem (ali bralnim pomnilnikom)

Ukaz	ALUsrc	MemToReg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp
format R: add, sub, and, or	0	0	1	0	0	0	10
lw	1	1	1	1	0	0	00
sw	1	X	0	0	1	0	00
beq	0	X	0	0	0	1	01

Primer programa:

Naslov	Ukaz	Format	Polja (f3 je funct3)	Strojna koda
0x1000 L7:	lw x6, -4(x9)	I	$imm_{11:0}$ rs1 f3 rd op 111111111100 01001 010 00110 0000011	FFC4A303
0x1004	sw x6, 8(x9)	S	$imm_{11:5}$ rs2 rs1 f3 $imm_{4:0}$ op 0000000 00110 01001 010 01000 0100011	0064A423
0x1008	or x4, x5, x6	R	funct7 rs2 rs1 f3 rd op 0000000 00110 00101 110 00100 0110011	0062E233
0x100C	beq x4, x4, L7	B	$imm_{12,10:5}$ rs2 rs1 f3 $imm_{4:0}$ op 1111111 00100 00100 000 10101 1100011	FE420AE3

Zmogljivost enocikelnega procesorja

- Najdalj traja ukaz LW:
  - PC se naloži po uri (t_{pcq} = propagation delay clock-to-q),
  - Branje ukaza iz pomnilnika (t_{mem}),
  - RF bere ReadData1 (t_{RFread}),
    - obenem se razširja predznak odmika (se: sign-extension) in gre preko mux-a na ALE (t_{se}+ t_{mux}),
  - ALE sešteje bazo in odmik (t_{ALE}),
  - Podatkovni pomnilnik bere iz tega naslova (t_{mem}),
  - Mux MemtoReg izbere ReadData (t_{mux}),
  - Result se mora pojaviti malo (vzpostavitveni čas, setup time, $t_{RFsetup}$) pred fronto ure

$$t_{CPE} = t_{pcq} + t_{mem} + \max(t_{RFread}, t_{se} + t_{mux}) + t_{ALE} + t_{mem} + t_{mux} + t_{RFsetup}$$

    - označeni z rdečo so običajno večji od ostalih - po grobi oceni je perioda malo večja kot

$$2t_{mem} + t_{RFread} + t_{ALE}$$

    - CPI (clocks per instruction) = 1

- Primer: Določi najmanjšo možno periodo ure pri zakasnitvah elementov, podanih v Tabeli.
  - Koliko časa traja program z 10^9 ukazi?

Parameter	Element	Zakasnitev [ps]
t_{pcq}	register propagation clock-to-q	30
t_{se}	razširitev predznaka (sign-extension)	20
t_{mux}	multipleksor	25
t_{ALE}	ALE	200
t_{mem}	branje iz pomnilnika	250
t_{RFread}	branje registrov (register file)	150
$t_{RFsetup}$	vzpostavitev RF	20

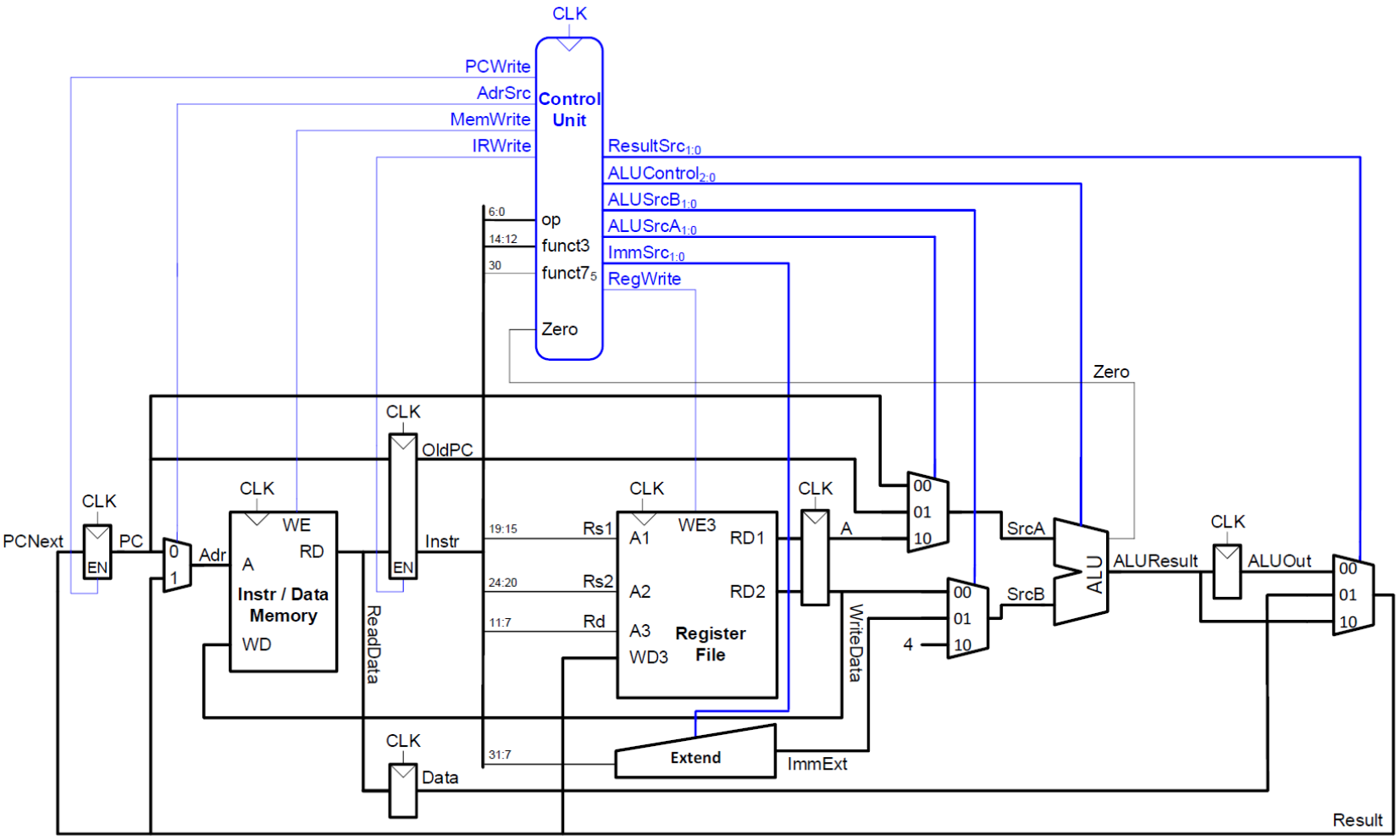
$t_{CPE,min}$ = 30+250+150+200+250+25+20 = 925 ps

Program traja 925 ms.

Implementacija večcikelnega procesorja

- Vsak ukaz potrebuje za izvedbo nekaj period ure
  - v vsaki periodi se izvrši ena podoperacija
- Preprost primer:
  - implementacija ukazov: lw, sw, beq, add, sub, and, or
- V tem primeru dodamo nekaj registrov:
  - Instr (ukazni reg.), Data, ALUOut, OldPC, A, ...

Večcikelni procesor



Harris & Harris: Digital Design and Computer Architecture (Morgan Kaufmann)

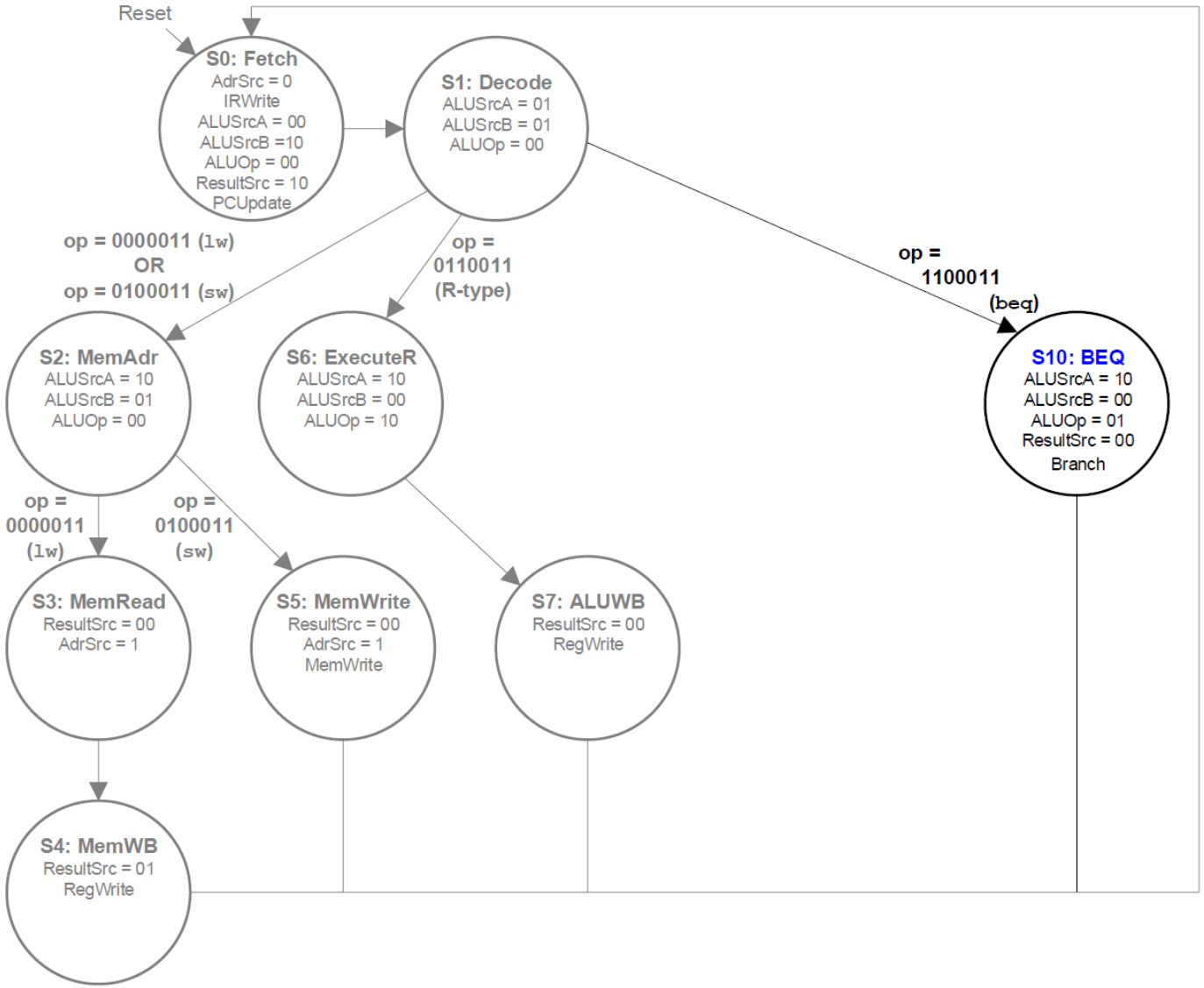
Kontrolna enota večcikelnega procesorja

- 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, FSM)

- 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

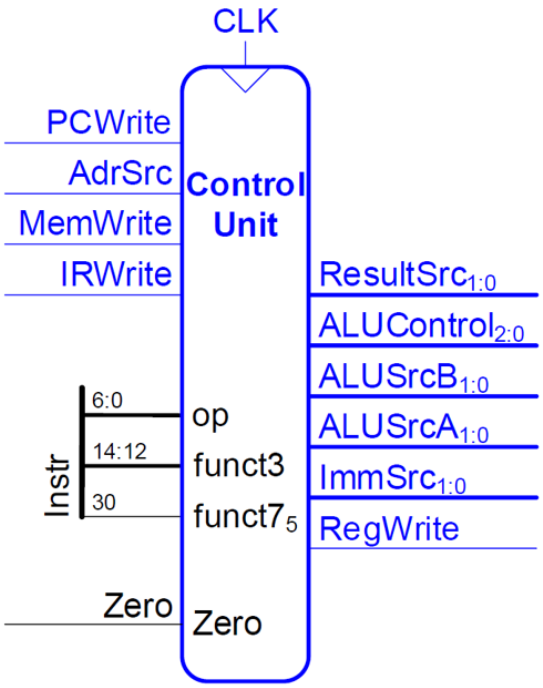
- Končni avtomat realiziramo s kombinacijskimi in sekvenčnimi vezji
  - npr. vrata in flip-flopi
- Najprej potrebujemo popoln DPS

DPS končnega avtomata, ki nadzoruje delovanje večcikelne CPE:

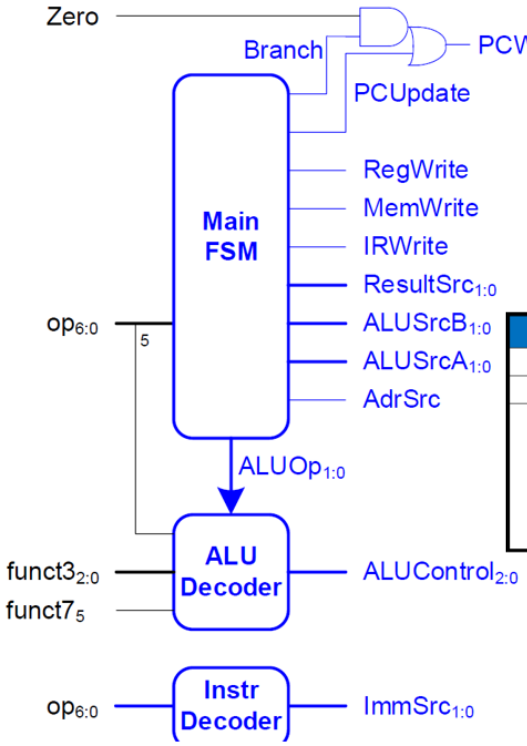


Kontrolna enota

High-Level View



Low-Level View



op	Instruction	ImmSrc
3	`lw`	00
35	`sw`	01
51	R-type	XX
99	`beq`	10

ALUOp	funct3	op5, funct75	Instruction	ALUControl2:0
00	x	x	`lw, sw`	000 (add)
01	x	x	`beq`	001 (subtract)
10	000	00, 01, 10	`add`	000 (add)
	000	11	`sub`	001 (subtract)
	010	x	`slt`	101 (set less than)
	110	x	`or`	011 (or)
	111	x	`and`	010 (and)

ALUControl2:0	Function
000	add
001	subtract
010	and
011	or
101	SLT

- DPS ima 9 stanj
  - 4-bitni register stanja
  - kombinacijsko logiko, ki iz stanja, registra IR in vhodnih signalov tvori naslednje stanje in izhodne signale
- Zaradi preproste zgradbe ukazov lahko precejšen del bitov IR peljemo mimo KE v PE
- Kontrolna enota ima precej izhodnih signalov (do 20)

Realizacija kontrolne enote

- CPE lahko izvršuje ukaze na 2 načina (to vpliva na realizacijo kontrolne enote):
  1. **Trdo ožičena logika (hard-wired logic)**
     - vezje (logična vrata, pomnilne celice, povezave)
     - trdo ožičena pomeni, da so povezave fiksne
     - 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

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**
    - za majhen primer 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

CPI

- Čas izvrševanja različnih ukazov na večcikelnem procesorju je različen
  - vsak ukaz rabi določeno celo št. urinih period, vendar nekateri ukazi ne potrebujejo vseh korakov
  - pri enocikelnem procesorju je trajanje periode ure določeno z najpočasnejšim ukazom, pri večcikelnem pa z najpočasnejšim *korakom ukaza*
    - pri enocikelnem procesorju je CPI vsakega ukaza seveda 1, pri večcikelnem pa več (tipično od 3 do 6)
- Povprečen **CPI** (Clocks Per Instruction) za nek program:

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

  - p_i je relativna pogostost (verjetnost) posamezne vrste ukaza
  - CPI_i je število urinih period za ukaz vrste *i*

- Pogostost posameznih skupin ukazov je precej odvisna tudi od programa, ki ga poganjamo
  - Npr. nek prevajalnik za C uporablja 46% ALE ukazov, 36% ukazov za prenos podatkov in 18% kontrolnih ukazov
  - Če upoštevamo CPI ukazov v spodnji tabeli in predpostavimo, da je število load ukazov trikrat večje od števila store ukazov, dobimo:

$$CPI_{idealni} = 0{,}46*4 + 0{,}36*(0{,}75*5+0{,}25*4) + 0{,}18*3 =$$
$$= 1{,}84 + 1{,}71 + 0{,}54 =$$
$$= \mathbf{4{,}09}$$

Vrsta ukaza	**CPI_i** (število urinih period na ukaz)
Ukazi load	5
Ukazi store	4
Ukazi ALE	4
Skoki	3

- Če za CPI_i vzamemo najmanjše možno število urinih period, dobimo $CPI_{idealni}$, ki ne vključuje izgubljenih urinih period zaradi zgrešitev v predpomnilniku
- Čas izvrševanja pa je v splošnem odvisen tudi od časa za dostop do pomnilnika, ta pa od pogostosti zgrešitev v predpomnilniku (PP)
  - Vsi ukazi potrebujejo en dostop do pomnilnika (za prevzem ukaza), ukazi za prenos podatkov (load in store) pa še enega
  - Povprečno število urinih period pri upoštevanju PP upošteva še povprečno število čakalnih urinih period ($CPEperiode_{čak}$), ki so zaradi zgrešitev v predpomnilniku potrebne pri dostopih do pomnilnika
    - pri vsaki zgrešitvi je določeno število čakalnih urinih period (npr. 10) – temu se reče tudi *zgrešitvena kazen* (K_Z)
  - Povprečno št. period = najmanjše št. period + povprečno št. čakalnih period:

$$CPI_{resnični} = CPI_{idealni} + CPEperiode_{čak}$$

- Povprečno št. čakalnih period = št. pom. dostopov (N) × verjetnost zgrešitve × št. čakalnih period:

$$CPEperiode_{čak} = N*(1-H)*K_Z$$

- Če je verjetnost zadetka (H) v predpomnilniku npr. 95% in $K_Z = 10$, je povprečno število čakalnih urinih period enako N × 0,05 × 10

