Mikroprocesorski sistemi

Mikroprocesorska elektronika

Osnovi o računarima

I poglavlje

- Opšti principi rada-

Računarski sistemi-definicija pojmova

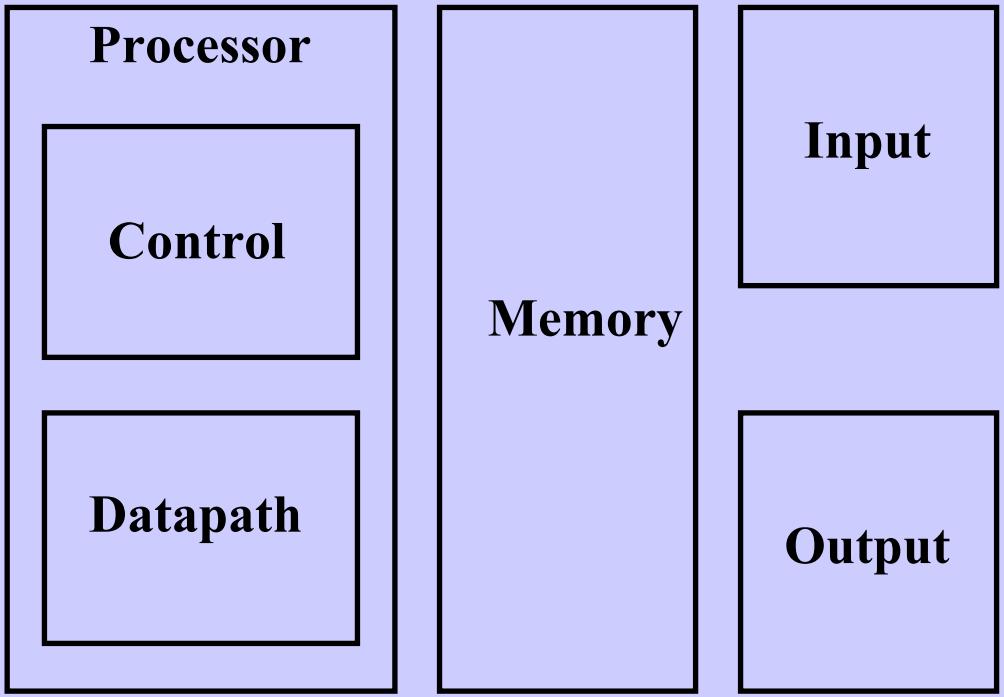
Računarski sistemi su mašine koje vrše brzo izračunavanje i čije su akcije upravljane od strane programa

Program se sastoji od unapred pripremljene liste instrukcija

Softver, **hardver**, **podaci**, i **komunikacije** su komponente koje ~ine ra~unarski sistem

The Big Picture

- Since 1946 all computers have had 5 components



Komponente računarskog sistema

- **Podaci** sa kojima se manipuliše - predstavljaju fundamentalnu prezentaciju ~injenica i opservacija.

- **Hardver** - obezbedjuje fizi~ke mehanizme za:
  i) unošenje (generisanje) podataka iz (ka) spoljnjeg sveta;
  ii) procesiranje podataka putem izvršenja instrukcija;
  iii) memorisanje podataka;
  iv) kopiranje podataka izmedju razli~itih ulazno-izlaznih uredjaja;
  v) elektronsko upravljanje razli~itim ulaznim, izlaznim, i memorijskim komponentama.

Komponente računarskog sistema- produžetak

- **Softver** - čine ga sistemski i aplikacioni programi, a definišu ga instrukcije koje se izvršavaju od strane hardvera.

- **Komunikaciona komponenta** - ~ine je hardver i softver koji obavljaju transport programa i podataka izmedju medjusobno povezanih ra~unara

Arhitektura i organizacija računara

- Pojam **arhitektura ra~unara** ti~e se funkcionalnog ponašanja ra~unarskog sistema i uklju~uje one atribute sistema koji su vidljivi programeru, tj. atribute koji imaju direktan uticaj na logi~ko izvršenje programa kao što su:
  - broj bitova koji se koriste za predstavljanje razli~itih tipova podataka
  - skup instrukcija
  - tehnike za adresiranje memorije
  - ulazno-izlazni (U/I) mehanizmi

Arhitektura i organizacija računara (produžetak)

- Pojam **organizacija ra~unara** odnosi se na operativne jedinice i njihovo medjusobno povezivanje, tj. detalje koji nisu direktno vidljivi programeru, kakvi su na primer:
  - interfejsi izmedju ra~unara i perifernih uredjaja,
  - taktna frekvencija,
  - tehnologija instalirane memorije,
  - namena i naponski nivoi raznih upravlja~kih signala, i dr.

- Pojam **organizacija** se alternativno, od strane većeg broja autora, naziva **implementacija**

Arhitektura i organizacija računara (produžetak)

- U toku svog `ivotnog veka arhitektura ra~unara se ne menja (ne menja se obim podataka, skup instrukcija, adresni na~ini rada itd.). Kada je arhitektura dobro projektovana, dobro se prilagodjava budućim tehnologijama, a prihvaćena je od strane tržišta, tada ona može da egzistira nekoliko desetina godina (kao na primer PC mašina).

- Organizacija ra~unara obi~no pretrpi ve}i broj promena (promene se gabariti uredjaja, taktna frekvencija rada, tehnologija izrade plo~a i modula itd.).

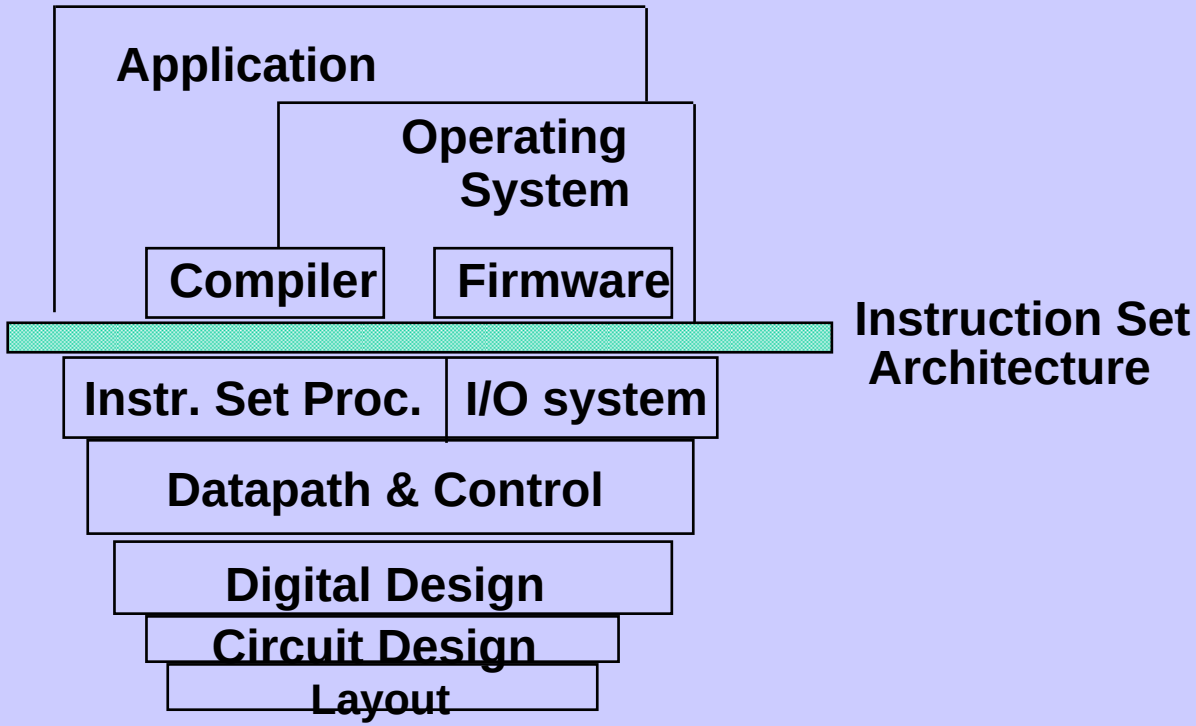
What is "Computer Architecture"

Computer Architecture =

Instruction Set Architecture +

Machine Organization + …..

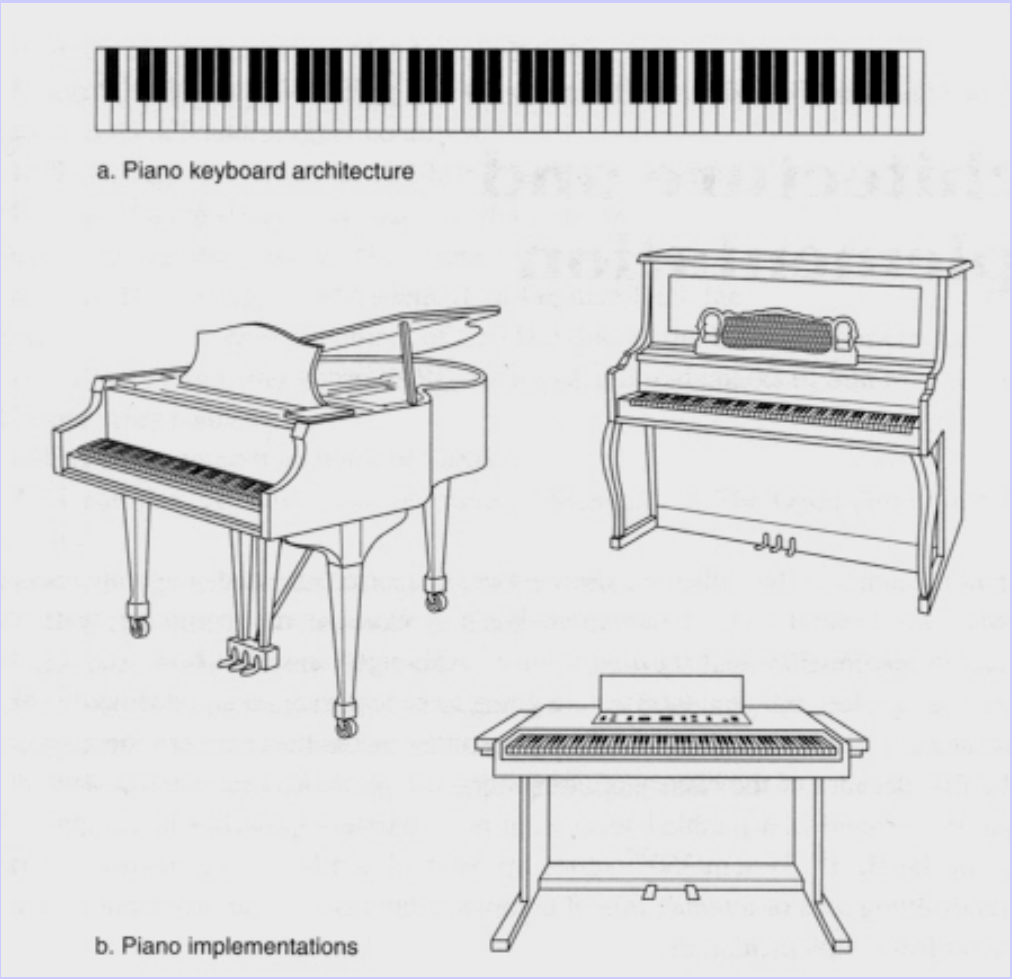
What is "Computer Architecture"?



- Coordination of many *levels of abstraction*
- Under a rapidly changing set of forces
- Design, Measurement, *and* Evaluation

Analogija sa piano arhitekturom

- **Piano arhitektura se definiše specifikacijom tastature.**
- **Tastatura je interfejs ovog muzičkog instrumenta prema korisniku.**
- **Tastatura se sastoji od 88 dirki, 36 crnih i 52 belih.**
- **Pritiskom na dirku generiše se ton.**



Uredjenje dirki je identično za sve tipove piana (arhitektura), tako da onaj koji zna da svira na jednom tipu piana zna i na svim ostalim tipovima. (organizacije, tj. implementacije)

Implementation Domains

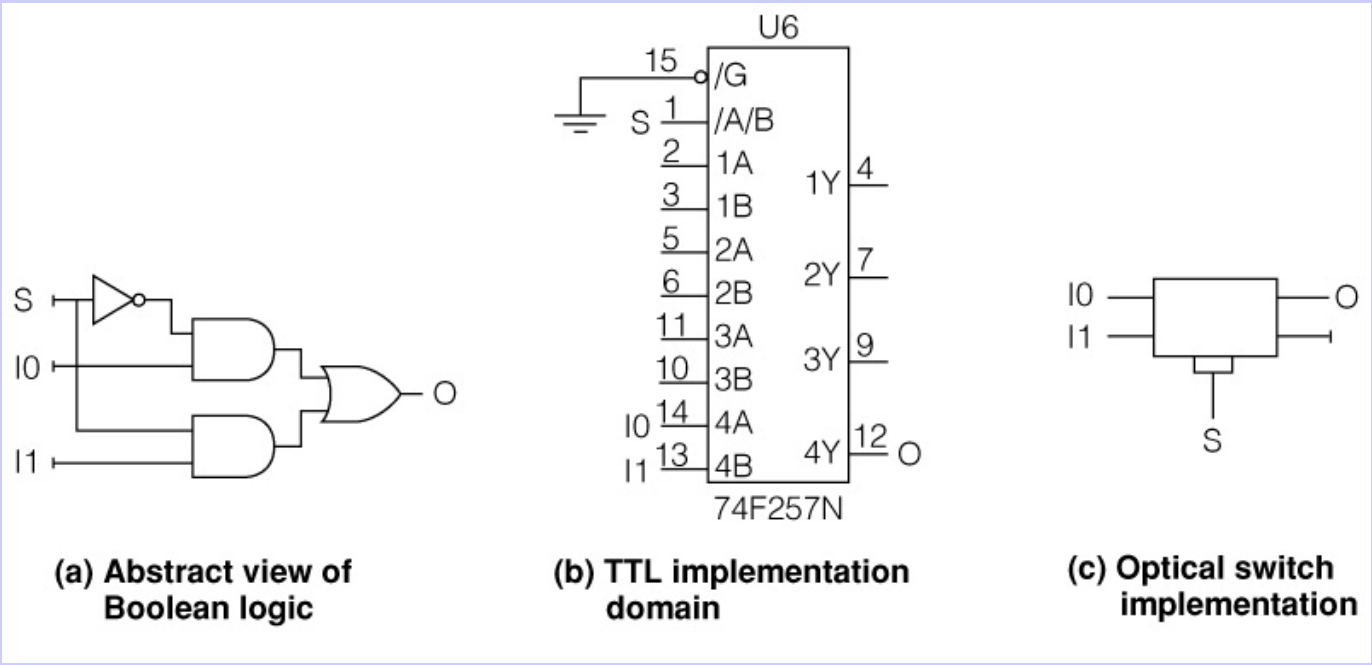
An implementation domain is the collection of devices, logic levels, etc. which the designer uses.

Possible implementation domains:

- VLSI on silicon
- TTL or ECL chips
- Gallium Arsenide chips
- PLA's or sea-of-gates arrays
- Fluidic logic or optical switches

Three Different Implementation Domains

- 2 to 1 multiplexer in three different implementation domains
  - generic logic gates (abstract domain)
  - National Semiconductor FAST Advanced Schottky TTL (VLSI on Si)
  - Fiber optic directional coupler switch (optical signals in LiNbO3)



Hijerarhijska organizacija računara

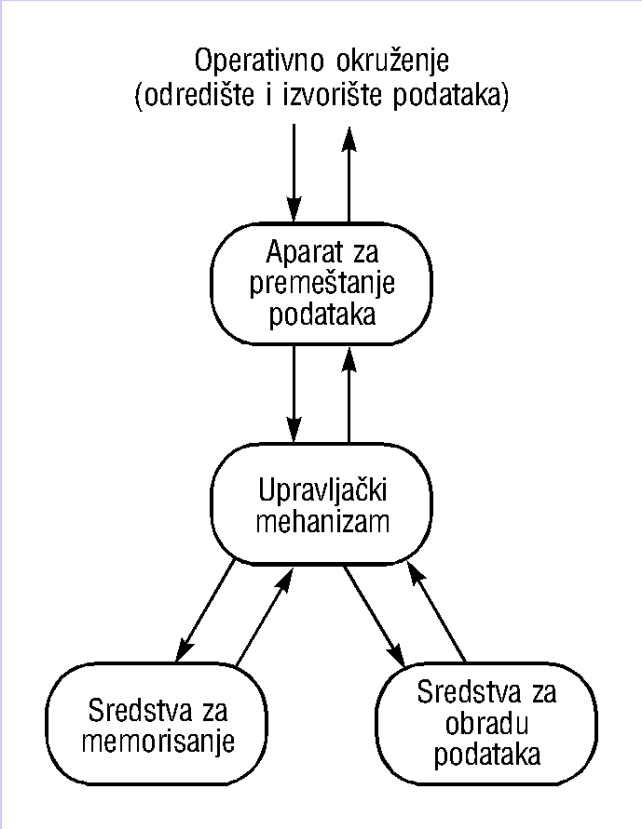
- Ra~unar je kompleksan sistem sastavljen od velikog broja elektronskih komponenata
- Kako se funkcionisanje ra~unara mo`e najefikasnije opisati i rad sa njim uspešno savladati ?
- Rešenje se ogleda u hijerarhijskoj organizaciji. Svaki korisnik treba da poznaje detalje koji su njemu od interesa.

Hijerarhijska organizacija računara – struktura i funkcija

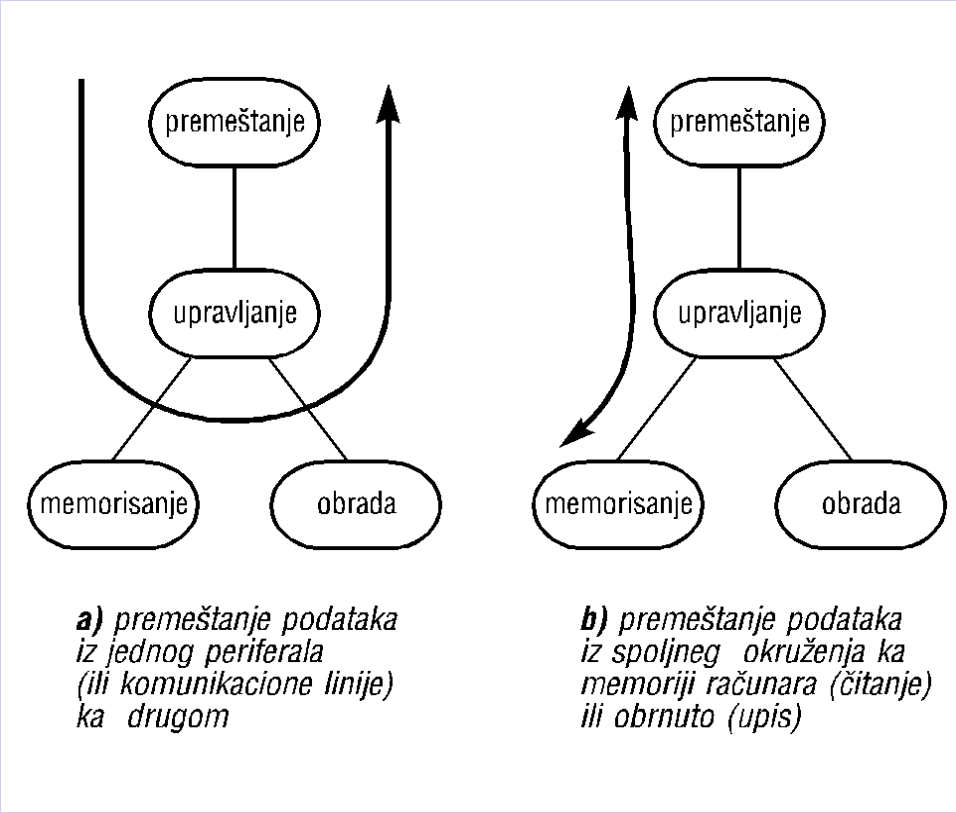
- Na svakom nivou projektant se suočava sa :
  - strukturom – ukazuje na medjusobni odnos komponenata
  - funkcijom – opisuje rad svake individualne komponente koja predstavlja deo sistema

Hijerarhijska organizacija računara – funkcija

- Osnovne funkcije koje računar treba da obavi su:
  - obrada podataka
  - memorisanje podataka
  - premeštanje podataka
  - upravljanje



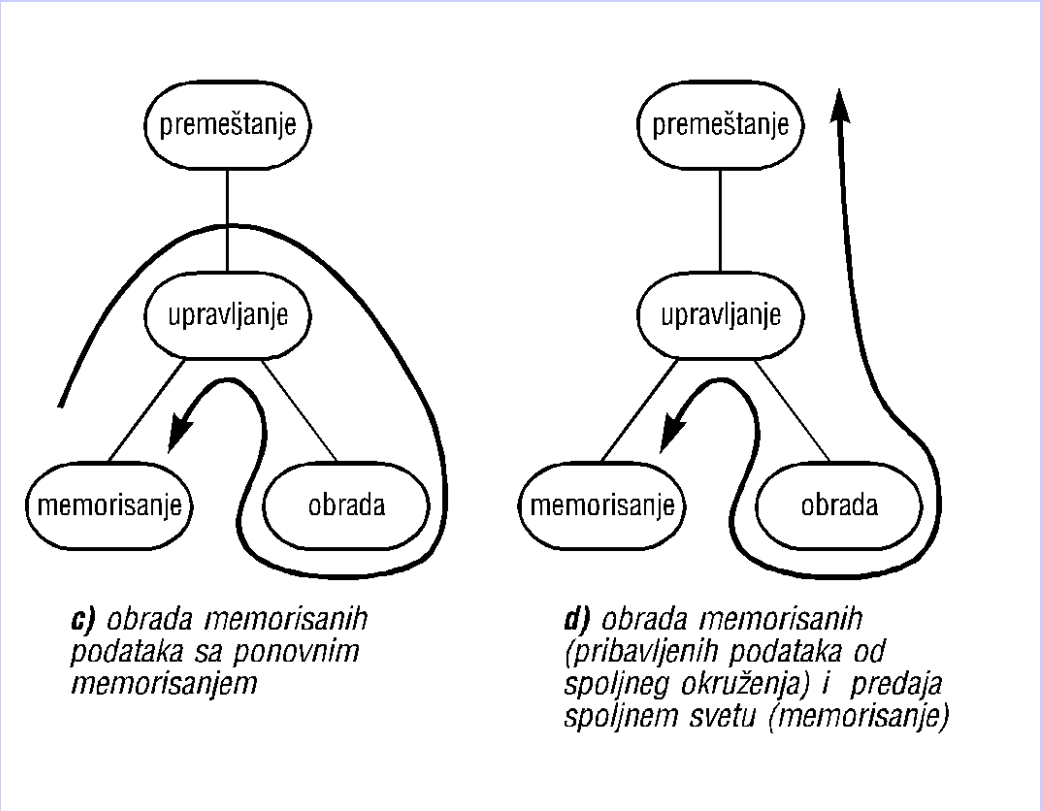
Operacije koje obavlja računar



a) *premeštanje podataka iz jednog periferala (ili komunikacione linije) ka drugom*

b) *premeštanje podataka iz spoljneg okruženja ka memoriji računara (čitanje) ili obrnuto (upis)*

Operacije koje obavlja računar – prod.

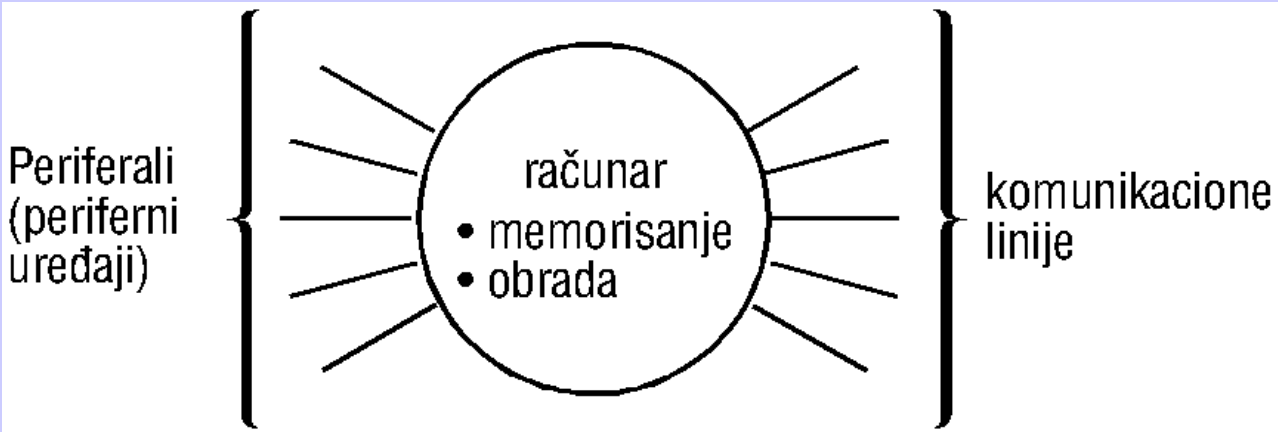


c) *obrada memorisanih podataka sa ponovnim memorisanjem*

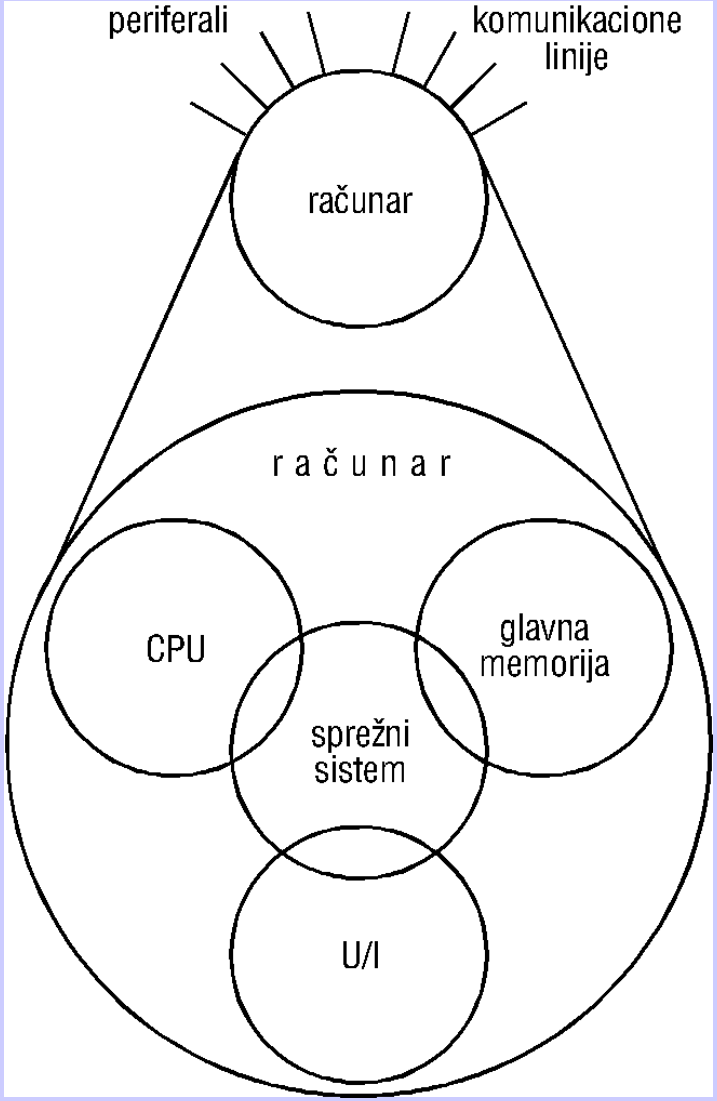
d) *obrada memorisanih (pribavljenih podataka od spoljneg okruženja) i predaja spoljnem svetu (memorisanje)*

Hijerarhijska organizacija računara – struktura

- Funkcionalne komponente računara:
  - centralna procesorska jedinica (CPU)
  - glavna memorija
  - ulazno-izlazni podsistem
  - sprežni sistem



Globalna struktura računara

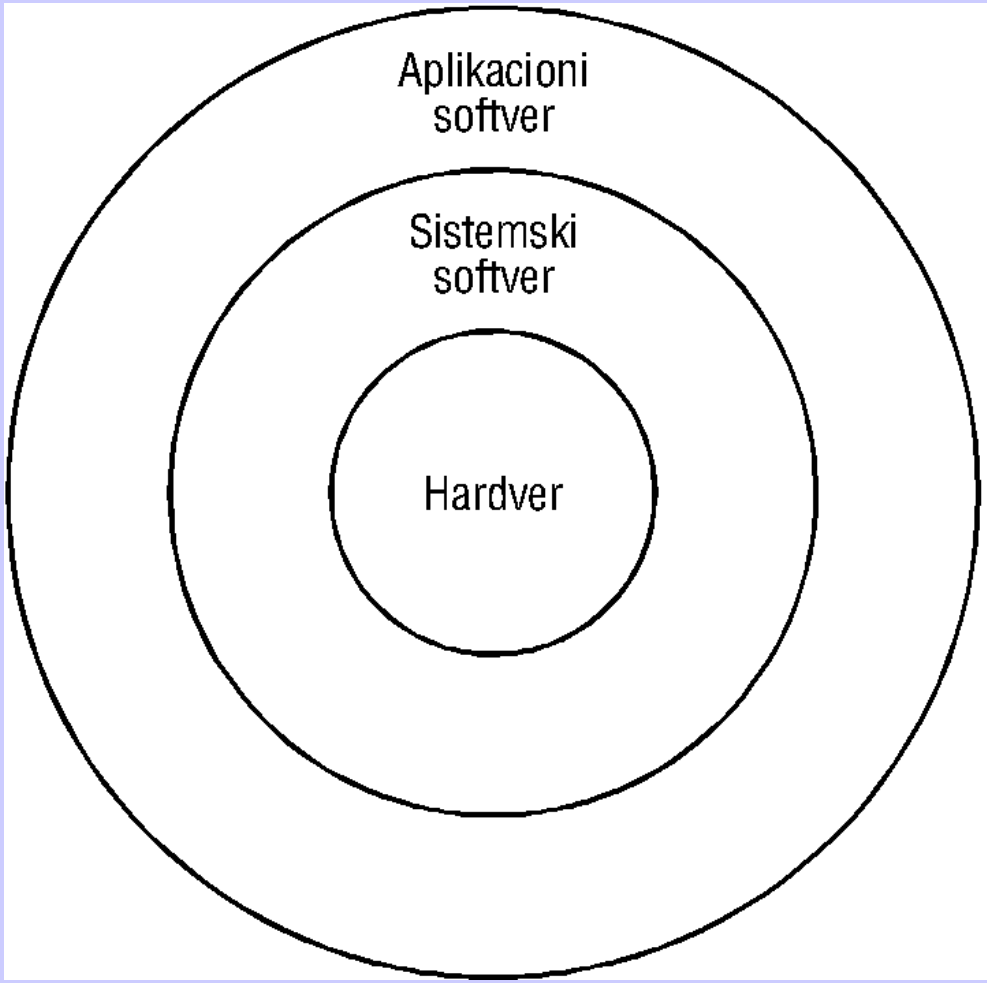


Osnovi o računarima

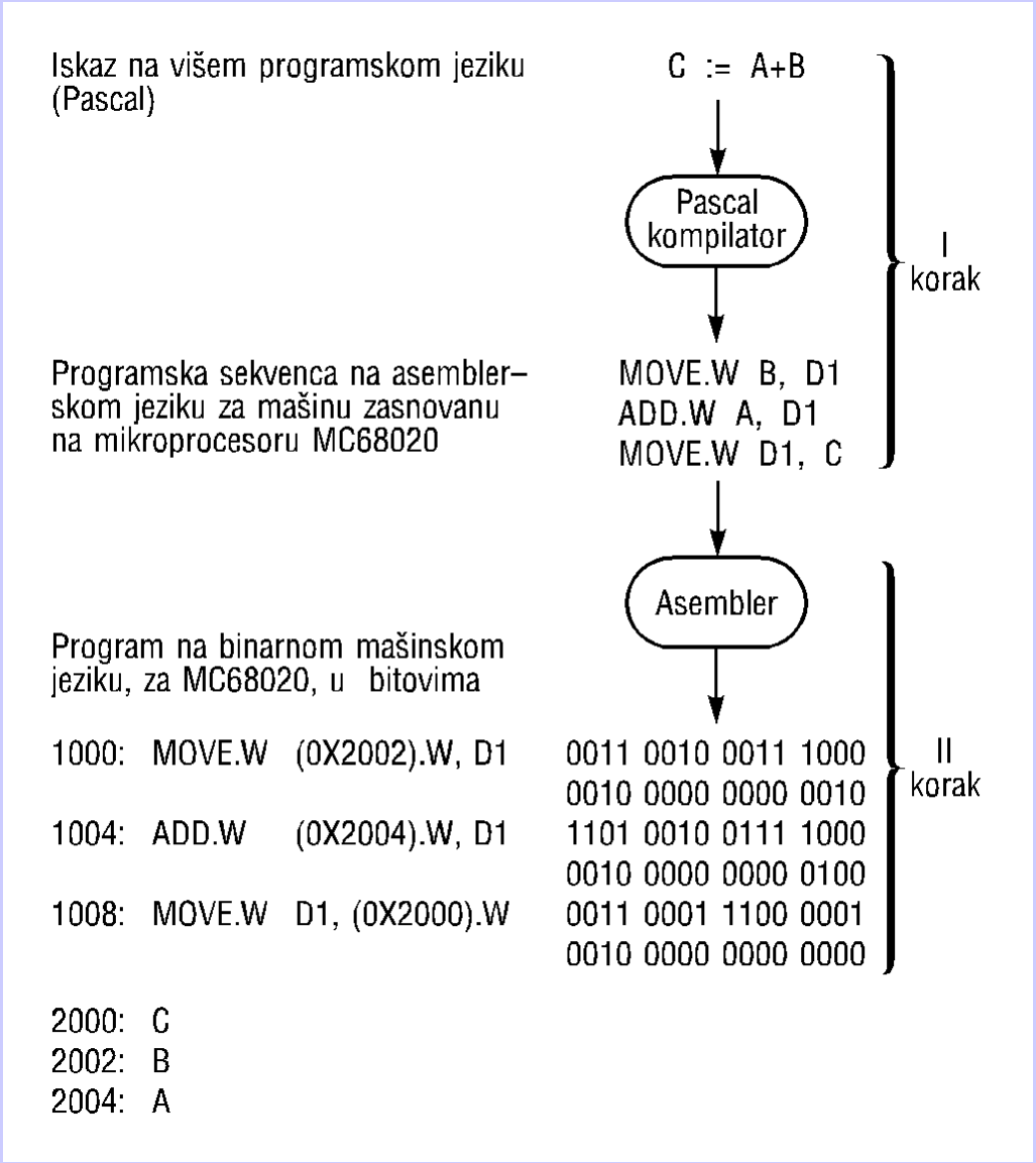
I poglavlje

- Jezici i prevodioci-

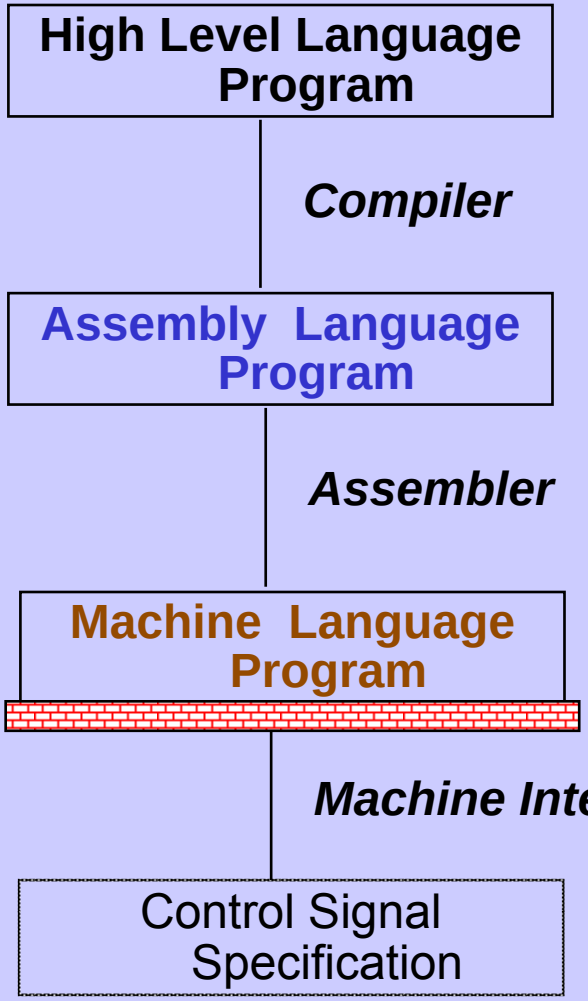
Odnos hardver, sistremski softver i aplikacioni softver kod računara



Pascal iskaz preveden u asemlerski jezik, a nakon toga asembliran u mašinski jezik