Število urinih period na ukaz

- 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	5	6
store	2	4	5
ALE (format R)	1	4	4,5
BEQ	1	3	3,5

- Če upoštevamo torej 95% verjetnost zadetka (H=0,95), dobimo:

$CPI_{resnični}$ = 0,46*4,5 + 0,36*(0,75*6+0,25*5) + 0,18*3,5 =

= 2,07 + 2,07 + 0,63 =

= 4,77 = $CPI_{idealni}$ + **0,68**

- Lahko določimo 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
  - npr., pri 2 GHz bi dobili za ta C-prevajalnik 2000/4,77 = 419 MIPS

Kritična pot, ki navzdol omejuje periodo ure, je vedno **med dvema registroma**:

- S_0 (PC+4):

$$t_{CPE,min} = t_{pcq(PC)} + t_{mux} + t_{ALE} + t_{mux} + t_{setup(PC)},$$

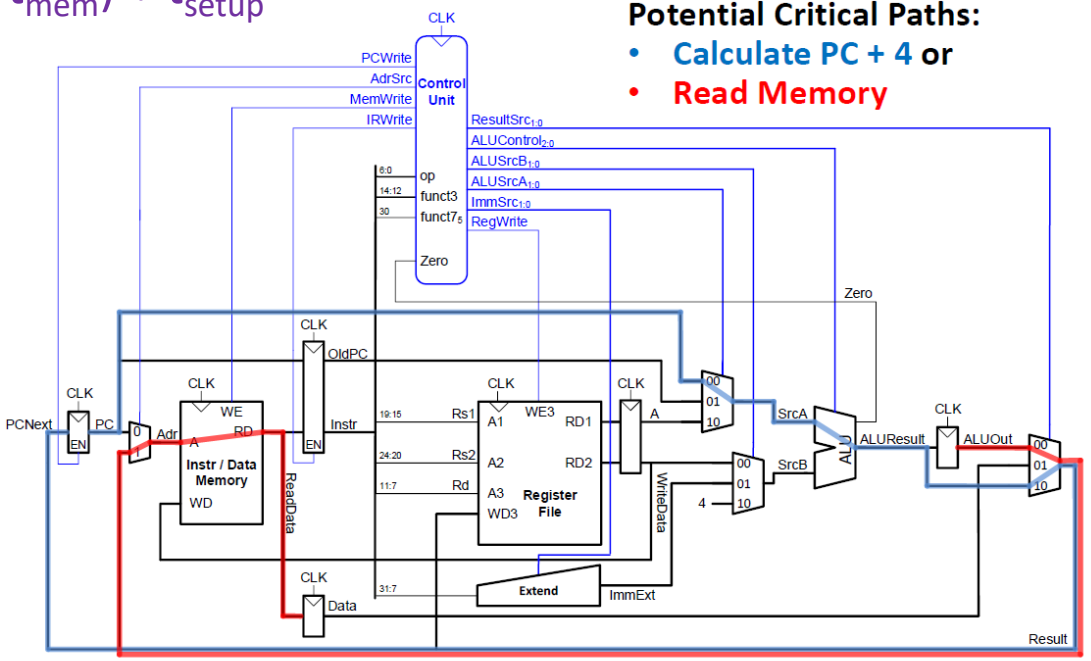
ali

- S_3 (MemRead):

$$t_{CPE,min} = t_{pcq(ALUOut)} + 2t_{mux} + t_{mem} + t_{setup(Data)},$$

Torej:

$$t_{CPE,min} = t_{pcq} + 2t_{mux} + \max(t_{ALE}, t_{mem}) + t_{setup}$$



- Primer:
  a) Kolikšna je najmanjša možna perioda ure, če uporabimo enake elemente (glej Tabelo spodaj), kot pri primeru enocikelnega procesorja?
  b) Koliko časa traja v povprečju en ukaz, če vzamemo primer C-prevajalnika?
  c) Koliko časa se izvaja program z 10^9 ukazi, če vpliva predpomnilnika ne upoštevamo?

- Izračuni:

a) $t_{CPE,min} = 30 + 2*25 + 250 + 20 = 350$ ps

b) $CPI_{idealni} * t_{CPE} = 4{,}09 * 350$ ps $= 1{,}432$ ns

c) Program z $I = 10^9$ ukazi traja:

$I * CPI_{idealni} * t_{CPE} = 1{,}432$ s

Parameter	Element	Zakasnitev [ps]
$t_{pcq(PC)}$	register propagation clock-to-q	30
t_{setup}	vzpostavitev registra	20
t_{mux}	multipleksor	25
t_{ALE}	ALE	200
t_{mem}	branje iz pomnilnika	250
t_{RFread}	branje registrov (register file)	150
$t_{RFsetup}$	vzpostavitev RF	20

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
- Zmogljivost (Performance) je obratno sorazmerna s časom izvajanja programa:

$$Zmog \propto \frac{1}{CPEčas}$$

- Pohitritev (Speed-up) pri primerjavi 2 procesorjev (ali pa stare in nove verzije enega procesorja):

$$S = \frac{Zmog_A}{Zmog_B} = \frac{CPEčas_B}{CPEčas_A}$$

- 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:

$$CPE\ \check{c}as = \frac{\check{S}tevilo\ \ ukazov}{MIPS \cdot 10^6}$$

  - Tudi MIPS ni popolnoma merodajen:
    - odvisen od števila in vrste ukazov
      - pri enostavnejših ukazih je MIPS večji (čeprav jih potrebujemo več)
    - odvisen od programa
    - celo *Meaningless Indication of Processor Speed* ☺

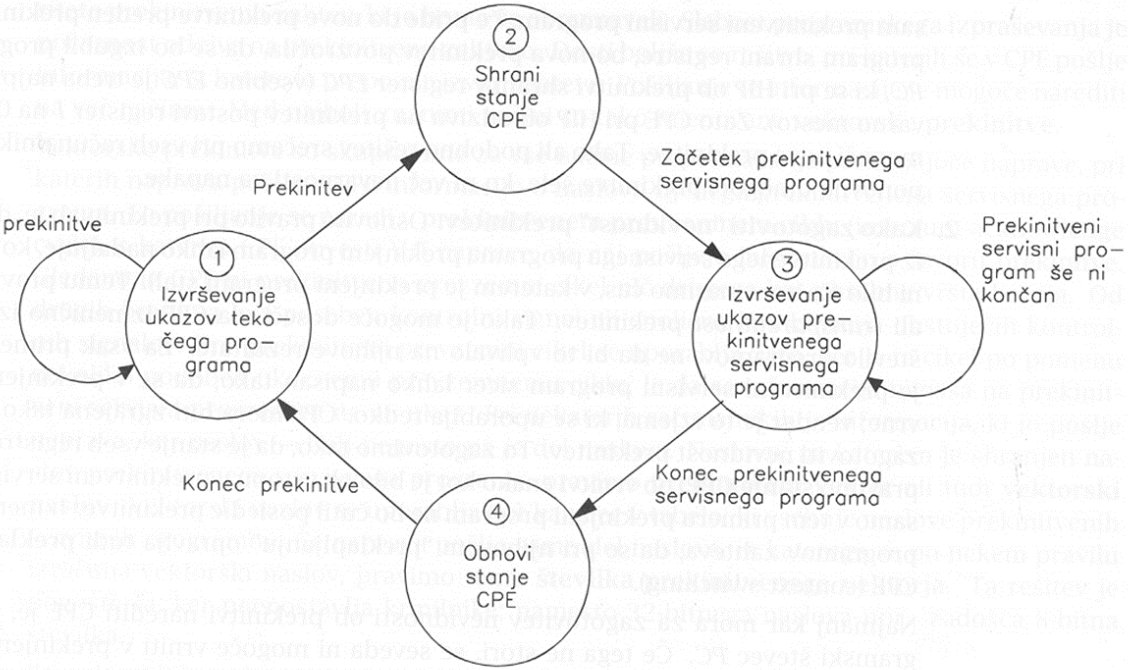
- **MFLOPS** (Million FLoating point Operations Per Second)
  - operacije v plavajoči vejici so si (na različnih računalnikih) bolj podobne kot ukazi
  - ima smisel samo za programe, ki uporabljajo operacije v plavajoči vejici
  - proizvajalci so začeli navajati maksimalno (teoretično) zmogljivost
    - 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

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:

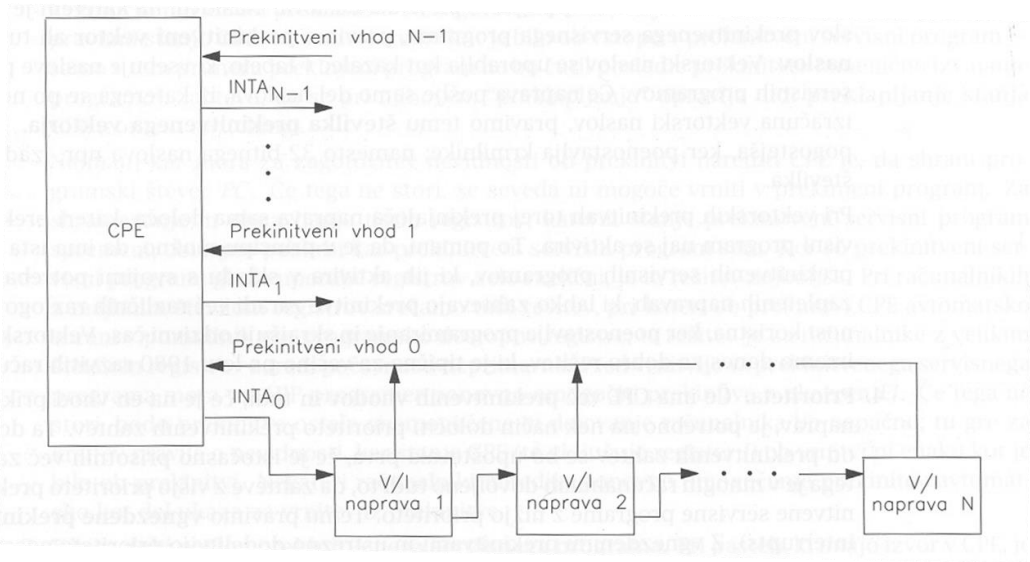
1. **Kdaj CPE reagira na prekinitveno zahtevo**
   - najenostavneje je po izvrševanju tekočega ukaza
     - v tem primeru se mora ohraniti samo stanje programsko dostopnih registrov (*R0-R31, PC, EPC, I*)
   - programer lahko onemogoči odziv CPE na prekinitvene zahteve (bit *I*, ukaza DI in EI)
     - 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

3. 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

7
PARALELIZEM NA NIVOJU UKAZOV

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

- zaporedno izvrševanje

Število ukazov na sekundo je

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

IPS ... Instructions Per Second

f_{CPE} ... frekvenca ure

CPI ... Clocks Per Instruction

IPS lahko povečamo:

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

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

- zato je druga varianta bolj perspektivna

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

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

Ena možnost je paralelno programiranje

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

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

Najpogostejši način je **cevovod** (pipeline)

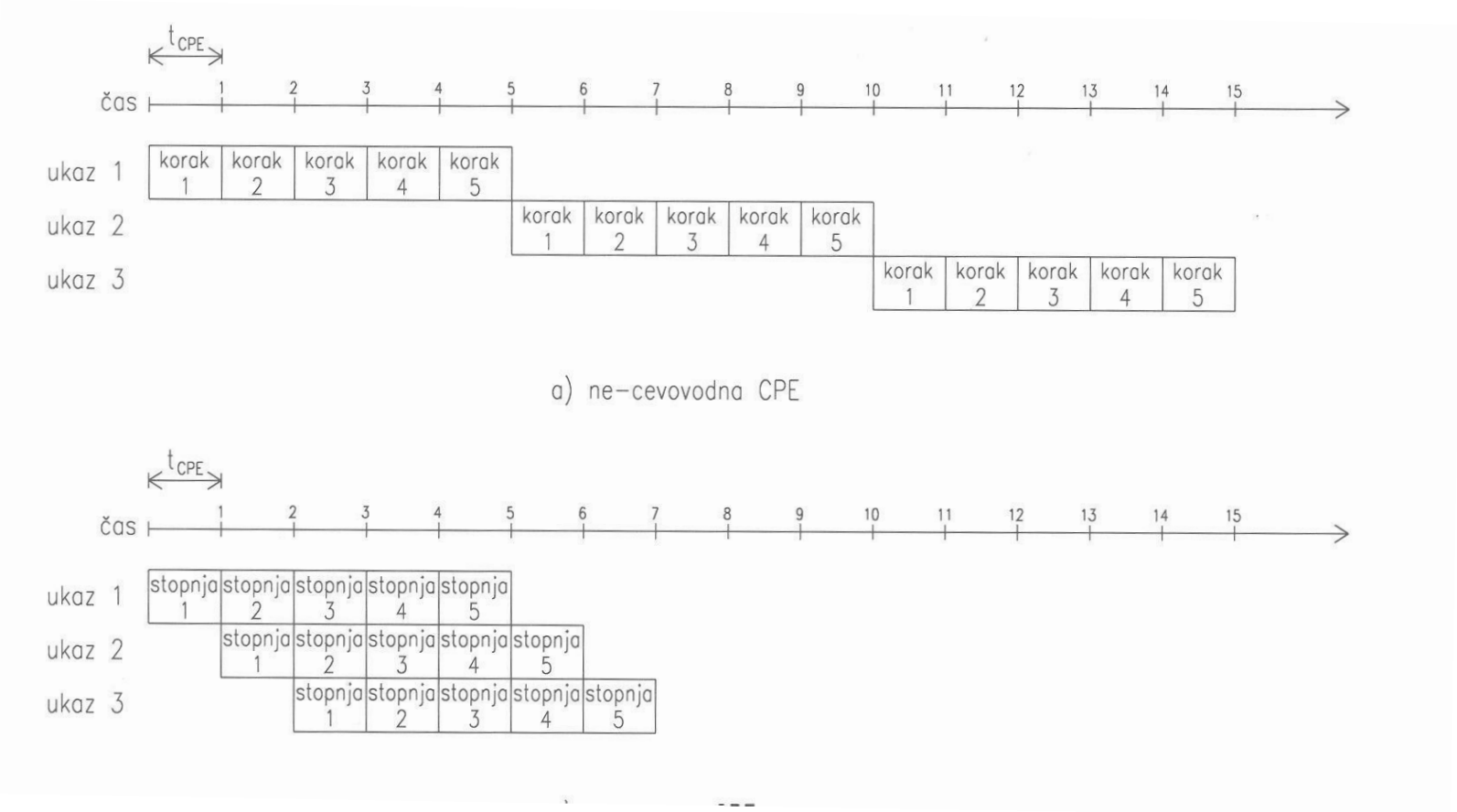
Cevovod - splošno

Cevovod (pipeline)

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

Primer: ne-cevovodna CPE in cevovodna CPE

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



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

Uravnoteženost:

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

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

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

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

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

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

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

Supercevovodni računalniki

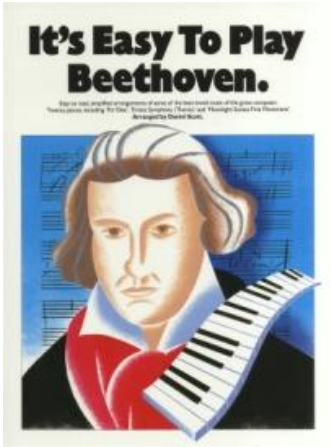
- Intel Pentium 4
  - 20-stopenjski, kasneje 31-stopenjski cevovod
    - želeli so doseči f_{CPE} 10GHz, a je bila poraba prevelika (problemi s hlajenjem, tj. odvajanjem toplote)
    - kasnejši CPU (npr. Core) imajo (le) 14-stopenjski cevovod

BTW: najvišja dosežena frekvenca je okrog 9 GHz

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

Zakaj se ne uporabljajo poljubno dolgi cevovodi?