Levels of Representation



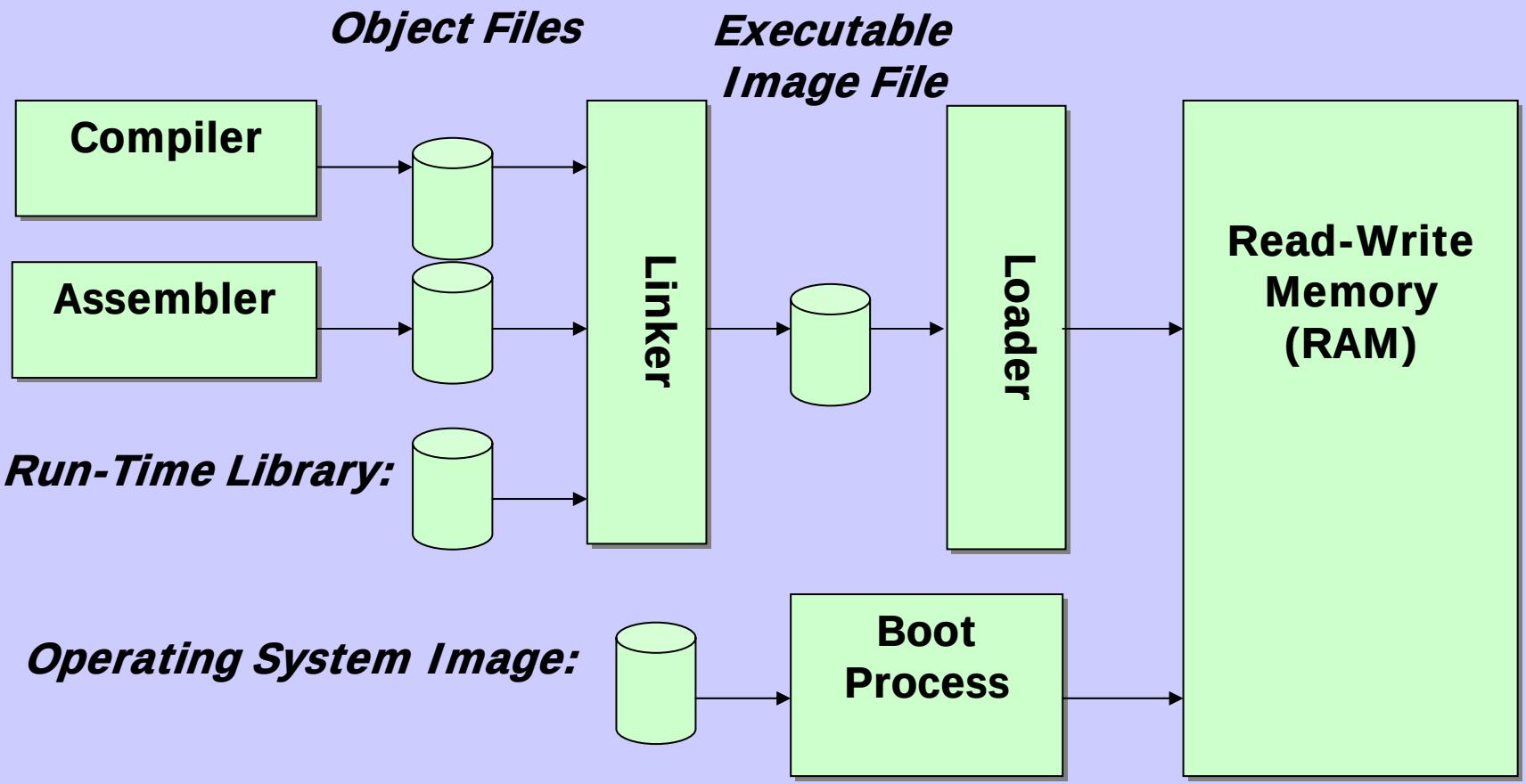

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw $15,   0($2)
lw $16,   4($2)
sw $16,   0($2)
sw $15,   4($2)
```

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

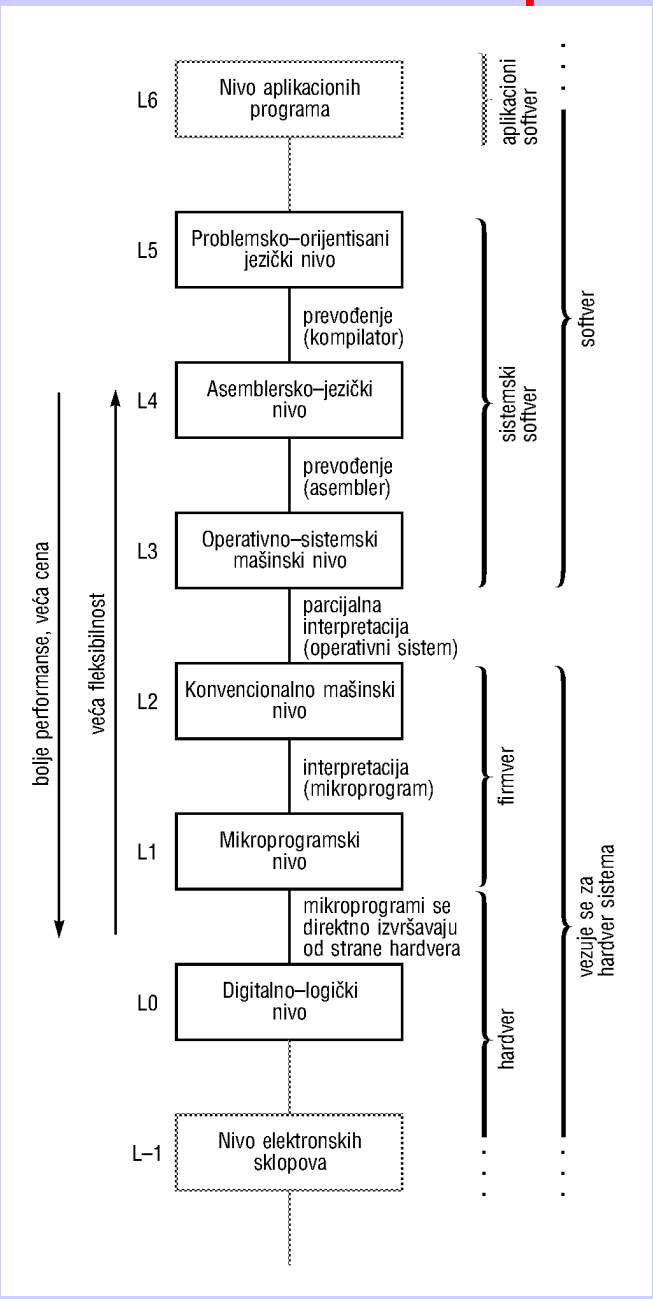
ALUOP[0:3] <= InstReg[9:11] & MASK

The build and load process for desktop application programs.



Hijerarhijska organizacija računara - koncept nivoa

- Nivo aplikacionog programa
- Nivo višeg programskog jezika
- Nivo mašinskog kôda (asemblerski jezik)
- Upravljački nivo
- Nivo funkcionalne jedinice
- Logička kola, tranzistori i veze



Generacije računarskih jezika

- Globalno razlikujemo četiri klase računarskih jezika

Generacija	Opis
prva generacija	mašinski jezik
druga generacija	asemblerski jezik
treća generacija	viši programski jezici (HLL)
četvrta generacija	novi jezici

Prva generacija – mašinski jezik

Karakteristike:

- svaka instrukcija, na hardverskom nivou, direktno upravlja radom mašine, tj. pojedinim gradivnim blokovima.
- instrukcije su numeričke, predstavljene u formi binarnih oblika od 0 i 1
- programiranje je naporno i podložno velikom broju grešaka
- efikasnost programiranja je niska
- programi nerazumljivi korisniku
- direktno se pristupa resursima mašine
- veća brzina izvršenja programa
- efikasnije korišćenje memorije

Druga generacija – asemblerski jezik

Karakteristike:

- svaka instrukcija se predstavlja mnemonikom, kao na primer ADD
- korespondencija izmedju asemblerskih i mašinskih instrukcija je jedan-na-prema-jedan
- postoje i direktive koje nemaju izvršno dejstvo, ali programu na asemblerskom jeziku olakšavaju prevodjenje, dodelu memorije i segmentaciju programa
- direktno se pristupa resursima mašine
- veća brzina izvršenja programa
- efikasnije korišćenje memorije

Treća generacija – HLL

Karakteristike:

- kompajler prevodi programske iskaze u odgovarajuće sekvence instrukcija na mašinskom nivou
- u principu jedan iskaz na HLL-u (*High Level Language*) se prevodi u *n* (n >= 1) instrukcija na mašinskom (asemblerskom) jeziku
- programiranje je jednostavnije
- efikasnost je veća
- ispravljanje grešaka lakše
- nema direktni pristup resursima mašine
- neefikasno iskorišćenje memorije
- duži programi

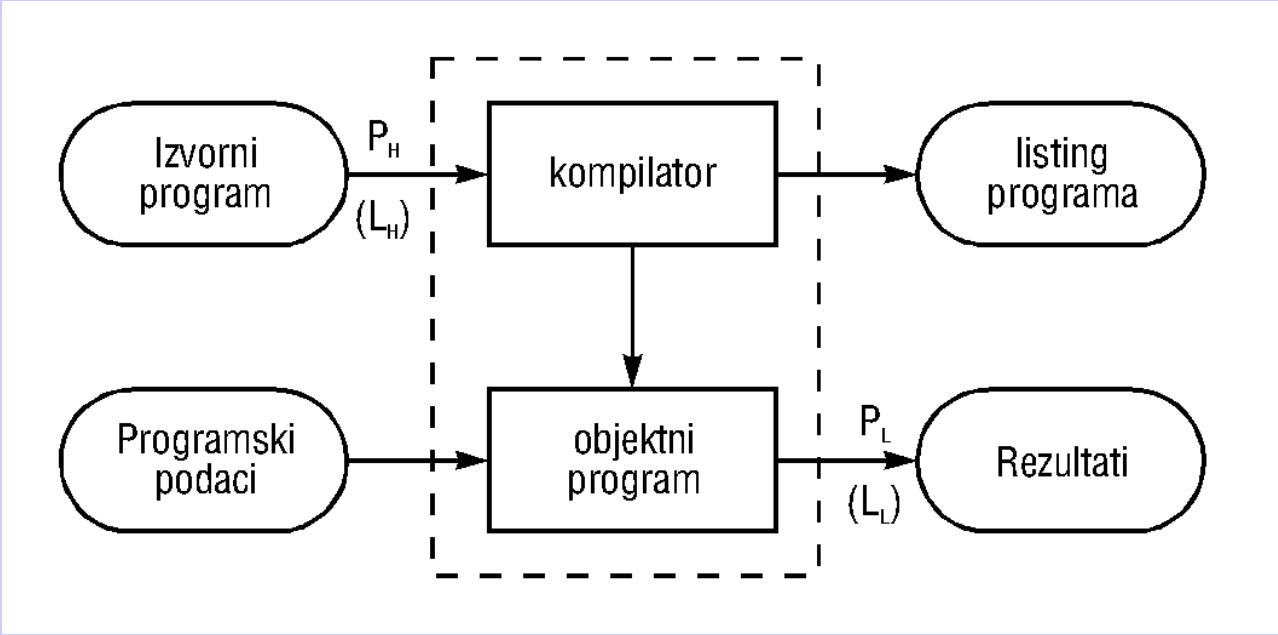
Četvrta generacija – novi jezici

Novi tipovi računarskih jezika se karakterišu sledećim osobinama:

- implementiraju veštačku inteligenciju (primer je LISP)
- jezici za pristup bazama podataka (primer je SQL)
- objektno-orijentisani jezici (primeri su C++, Java i dr.)

Mehanizmi za prevodjenje sa HLL-a na mašinski kôd

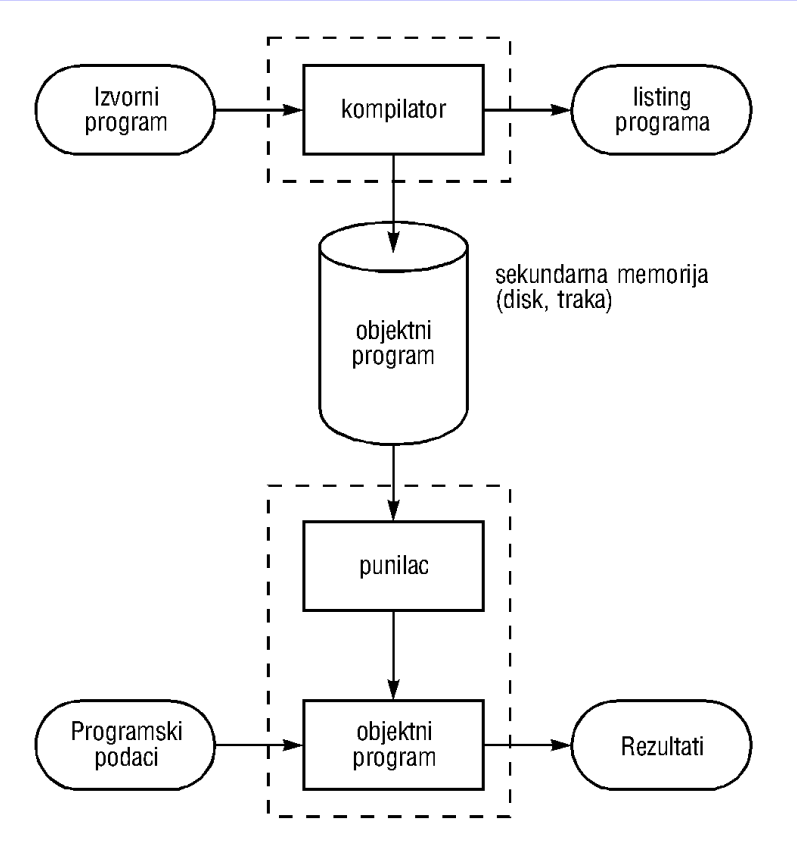
kompilacija – kompilator je program koji na svom ulazu prihvata program napisan na nekom od programskih jezika nazvan izvorni program, a na svom izlazu generiše ekvivalentan program na mašinskom kôdu nazvan objektni program, tj. program u formi mašinskog kôda koji se može direktno izvršavati od strane hardvera računara



Ciklus kompilacije

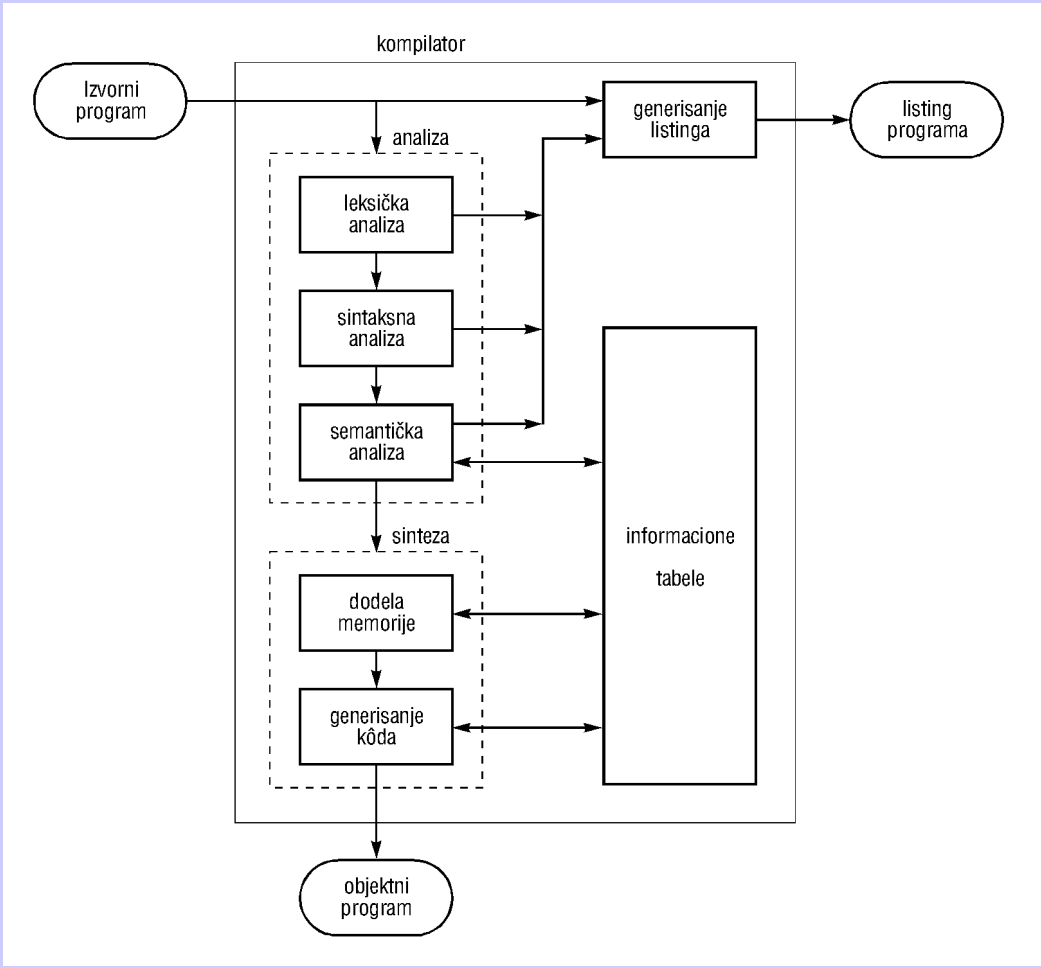
- Objektni program koji se generiše na izlazu kompilatora upisuje se u sekundarnu memoriju

- Korisnički program poznat kao punilac (*loader*) smešta program iz sekundarne meorije u glavnu čime je objektni program spreman za izvršenje



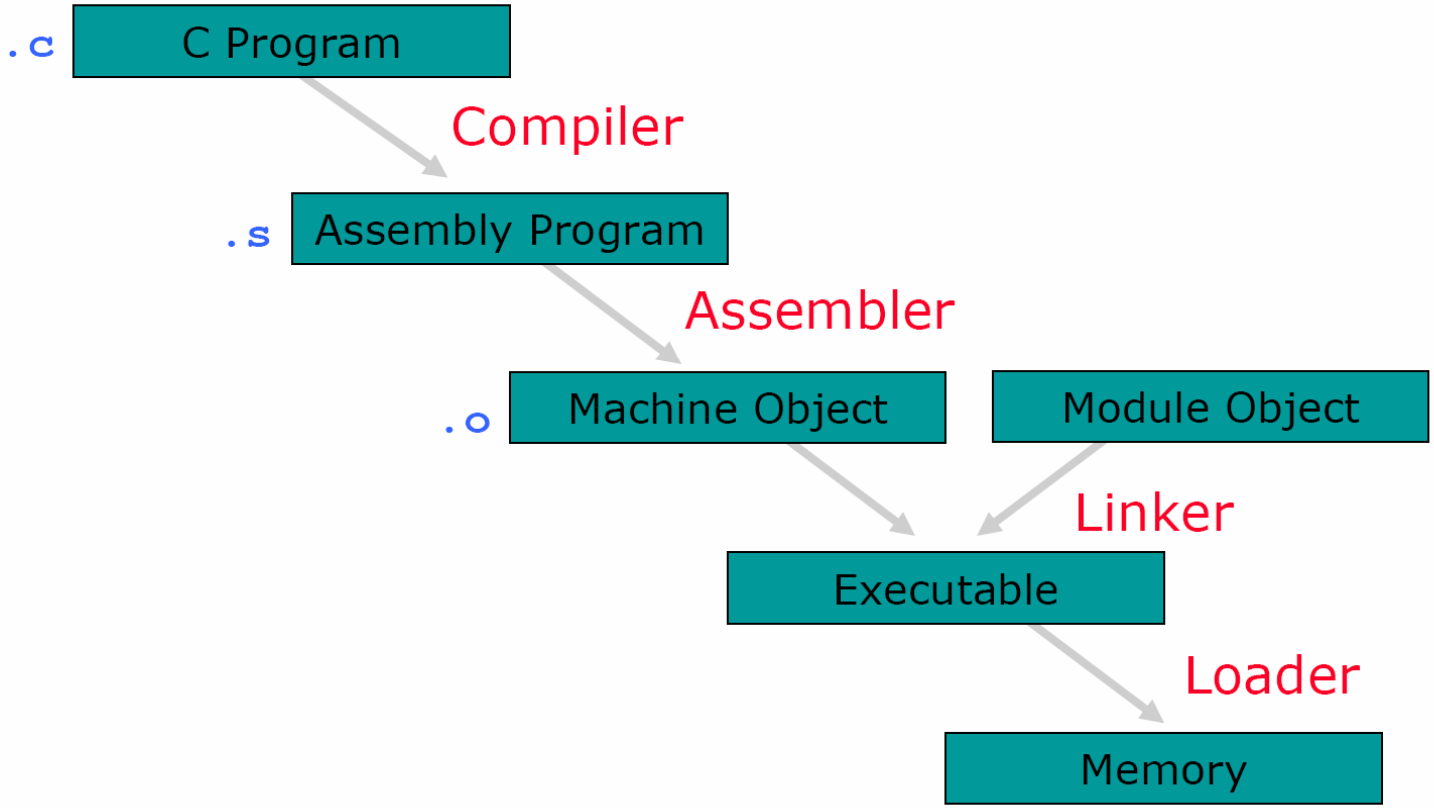
Proces kompilacije

- Proces kompilacije se može podeliti na dva glavna dela:
  - analiza izvornog programa
  - sinteza objektnog programa



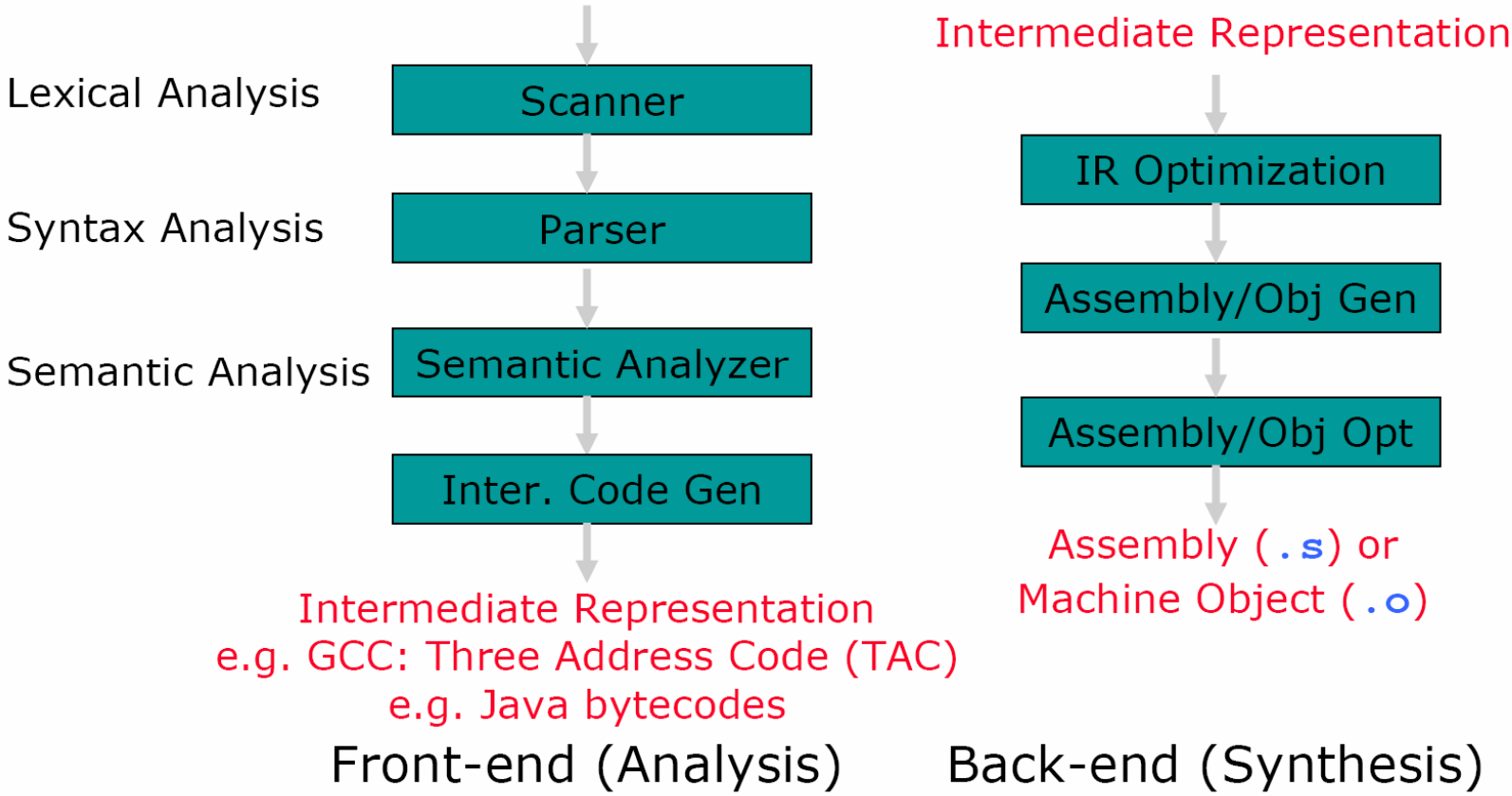
Translation Hierarchy

- High-level → Assembly → Machine



Compiler Phase

- Compiler phases



Intermediate Code Generation

- Three-address code (TAC)

```
j = 2 * i + 1;
if (j >= n)
   j = 2 * i + 3;
return a[j];
```

```
        t1 = 2 * i
        t2 = t1 + 1
        j = t2
        t3 = j < n
        if t3 goto L0
        t4 = 2 * i
        t5 = t4 + 3
        j = t5
L0:     t6 = a[j]
        return t6
```

Preview: Code Optimisation

- Goal: Improve performance by:
  - Removing redundant work
    - Unreachable code
    - Common-subexpression elimination
    - Induction variable elimination
  - Creating simpler operations
    - Dealing with constants in the compiler
    - Strength reducing multiplies (convert to shifts)
  - Managing registers well

$$Execution\ Time = Instructions \times CPI \times Clock\ Cycle\ Time$$

Code Optimisation

- Example

```
        t1 = 2 * i
        t2 = t1 + 1
        j = t2
        t3 = j < n
        if t3 goto L0
        t4 = 2 * i
        t5 = t4 + 3
        j = t5
L0:     t6 = a[j]
        return t6
```

```
        t1 = 2 * i

        j = t1 + 1
        t3 = j < n
        if t3 goto L0

        j = t1 + 3

L0:     t6 = a[j]
        return t6
```

- Copy propagation
- Common subexpression elimination

Optimizing Compilers

- Provide efficient mapping of program to machine
  - Code selection and ordering
  - eliminating minor inefficiencies
  - register allocation
- Don't (usually) improve asymptotic efficiency
  - Up to programmer to select best overall algorithm
  - Big-O savings are (often) more important than constant factors
    - but constant factors also matter

- Optimization types
  - Local: inside basic blocks
  - Global: across basic blocks e.g. loops

Limitations of Optimizing Compilers

- Operate Under Fundamental Constraints
  1. Improved code has same output
  2. Improved code has same side effects
  3. Improved code is as correct as original code

- Most analysis is performed only within procedures
  - Whole-program analysis is too expensive in most cases
- Most analysis is based only on static information
  - Compiler has difficulty anticipating run-time inputs
  - Profile driven compilation becoming more prevalent (JIT)

- **When in doubt, the compiler must be conservative**

Constant Folding

- Detect & combine values that will be constant.
- Respect laws of arithmetic on target machine.

```
a = 10 * 5 + 6 - b;
```

```
_t0 = 10 ;
_t1 = 5 ;
_t2 = _t0 * _t1 ;
_t3 = 6 ;
_t4 = _t2  * _t3 ;
_t5 = _t4  - b ;
a = _t5 ;
```

```
_t0 = 56 ;
_t1 = _t0 - b ;
a = _t1 ;
```

Constant Propagation

- When x gets assigned a constant c, replace uses of x with uses of c, as long as x remains unchanged.

```
_tmp4 = 0 ;
f0 = _tmp4 ;
_tmp5 = 1 ;
f1 = _tmp5 ;
_tmp6 = 2 ;
i = _tmp6 ;
```

```
f0 = 0 ;
f1 = 1 ;
i = 2 ;
```

- VERY useful for turning constants into immediates, for code generator.

Constant Propagation – cont.

```
_tmp0 = 12 ;
_tmp1 = arr + _tmp0 ;
_tmp2 = *(_tmp1) ;
```

```
addiu $t0, $zero, 12
lw $t1, -8($fp)
addu $t2, $t1, $t0
lw $t3, 0($t2)
```

- Replace explicit add instructions with offsets or immediates in the MIPS instruction.

```
_tmp0 = *(arr + 12) ;
```

```
lw $t0, -8($fp)
lw $t1, 12($t0)
```

Reduction in Strenght

- Some operations are faster (lower CPI) than others:
  - Multiplication slower than Addition
  - Mult or Divide by 2^n slower than shift left or right
  - Multiplies take 12 cycles on MIPS R2000 ALU instructions take 1 cycle
- Can replace complex instruction with equivalent simple instructions
- Correctness:
  - Overflow behavior preserved?
  - Other side-effects of operation? (interrupts)

Reduction in Strenght – cont.

- How to replace multiplication with addition?
- Often presents in *for* loops & array walks.

```
while (i<100) arr[i++] = 0;
```

```
L0:    _tmp2 = i < 100;
   IfZ _tmp2 Goto _L1 ;
   _tmp4 = 4 * i ;
   _tmp5 = arr + _tmp4 ;
   *(_tmp5) =  0 ;
   i = i + 1 ;
   Goto L0 ;
L1:
```

```
   _tmp4 = arr ;
L0:    _tmp2 = i < 100;
   IfZ _tmp2 Goto _L1 ;
   *_tmp4 =  0;
   _tmp4 = _tmp4 + 4;
   i = i + 1 ;
   Goto L0 ;
L1:
```

Common Sub-Expression Elimination

```
main() {
   int x, y, z;
   // They get
   // initial values
   x = (1+20)* -x;
   y = (x*x)+(x/y);
   y = z = (x/y)/(x*x);
}
```

```
tmp1 = 1 + 20 ;
tmp2 = -x ;
x = tmp1 * tmp2 ;
tmp3 = x * x ;
tmp4 = x / y ;
y = tmp3 + tmp4 ;
tmp5 = x / y ;
tmp6 = x * x ;
z = tmp5 / tmp6 ;
y = z ;
```

Common Sub-Expression Elimination – cont.

```
t1 = 1 + 20 ;
t2 = -x ;
x = t1 * t2 ;
t3 = x * x ;
t4 = x / y ;
y = t3 + t4 ;
t5 = x / y ;
t6 = x * x ;
z = t5 / t6 ;
y = z ;
```

```
t2 = -x ;
x = 21 * t2 ;
t3 = x * x ;
t4 = x / y ;
y = t3 + t4 ;
t5 = x / y ;
z = t5 / t3 ;
y = z ;
```

Induction Variable Elimination

```
while (i<100) arr[i++] = 0;
```

- Remember replacing multiply with add in loop?
- Why not get rid of induction variable i altogether?

```
   t4 = arr ;
L0:   t2 = i < 100;
   IfZ t2 Goto _L1 ;
   *t4 =  0;
   t4 = t4 + 4;
   i = i + 1 ;
L1:
```

```
   t4 = arr ;
L0:   t2 = t4 < 400;
   IfZ t2 Goto _L1 ;
   *t4 =  0;
   t4 = t4 + 4;
L1:
```

Register Allocation & Assignement

- Registers provide fastest data access in the processor.
- Two terms:
  - What variables belong in registers? (allocation)
  - What register will hold this value? (assignment)
- Once a value is in a register, we'd like not to spill & restore it to use same value again.

Assembly Code Generation

- Example: a in $a1, i in $a0, n in $a2

```
        t1 = 2 * i

        j = t1 + 1
        t3 = j < n
        if t3 goto L0

        j = t1 + 3

L0:     t6 = a[j]
        return t6
```

```
        slli  $t0, $a0, 1

        addiu $t2, $t0, 1
        slt   $t2, $t2, $a2
        bne   $t2, $zero, L0
        addiu $t2, $t0, 3

L0:     slli  $t2, $t2, 2
        addu  $t2, $t2, $a1
        lw    $v0, 0($t2)
        jr    $ra
```

- Register allocation

Assembler

- Expands macros and pseudoinstructions as well as converts constants
- Primary purpose is to produce object file
  - Machine language instructions
  - Application data
  - Information for memory organization

Pseudoinstructions

- Assembler expands *pseudoinstructions*

```
move $t0, $t1                # Copy $t1 to $t0

addu $t0, $zero, $t1         # Actual instruction
```

- Some pseudoinstructions need a temporary register
  - Cannot use `$t`, `$s`, etc. since they may be in use
  - The `$at` register is reserved for the assembler