- pač povečujemo N in višamo hitrost CPU …



Cevovodne nevarnosti

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

3 vrste cevovodnih nevarnosti:

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

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

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

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

Cevovodna podatkovna enota

Cevovodna realizacija je pri računalnikih RISC enostavnejša kot pri CISC

- preprostejši ukazi

Cevovodno CPE si bomo zato ogledali na primeru računalnika RISC

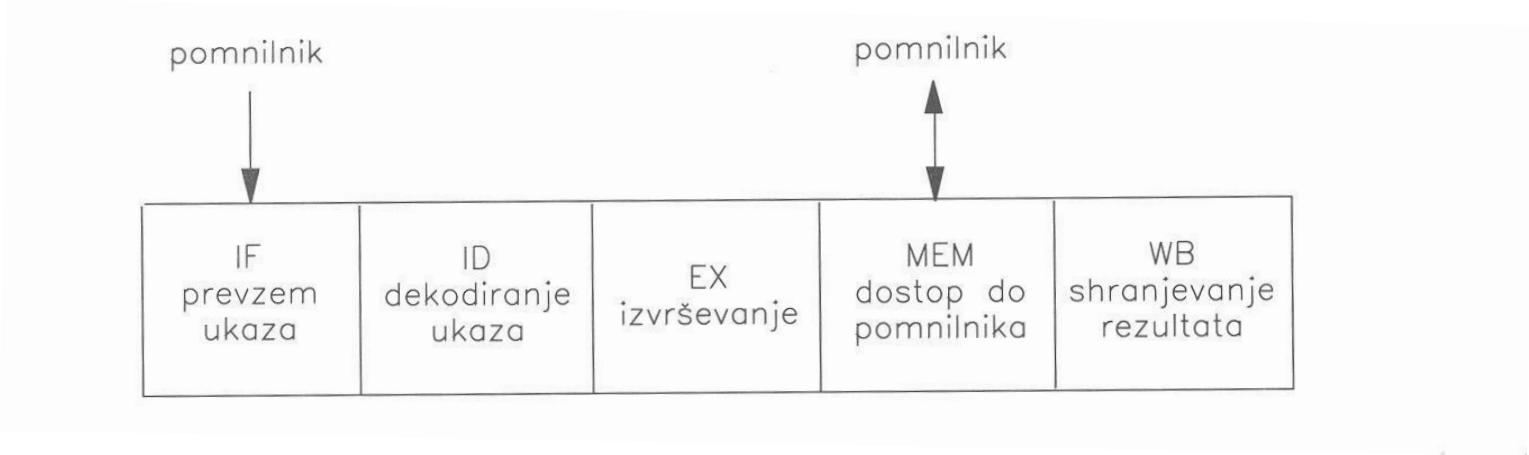
5 korakov izvrševanja ukazov: 5 stopenj cevovoda

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

Vsaka stopnja mora opraviti svoje delo v eni urini periodi

- prej so nekateri koraki potrebovali 2 periodi

Klasična petstopenjska cevovodna podatkovna enota:

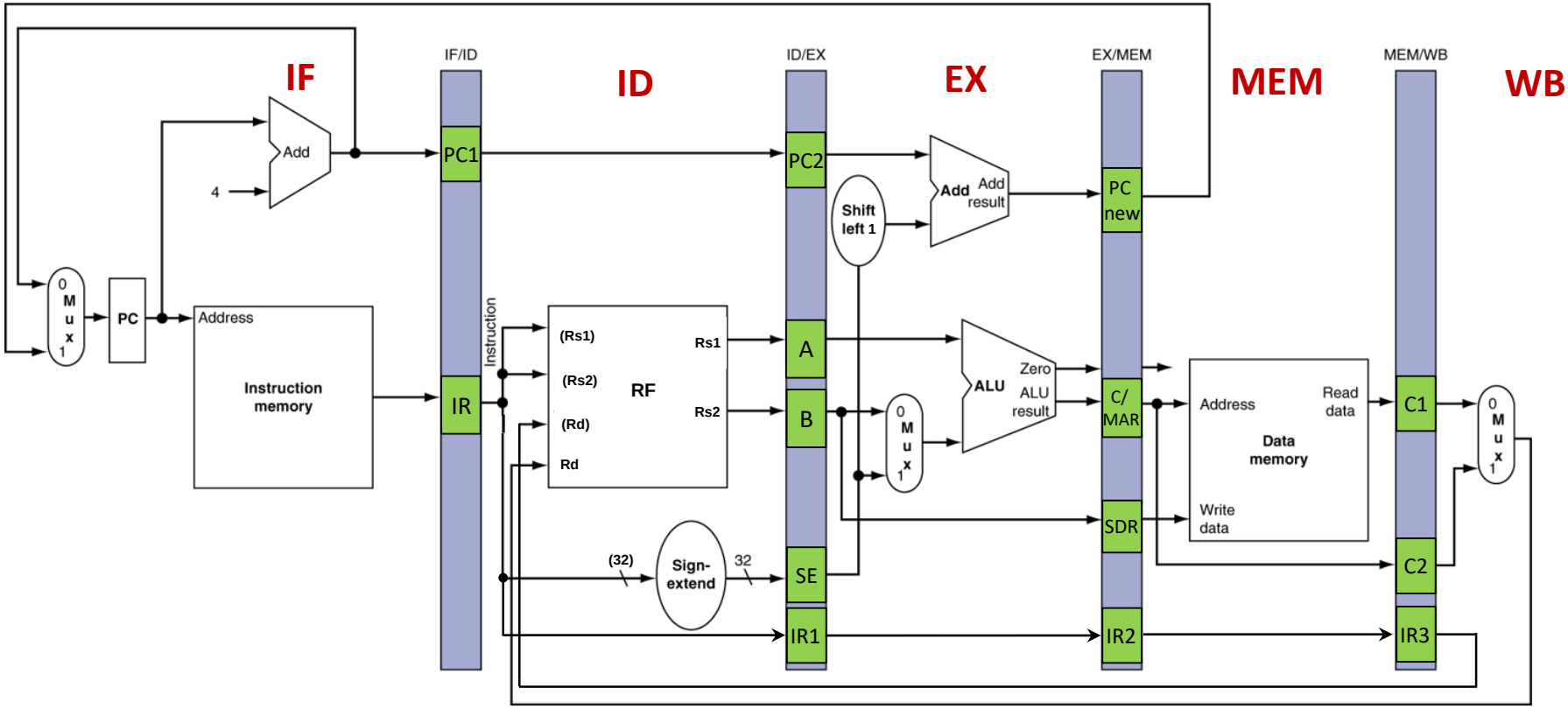


Včasih sta potrebna dva hkratna dostopa do pomnilnika

Cevovodni procesorji imajo zato 2 predpomnilnika:

- ukazni
- operandni

Cevovodna podatkovna enota



1. Stopnja IF

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

PC $\leftarrow$ PC + 4 ali newPC

PC1 $\leftarrow$ PC+4

2. Stopnja ID (za load/store)

PC2 $\leftarrow$ PC1

A $\leftarrow$ Rs1

B $\leftarrow$ Rs2

SE $\leftarrow$ se(imm)

3. Stopnja EX

prehod v to stopnjo se imenuje **izstavitev ukaza** (instruction issue) - tukaj se ukaza več ne da na preprost način izničiti (v IF in ID ni problema)

Delovanje stopnje EX je odvisno od vrste ukaza:

Ukazi za prenos podatkov

MAR $\leftarrow$ A + se(imm)

SDR $\leftarrow$ B

ALE ukazi

SDR $\leftarrow$ B

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

Skoki

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

4. Stopnja MEM

load

IR3 ← IR2

C1 ← M[MAR] branje iz operandnega PP

store

IR3 ← IR2

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

Ostali ukazi

IR3 ← IR2

C1 ← C (zaradi WB)

5. Stopnja WB

ALE ukazi, load:

Rd ← C1

Idealen potek izvrševanja ukazov v cevovodu:

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

CEVOVODNE NEVARNOSTI

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

Programsko odpravljanje cevovodnih nevarnosti

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

Strukturne nevarnosti

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

Podatkovne nevarnosti

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

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

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

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

Rešitev 1:
Cevovodna zaklenitev (pipeline interlock)

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

1. Nevarnost v stopnji EX:

   EX/MEM.rd = ID/EX.rs1

   EX/MEM.rd = ID/EX.rs2

2. Nevarnost v stopnji MEM:

   MEM/WB.rd = ID/EX.rs1

   MEM/WB.rd = ID/EX.rs2

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

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

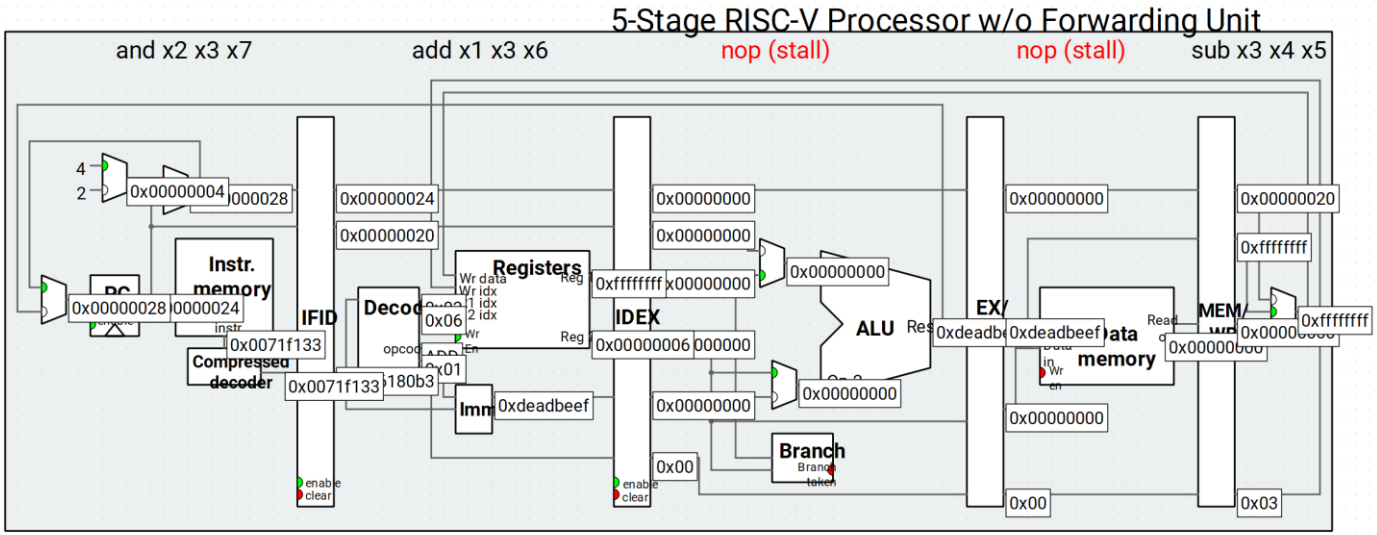
urina perioda	IF (IM)	ID (IR)	EX (IR1)	MEM (IR2)	WB (IR3)
1	addi				
2	sub	addi			
3	add	sub	addi		
4	and	add	sub	addi	
5	and	add	nop (stall)	sub	addi
6	and	add	nop (stall)	nop (stall)	sub
7	xor	and	add	nop (stall)	nop (stall)
78	or	xor	and	add	nop (stall)

6. perioda

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

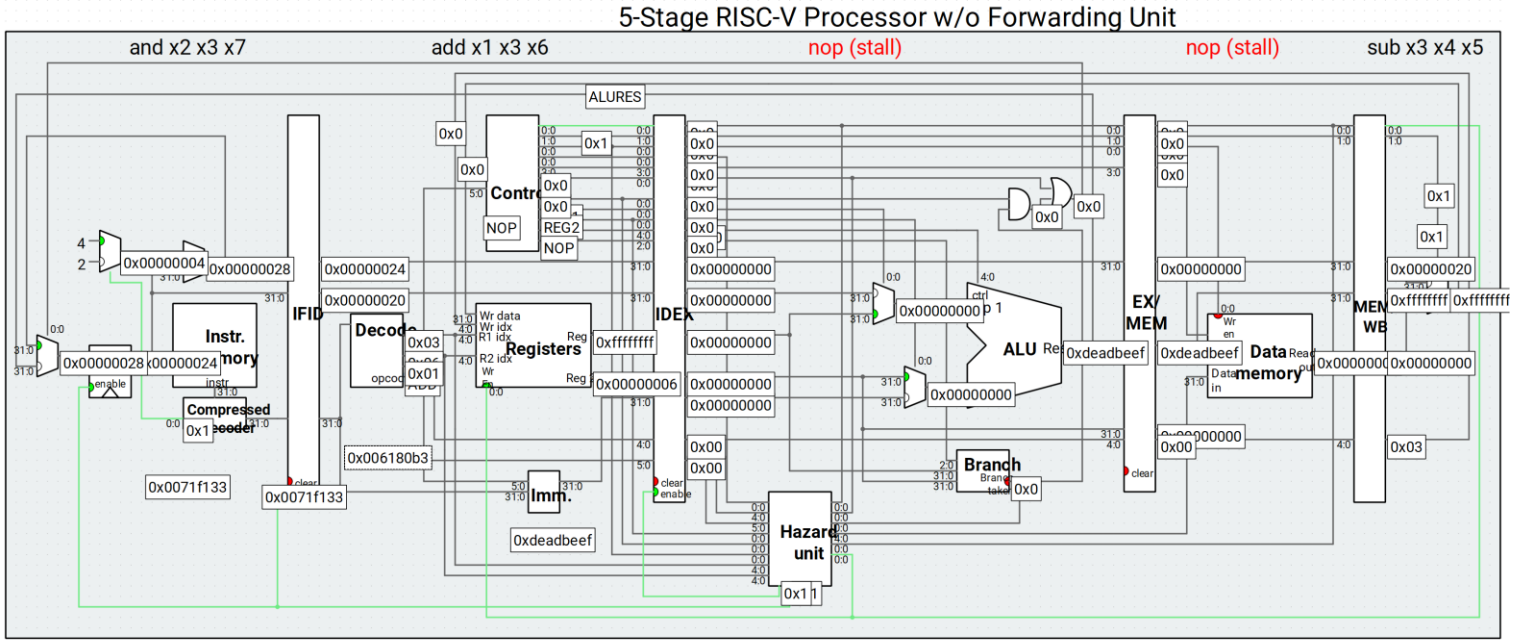
Simulator Ripes:

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



Layout: Standard

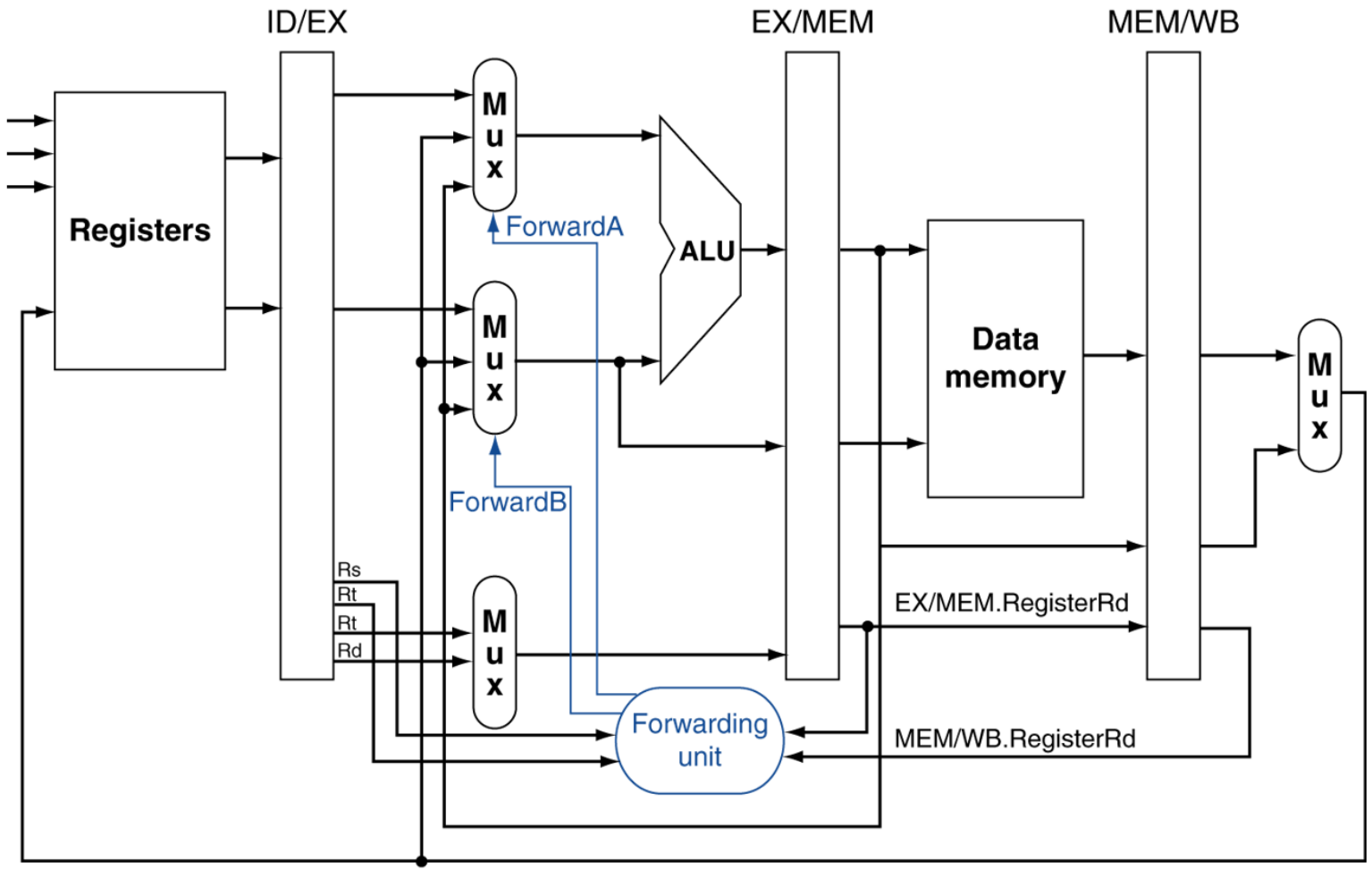
Layout: Extended



Rešitev 2:
Premoščanje (bypassing, data forwarding)

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

Cevovodna PE s premoščanjem (forwarding unit)



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

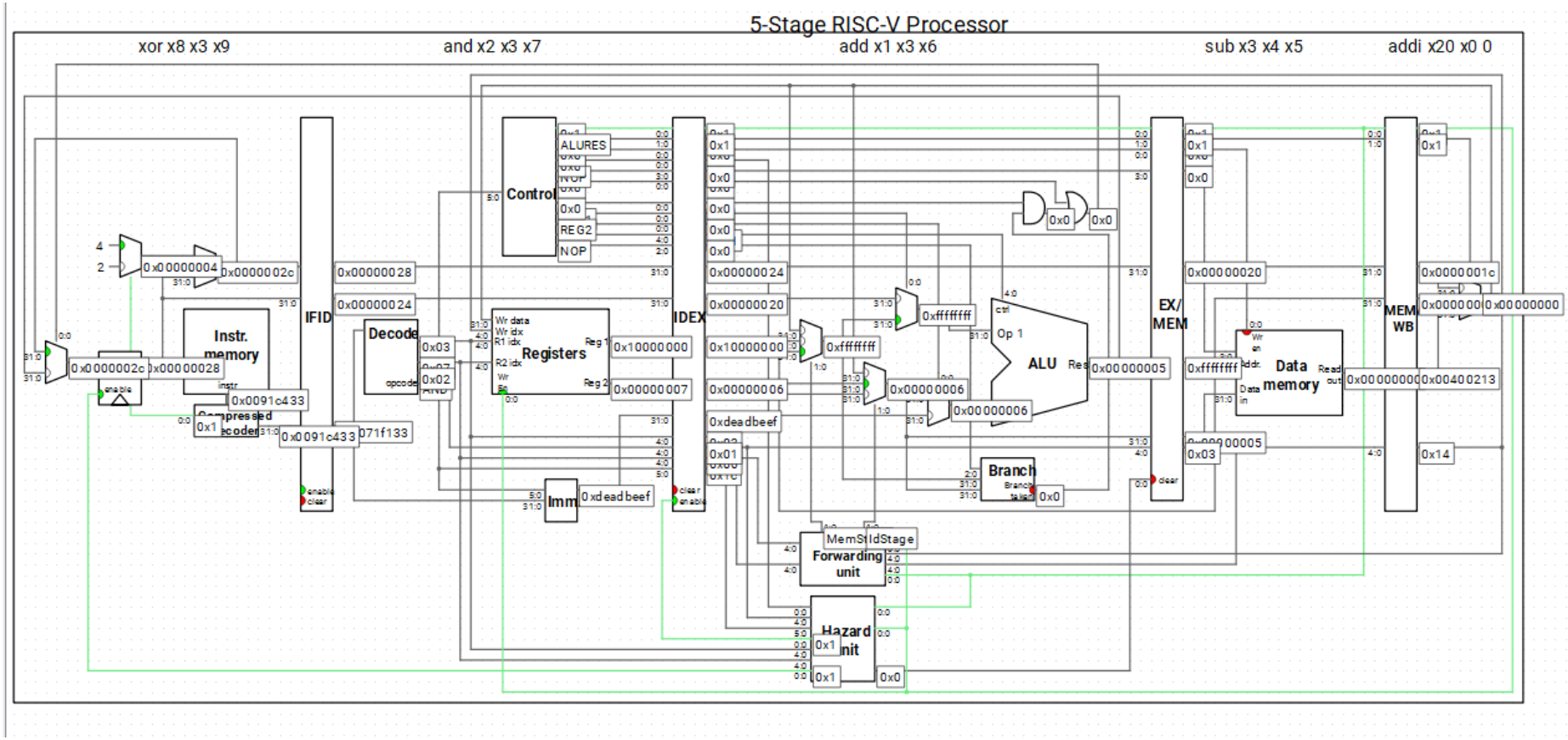
1. Nevarnost v stopnji EX:

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

2. Nevarnost v stopnji MEM:

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

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



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

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

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

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

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

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

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

Potek izvajanja ukazov pri premoščanju

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

Rešitev 3: Cevovodno razvrščanje (pipeline scheduling)

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

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

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

Večina prevajalnikov danes uporablja cevovodno razvrščanje

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

Povzetek odpravljanja podatkovnih nevarnosti

Če povzamemo, imamo glede PN naslednje rešitve:

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

7
PARALELIZEM NA NIVOJU UKAZOV 2

Kontrolne nevarnosti

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

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

Odpravljanje kontrolnih nevarnosti

Prvi korak je zmanjšanje skočne zakasnitve:

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

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

Vstavljanje mehurčkov

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

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

Primer

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

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

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

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

Primer:

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

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

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

Rešitev:

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

Če smemo spremeniti tudi odmike:

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

Samo sprememba vrstnega reda:

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

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

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

2 skupini:

- s statično predikcijo
- z dinamično predikcijo

Statična predikcija

Prevajalnik skuša napovedati bolj verjeten rezultat preverjanja skočnega pogoja

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

Statična predikcija

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

Statična predikcija z zakasnjenimi skoki

Ta metoda je bila priljubljena pri starejših RISC

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

Pri pogojnih skokih je težje:

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

Dokler je bil cevovod, razmeroma preprost, je to delovalo

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

Dinamična predikcija skokov

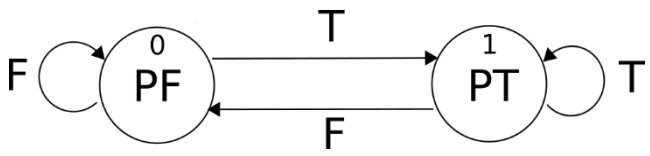
- Danes se bolj kot statična predikcija uporabljajo strojni načini za dinamično predikcijo skokov
- Dinamična predikcija se prilagaja dogajanju v programu

Več vrst dinamične predikcije:

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

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

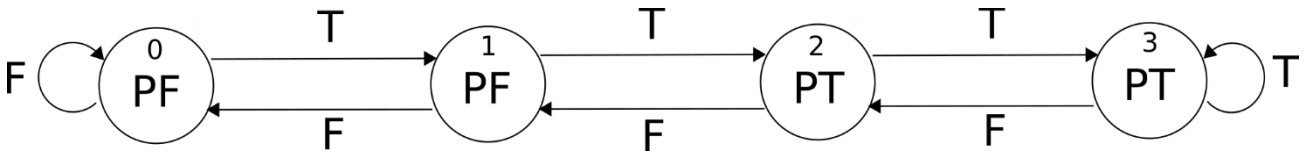
Stanje	Skočni pogoj ni izpolnjen (F)	Skočni pogoj izpolnjen (T)
0 (skočni pogoj ne bo izpolnjen – PF (predict false))	napoved pravilna -> stanje ostane 0 (PF)	napoved napačna -> stanje 1 (PT)
1 (skočni pogoj bo izpolnjen – PT (predict true))	napoved napačna -> stanje 0 (PF)	napoved pravilna -> stanje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



2. 2-bitna prediktorska tabela

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

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



Možna tudi n-bitna prediktorska tabela, n > 2

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

Primer:

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

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

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

Rešitev:

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

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

Enobitna predikcijska (napovedna) tabela:

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

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

Pravilnih napovedi: 5/10 = 50%

2-bitna predikcijska tabela:

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

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

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

Pravilnih napovedi: 5/10 = 50%

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

3. Korelacijski prediktor

Correlating branch prediction table

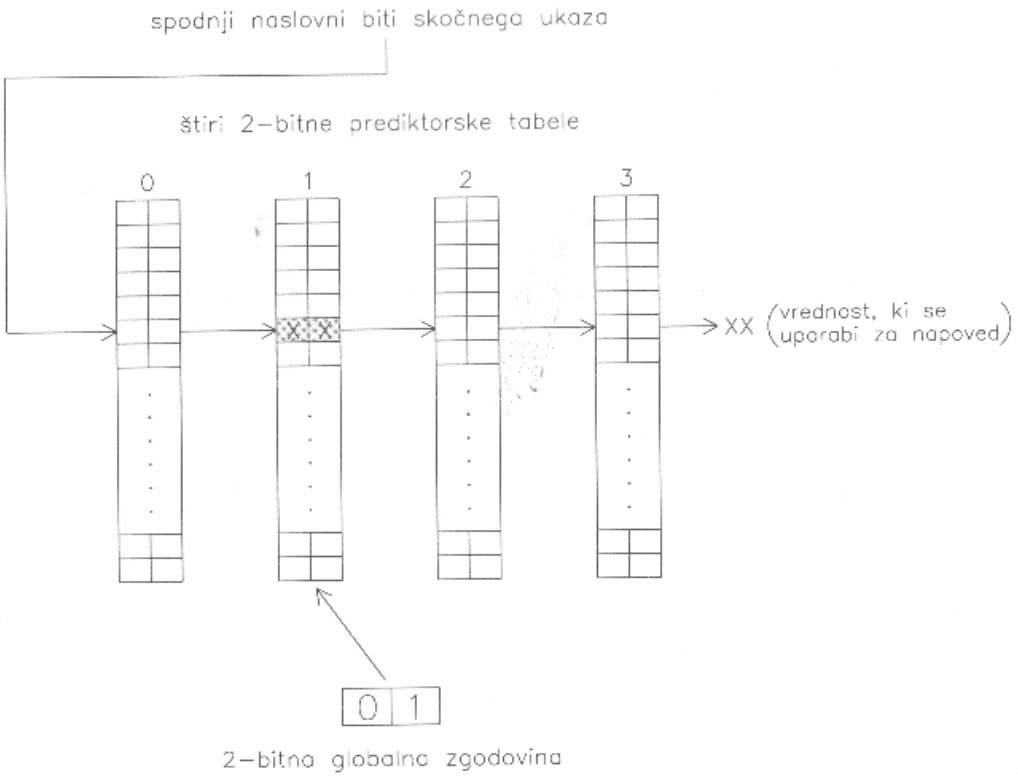
Primer:

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

```
        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

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



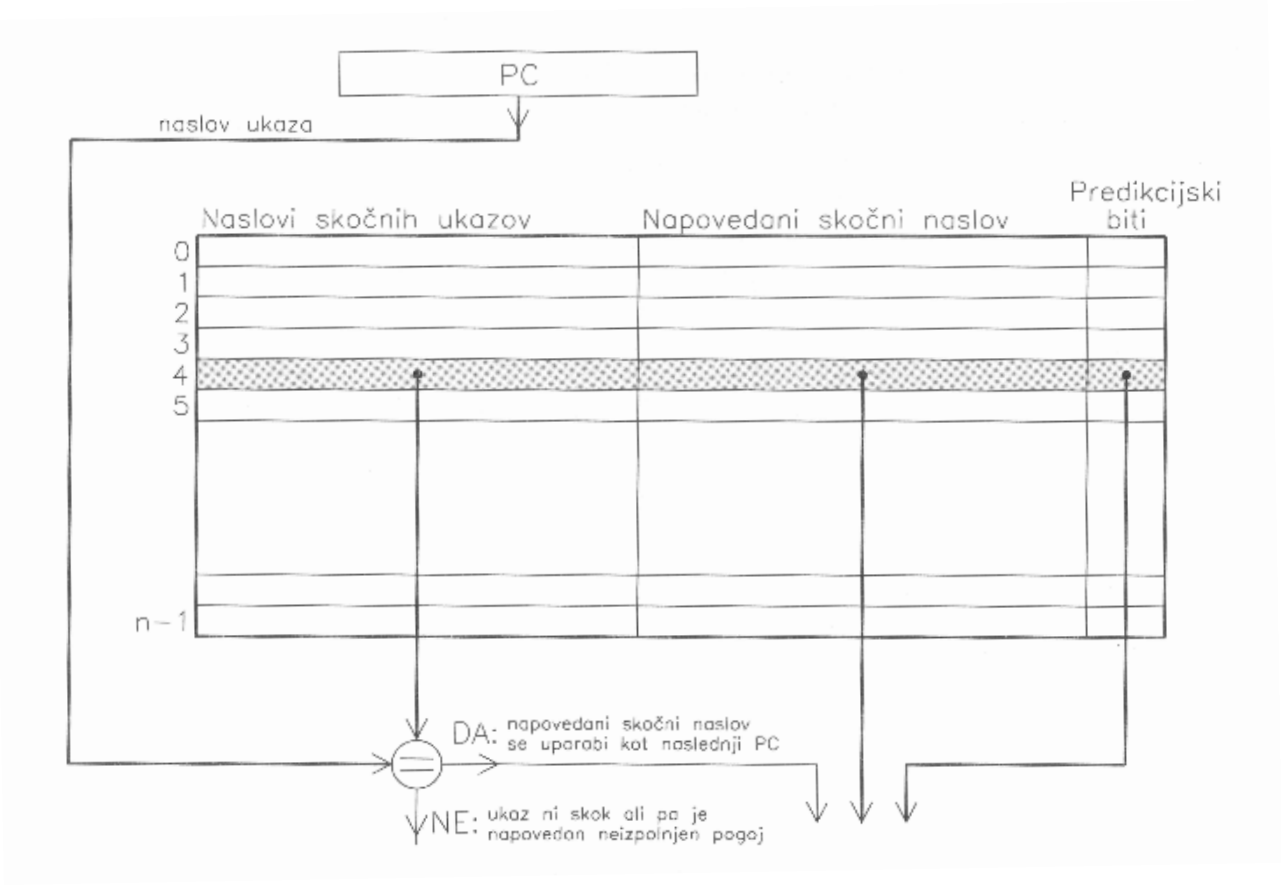
4. Turnirski prediktor

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

5. Skočni predpomnilnik

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

Skočni predpomnilnik:



Skočni PP je bolj zapleten od prediktorjev na osnovi tabel

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

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

- Sicer je naslov poznan (naslednji po vrsti)

6. Vrnitveni prediktor

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

7. Enota za prevzem ukazov

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

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
   - prekinitveno-prevzemni cikel je najbolje izvesti izven cevovoda (sicer bi rabili 6 stopenj v cevovodu)

2. Programske pasti

- v bistvu gre za 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

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

- napaka strani (page fault): pri navideznem (virtualnem) pomnilniku, kadar stran ni (fizično) v GP (ne gre za resnično napako)
- zaščita pomnilnika: dostop do naslova, ki ne pripada programu (segmentation fault)
- pri napaki strani se po servisiranju program nadaljuje na prekinjenem mestu
- pri ostalih pasteh se običajno zaključi z diagnostičnim sporočilom

Operacije, ki trajajo več urinih period

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

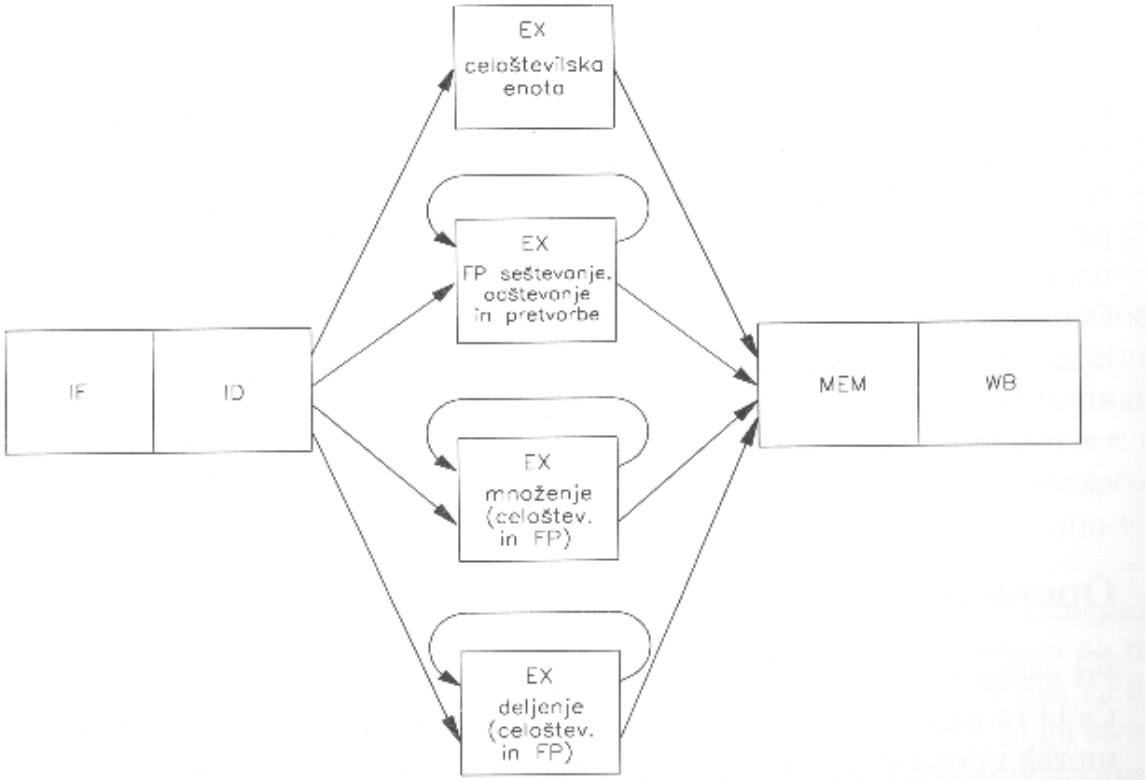
- cevovod bi pri mnogih programih postal prepočasen

Zato so uvedli funkcijske enote:

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

Predpostavimo, da FE niso cevovodne

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

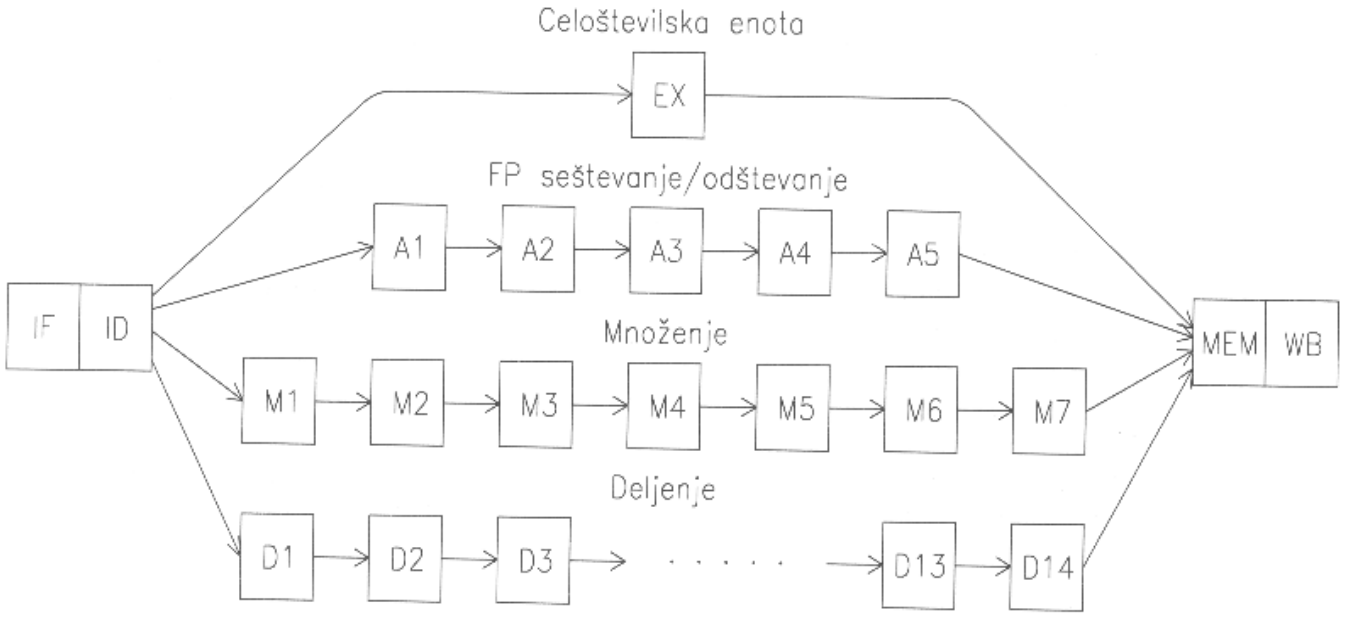


Če so FE cevovodne (danes običajno), lahko odpravimo strukturne nevarnosti v ID in EX

- lahko pa se SN pojavijo v MEM in WB

Dodatni problemi:

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



3 vrste PN:

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

Pri preprostih RISC je edina možnost RAW

- s premoščanjem jo običajno odpravimo (razen pri load)

Primer: zaporedje ukazov v plavajoči vejici

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

	Urina perioda																
Ukaz	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
FLD F4,0(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

Odpravljanje podatkovnih nevarnosti z dinamičnim razvrščanjem

Dinamično razvrščanje:

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

Primer PN:

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