```
blt $t0, $t1, L1     # Goto L1 if $t0 < $t1

slt $at, $t0, $t1    # Set $at = 1 if $t0 < $t1
bne $at, $zero, L1   # Goto L1 if $at != 0
```

Object File

- Includes
  - Object header – Describes file organization
  - Text segment – Machine code
  - Data segment – Static and dynamic data
  - Relocation information – Identifies instructions/data that depend on absolute addresses when program is loaded
  - Symbol table – List of labels that are not defined (ex. external references)
  - Debugging information – Describes relationship between source code and machine instructions

Linker

- Linker combines multiple object modules
  - Identify where code/data will be placed in memory
  - Resolve code/data cross references
  - Produces executable if all references found

- Steps
  1. Place code and data modules in memory
  2. Determine the address of data and instruction labels
  3. Patch both the internal and external references

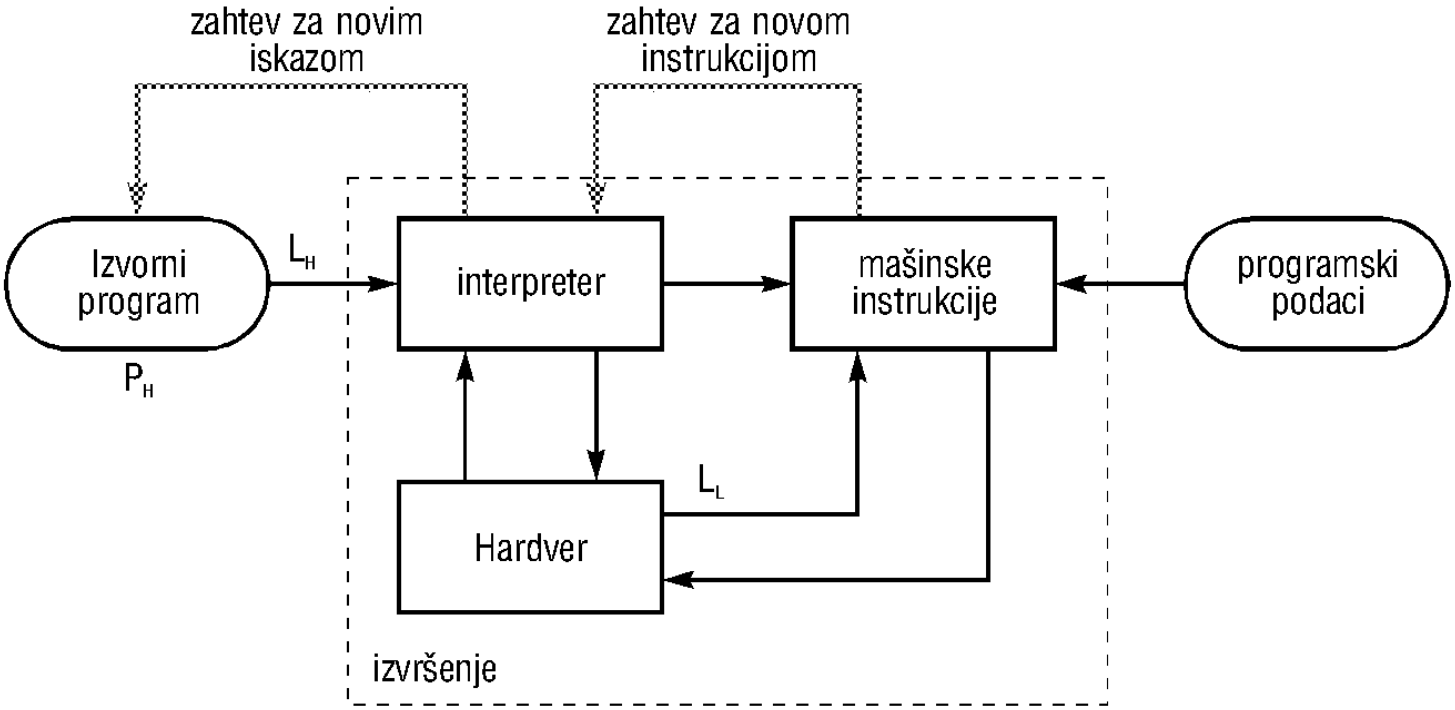
- Separation between compiler and linker makes standard libraries an efficient solution to maintaining modular code

Loader

- Loader used at run-time
  1. Reads executable file header for size of text/data segments
  2. Create address space sufficiently large
  3. Copy program from executable on disk into memory
  4. Copy arguments to main program's stack
  5. Initialize machine registers and set stack pointer
  6. Jump to start-up routine
  7. Terminate program when execution completes

Interpretacija

Interpreter uzima jednu instrukciju iz programa P_H, analizira je i uslovljava da se sa istim efektom izvrši niz instrukcija sa nivoa L_L. Ovaj proces se nastavlja sve dok se ne izvrši kompletan program



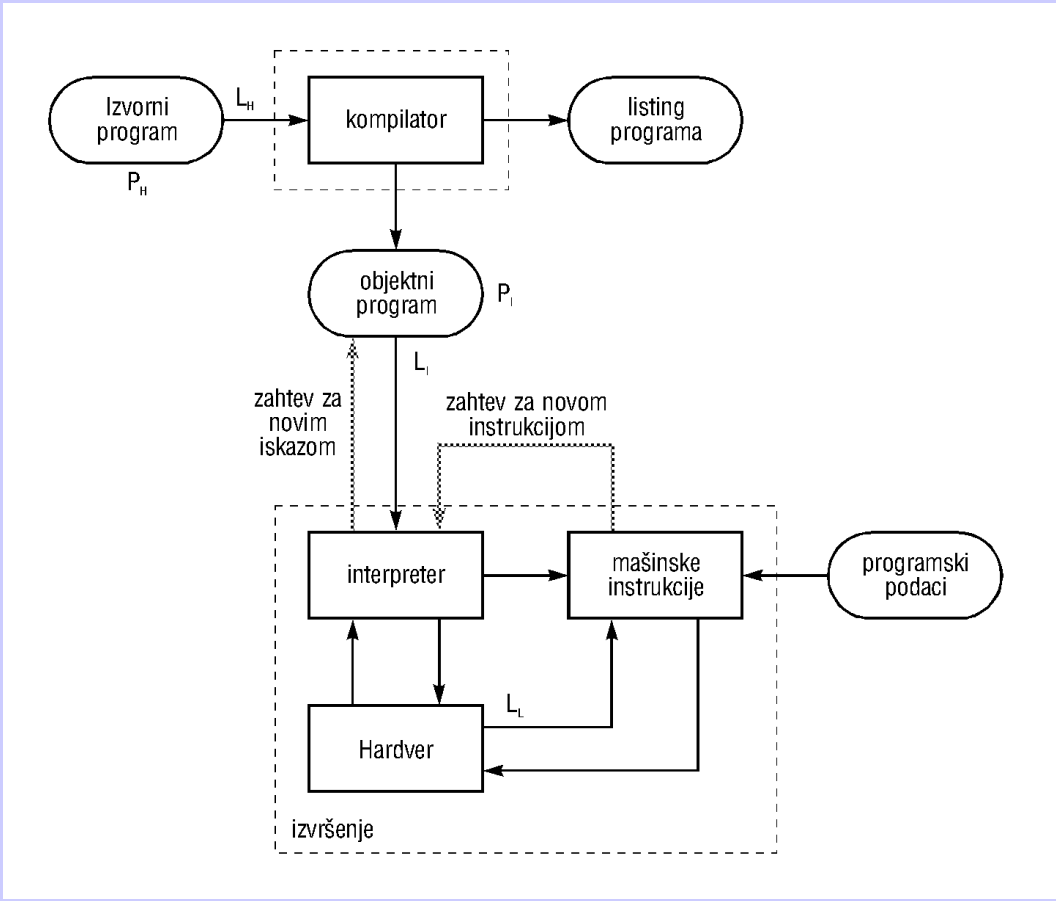
Interpretacija - prednosti i nedostaci

- Prednost interpeterskog mehanizma konverzije je ta što je interpreter relativno mali i što se u odnosu na kompilator lakše imlementira na nivou mašine.
- Nedostatak je taj što je izvršenje izvornih programa postupkom interpetacije sporije u odnosu na kompilovane programe
- Razlika u brzini izvršenja je reda 10

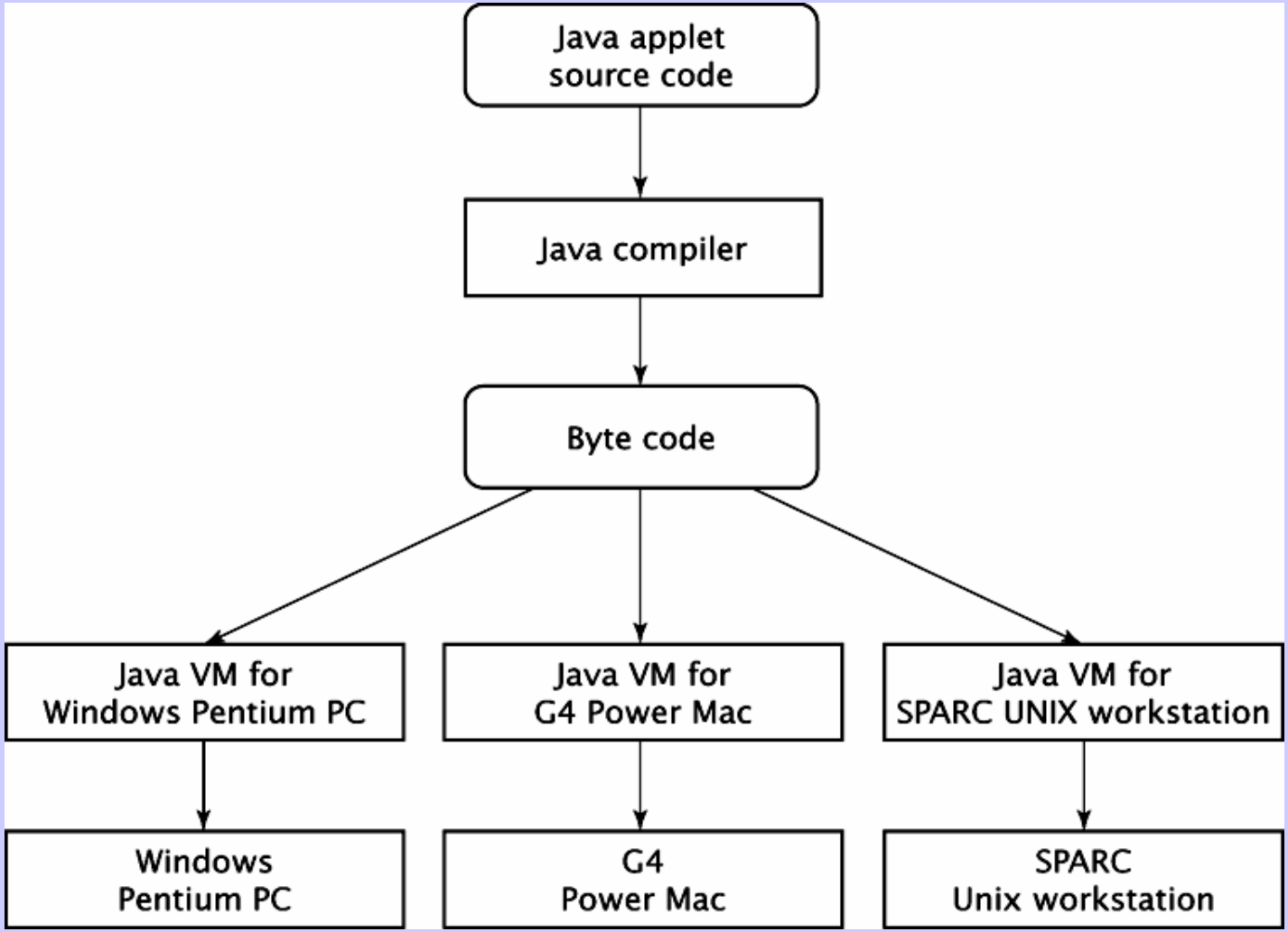
Kombinovanje interpetacije i kompilacije

Kombinovani mehanizam kompilacija-interpretacija koristi me|u-jezik L_I, koji se nalazi izme|u nivoa L_L i L_H. Program P_H, napisan na jeziku L_H, kompajlira se u program P_I na jeziku L_I. Program P_I se zatim intepretira na nivou L_L pomo}u interpretera

Prednost ove metode ogleda se u boljoj prenosivosti programa, kao i u tome što se zadr`avaju ostale dobre osobine tehnika kompilacije i interpretacije



Proces kompilacije za Java aplete



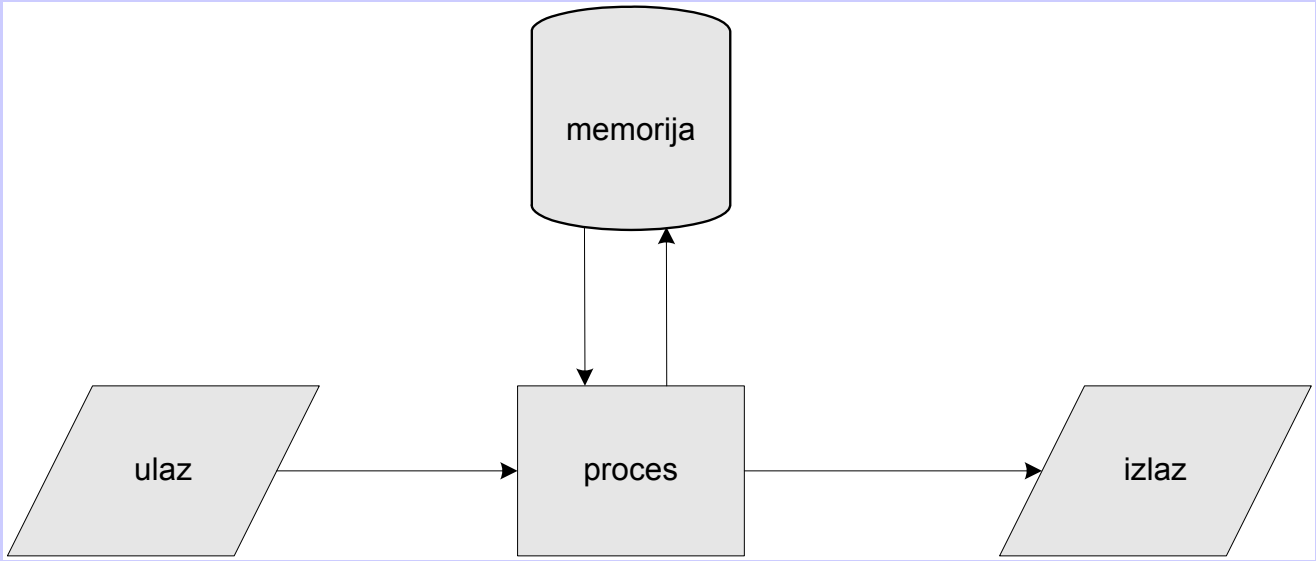
Osnovi o računarima

I poglavlje

- Odnos programer-računar -

Pogled programera na sistem

- Sa ta~ke gledišta korisnika, ra~unarski sistem je namenjen obavljanj nekog korisničkog zadatka
- Svi zadaci se obavljaju shodno modelu ulaz-proces-izlaz:
  - program prihvata ulazne podatke
  - obavlja neki tip procesiranja nad ulaznim podacima
  - generiše izlazne podatke



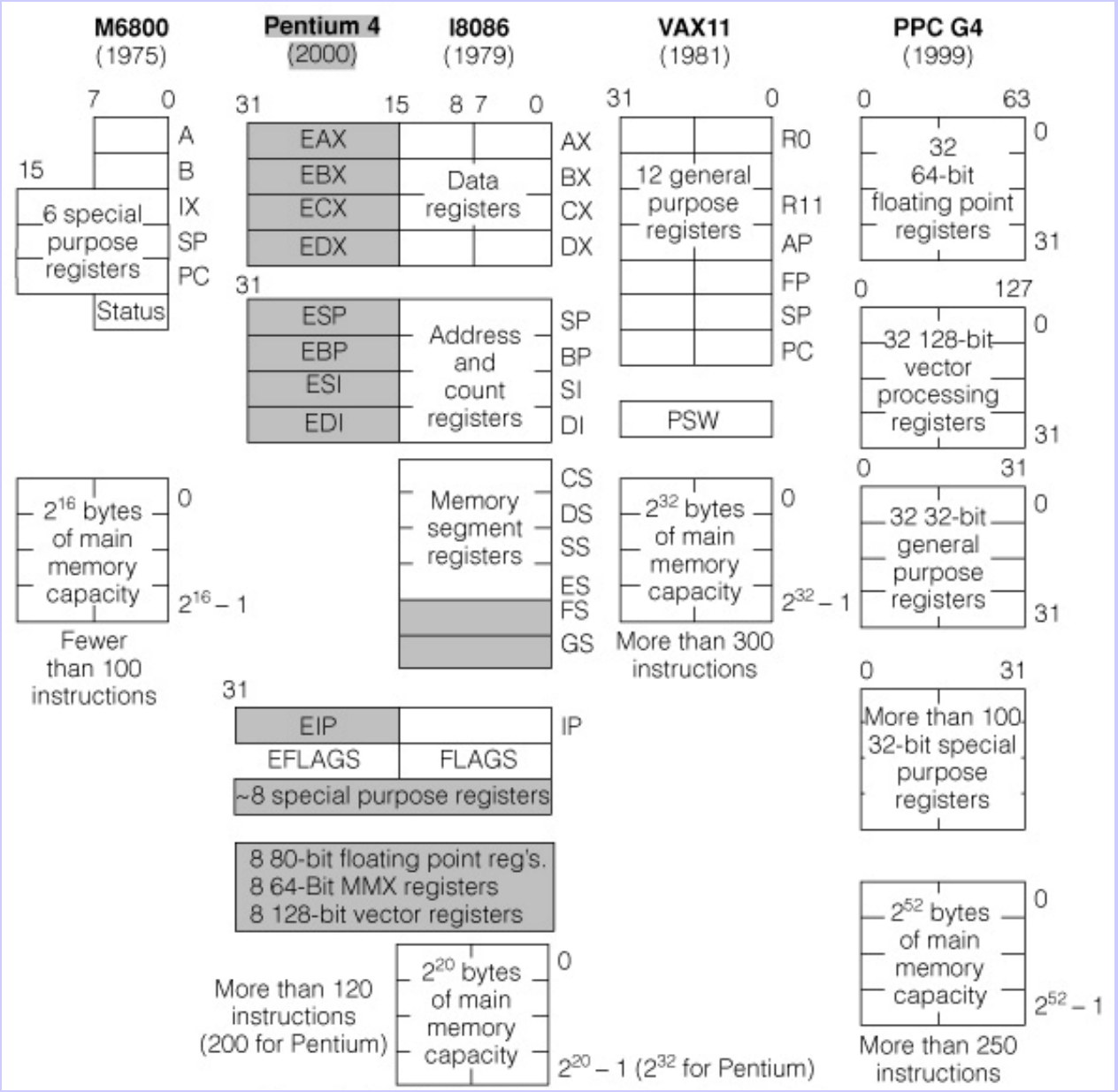
What are the components of an ISA?

- Sometimes known as *The Programmers Model* of the machine
- Storage cells
  - General and special purpose registers in the CPU
  - Many general purpose cells of same size in memory
  - Storage associated with I/O devices
- The Machine Instruction Set
  - The instruction set is the entire repertoire of machine operations
  - Makes use of storage cells, formats, and results of the fetch/execute cycle
  - i. e. Register Transfers
- The Instruction Format
  - Size and meaning of fields within the instruction
- The nature of the Fetch/Execute cycle
  - Things that are done before the operation code is known

The Computer Architect's View

- Architect is concerned with design & performance
- Designs the ISA for optimum programming utility and optimum performance of implementation
- Designs the hardware for best implementation of the instructions
- Uses performance measurement tools, such as benchmark programs, to see that goals are met
- Balances performance of building blocks such as CPU, memory, I/O devices, and interconnections
- Meets performance goals at lowest cost

Pogled programera na različite mašine

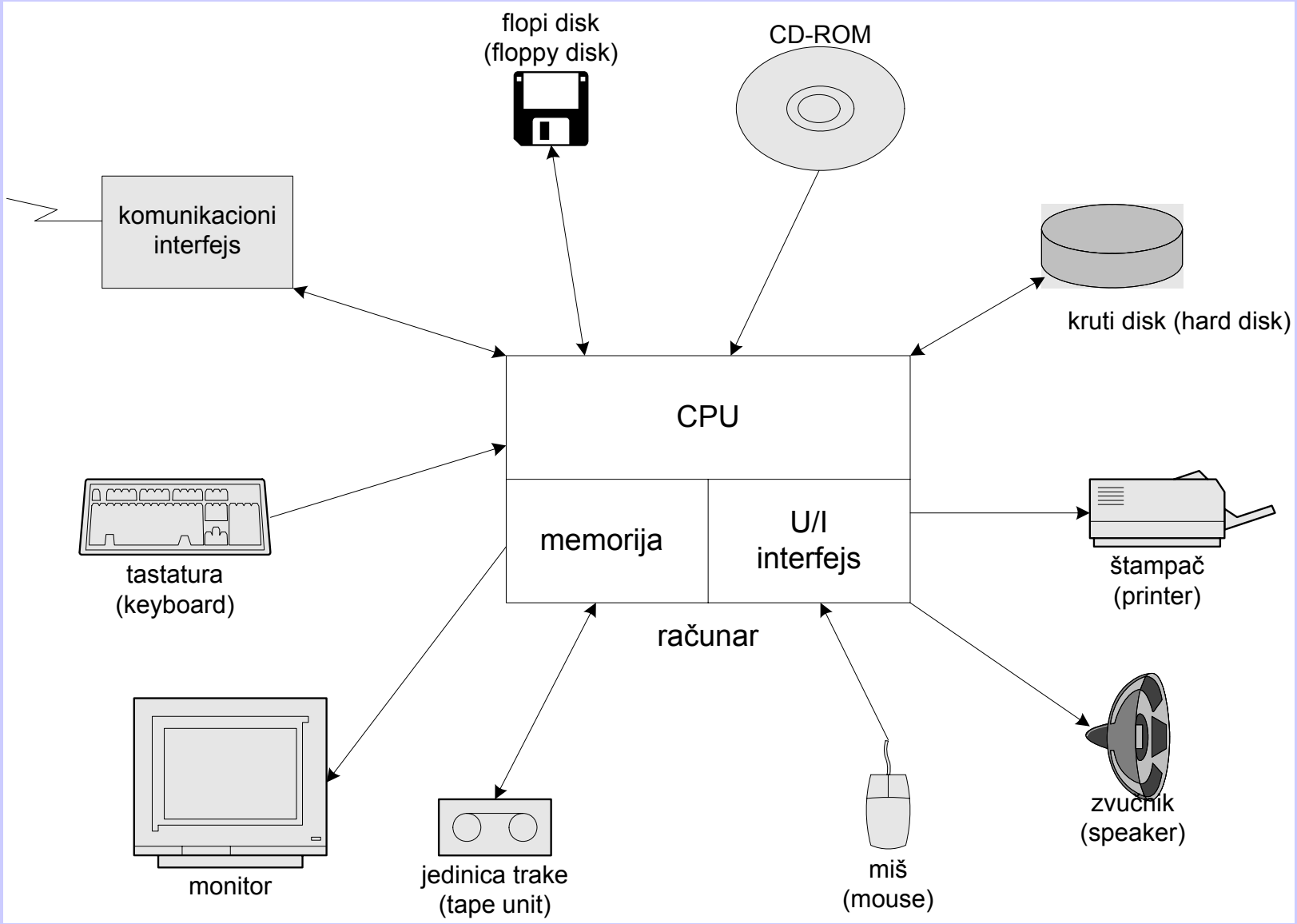


Osnovi o računarima

I poglavlje

- Tipičan računarski sistem -

Tipičan računarski sistem



Hardverske komponente računarskog sistema

- Hardversku organizaciju računarskog sistema čine sledeća četiri podsistema:
  - magistrala
  - CPU
  - ulazno-izlazni podsistem
  - memorijski podsistem

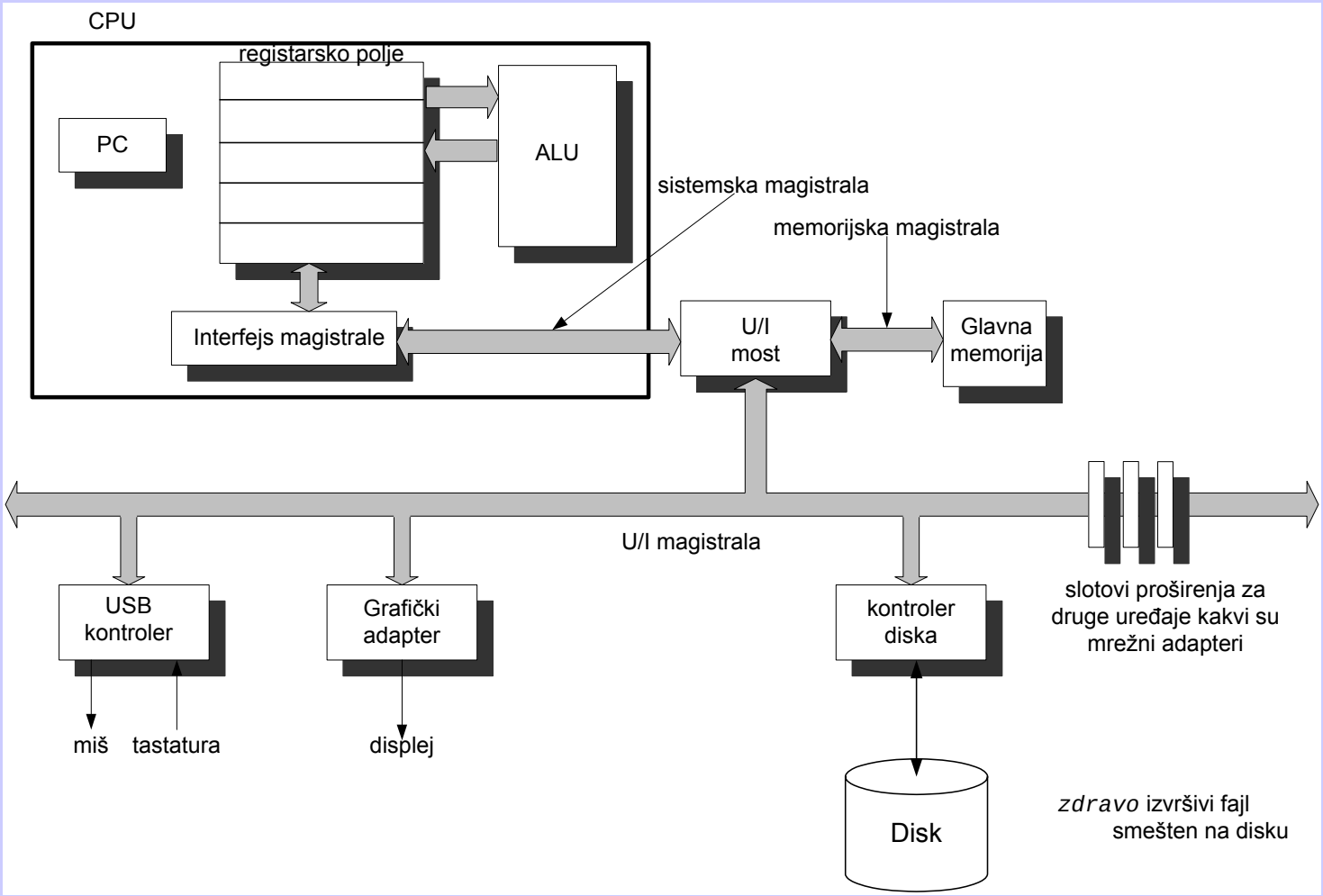
Hardverske komponente -magistrala

- **Magistrala** predstavlja skup veza (linija) po kojima se u paralelnoj formi prenose signali i napajanje.
  - obezbedjuje put preko koga sastavni delovi sistema mogu komunicirati.
  - može biti jednosmerna ili dvosmerna
  - trostatička magistrala – označava da svaka linija može biti u stanju 0, 1, ili isključeno
  - svakom magistralom mogu se prenositi tri/četiri grupe signala:
    - podaci
    - adrese
    - upravljački signali
    - takt i napajanje

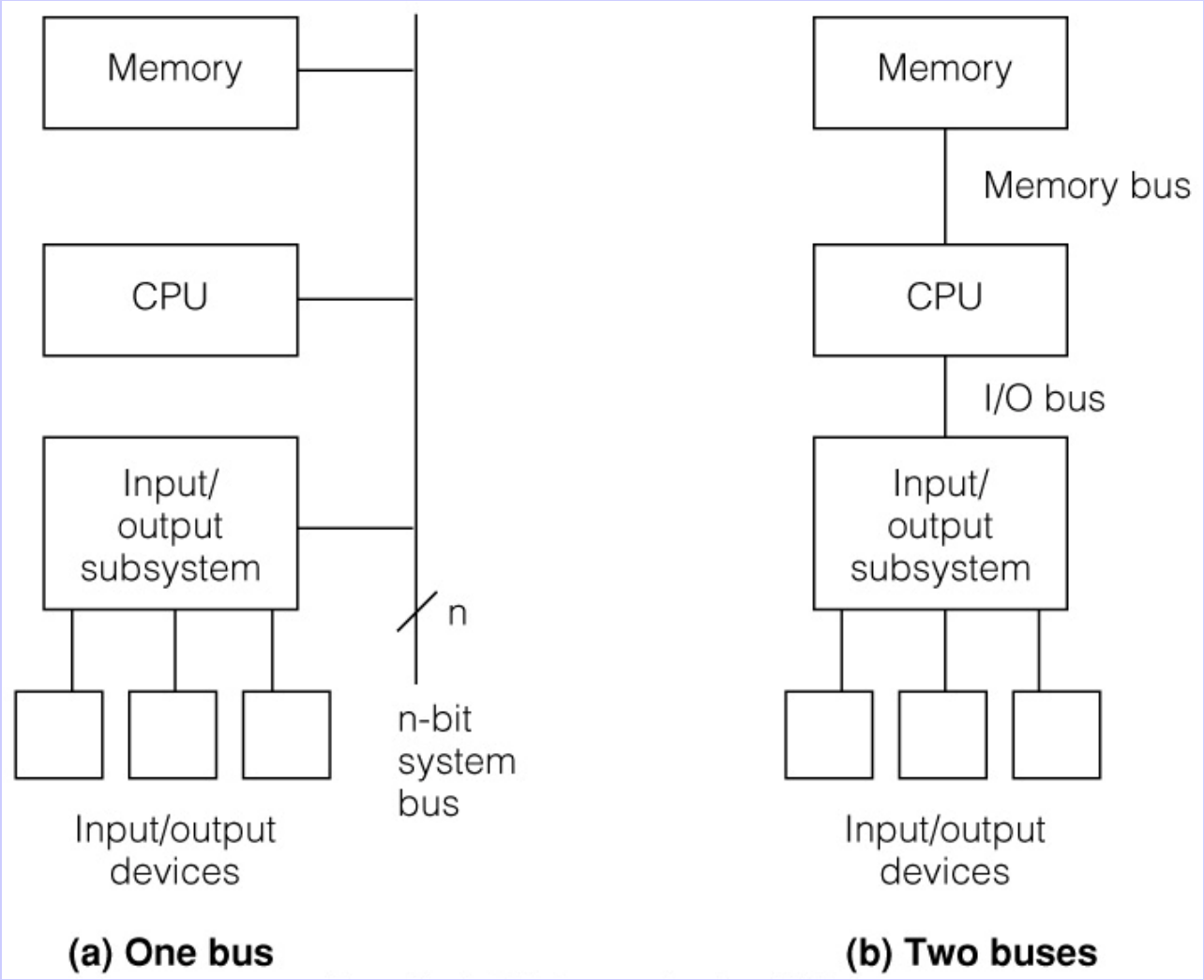
Buses as Multiplexers

- Interconnections are very important to computer
- Most connections are shared
- A bus is a time-shared connection or multiplexer
- A bus provides a data path and control
- Buses may be serial, parallel, or a combination
  - Serial buses transmit one bit at a time
  - Parallel buses transmit many bits simultaneously on many wires

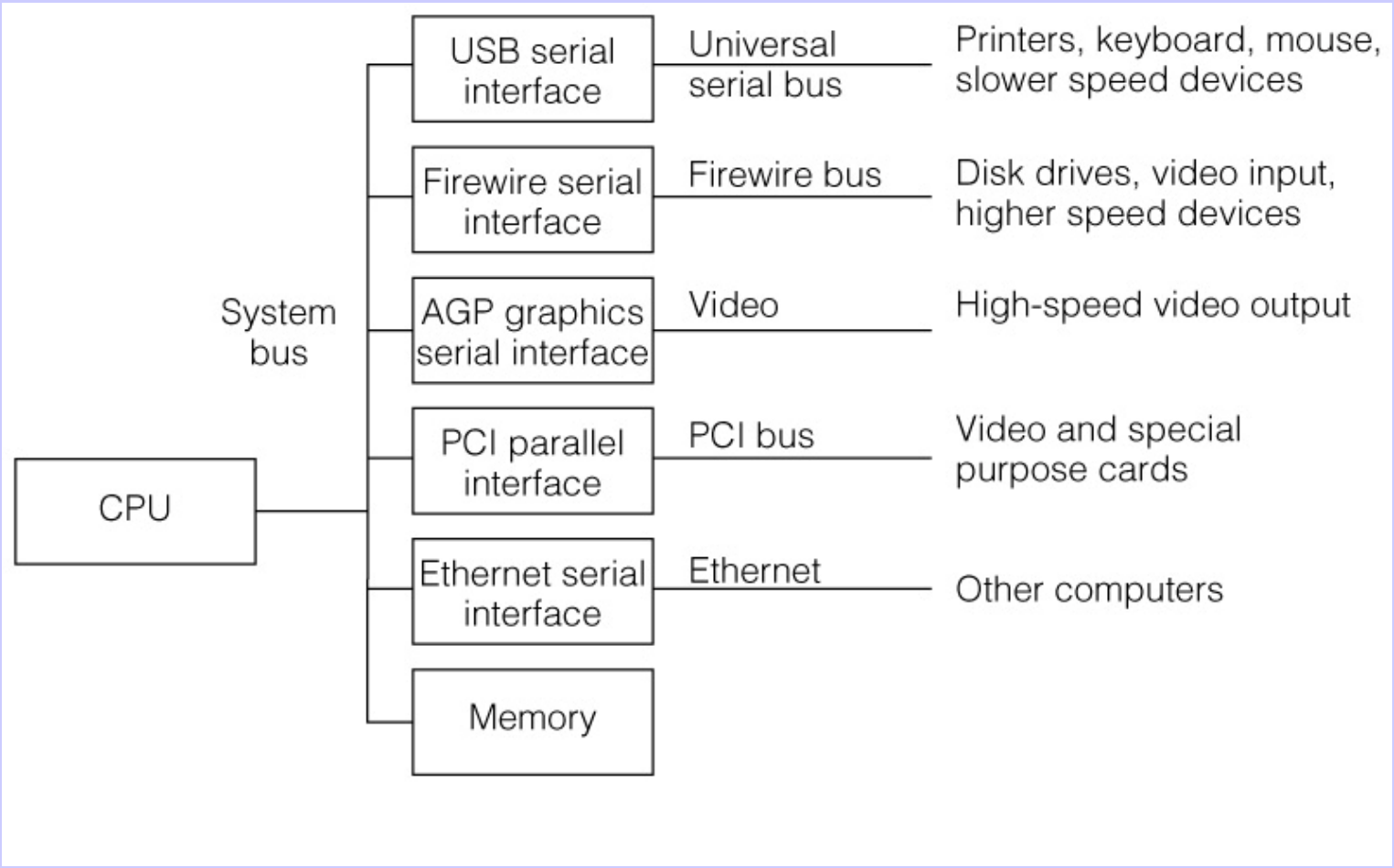
Hardverska organizacija jednog tipičnog računarskog sistema sa aspekta magistrala



One and Two Bus Architecture Examples



Tipične magistrale kod jednog računarskog sistema



Hardverske komponente- ulazno-izlazni podsistem

- **Ulazno izlazni podsistem** čine ulazno-izlazni uredjaji koji povezuju računar sa spoljnim svetom.
- Kod tipičnih sistema U/I podsistem čine sledeći U/I uredjaji:
  - tastatura i miš - unos podataka i komandi od strane korisnika
  - displej - izlaz za korisnika
  - disk - dugoročno memorisnje podataka i programa

Ulazno-izlazni uredjaji

- Svaki U/I uredjaj je povezan na U/I magistralu preko kontrolera ili adaptera

Kontroleri su ~ip-setovi (jedinstveno ili skup integrisanih kola) koji su locirani na glavnoj **sistemskoj plo~i** (~esto nazvanoj *motherboard*)

Adapter je kartica koja se smešta u slot na sistemskoj plo~i.

- Obično na svaki U/I kontroler ili adapter moguće je povezati više od jednog uredjaja ili tzv. periferala

Ulazno-izlazni uredjaji - interfejsi

- U/I interfejsi ne predstavljaju samo mehani~ke konektore koji povezuju periferal na U/I magistralu, nego sadr`e i odredjeni iznos "inteligencije"
- U/I periferali ne povezuju direktno na magistralu iz sledećih razloga :
  - postoji širok dijapazon periferala koji se razlikuju po načinu rada
  - brzina prenosa podataka koja karakteriše periferale je često sporija u odnosu na brzinu prenosa podataka izmedju memorije i CPU-a
  - periferali često koriste različite formate podataka i obim reči u odnosu na računar na koji se priključuju

Hardverske komponente - CPU

Centralna procesorska jedinica – CPU ili jednostavno rečeno procesor je mašina koja interpretira (ili izvršava) instrukcije smeštene u glavnoj memoriji

- CPU se može podeliti na sledeće gradivne blokove:
  - programsko-upravljačka jedinica (*control unit*)
  - staza podataka (*datapath*) – sadrži sledeće gradivne blokove:
    - izvršne jedinice kakve su ALU, množači, pomerači i td.
    - registre opšte namene vidljive korisniku (R0 ... Rn), programski brojač (PC), programsko statusni registar (PSW), instrukcioni registar (IR), i td.

Osnovi o računarima

I poglavlje

- Principi rada CPU-a -

Princip rada CPU-a – osnovni koraci

- čita instrukciju iz glavne memorije na koju pokazuje PC,
- interpretira bitove instrukcija,
- obavlja neku jednostavnu operaciju diktiranu od strane instrukcije, i
- ažurira PC da bi ukazao na narednu instrukciju, koja može, ali ne mora biti kao naredna u memoriji u odnosu na instrukciju, koja se tekuće izvršava.

Operacije koje obavlja CPU

- Neke od jednostavnih aktivnosti koje CPU obavlja po nalogu instrukcije su sledeće:
  - punjenje (*load*)
  - pamćenje (*store*)
  - ažuriranje (*update*)
  - U/I čitanje (*I/O read*)
  - U/I upis (*I/O write*)
  - grananje (*jump*)

Osnovi o računarima

I poglavlje

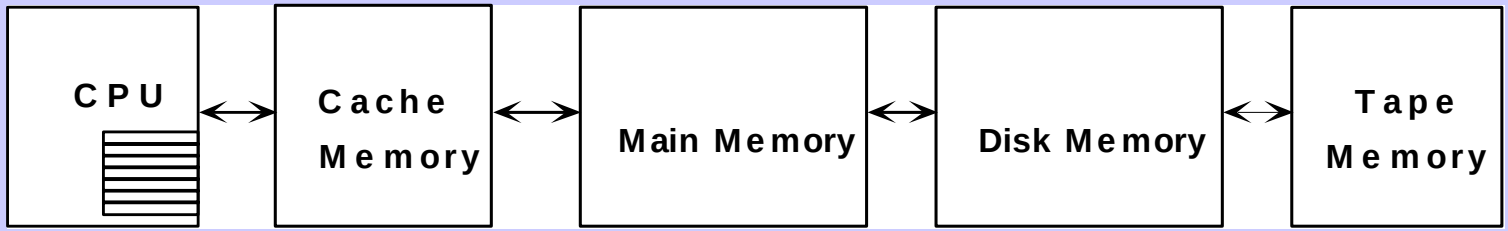
- Memorija -

Hardverske komponente - memorija

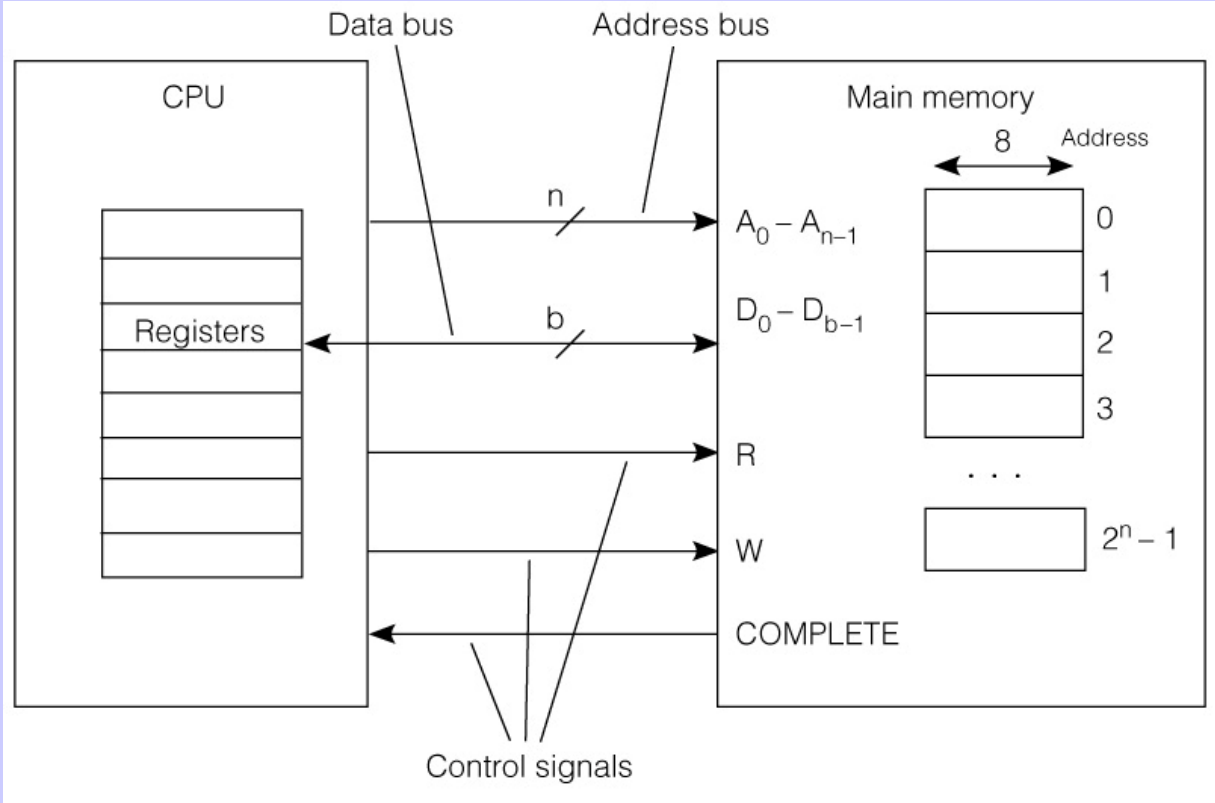
Memorija – koristi se za čuvanje programa, konstanti, medjurezultata, i rezultata izračunavanja.

- Postoji razlika u brzini rada procesora sa jedne i memorije sa druge strane koju projektanti teže da reše odgovarajućom hijerarhijskom organizacijom

- Jedan tipi~ni ra~unarski sistem sadr`i nekoliko tipova memorije, po~ev od brzih i skupih internih registara pa sve do sporih i jeftinih prenosivih diskova i traka

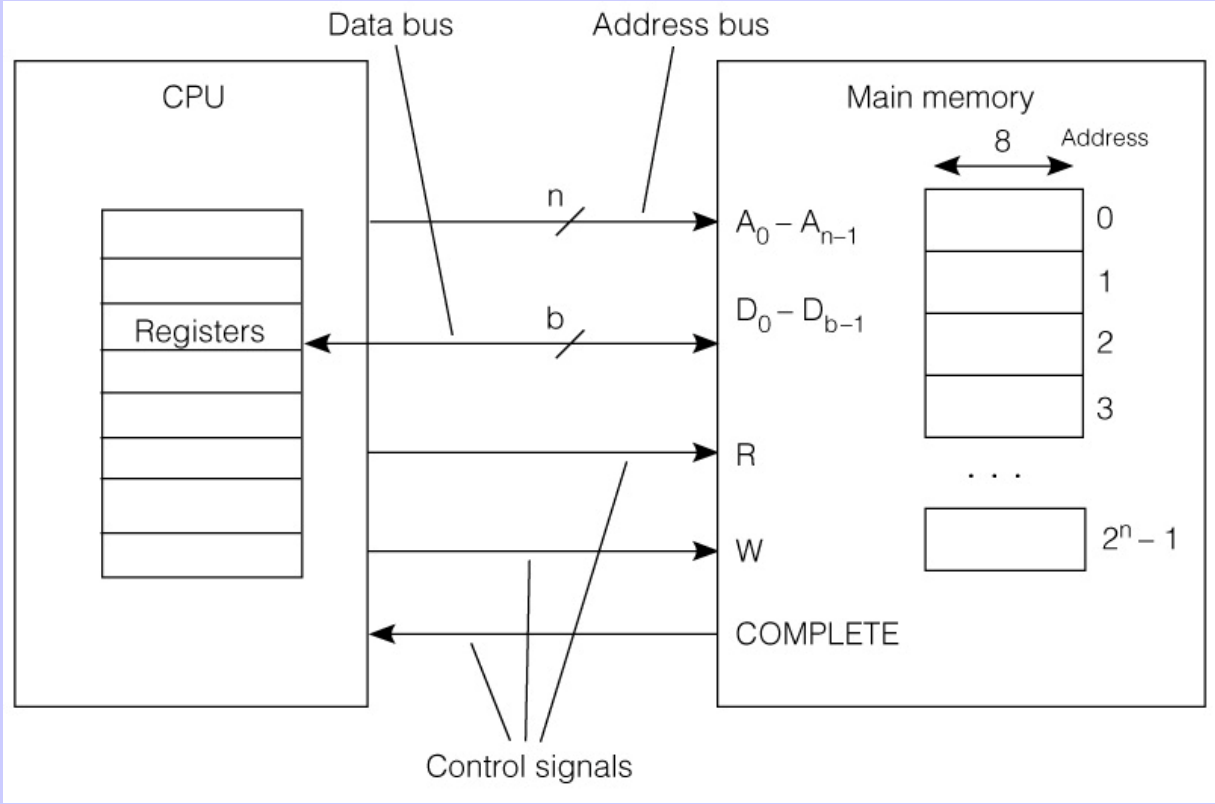


Accessing Memory—Reading from Memory



For a Memory Read:
CPU applies desired address to Address lines A_0-A_{n-1}
CPU issues Read command, R
Memory returns the value at that address on Data lines D_0-D_{b-1} and asserts the COMPLETE signal

Accessing Memory—Writing to Memory



For a Memory Write:
CPU applies desired address to Address lines A_0-A_{n-1} and and data to be written on Data lines D_0-D_{b-1}
CPU issues Write command, W
Memory asserts the COMPLETE signal when the data has been written to memory.

Memorijski sistem - karakteristike

med.mem	t_{ACC}	*BW (B/s)*	kapacitet	obim bloka	upravljanje	tehno
registri CPU-a	200 ps-1 ns	0.5–60 GB/s	256 B – 1 kB	reč 2 ili 4 B	CPU	CMOS SRAM
L1 keš memorija	5 – 10 ns	0.1-0.8 GB/s	16 – 64 kB	linija 4 – 32 B	primarni keš kontroler	CMOS SRAM
L2 keš memorija	15 –40 ns	0.1-0.3 GB/s	128 kB – 1GB	linija 4 – 128 B	sekundarni keš kontroler	CMOS SRAM
glavna memorija-GM	50 – 100 ns	20-80 MB/s	256 MB – 1GB	stranica 4 kB	MMU	CMOS DRAM
slotovi proširenja GM	75 – 500 ns	800 kB/s-300 MB/s	1 – 10 GB	stranica 4 kB	MMU	CMOS DRAM
disk keš	60 – 500 ns	900 kB/s-30MB/s	1 – 10 MB	stranica 4 kB	kontroler uredjaja	CMOS DRAM
kruti disk	5 – 50 ms	1200-6000 kB/s	100 – 500 GB	fajlovi obima MB	kontroler uredjaja	magnetni medijum
flopi disk	95 ms	100-200 kB/s	1.44 MB	fajlovi obima kB	kontroler uredjaja	magnetni medijum
CD-ROM	100 – 500 ms	500-4000 kB/s	600 MB–20 GB	fajlovi obima MB	kontroler uredjaja	optički zapis
trake (*cartrige*)	0.5 s pa naviše	2000 kB/s	1 – 10 TB	fajlovi obima MB	kontroler uredjaja	magnetni medijum

Osnovne karakteristike glavne memorije

Glavna memorija – često se naziva primarna ili radna memorija, koristi se za čuvanje programa i podataka kojima se pristupa od strane CPU-a

- Obim podatka (reči) koji se čuva u memoriji odgovara obimu magistrale
- Kapacitet glavne memorije odredjuje maksimalni broj instrukcijskih re~i i re~i podataka koje se mogu istovremeno napuniti (~uvati) u memoriju

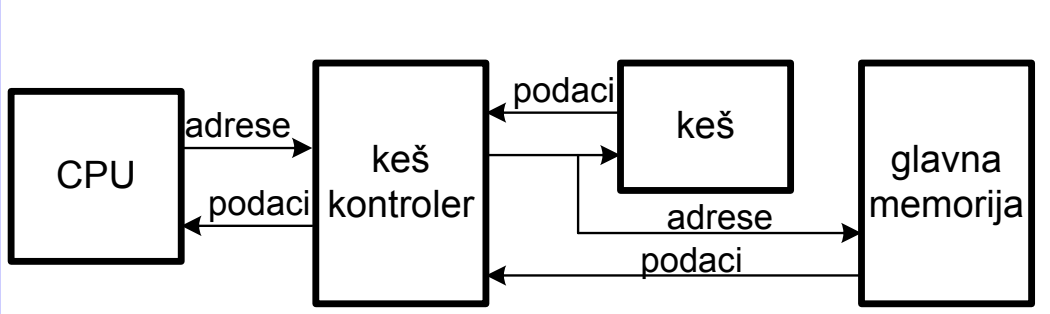
Sekundarna memorija

- čine je uredjaji za masovno memorisanje kakvi su kruti diskovi, trake, CD, i dr.
- koristi ne-nestalan (*non-volatile*) tip memorisanja – informacija se ne gubi nakon isključenja napajanja
- karakteriše je veliki kapacitet

Keš memorija

- Princip rada keša se zasniva na lokalnosti- koja može biti:
  - vremenska lokalnost
  - prostorna lokalnost

Keš kontroler posreduje izmedju CPU-a i memorijskog sistema koga ~ine keš i glavna memorija.

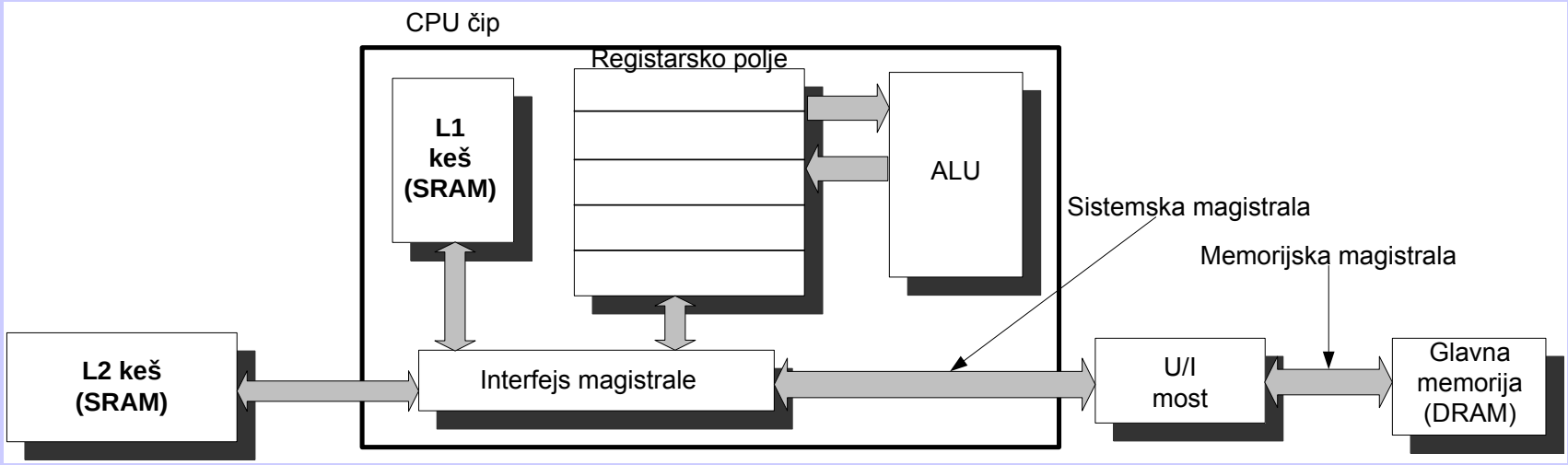


Princip rada keš memorije

- Keš kontroler predaje zahtev za ~itanje memorije prema kešu i glavnoj memoriji.
- Ako se sadr`aj zahtevane lokacije nalazi u kešu, keš kontroler prosledjuje taj sadr`aj CPU-u i ukida zahtev prema glavnoj memoriji. Ovaj uslov se naziva **keš pogodak** (*cache hit*).
- U slu~aju kada lokacija kojoj se obra}amo nije prisutna u kešu, prvo se kreira slobodna lokacija u kešu pa se referencirana re~ iz glavne memorije dobavlja prvo u keš, a zatim se iz keša dostavlja CPU-u. Ova situacija se naziva **keš promašaj** (*cache miss*).

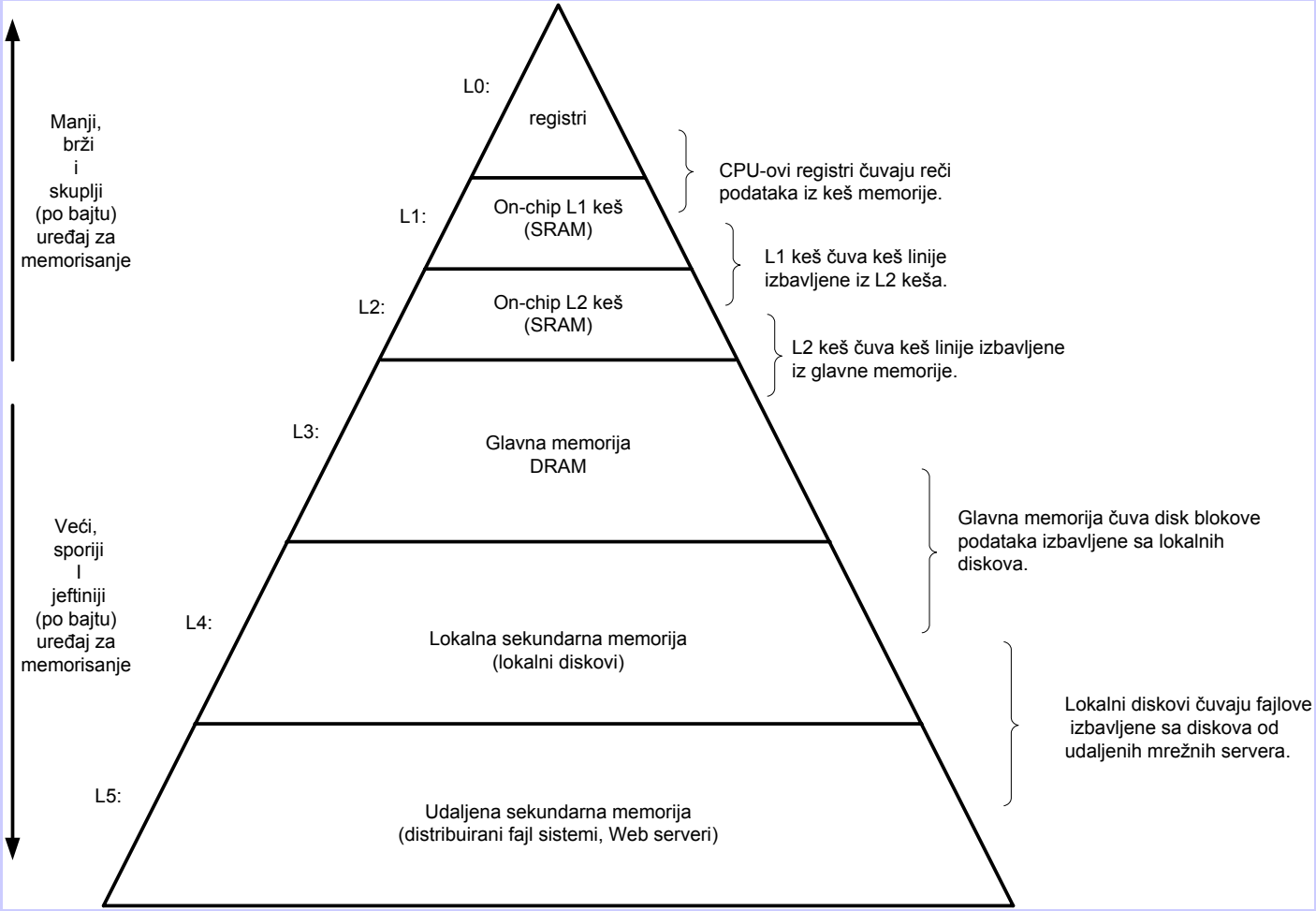
Nivoi keš memorije

- Kod savremenih ra~unarskih sistema postoje nekoliko nivoa keša, poznatih kao nivo L1, nivo L2, a u odredjenim slu~ajevima i nivo L3.



Hijerarhijska organizacija memorije

Memorijske komponente u okviru ra~unarskog sistema su hijerarhijski organizovane



Osnovi o računarima

I poglavlje

- Softverske komponente -

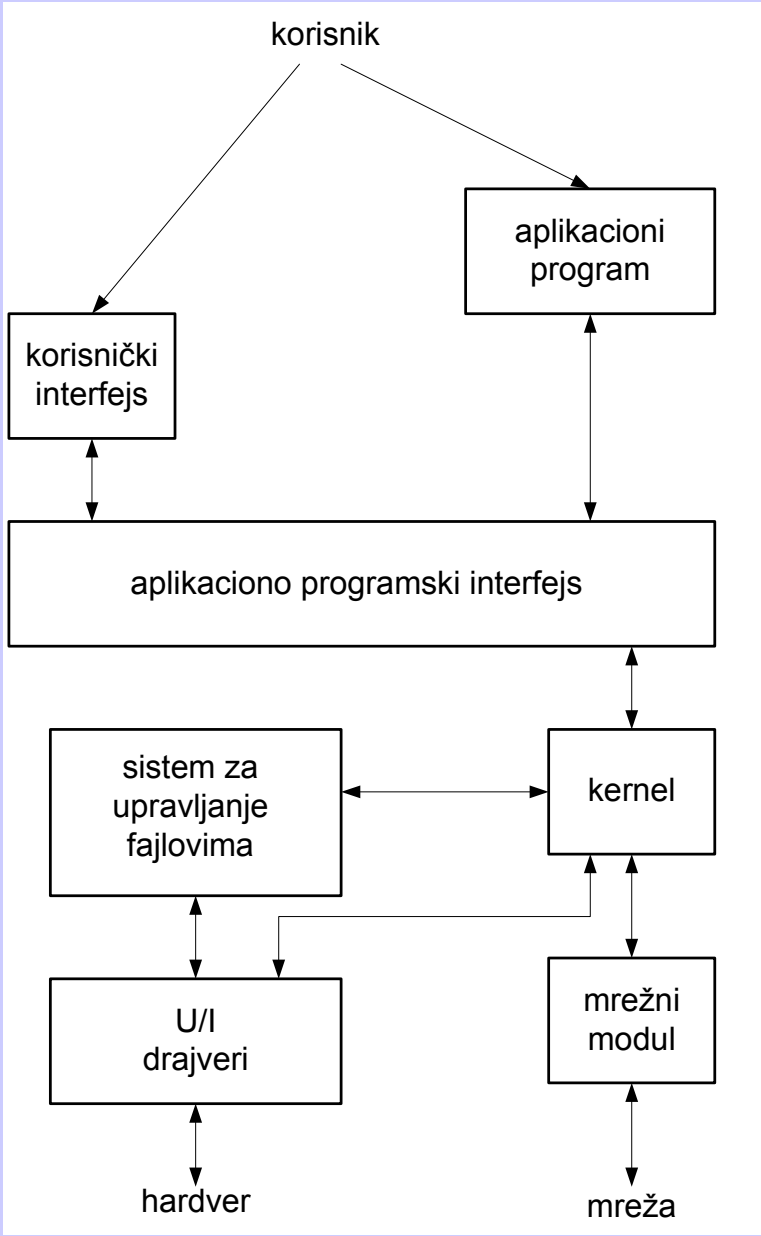
Softverska komponenta

- Pored hardvera ra~unarskom sistemu je potreban i softver.
- Softver ~ine programi koji ukazuju ra~unaru šta da radi
- Postoje dve glavne kategorije softvera:
  - sistemski softver
  - aplikacioni softver
- Sistemski softver nam omogu}ava da manipulišemo fajlovima, punimo i izvršavamo programe, i prihvatamo podatke sa miša i tastature

Sistemsko-softverski programi koji upravljaju ra~unarom nazivaju se operativni sistem

- Aplikacioni softver čine programi, kakvi su *Microsoft Word*, *Netscape*, ili programi koje je korisnik kreirao, a karakteristični su po tome što izvršavaju specifi~ne zahteve korisnika

Pojednostavljeni operativni sistem



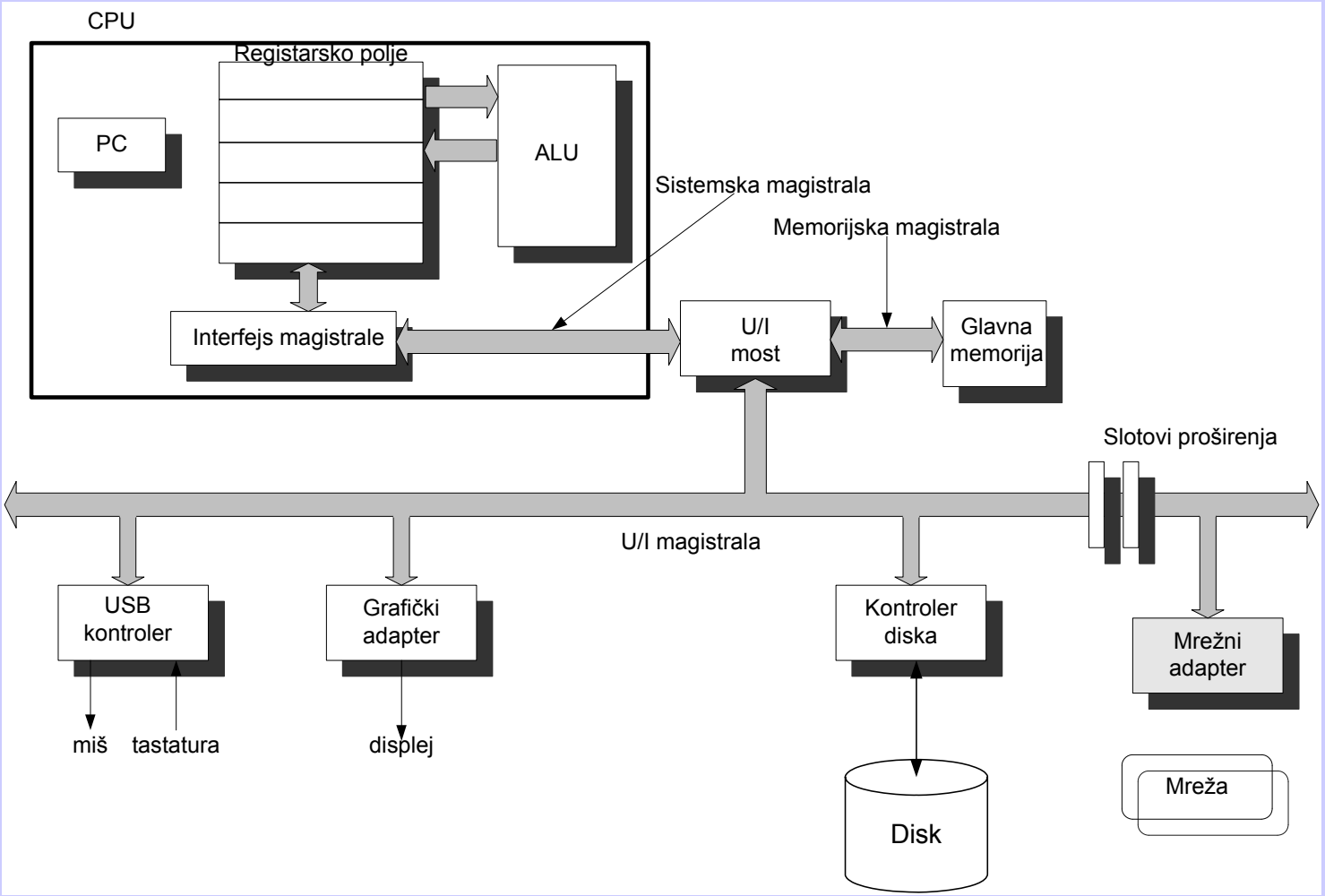
Osnovi o računarima

I poglavlje

- Komunikacione komponente -

Komunikaciona komponenta

- Veliki broj savremenih ra~unara danas ne radi samostalno. Umesto toga oni se povezuju sa drugim ra~unarima preko mre`nog adaptera na mre`u



Osnovi o računarima

I poglavlje

- Evolucija i klasifikacija -

Evolucija arhitekture računara

generacija period	tehnologija i arhitektura	softver i operativni sistem	reprezentativ ni sistemi
prva 1946-1956	vakuumske cevi i relei kao memorijski elementi, jedno-bitni CPU sa akumulatorsko baziranim skupom instrukcija	mašinsko/asemblerski jezici, programi bez potprograma	ENIAC, IBM 701, Princeton IAS
druga 1956-1967	diskretni tranzistori, mamorijska jezgra, akceleratori za rad sa brojevima pokretnog zareza, U/I kanali	Algol i Fortran sa kompajlerima, *batch* operativni sistem	IBM 7030, CDC 1604, Univac LARC
treća 1967-1978	integrisana kola, protočno organizovan CPU, mikroprogramsko upravljanje	C jezik, multiprogramiranje, *time-scharing* operativni sistem	PDP 11, IBM 360/370, CDC 6600
četvrta 1978-1989	VLSI mikroprocesori, poluprovodničke memorije, multiprocesori, vektor procesori	simetrično multiprocesiranje, paralelni kompajleri, *message-passing* biblioteke	IBM PC, VAX 9000, Cray X/MP
peta 1990- danas	ULSI kola, skalabilni paralelni računari, *cluster* računari, Intranet i Internet	Java, mikrokarneli, *multithreading*, distribuirani operativni sistem, *World Wide Web*	IBM SP2, SGI Origin 2000, Digital TruCluster

Klasifikacija računarskih arhitektura

- Klasifikovati objekte koji se u vremenu dinami~ki menjaju veoma je te`ak posao jer uklju~uje predvidjanje pravaca razvoja ovih objekata u budu}nosti.
- Klasifikacija zahteva pa`ljivo prou~avanje i najsitnijih detalja kao i sagledavanje svih aspekata na osnovu kojih je mogu}e ustanoviti kriterijume za klasifikaciju.
- Kada se govori o ra~unarima, njihov izbor na svetskom tr`ištu danas je tako veliki da je ponekad veoma teško uo~iti one elemente koji su zajedni~ki za ve}i broj ra~unara, kao i one koji ih ~ine razli~itim.

Klasifikacione šeme

- Analiziraćemo tri klasifikacione šeme:
  - Prva, razliku izme|u ra~unara prave na osnovu fizi~kog obima, kapaciteta i performansi,
  - Druga, se zasniva na identifikaciji dva razli~ita nezavisna koncepta: toka instrukcija i toka podataka.
  - Tre}a pravi razliku u obimu obrade koja se, kod izvršenja programa napisanih na nekom od viših programskih jezika, obavlja od strane hardvera ili softvera.

Prva klasifikaciona šema

- Medju in`enjerima, proizvodja~ima i korisnicima ra~unara standardizovano je grupisanje ra~unara u pet glavnih klasa:

  - mikroračunari
  - miniračunari
  - super-miniračunari
  - veliki računari
  - super-računari

Druga klasifikaciona (*Fllyn*-ova) šema

- Zasniva na identifikaciji dva nezavisna koncepta:
  - toka podataka – definiše se kao sekvenca podataka koji se pozivaju od strane toka instrukcija (uključujući ulazne podatke i privremene podatke)
  - toka instrukcija – se definiše kao sekvenca instrukcija koja se obavlja od strane računara

- Postoje četiri klase računara:
  - SISD
  - SIMD
  - MISD
  - MIMD

	jednostruki tok podataka	višestruki tok podataka
jednostruki tok instrukcija	SISD	SIMD
višestruki tok instrukcija	MISD	MIMD

SISD, SIMD, MISD i MIMD

SISD - jedinstveni tok instrukcija - jedinstveni tok podataka (*Single Instruction stream Single Data stream*) klasu ra~unara ~ine konvencionalne *von Neumann*-ove arhitekture. Kod ovih ra~unara u jednom trenutku procesira se samo jedna instrukcija koja operiše nad jedinstvenim tokom podataka, operanada i rezultata.

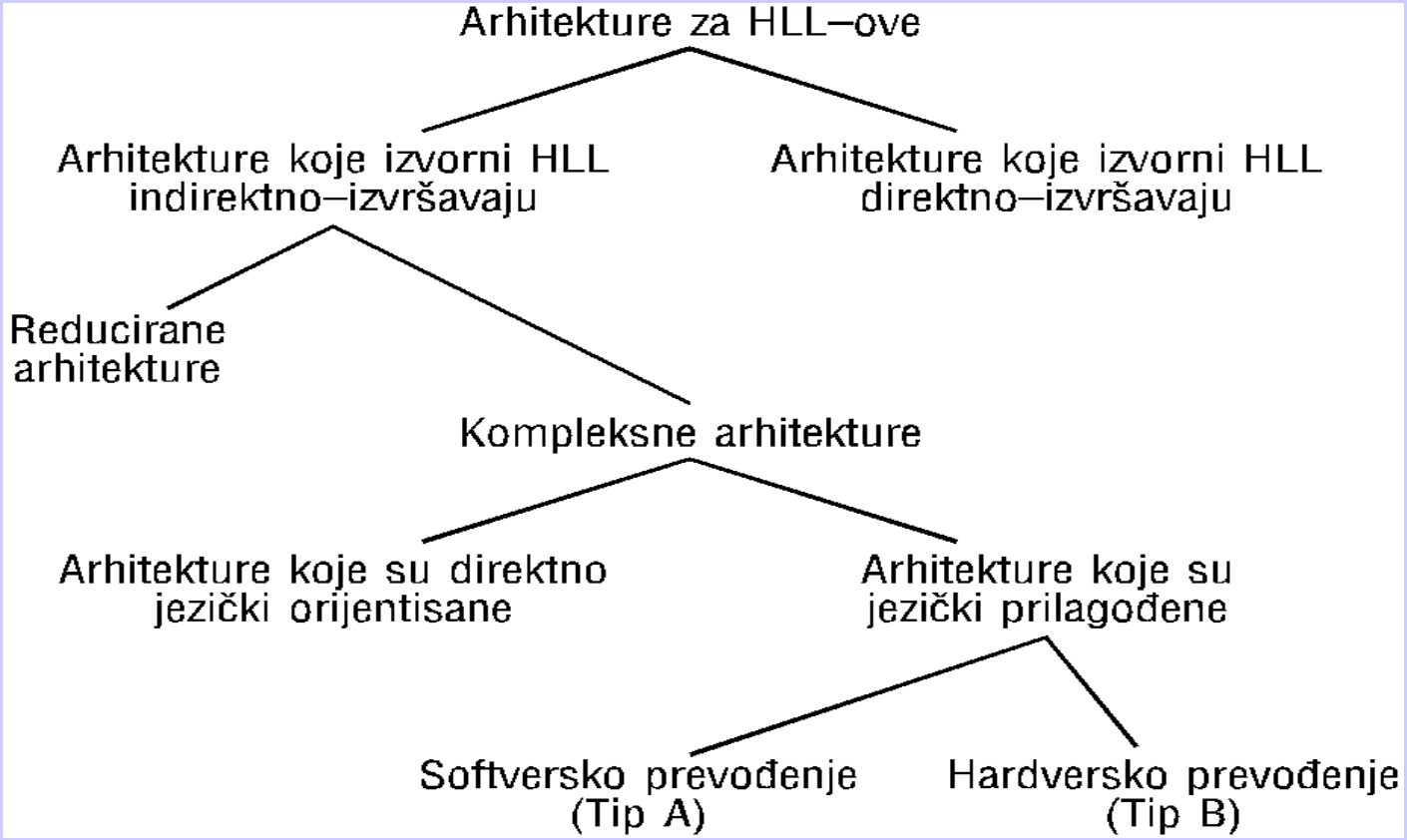
SIMD - jedinstveni tok instrukcija i višestruki tok podataka (*Single Instruction stream Multiple Data stream*). Kod ovih ra~unara jednom instrukcijom simultano se operiše nad nekoliko elemenata podataka.

MISD - višestruki tok instrukcija i jedinstveni tok podataka (*Multiple Instruction stream Single Data stream*). Kod ove klase ra~unara nad jedinstvenim podacima operiše se od strane ve}eg broja instrukcija.

MIMD - višestruki tok instrukcija višestruki tok podataka (*Multiple Instruction stream Multiple Data stream*). Ova klasa ra~unara je veoma široka jer pokriva sve multiprocesorske i multikompjuterske sisteme, tj. sisteme sa privatnom ili deljivom memorijom, slabo, umereno ili ~vrsto spregnute sisteme itd

Treća klasifikaciona šema

- Ovaj kriterijum za klasifikaciju arhitektura bazira se na iznosu obrade koji se, kod izvršenja HLL programa, obavlja od strane softvera ili hardvera



Performanse

II poglavlje

- Karakteristike -

Performanse

- **Performanse** su klju~ni kriterijum kod projektovanja, nabavke, i koriš}enja ra~unarskih sistema.
- Cilj kupca i korisnika je da dobiju sistem sa najboljim performansama po najni`oj ceni.
- Da bi ostvarili ovaj cilj, projektanti treba dobro da poznaju metodologije i tehnike na osnovu kojih se vrši procena performansi ra~unarskih sistema, a korisnici da poseduju osnovna znanja o koriš}enju terminologije i zna~enju tih performansnih metrika.
- Kada projektanti saopšte rezultate o evaluaciji, kupci treba dobro da razumeju na šta se te performanse odnose i šta one konkretno zna~e.

Performanse

Primer :

Kao ilustraciju koliko se brzo računarska tehnologija menja, analizirajmo šta bi se desilo u slučaju kada bi poboljšanje performansi automobila bilo isto kao i računara. Usvojićemo da je maksimalna brzina limuzine 1977. godine bila 150 km/h, a da je prosečna potrošnja goriva iznosila 15 litara na 100 km. Ako su se u periodu od 1977. do 1987. brzina i efikasnost povećavale za 35% godišnje, a u periodu od 1987. do 2000. godine porast na godišnjem nivou je bio 50%, sledeći performanse računara, odrediti kolika se maksimalna brzina i prosečna potrošnja goriva od limuzine očekuje na kraju 1987. i 2000. godine?

Performanse

Rešenje:

Razmak od 1977. do 1987. je 10 godina, a to zna~i da su se brzina i potrošnja poboljšavale za isti faktor:

$$(1.35)^{10} = 20.1$$

Odgovaraju}a brzina na kraju 1987. godine bi bila:

$$150\,\text{km/h} * 20.1 = 3015\,\text{km/h}$$

a prose~na potrošnja:

$$(15\,\text{litara}/100\,\text{km})/20.1 = 0.746\,\text{litara}/100\,\text{km}$$

Narednih 13 godina poboljšanje performansi na godišnjem nivou bilo je 50%, što odgovara ukupnom faktoru poboljšanja u odnosu na vrednost iz 1987. godine od:

$$(1.5)^{13} = 194.6$$

Performanse

Rešenje-produžetak:

Odgovaraju}a brzina limuzine na kraju 2000. godine bi bila:

$$3015\,\text{km/h} * 194.6 = 586719\,\text{km/h}$$

a potrošnja bi bila:

$$(0.746\,\text{litara/100km})/194.6 = 0.0038\,\text{litara/100km}$$

To znači da rastojanje od zemlje do meseca (384000 km) i natrag sa ovakvom brzinom bi mogli da predjemo za:

$$\frac{2 * 384000\,\text{km}}{586719\,\text{km/h}} = \frac{768000\,\text{km}}{586719\,\text{km/h}} = 1.3\,\text{h}$$

a ukupna potrošnja goriva bi bila:

$$7680 * 0.0038 = 29.184\,\text{litara}$$

Technology => dramatic change

- Processor
  - Logic capacity: about 30% per year
  - Clock rate: about 20% per year
- Memory
  - DRAM capacity: about 60% per year (4x every 3 years)
  - Memory speed: about 10% per year
  - Cost per bit: improves about 25% per year
- Disk
  - Capacity: about 60% per year
  - Total use of data: 100% per 9 months!
- Network Bandwidth
  - Bandwidth increasing more than 100% per year!

Evaluacija performansi

- Svaka evaluacija zahteva izvrsno poznavanje sistema koji se modelira, pa`ljivu selekciju metodologije, radnog optere}enja, i softverskih sredstava koja se koriste za procenu.

- Za dati problem, dva analiti~ara mogu da izaberu razli~ite performansne metrike, a takodje i razli~ite metodologije za evaluaciju. U suštini, jedne iste podatke, dva analiti~ara mogu da interpretiraju razli~ito.

Evaluacija performansi

-jedan tipičan problem-

Primer :

- Na slici su dati rezultati merenja sistemske propusnosti, izra`ene brojem transakcija u sekundi, dvaju sistema, A i B, za dva razli~ita radna optere}enja.