```
FDIV  F0,F5,F6          ; F0 ← F5/F6
FADD  F4,F0,F2          ; F4 ← F0 + F2
FSUB  F8,F2,F1          ; F8 ← F2 - F1
```

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

ID moramo razdeliti na 2 stopnji:

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

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

Tomasulov algoritem (1967)

- uvede *rezervacijske postaje* za dinamično razvrščanje ukazov

Podatkovne odvisnosti lahko delimo na

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

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

- Imenske odvisnosti: WAR in WAW
  - med FADD in FSUB zaradi R2
    - nevarnost WAR (*antiodvisnost*)
  - med FADD in FMUL zaradi F4
    - nevarnost WAW (*izhodna odvisnost*)
- Prave podatkovne odvisnosti: RAW
  - med FDIV in FADD
  - med FADD in FST
  - med FSUB in FMUL

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

```
FDIV   F0,F5,F6      F0 ← F5/F6
FADD   FT2,F0,F2     FT2 ← F0+F2
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

Špekulativno izvajanje ukazov

Pri dinamičnem razvrščanju ukazov se problemi zaradi KN zelo povečajo

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

Špekulativno izvajanje ukazov (speculative execution)

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

Preureditveni izravnalnik je realiziran kot FIFO vrsta v obliki krožnega bufferja

- ukazi so v njem v pravilnem vrstnem redu

Vsako polje v njem ima 4 parametre:

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

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

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

Večizstavitveni procesorji

Približevanje CPI vrednosti 1

- dinamična predikcija skokov
- dinamično razvrščanje
- špekulativno izvrševanje ukazov

CPI < 1:

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

-> večizstavitveni procesorji (multiple issue processors)

- dejanska paralelnost na osnovi več enot

- običajno se uporablja IPC = 1 / CPI

Vidiki prevzema in izstavljanja ukazov

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

Strojno ugotavljanje podatkovnih odvisnosti je zahtevno za realizacijo, zato sta se pojavili 2 rešitvi:

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

Superskalarni procesorji

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

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

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

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

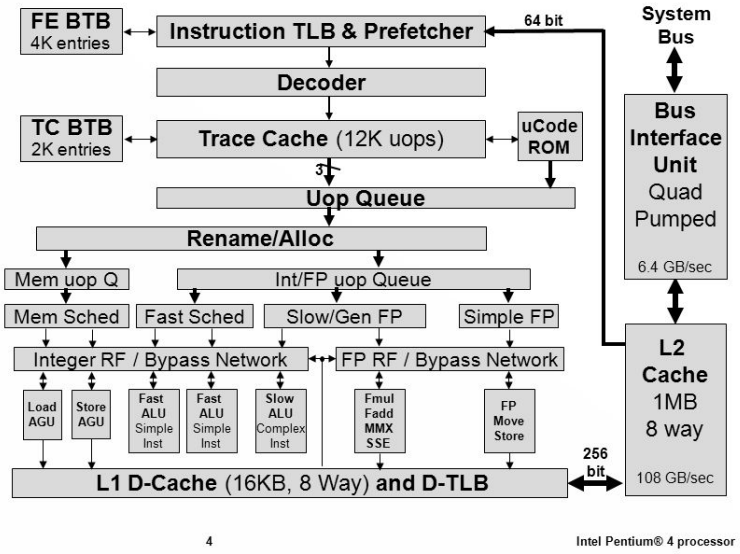
Najbolj zapleteni del superskalarnega procesorja je ROB

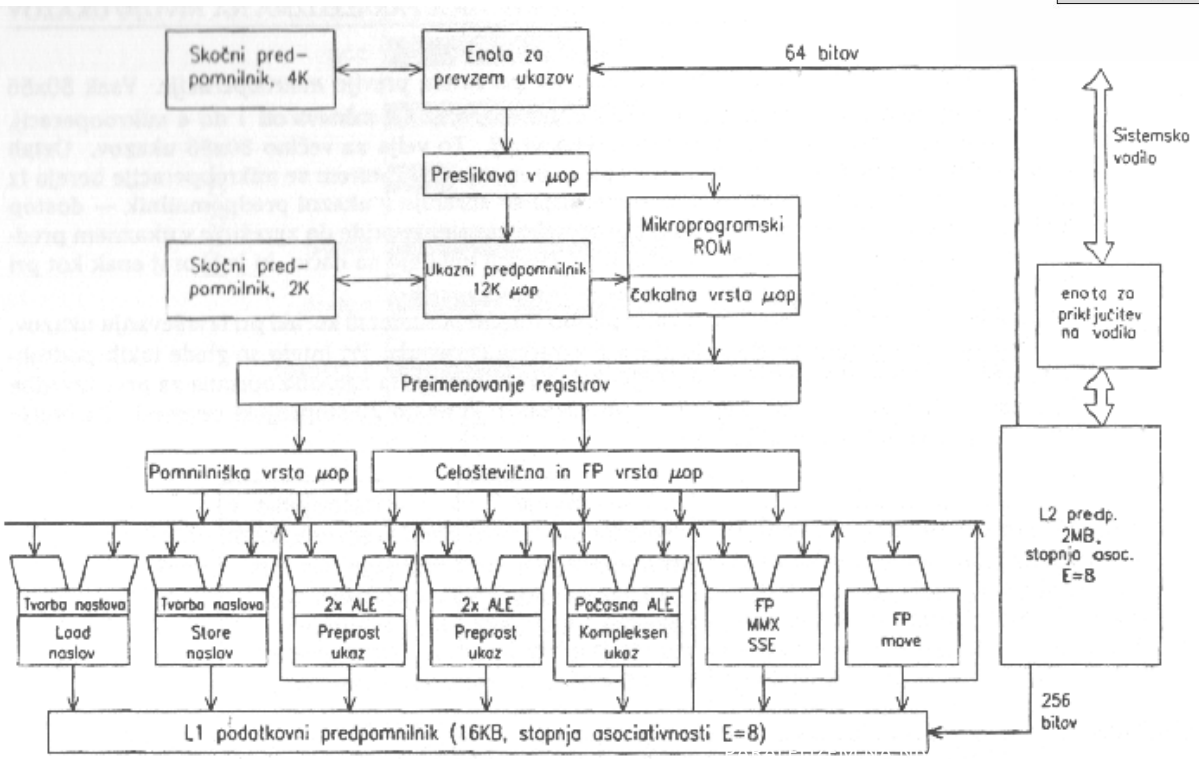
Zato procesorji po letu 2000 uporabljajo **eksplicitno preimenovanje registrov**

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

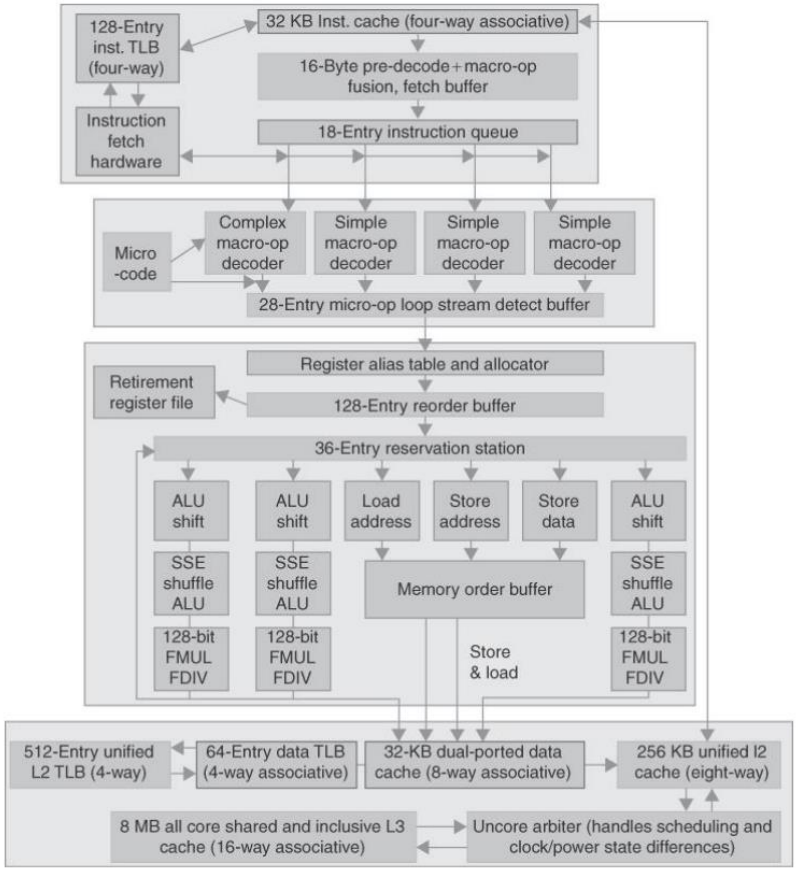
Primer superskalarnega procesorja: Intel Pentium 4

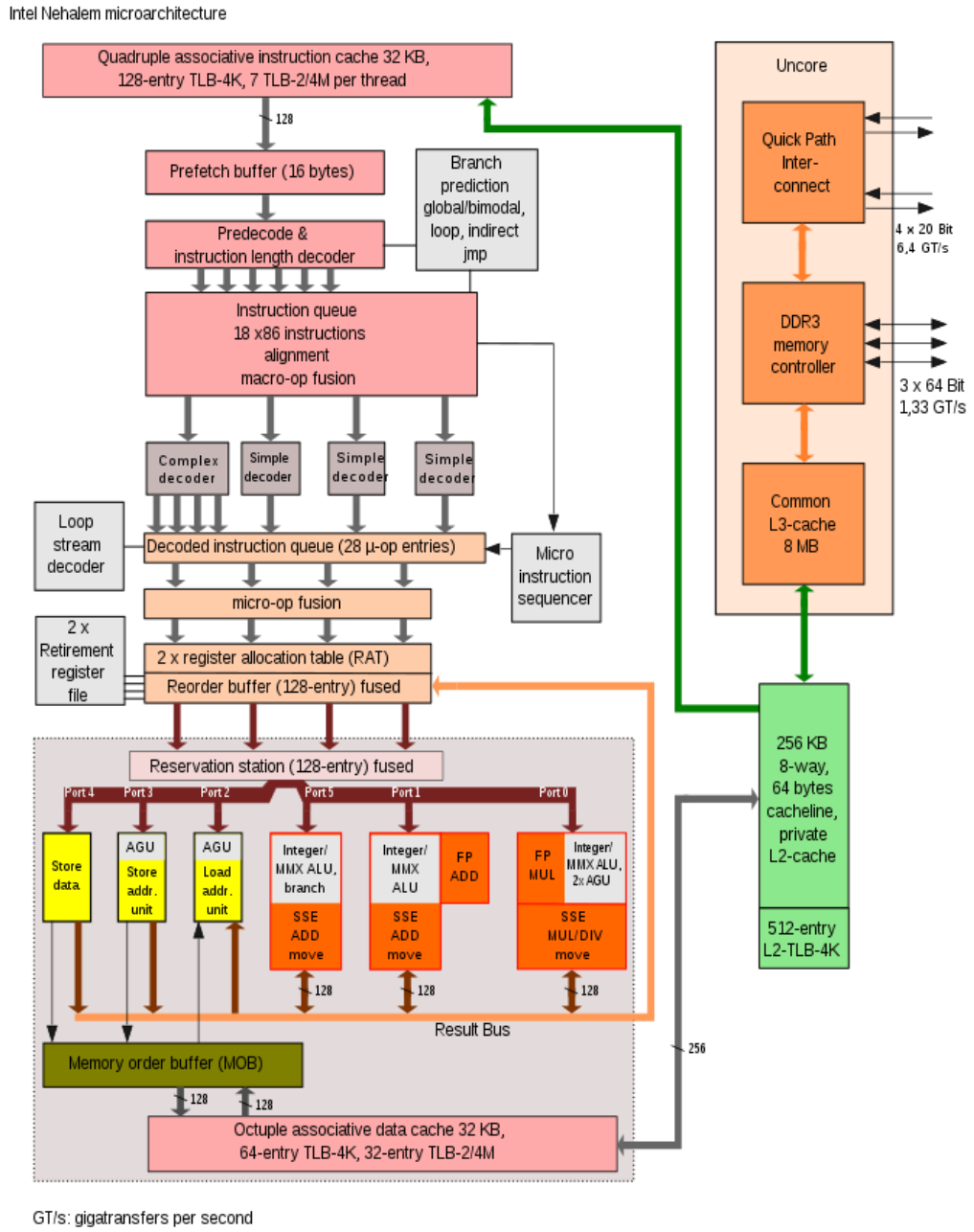
- mikroarhitektura NetBurst
- 7 FE:
  - load
  - store
  - preproste celoštevilske operacije (x2)
  - zahtevne celoštevilske operacije
  - FP operacije
  - prenosi FP operandov iz/v pomnilnik





Intel Core i7 (prvi)





Procesorji VLIW

Procesorji VLIW (very long instruction word) imajo dolge ukaze

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

Dvo-izstavitveni cevovod:

- hkrati se izvajata 2 ukaza različne vrste

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

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

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

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

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

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

Potencialne prednosti VLIW:

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

Digitalno procesiranje signalov

- Veliko paralelnosti

EPIC (explicitly parallel instruction computing)

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

Omejitve paralelizma na nivoju ukazov

Količina paralelnosti v programih je omejena

- S povečevanjem količine logike lahko pridobimo le do neke meje

Koliko paralelnosti na nivoju ukazov je v nekem programu?

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

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

- ostanejo le prave podatkovne nevarnosti

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

Paralelizem na nivoju niti

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

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

Več oblik večnitnosti:

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

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

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

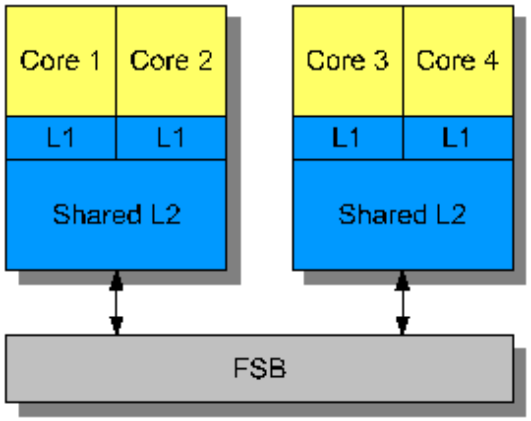
2. **Istočasna večnitnost** (simultaneous multithreading, SMT)
   - pri večizstavitvenih procesorjih
     - Intel Pentium 4: Hyper-threading (običajno 2 niti)
   - ni potrebno veliko sprememb
   - prednost: ni medsebojnih odvisnosti
   - hiba: v določenih primerih se zmogljivost tudi poslabša
     -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

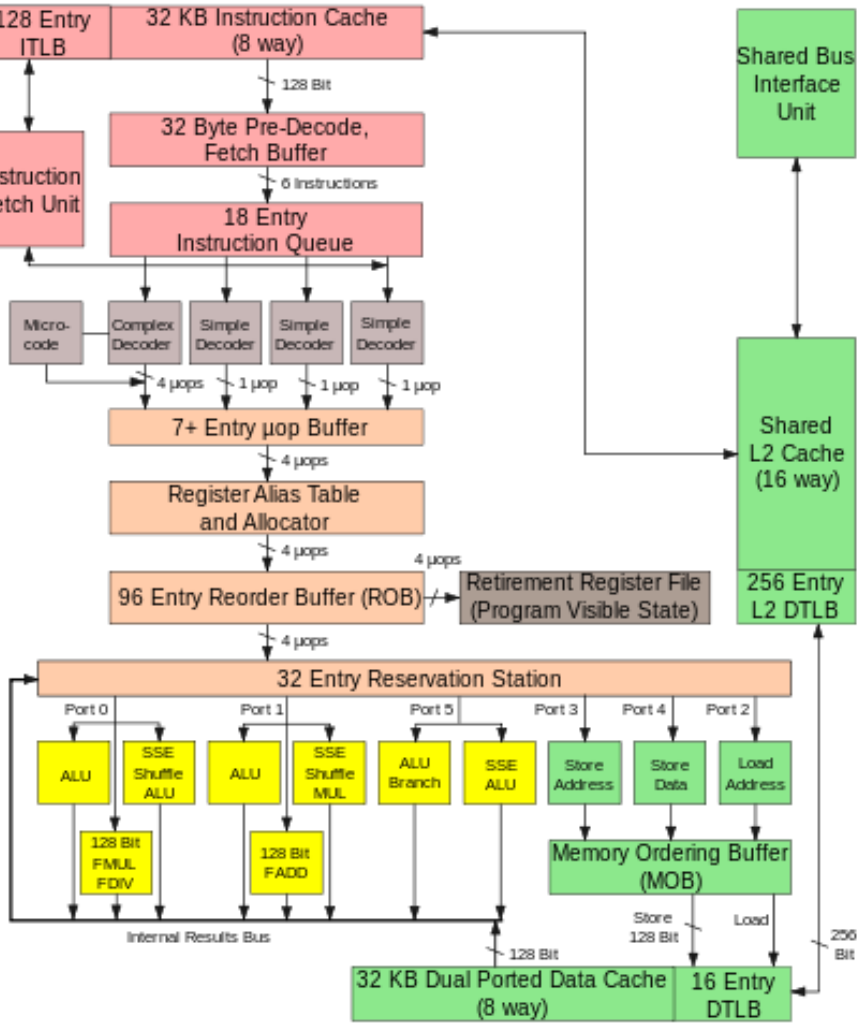
Večjedrni procesorji (multicore)