sistem	radno opterećenje 1	radno opterećenje 2
A	20	10
B	10	20

Evaluacija performansi

-prvi pristup za procenu-

Primer :

- Postoje dva na~ina da se uporede performanse razmatranih sistema.
- Prvi je da se odredi aritmeti~ka sredina radnih optere}enja.
- Rezultati dobijeni ovakvim pristupom prikazani su na slici.
- Zaklju~ak bi bio da su oba sistema podjednako dobra

sistem	radno opterećenje 1	radno opterećenje 2	prosek
A	20	10	15
B	10	20	15

Evaluacija performansi

-drugi pristup za procenu-

Primer :

- Drugi na~in se bazira na pristupu da se jedan od sistema, recimo B, tretira kao referentni, a da se performanse drugog sistema izraze u odnosu na referentni

sistem	radno opterećenje 1	radno opterećenje 2	prosek
A	2	0.5	1.25
B	1	1	1

Evaluacija performansi

-treći pristup za procenu-

Primer :

- Alternativno, ako usvojimo da je sistem A referentni dobi}emo rezultate prikazane na slici

sistem	radno opterećenje 1	radno opterećenje 2	prosek
A	1	1	1
B	0.5	2	1.25

Izbor metrike za procenu performansi

- Osnovne karakteristike ra~unarskog sistema obi~no se izra`avaju slede}im merama:
  - broj (*count*) - koliko puta se neki dogadjaj javlja,
  - trajanje (*duration*) - trajanje nekog vremenskog intervala,
  - veli~ina (*size*) - obim nekog parametra.
- Na osnovu izmerenih vrednosti, mogu}e je odrediti aktuelnu vrednost koju `elimo koristiti za odredjivanje performansi sistema.
- Dobijena vrednost se naziva **performansna metrika**.
- Izmerene vrednosti, kakve su recimo broj, vremenski interval i veli~ina, mogu se direktno koristiti kao performansne metrike

Izbor metrike za procenu performansi

- Sa ciljem da izvedemo metriku o brzini, od interesa je da se normalizuje mera kakva je na primer **broj** u odnosu na neku standardizovanu vremensku jedinicu. Kao rezultat ovakve normalizacije dobijamo metriku koja se naziva **propusnost** (*throughput* ili *rate metric*).

- Propusnost se izra~unava deljenjem vrednosti **broj**, koja odgovara broju dogadjaja koji su se javili u datom vremenskom intervalu, sa vremenskim intervalom opservacije

- Ovakva performansna metrika normalizovana u odnosu na standardizovanu vremensku bazu, kakva je sekunda, i korisna je za uporedjivanje razli~itih merenja koja se obavljaju za razli~ite vremenske intervale.

Procesorske i sistemske perfomansne metrike

- Na polju ra~unarske tehnike predlo`en je i koriš}en veliki broj performansnih metrika.
- Veliki broj faktora uti~e na performanse tako da donošenje odluke šta od ponudjenog treba kupiti nije uvek lako i jednostavno.
- Često jednostavnost programiranja mašine mo`e da ima ve}u te`inu u odnosu na to da li se neki program br`e ili sporije izvršava na jednoj ili na drugoj mašini.
- Va`an aspekt mo`e da bude i kompatibilnost sa prethodnom generacijom mašina.

Kriterijumi perfomansne metrike

- Kriterijumi, koje svaka performansna metrika treba da zadovolji da bi bila prihvatljiva kako za projektante ra~unarskih sistema tako i za kupce, su sledeći :

  - linearnost
  - pouzdanost
  - repetitivnost
  - jednostavnost merenja
  - konzistentnost
  - nezavisnost

Linearnost

- Vrednost metrike treba da je linearno proporcionalna sa ostvarenim performansama mašine.

- Ako se sistem modifikuje tako da je radna frekvencija CPU-a dva puta veća od teku}e, tada je realno o~ekivati da sistem dva puta br`e izvršava program.

- Sve metrike ne zadovoljavaju zahtev proporcionalnosti.

- Logaritamske metrike, kakva je dB skala, koja se koristi za prikazivanje intenziteta (nivoa) signala, nema linearnu metriku, jer pove}anje za jedini~nu vrednost performansne metrike odgovara faktoru pove}anja od 10 veli~ina fenomena koji se opservira

Primer : $y = \log x$

- Ako je *y* performansna metrika, a *x* veli~ina fenomena koji se opservira, tada da bi se *y* inkrementirao za 1, *x* treba da je 10 puta ve}i u odnosu na prethodnu vrednost.

Pouzdanost

- Ako sistem A ima uvek bolje radne karakteristike od sistema B, tada i odgovaraju}e performansne mere treba da ukazuju na to.
- Pored toga što ovaj zahtev izgleda logi~an i razumljiv postoje performansne metrike koje ne ispunjavaju ovaj kriterijum.
- Performansna metrika MIPS je nepouzdana. Naime, nije ništa neobi~no da neki procesor koji ima ve}i broj MIPS-ova u odnosu na drugi izvršava isti program za du`i vremenski period

Repetitivnost

- Za metriku ka`emo da je **repetitivna**, ako se svaki put kada se jedan isti eksperiment ponovi, dobija ista vrednost

Jednostavnost merenja

- Ako se metrika ne mo`e lako izmeriti mala je verovatno}a da }e je neko koristiti

Konzistentnost

- **Konzistentnost performansne metrike** se odnosi na zahtev da jedinica u kojoj se metrika izra`ava kao i precizna definicija metrike budu identi~ne i nezavisne, kako od razli~itih tipova sistema, tako i od razli~itih konfiguracija jednog istog sistema.
- Ako jedinice metrike nisu konzistentne, nemogu}e je koristiti tu metriku za uporedjivanje performansi razli~itih sistema.
- MIPS, MFLOPS, nisu konzistentne za sve tipove sistema

Nezavisnost

- Dobra metrika treba da bude nezavisna od "namerno naštimanih" rešenja
- Najve}i broj korisnika ra~unarskih sistema odlu~uje o tome koji sistem treba da kupi na osnovu nekih standardnih performansnih metrika.
- Kao rezultat ovakvog pristupa vrši se veliki pritisak na projektante ra~unara (uglavnom od slu`be za prodaju) da optimizuju sistem za pojedinu metriku, tj. da konstruišu mašine koje }e imati superiorne performanse samo za tu metriku.

Standardizovana sredstva

- U najve}em broju in`enjerskih i nau~nih oblasti postoje standardizovana sredstva za procenu (merenje) i uporedjivanje fenomena od interesa.

Primer : brzina automobila se mo`e lako izmeriti u nekim standardnim jedinicama kakva je, recimo, predjeni metri u sekundi. Koriš}enjem standardnih mera mogu}e je zatim direktno uporediti brzinu automobila sa brzinom aviona.

- Uporedjivanje performansi razli~itih ra~unarskih sistema se ne mo`e direktno izvesti.

- Kada se procenjuju performanse ra~unara problemi obi~no nastaju oko toga koju je metriku najbolje koristiti.

- Da li to treba da bude MIPS, MFLOPS, vreme, ili neka druga mera ?

Taktna frekvencija

- Kod najve}eg broja oglasa u novinama ili kataloškim podacima kao najistaknutiji indikator performansi ~esto se navodi **frekvencija rada** (*operating frequency*) ili **takt CPU-a** (*clock period*, *tick*, *clock tick*, *clock*, *clock cycle*).

- Indirektna sugestija potencijalnom kupcu je ta da sistem koji radi na 2 GHz je uvek br`i kod rešavanja problema korisniku od, recimo, sistema koji radi na 1 GHz

Taktna frekvencija - produžetak

- Na `alost, taktna frekvencija kao performansna metrika:
  - ❖ Ne ukazuje u potpunosti na obim izra~unavanja koji se obavlja u svakom taktnom intervalu;
  - ❖ Ignoriše kompleksne interakcije izmedju procesora i memorije, i procesora i U/I podsistema; i
  - ❖ Namerno pre}utkuje ~injenicu da procesor nije taj koji predstavlja usko grlo u radu ra~unarskog sistema.

Taktna frekvencija - produžetak

- Kao metrika, taktna frekvencija poseduje sledeće osobine:
  - S obzirom da je **konstantna za dati sistem**, repetitivna je (kriterijum 3), lako se meri (kriterijum 4), precizno je definisana, a to zna~i da je konzistentna (kriterijum 5), i nezavisna je od tipa proizvodja~a (kriterijum 6).
  - Ova metrika je nelinearna (kriterijum 1) i nepouzdana (kriterijum 2).
  - Tvrdnja da se pri višoj frekvenciji mogu ostvariti bolje performanse samo je delimi~no ta~na.
  - Frekvencija rada procesora, kao metrika, nije linearno proporcionalna sa brzinom izvršenja programa pa }e o~ekivani rezultati izostati.
  - Kao zaklju~ak, radna frekvencija CPU-a jeste korisna, ali nije uvek dobra metrika za procenu ukupnih performansi sistema.

MIPS

- MIPS predstavlja pokušaj da se razvije metrika za brzinu izvršenja (*throughput* ili *execution rate*), tj. iznos izra~unavanja koji se obavlja u jedinici vremena.
- Ideja je bila da se na osnovu ove metrike direktno uporedjuju brzine ra~unara.
- MIPS odgovara broju izvršenih instrukcija u jedinici vremena.
- MIPS (*Million Instructions Per Second*) je skra}enica od milion instrukcija u sekundi i definiše se kao:

$$MIPS = \frac{N}{T_{CPU} * 10^6}$$

gde je: N - broj izvršenih instrukcija, a T_{CPU} - vreme potrebno da CPU izvrši N instrukcija

MIPS - produžetak

- Dobre osobine MIPS-a su:
  - lako se meri (kriterijum 4);
  - repetitivan je (kriterijum 3);
  - nezavistan je (kriterijum 6).

- Loše osobine MIPS-a su:
  - nije linearan (kriterijum 1- dupliciranje MIPS-ova ne zna~i i dupliciranje rezultantnih performansi)
  - nije pouzdan (kriterijum 2),
  - nije konzistentan (kriterijum 5- nije u dobroj korelaciji sa performansama).

Korišćenje MIPS-a kao metrika

- Koriš}enje MIPS-a kao merilo za ocenu performansi ra~unara nije opravdano iz slede}ih razloga:
  - MIPS zavisi od skupa instrukcija, što ~ini nelogi~nim da se dva ra~unara sa razli~itim skupovima naredbi uporedjuju na bazi kataloški deklarisanih MIPS-ova.
  - broj MIPS-ova za isti ra~unar mo`e da varira u zavisnosti od programa, i
  - broj MIPS-ova mo`e biti (da se menja) obrnuto proporcionalan performansama.

Relativni MIPS

- Da bi se otklonili nedostaci kao alternativna mera se koristi relativni MIPS
- Relativni MIPS se definiše na slede}i na~in:

$$MIPS_{rel} = \frac{T_{ref}}{T_{mer}} * MIPS_{ref}$$

gde je: T_{ref} - vreme izvršenja programa na referentnoj mašini, T_{mer} - vreme izvršenja programa na mašini koja se ocenjuje, $MIPS_{ref}$ - broj MIPS-ova referentne mašine.

- Osamdesetih godina prošlog veka dominantna referentna mašina je bila VAX-11/780 koja je nazvana 1 MIPS mašina.

MFLOPS

- Alternativna mera za ocenu performansi ra~unara sa aspekta brzine izvršenja operacija u FP aritmetici je MFLOPS.

- MFLOPS-u (*Million Floating Operation Per Second*) odgovara milion FP operacija u sekundi

$$MFLOPS = \frac{N_{FP}}{T_{CPU} * 10^6}$$

gde je N_{FP} - broj FP operacija u programu

- Kao i kod MIPS-ova, broj MFLOPS-ova je zavistan od mašine i programa

- Slični zaključci koji se odnose na dobre i loše osobine MIPS-a važe i za MFLOPS

- I u ovom slučaju se definiše relativni MFLOPS

Broj ciklusa po instrukciji / Broj izvršenih instrukcija po ciklusu

- Najbolja mera za procenu performansi ra~unara je vreme izvršenja programa
- Medju korisnicima ra~unara pre}utno je usvojeno da va`i slede}e pravilo: Za one sisteme koji izvršavaju svoje programe za kra}e vreme ka`emo da imaju bolje performanse.
- Vreme potrebno CPU-u da izvrši neki program iznosi:

$$CPU_{vreme} = CPU_{broj\_taktnih\_ciklusa\_u\_programu} * Takt_{CPU}$$

ili