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

Primer:

- Intel Core 2 Quad





Intel Core 2 Architecture

8 PREDPOMNILNIKI

BRANKO ŠTER

PO KNJIGI - DUŠAN KODEK: ARHITEKTURA IN ORGANIZACIJA RAČUNALNIŠKIH SISTEMOV

Lokalnost pomnilniških dostopov

- Tipično je, 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**

- 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)
- Zaporedje naslovov A(1), ..., A(*N*)
  - pri *N* dostopih do pomnilnika je število različnih naslovov mnogo manjše od *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

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

Pomožni pomnilnik

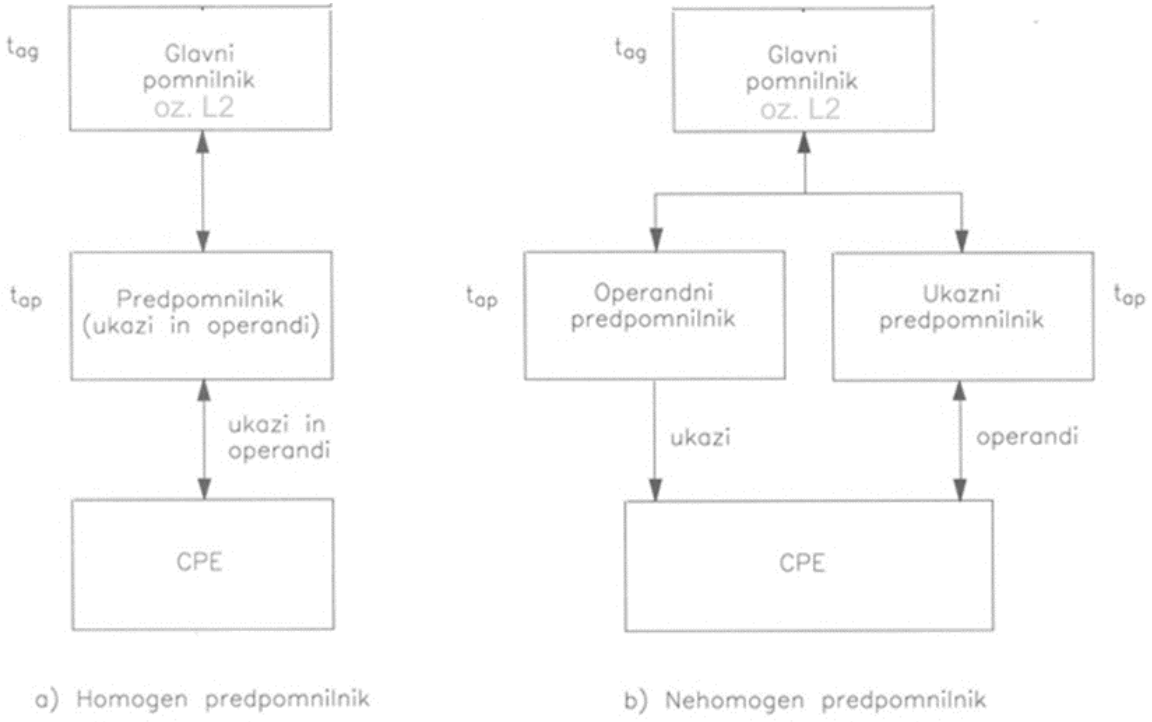
- Pomnilniška hierarhija vključuje tudi *pomožni pomnilnik* (oz. sekundarni, masovni). Kakšna je razlika?
  - Do GP ima CPE **neposreden dostop** (tako, da poda naslov pomnilniške besede)
  - pri pomožnih pomnilnikih je dostop **posreden** preko V/I ukazov, ki najprej prenesejo zahtevano besedo v GP, 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
  - t_{ap} ... čas dostopa do PP
  - t_{ag} ... čas dostopa do GP
- Razmerje t_{ag}/t_{ap} je lahko tudi do nekaj sto
- 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

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

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

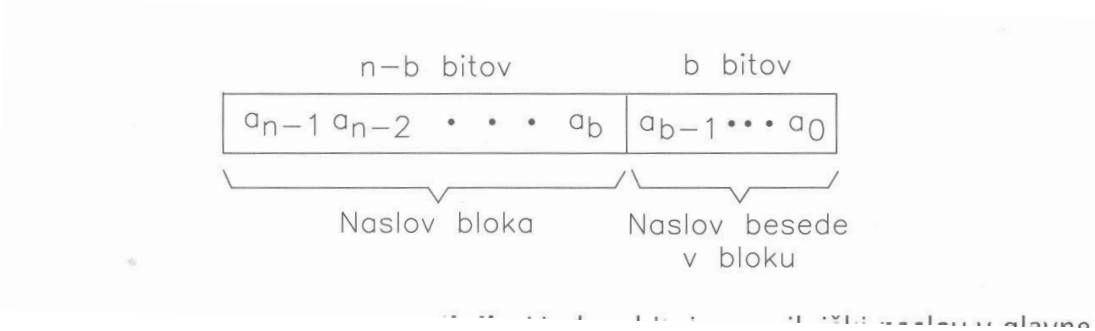
$$t_a = t_{ap} + (1-H)t_B$$

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

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

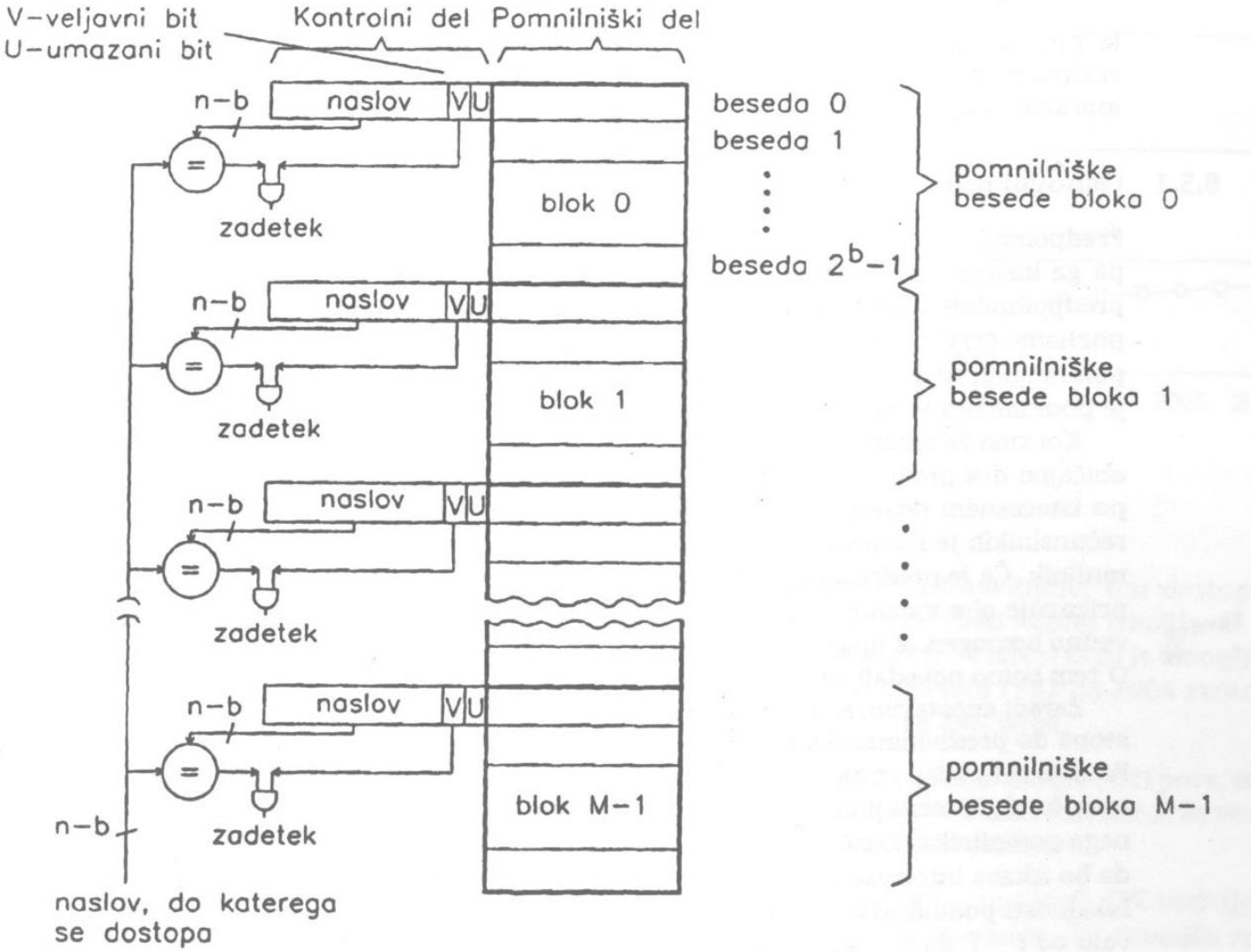
- Možno je definirati področja v GP, katerih besede se nikoli ne prenesejo v PP (*uncacheable* področja)
  - dostop do besede v takem področju je vedno zgrešitev
  - beseda se nikoli ne prenese v PP
  - npr. pri računalnikih, ki uporabljajo pomnilniško preslikan V/I, se določeni pomnilniški naslovi nanašajo 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:
  - **naslov bloka** v glavnem pomnilniku (ang. **TAG**)
    - pove, kateri del GP je trenutno v bloku (rečemo, da je *preslikan* v PP)
  - običajno pa še **veljavni** in **umazani bit**
    - veljavni bit pove, ali je vsebina PP veljavna
      - V=1: je
      - V=0: ni $\rightarrow$ zgrešitev
    - umazani bit U se ob prenosu bloka v PP postavi na 0. Če pride do pisanja v blok, se postavi na 1.

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

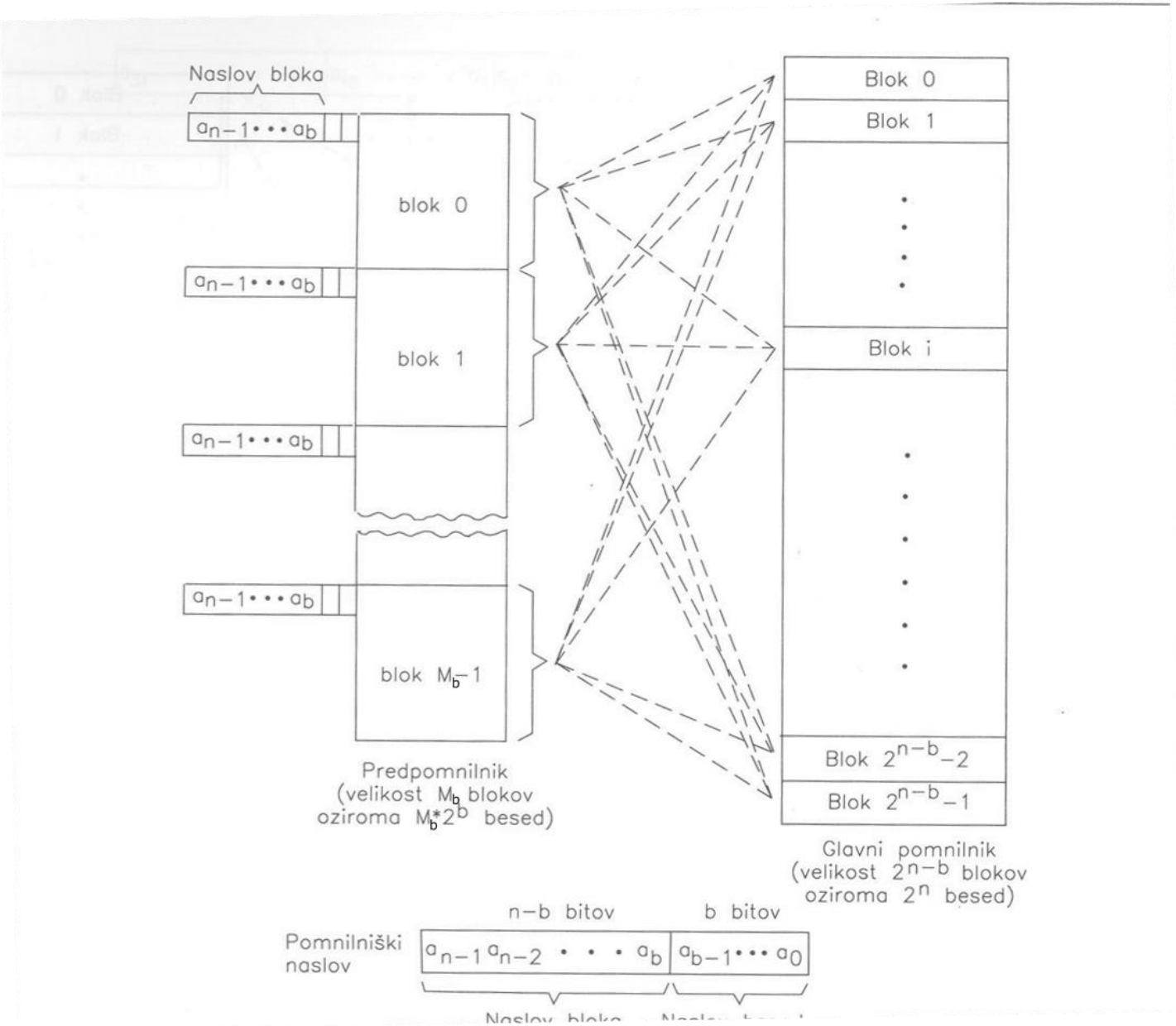
- Zgornjih *n* – *b* bitov naslova se v PP primerja z naslovi v kontrolni informaciji vseh blokov
  - če obstaja pri nekem bloku enakost, je zahtevana beseda v PP (zadetek); poleg tega mora biti še V=1
  - sicer imamo zgrešitev; potreben je dostop do GP; blok iz GP se prenese v PP
    - Če so vsi bloki 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

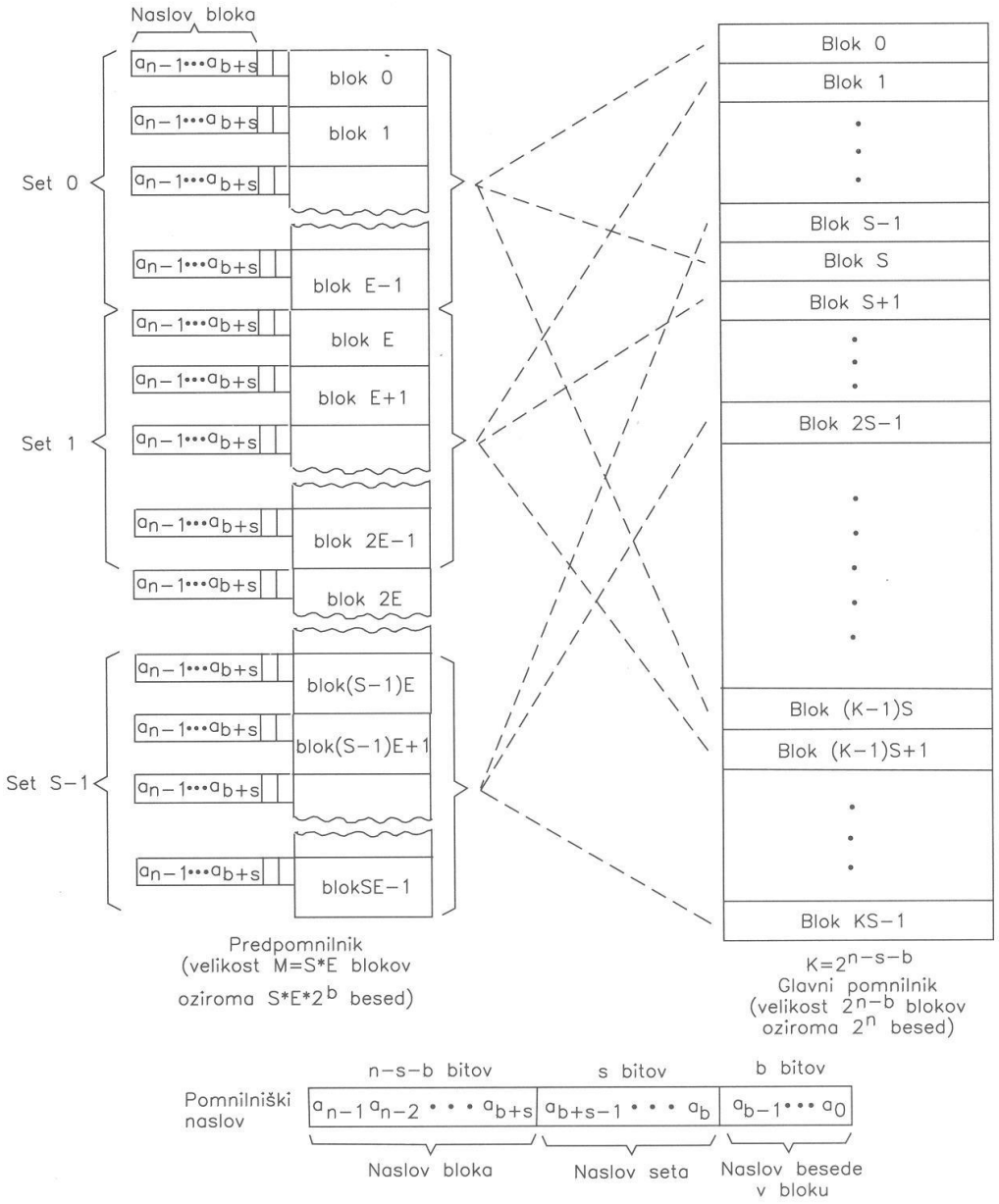
Čisti APP


Naslov bloka
$a_{n-1} \cdots a_b$
blok 0
blok 1
blok $M_b - 1$
Predpomnilnik
(velikost M_b blokov
oziroma $M_b * 2^b$ besed)
Blok 0
Blok 1
Blok i
Blok $2^{n-b}-2$
Blok $2^{n-b}-1$
Glavni pomnilnik
(velikost 2^{n-b} blokov
oziroma 2^n besed)
n−b bitov
b bitov
Pomnilniški naslov
$a_{n-1}\, a_{n-2} \cdots a_b$
$a_{b-1} \cdots a_0$

- 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


Naslov bloka
Set 0
Set 1
Set S−1
blok 0
blok 1
blok E−1
blok E
blok E+1
blok 2E−1
blok 2E
blok(S−1)E
blok(S−1)E+1
blokSE−1
Predpomnilnik
(velikost M=S*E blokov
oziroma S*E*2^b besed)
Blok 0
Blok 1
Blok S−1
Blok S
Blok S+1
Blok 2S−1
Blok (K−1)S
Blok (K−1)S+1
Blok KS−1
K=2^(n−s−b)
Glavni pomnilnik
(velikost 2^(n−b) blokov
oziroma 2^n besed)
n−s−b bitov
s bitov
b bitov
Pomnilniški naslov
Naslov bloka
Naslov seta
Naslov besede v bloku

- 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

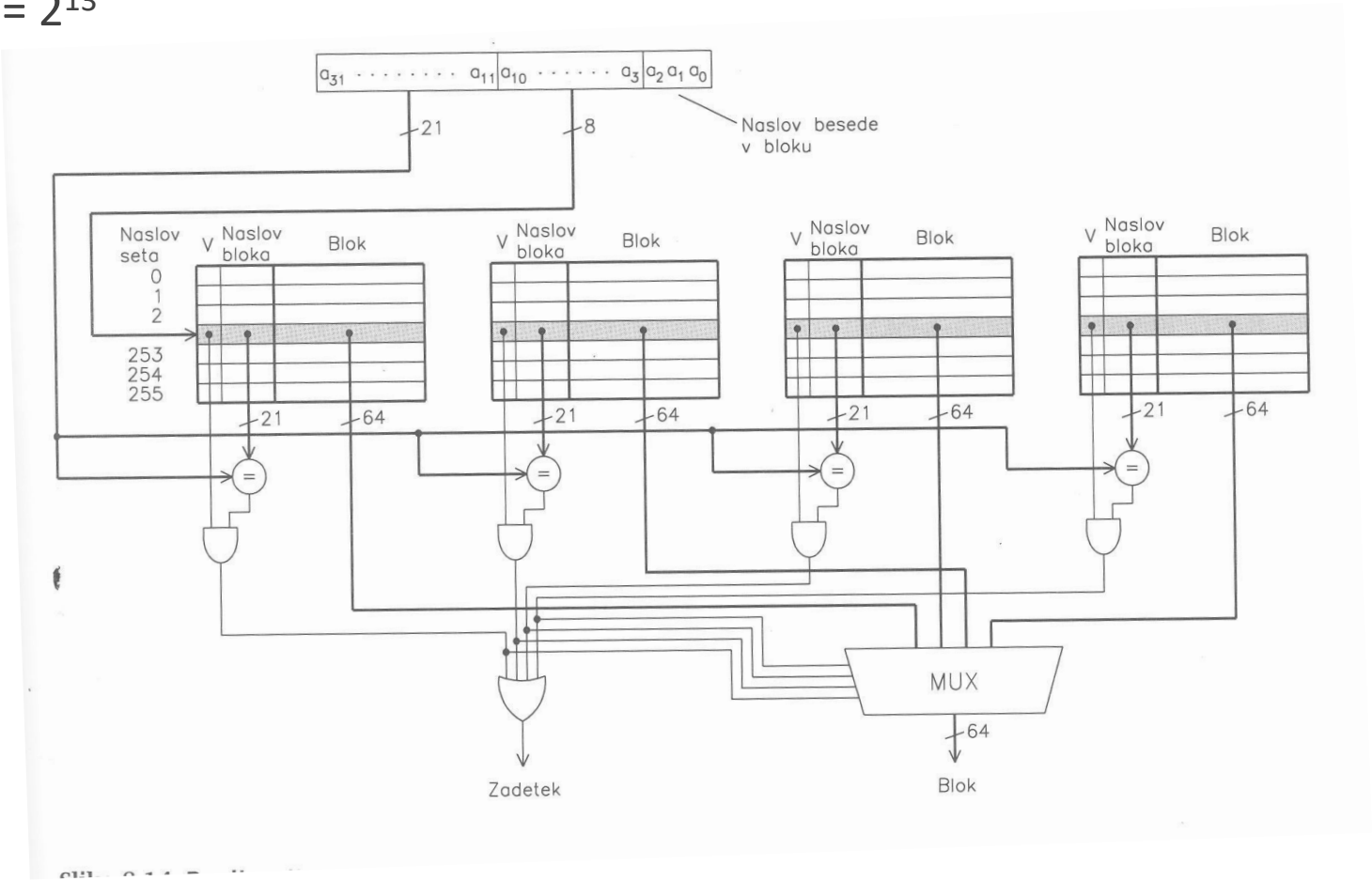
- Pri SAPP se pojavi omejitev pri preslikovanju naslovov:
  - za vsako besedo GP je vnaprej določeno, v katerega od setov se lahko preslika
    - to določajo naslovni biti $a_{b+s-1}, a_{b+s-2}, \ldots, a_b$
    - ta naslov seta (SET) se imenuje tudi **predpomnilniški indeks** (cache index, CI)
      - 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

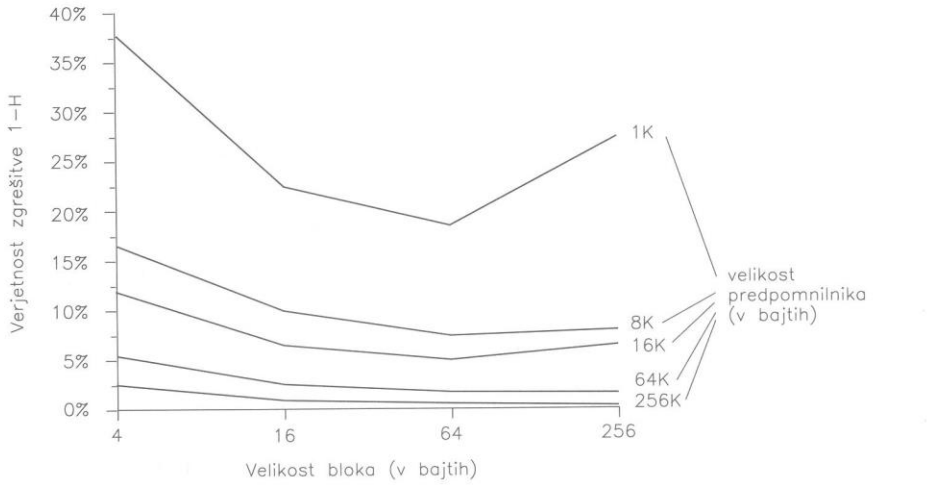
- n=32, b=3, e=2, s=8, 8-bitna pomnilniška beseda
- Vsak izmed 256 setov ima 4 bloke, vsak blok 8 besed (=8x8=64 bitov)
- $M = 2^{13}$



- Pri podani velikosti PP (M) se z E spreminja tudi H
  - manjši E $\rightarrow$ manjša cena in tudi manjši H
- *Predpomnilniško pravilo 2:1*
  - velja za direktne PP
  - (1-H) dir. PP velikosti M $\approx$ (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

- Čas izvrševanja programa:

$$CPEčas = (CPEperiode_{izvrš.} + CPEperiode_{čak.})*t_{CPE}$$

$$CPEperiode_{izvrš} = I*CPI_{idealni}$$

$$CPEperiode_{čak} = N*(1-H)*K_Z$$

$$N = I*(1+M_I)$$

$$CPEčas = I*(CPI_{idealni} + M_I*(1-H)*K_Z)*t_{CPE}$$

N_R ... število bralnih dostopov

N_W ... število pisalnih dostopov

N ... število vseh pom. dostopov

I ... število ukazov

H ... povprečna verjetnost zadetka

K_Z … povprečna zgrešitvena kazen

M_I ... povprečno število operandnih pomnilniških dostopov na ukaz

$CPI_{idealni}$ predpostavi, da ni zgrešitev

- Če $H_R \neq H_W$:

  $CPEperiode_{čak} = N_R*(1-H_R)*K_{Z,R} + N_W*(1-H_W)*K_{Z,W}$

- Če $H_{UPP} \neq H_{OPP}$:

  Število zgr.: $N*(1-H) \rightarrow I*(1-H_{UPP}) + I*M_I*(1-H_{OPP})$

  Verj. zgr.: $1-H = 1-H_{UPP} + M_I*(1-H_{OPP})$

  $CPEperiode_{čak} = (I*(1-H_{UPP}) + I*M_I*(1-H_{OPP}))*K_z$

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
- Za koliko zgrešitve upočasnijo delovanje računalnika?

Prenos bloka zahteva 4 64-bitne prenose, torej 90ns

- zgrešitvena kazen torej 27 period ure

$CPEperiode_{čak,UPP} = I*(1-H) * K_Z = I*0,02*27 = 0,54*I$

$CPEperiode_{čak,OPP} = I*M_I*(1-H) * K_Z = I*0,36*0,04*27 = 0,39*I$

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

$CPEperiode_{čak,UPP} + CPEperiode_{čak,OPP}$

= 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

$$CPEperiode_{čak} = N*(1-H_{L1}) * K_{Z,L1}$$

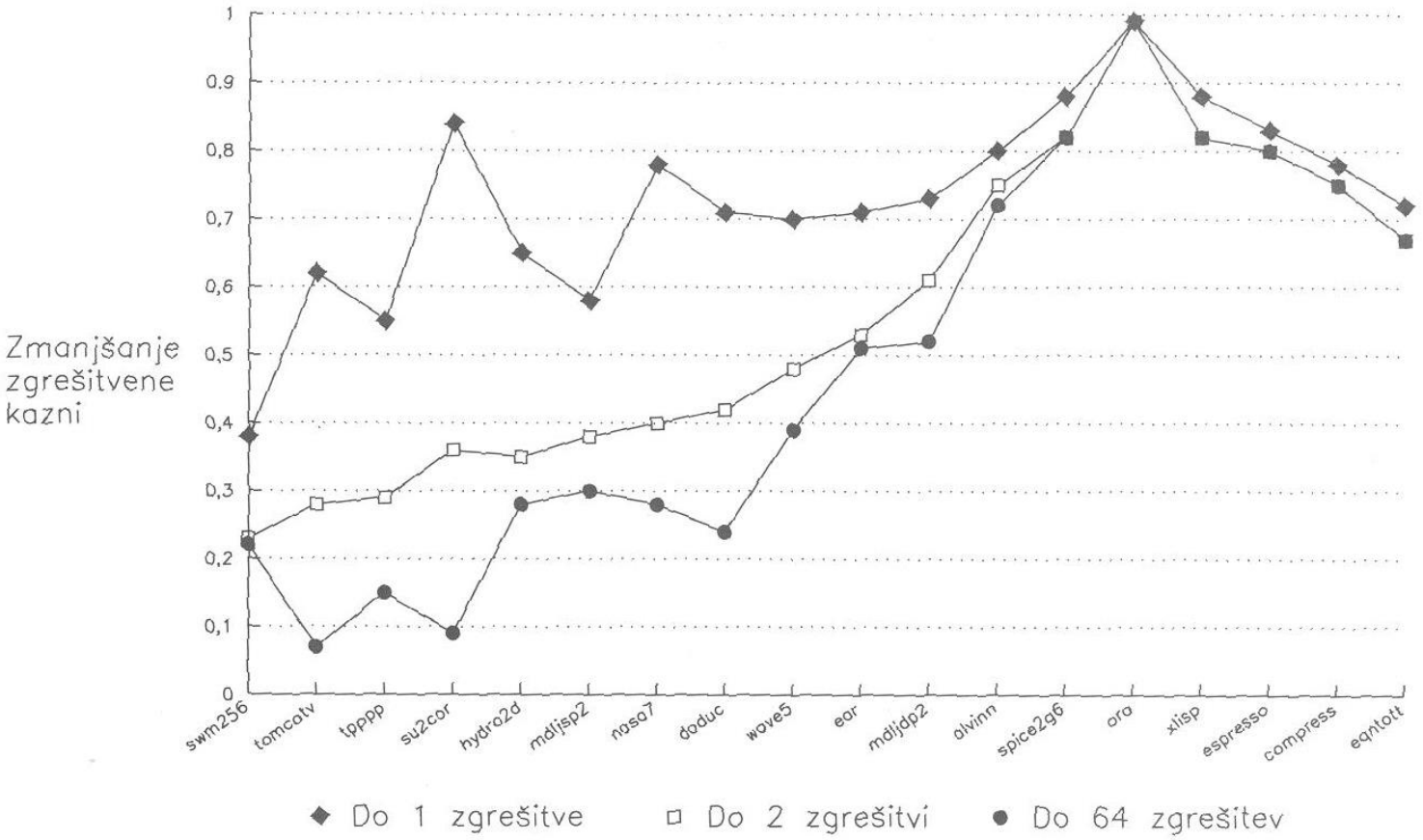
$$K_{Z,L1} = t_{B,L2} + (1-H_{L2}) * K_{Z,L2}$$

- $t_{B,L2}$ ... 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:



Primer. Oglejmo si na praktičnem primeru, kako način dostopa do pomnilnika in predpomnilnikov lahko vpliva na CPEčas.
Množimo matriki **A** in **B**, obe velikosti 1000 x 1000 double (dvojna natančnost v plavajoči vejici), rezultat je **C**. Do elementov matrik dostopamo na dva različna načina.

```
#define N 1000

double **A = new double *[N];
double **B = new double *[N];
double **C = new double *[N];

for (i = 0; i < N; i++) {
   A[i] = new double[N];
   B[i] = new double[N];
   C[i] = new double[N];
}
```

a)
```
   for (i = 0; i < N; i++)
      for (j = 0; j < N; j++)
         for (k = 0; k < N; k++)
            C[i][j] += A[i][k] * B[k][j];
```

V primeru a) do j-tega stolpca (vektorja) matrike **B** dostopamo s stalnim menjavanjem blokov PP, medtem ko je dostop do i-te vrstice matrike **A** zaporeden.

b)
```
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        Bt[i][j] = B[j][i];

for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        for (k = 0; k < N; ++k)
            C[i][j] += A[i][k] * Bt[j][k];
```

V primeru b) najprej matriko **B** transponiramo, zato da pri dostopanju do stolpca ne skačemo med različnimi bloki PP.

Poskus smo izvedli na računalniku s procesorjem Intel Core i7-4790 @ 3.6 GHz, 8 GB RAM, in predpomilniki s podatki spodaj:

Cache:	L1 data	L1 instruction	L2	L3
Size:	4 x 32 KB	4 x 32 KB	4 x 256 KB	8 MB
Associativity:	8-way set associative	8-way set associative	8-way set associative	16-way set associative
Line size:	64 bytes	64 bytes	64 bytes	64 bytes
Comments:	Direct-mapped	Direct-mapped	Non-inclusive Direct-mapped	Inclusive Shared between all cores

V prvem primeru dobimo CPEčas okrog 5 s, v drugem okrog 2 s (pohitritev torej kar **2.5**).

9 POMNILNIKI

BRANKO ŠTER

PO KNJIGI - DUŠAN KODEK: ARHITEKTURA IN ORGANIZACIJA RAČUNALNIŠKIH SISTEMOV

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)

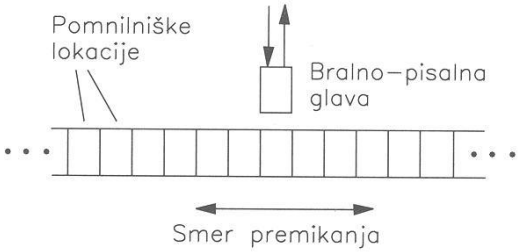
- čas dostopa je odvisen od prejšnjega naslova
  - če smo bili na naslovu A, je takoj dostopen le naslov A+1
- npr. magnetni trak

3.3 Krožni dostop (rotational access)

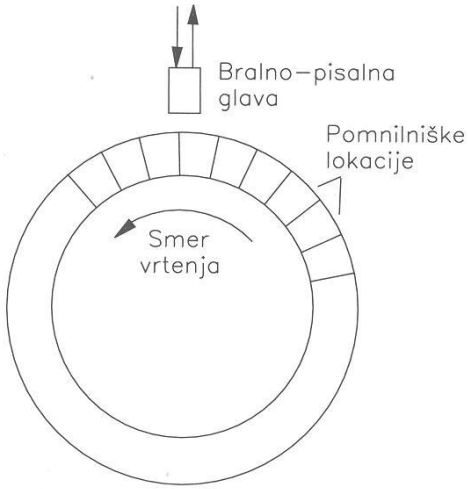
- posebna vrsta zaporednega dostopa
  - kot npr. magnetni trak, ki bi bil zlepljen v zanko
- npr. magnetni disk s fiksnimi glavami
- povprečen čas dostopa t_a je enak ½ periode vrtenja

3.4 Kombinacija zaporednega in krožnega dostopa

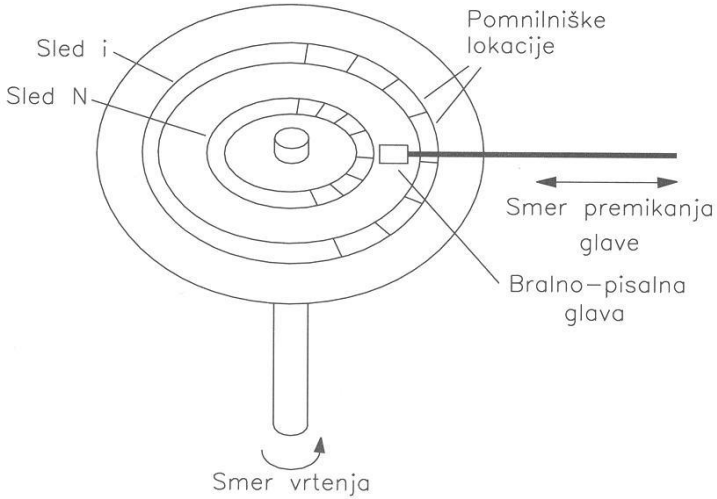
- magnetni in optični diski s premičnimi glavami
- bralno-pisalna glava se najprej premakne na ustrezno sled (zaporedni dostop), nato pa imamo krožni dostop
- hitrejši od zaporednega ali krožnega dostopa



a) Zaporedni način dostopa



b) Krožni način dostopa



c) Direktni način 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)

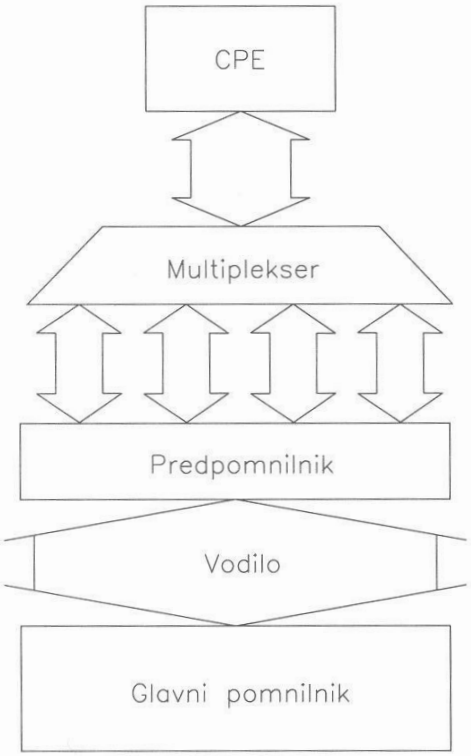
- Problem se je še povečal s pojavom večuporabniških (multiuser) in večopravilnih (multitasking) OS
  - istočasno se izvaja le en program (če imamo eno CPE), vendar si programi delijo isti pomnilniški prostor
  - treba je poskrbeti, da en program ne posega v prostor drugega (namenoma ali nehote, vseeno)
    - 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

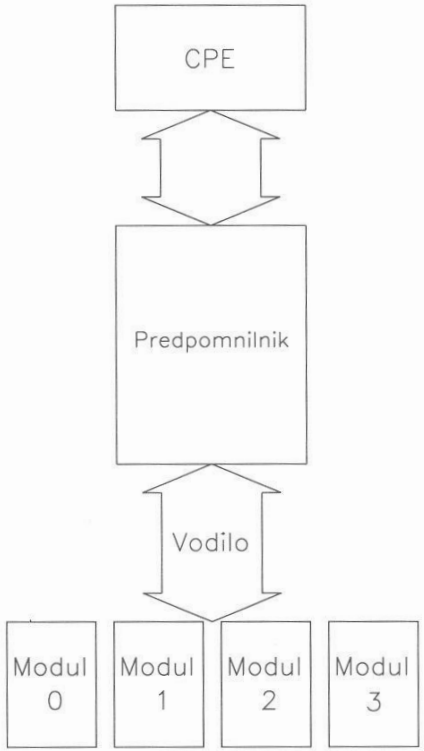
    spodnja meja <= A <= *zgornja meja*
  - slabosti:
    - programi morajo zasedati zvezen prostor v pomnilniku
    - vse besede so zaščitene na enak način
      - raje bi imeli “samo branje”, “branje ali pisanje”, ...
- Boljše rešitve uporabljajo **bloke** ali **strani** (pages) velikosti 1024, 2048 ali 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



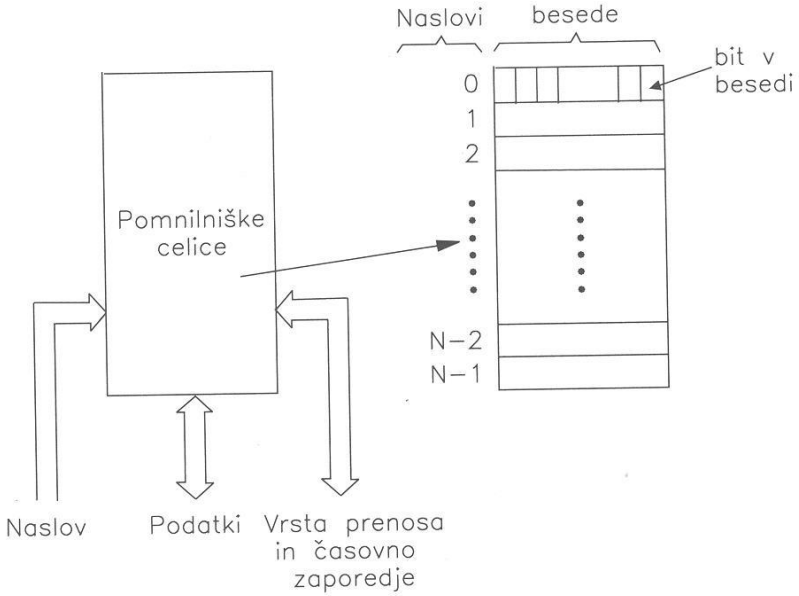
a) Glavni pomnilnik s široko podatkovno potjo.



b) Glavni pomnilnik z ozko podatkovno potjo in pomnilniškim prepletanjem.

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



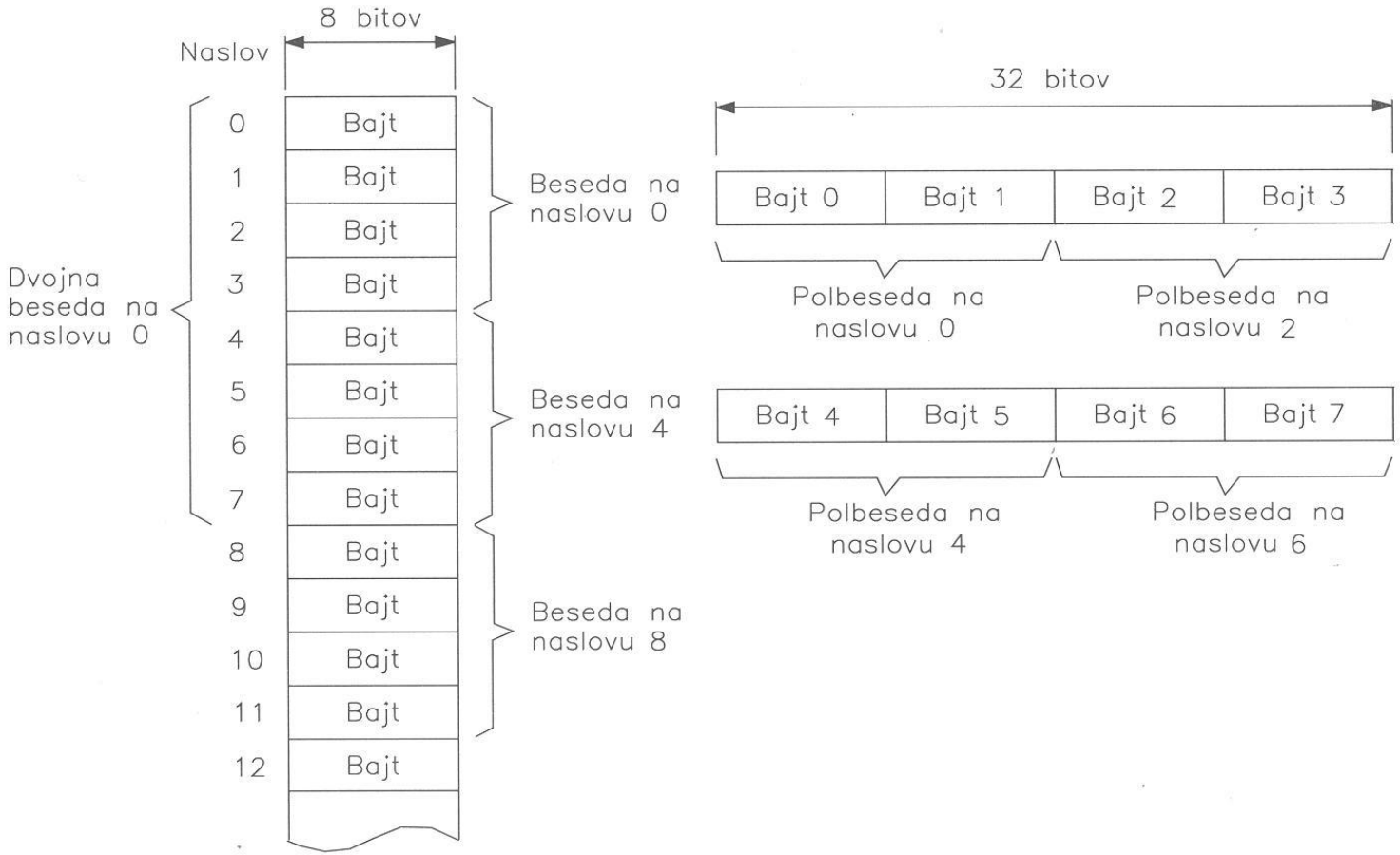
- Osnovna parametra pomnilnika sta:
  1. **Pomnilniška beseda**
     - to je najmanjše število bitov s svojim naslovom
       - **dolžina besede** (običajno 1B oz. 8 bitov)
     - običajno je možen dostop do več besed
  2. **Pomnilniški naslov**
     - binarno število
     - **dolžina naslova** določa 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

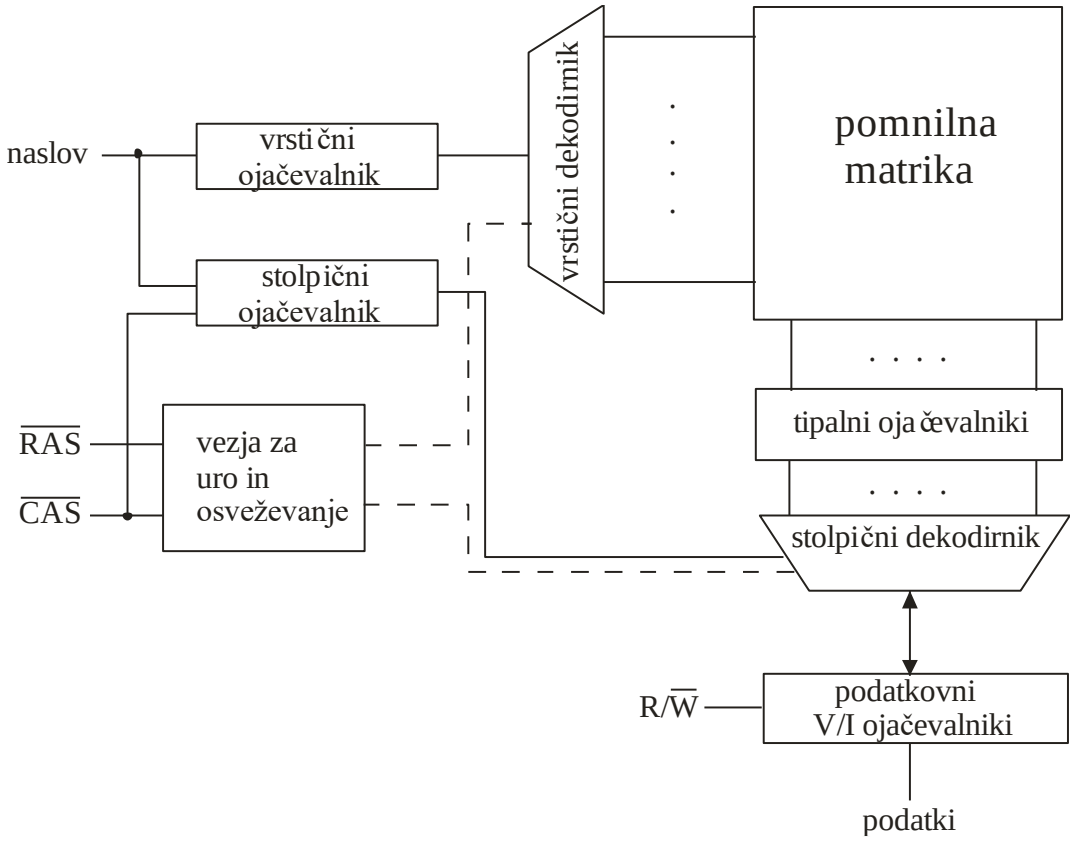
- Kjer je možen dostop do sestavljenih besed, je dobro, če je podatkovno vodilo temu ustrezno široko, sicer je potrebnih več prenosov
  - tudi, če je več prenosov, programer tega ne vidi

- 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!

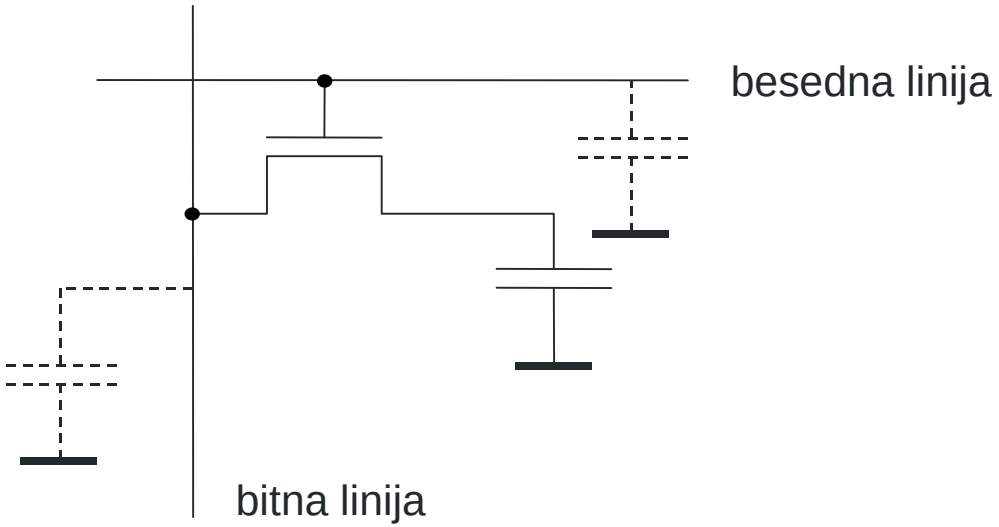

8 bitov
Naslov
0
1
2
3
4
5
6
7
8
9
10
11
12
Bajt
Dvojna beseda na naslovu 0
Beseda na naslovu 0
Beseda na naslovu 4
Beseda na naslovu 8
32 bitov
Bajt 0
Bajt 1
Bajt 2
Bajt 3
Polbeseda na naslovu 0
Polbeseda na naslovu 2
Bajt 4
Bajt 5
Bajt 6
Bajt 7
Polbeseda na naslovu 4
Polbeseda na naslovu 6

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)



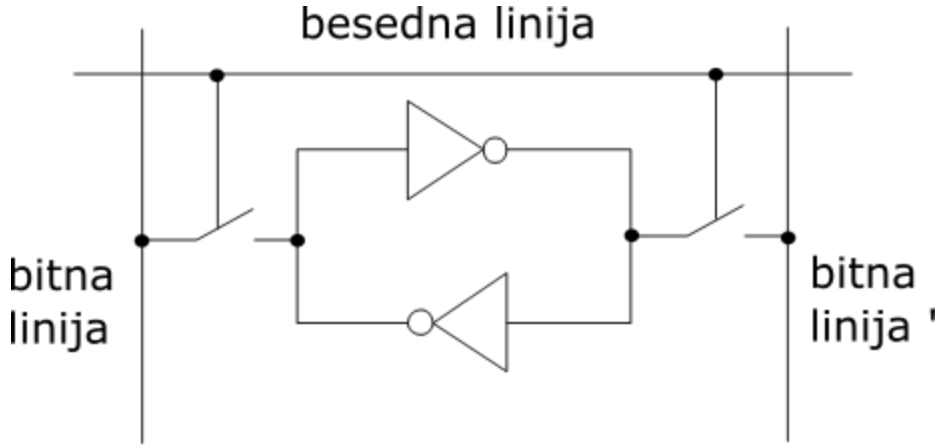
- DRAM vsebuje bitno ravnino oz. matriko ALI
  - v njej so besedne in bitne linije, na presečiščih pa so pomnilne celice
  - razlog za 2D organizacijo je velikost dekodirnika in število ter dolžina linij
    - npr. 1Mb pri 1D: dekodirnik 20/1M, 1M besednih linij, zelo dolga bitna linija (z 1M celicami! – 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

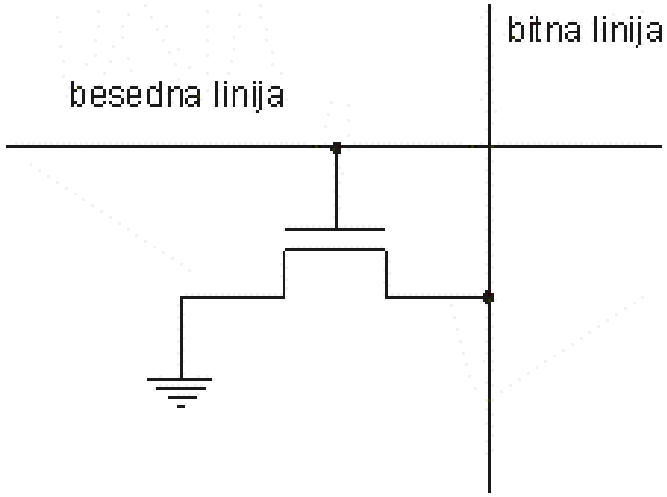
- DRAMi uporabljajo *naslovno multipleksiranje*
  - naslov vrstice in naslov stolpca sta na istih pinih
    - s tem se zmanjša število priključkov (pinov) za bite naslova
      - naslovi so pri DRAMih seveda dolgi (npr. 30 bitov pri 1Gb)
      - priključki so glavni dejavnik pri ceni čipa
    - ne izgubimo 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 je *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)