$$CPU_{vreme} = \frac{CPU_{broj\_taktnih\_ciklusa\_u\_programu}}{Frekvencija_{CPU}}$$

Taktni ciklus po instrukciji

- Metrika koja se ~esto koristi za procenu performansi ra~unara predstavlja broj taktnih ciklusa potrebnih da se izvrši program.
- Ova metrika naziva se **taktni ciklusi po instrukciji** (*clock cycles per instruction* - ***CPI***) i definiše se kao:

$$CPI = \frac{CPU_{broj\_taktnih\_ciklusa\_u\_programu}}{IC}$$

gde je *IC* (*instruction count*) broj izvršenih instrukcija u programu.

pa je $$CPU_{broj\_taktnih\_ciklusa\_u\_programu} = IC * CPI$$

tako da je $$CPU_{vreme} = IC * CPI * Takt_{CPU}$$

ili $$CPU_{vreme} = \frac{IC * CPI}{Frekvencija_{CPU}}$$

Taktni ciklus po instrukciji- produžetak

- Jednačina

$$CPU_{vreme} = IC * CPI * Takt_{CPU}$$

ukazuje da su performanse CPU-a zavisne od slede}ih triju parametara:

- trajanje taktnog ciklusa (intervala) - zavisi od tehnologije hardvera i organizacije mašine
- broj taktnih ciklusa po instrukciji, CPI - odredjen je organizacijom mašine, tj. skupom instrukcija i brojem funkcionalnih jedinica mašine
- broj instrukcija koje se izvršavaju u programu, IC - zavisi od arhitekture ra~unara i tehnologije kompajlera.

Taktni ciklus po instrukciji- produžetak

- Broj taktnih ciklusa CPU-a potrebnih da se izvrši neki program mo`e se odrediti na slede}i način :

$$CPU_{broj\_taktnih\_ciklusa\_u\_programu} = \sum_{i=1}^{n} CPI_i * IC_i$$

gde se IC_i odnosi na to koliko puta se instrukcija i izvršava u toku programa;

CPI_i odgovara prose~nom broju taktnih ciklusa za instrukciju i ;

n je ukupan broj razli~itih instrukcija u programu.

Pri čemu važi :

$$CPI = \frac{\sum_{i=1}^{n} CPI_i * IC_i}{IC}$$

i

$$CPU_{vreme} = \left(\sum_{i=1}^{n} CPI_i * IC_i \right) * Takt_{CPU}$$

SPEC

- Da bi standardizovali aktuelne rezultate dobijene izvršenjem programa nekoliko proizvodja~a ra~unara formiralo je neprofitabilnu grupu (komitet) nazvanu **SPEC** (*Systems Performance Evaluation Cooperative*).
- *SPEC* grupa je :
  - identifikovala skup *integer* i *floating point* ben~mark programa, i
  - standardizovala metodologiju merenja,
  - standardizovala na~in prezentacije performansi dobijenih u toku izvršenja ovih programa

Koraci SPEC metodologije

- *SPEC* metodologiju ~ine slede}i koraci:
  - merenje potrebnog vremena da se odredjeni program, iz nekog skupa programa, izvrši na sistemu koji se testira.
  - deljenje izmerenog vremena za svaki program iz prvog koraka sa vremenom koje je potrebno da se svaki od tih programa izvrši na standardnoj baznoj mašini (referentna mašina), sa ciljem da se normalizuju vremena izvršenja.
  - usrednjavanje svih normalizovaih vrednosti koriš}enjem geometrijske sredine sa ciljem da se generiše jedinstvena performansna metrika.

SPEC osobine

- SPEC metodologija je znatno rigoroznija od onih koje koriste MIPS i MFLOPS

Nedostaci:

- usrednjavanje svih individualno normalizovanih rezultata generiše metriku koja nije u linearnoj vezi sa aktuelnim izvršenjem programa (kriterijum 1).
- nije pouzdana (kriterijum 2), jer se program mo`e izvršavati br`e na sistemu ~iji je SPEC rejting ni`i u odnosu na mašinu ~iji je rejting viši.

Prednost:

- proklamuje nepristrasnu metriku koja je nezavisna od spoljnih uticaja (kriterijum 6)
- predstavlja zna~ajan korak na korektnom putu u definisanju dobre performansne metrike

Statistička sredstva i tehnike

- Kupci i projektanti uporedjuju performanse razli~itih ra~unarskih sistema na jedan pojednostavljen na~in.
- Oni insistiraju da se performanse izraze kao jedinstveni broj.
- Taj broj treba da bude neka srednja vrednost i da sadr`i sve esencijalne metrike
- Problem je na koji na~in odrediti srednju vrednost

Standardne tehnike, koje se koriste za odredjivanje srednjih vrednosti, su:

- aritmetička sredina
- geometrijska sredina
- harmonijska sredina
- ponderisana sredina

Aritmetička sredina

Aritmeti~ka sredina se definiše kao:

$$x_A = \frac{1}{n}\sum_{i=1}^{n} x_i$$

gde je: x_i - vrednost individualnog merenja koje se usrednjava; n - broj merenja.

Za slu~aj da je $x_i = T_i$, gde je T_i vreme izvršenja programa i, srednje vreme bi}e:

$$T_A = \frac{1}{n}\sum_{i=1}^{n} T_i$$

Ova jedna~ina ukazuje da je T_A direktno proporcionalno ukupnom vremenu izvršenja.

Aritmeti~ka sredina je korektna kada se procenjuje vreme izvršenja.

Aritmetička sredina - produžetak

- Kada koristimo aritmeti~ku sredinu da procenimo brzinu izvršenja imamo:

$$M_A = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{T_i}$$

Ova jedna~ina daje rezultat koji je direktno proporcionalan zbiru inverznih vremena izvršenja.

- S obzirom da se radi o inverznoj proporcionalnosti, evidentno se name}e zaklju~ak da :

Aritmeti~ka sredina nije pogodno sredstvo za procenu brzine izvršenja

Harmonijska sredina

Uobi~ajeno se koristi od strane analiti~ara performansi.

Definiše se kao:

$$x_H = \frac{n}{\sum_{i=1}^{n} \frac{1}{x_i}}$$

Ako koristimo harmonijsku sredinu da bi smo odredili vrednost vremena izvršenja tada je $x_i = T_i$, pa shodno prethodnom dobijamo slede}i izraz

$$T_H = \frac{n}{\sum_{i=1}^{n} \frac{1}{T_i}}$$

Jasno je da ova vrednost nije direktno proporcionalna ukupnom vremenu izvršenja, pa zaklju~ujemo da je:

Harmonijska sredina neodgovaraju}a mera za odredjivanje vremena izvršenja.

Harmonijska sredina - produ`etak

Kada se procenjuje brzina izvr{enja tada va`i da je $x_i = 1/T_i$, pa dobijamo:

$$M_H = \frac{n}{\sum_{i=1}^{n} T_i}$$

Vrednost M_H predstavlja ukupan broj operacija koji se izvr{ava od strane svih programa podeljen sumom svih vremena izvr{enja.

M_H je inverzno proporcionalna ukupnom vremenu izvr{enja, pa shodno tome zaklju~ujemo da :

Harmonijska sredina predstavlja zadovoljavaju}e sredstvo za procenu brzine izvr{enja.

Geometrijska sredina

- Analiti~ari performansi koriste geometrijsku sredinu kao sredstvo za sumiranje normalizovanih brojeva.
- **Geometrijska sredina** se definiše kao ***n***-ti koren proizvoda od ***n*** individualnih vrednosti, tj. kao:

$$x_G = \sqrt[n]{x_1 x_2 \ldots x_i \ldots x_n} = \left(\prod_{i=1}^{n} x_i \right)^{1/n}$$

- Za slu~aj kada je $x_i = T_i$ prose~no vreme izvršenja odredjuje se kao

$$T_G = \left(\prod_{i=1}^{n} T_i \right)^{1/n}$$

T_G je direktno proporcionalno ukupnom vremenu izvršenja.

Geometrijska sredina - produžetak

- Usrednjavanje brzine izvršenja pomo}u geometrijske sredine izra~unava se kao:

$$M = \left(\prod_{i=1}^{n} \frac{1}{T_i} \right)^{1/n}$$

što je inverzno proporcionalno ukupnom vremenu izvršenja.

Primer

Odrediti aritmeti~ke i geometrijske sredine vrednosti 4, 2, 4, 82.
Aritmeti~ka sredina niza je

$$\frac{4+2+4+82}{4}=23$$

Geometrijska sredina iznosi

$$\sqrt[4]{4*2*4*82}=7.16$$

Na osnovu dobijenih rezultata zaklju~ujemo da ekstremna vrednost u nizu ima ve}i efekat kod aritmeti~ke sredine u odnosu na geometrijsku.

Ponderisana sredina

- **Ponderisana sredina** (*weighted mean*) se izra~unava dodelom odgovaraju}e te`ine vremenu izvršenja svakog programa,

vremenu izvršenja programa i dodeljuje se te`ina w_i tako da važi

$$\sum_{i=1}^{n} w_i = 1$$

- Zadavanjem te`ina, izraz za odredjivanje aritmeti~ke ponderisane sredine dobija formu:

$$T_{A,w} = \sum_{i=1}^{n} w_i T_i$$

dok izraz za odredjivanje harmonijske ponderisane sredine je oblika:

$$T_{H,w} = \frac{1}{\sum_{i=1}^{n} \frac{w_i}{T_i}}$$

Benčmark programi

- Ako `elimo da ra~unaru procenimo performanse on mora da izvršava odredjeni program.
- Najbolju procenu dobi}emo ako se izvršava onaj aplikacioni program za koji je ra~unar namenjen
- Ovo nije mogu}e uvek ostvariti zbog :
  - potrebno je ulo`iti zna~ajan napor da se aplikacija brzo i jednostavno prenese na novi ra~unarski sistem
  - nije jednostavno preneti aplikaciju ako je pri tome jedini cilj samo da se procene performanse
  - za slu~aj da je aplikacija nova i da nije u potpunosti razvijena nemogu}e ju je iskoristiti kao test program

Benčmark programi - produžetak

- Surogat program koji se koristi kao standardna referenca za uporedjivanje performansnih rezultata naziva se **ben~mark program** (*benchmark program*)
- U suštini ben~mark je program napisan na nekom od HLL-ova kao što su C, Fortran, Pascal, Algol, Ada, Java, ili neki drugi jezik.
- Termini ben~mark i **radno-optere}enje** se ~esto koriste kao sinonimi
- Kao program, ben~mark treba da bude jednostavan za koriš}enje i da se bez problema izvršava na razli~itim sistemima

Tipovi benčmark programa

- Slede}i tipovi ben~mark programa se koriste za uporedjivanje ra~unarskih sistema:

  - instrukcija sabiranja,
  - mešavina instrukcija,
  - sinteti~ki programi,
  - mikroben~markovi,
  - kernel programi,
  - aplikacioni programi

Benčmark - instrukcija sabiranja

- Istorijski posmatrano, kada su se ra~unarski sistemi prvi put pojavili, procesori su bili najskuplji delovi ra~unara, pa su performanse ra~unara uglavnom bile poistove}ivane sa performansama procesora.
- U suštini prvi procesori su imali implementiran veoma mali skup instrukcija.
- Jedna od instrukcija koja se naj~ešće izvršavala bila je *Add*.
- Imaju}i to u vidu, u prvoj aproksimaciji, za ra~unar koji najbr`e izvršava instrukciju *Add* smatralo se da ima najbolje performanse.
- Izvršenje instrukcije *Add* bilo je jedino koriš}eno radno optere}enje, a vreme izvršenja operacije sabiranja predstavljalo je jedinu performansnu metriku.

Benčmark – mešavina instrukcija

- Kako se broj i kompleksnost instrukcija podr`anih od strane procesora pove}avala, instrukcija *Add* nije više predstavljala pravo merilo za procenu performansi
- Od projektanata se zahtevao sve detaljniji opis radnog optere}enja
- Kao rezultat, vreme izvršenja jedinstvene operacije, kakva je bila *Add*, nije više bilo adekvatno i merodavno za procenu performansi.
- *Jack C. Gibson* predlo`io je da se kao performansna metrika koristi **mešavina instrukcija** (*instruction mix*).

Benčmark – mešavina instrukcija

-produžetak-

- Ideja mešavine instrukcija bila je da se sve instrukcije podele u razli~ite klase, tako da je instrukcijama u okviru jedne klase potreban isti broj procesorskih ciklusa za izvršenje
- Broj instrukcija svake klase koji se izvršavao od strane pojedinih kolekcija programa koristio se za formiranje ponderisane sredine.
- Ponderisana sredina je predstavljala metriku za uporedjivanje ra~unarskih sistema.
- *Gibson* je predlo`io neke specifi~ne te`ine za skup unapred definisanih klasa instrukcija

Benčmark – mešavina instrukcija

- *Gibson* predložio 13 različitih klasa instrukcija -

1. Load i Store	31.2
2. Fixed-point Add i Sub	6.1
3. Kompariranje, Cmp	3.8
4. Grananje, Bra	16.6
5. Floating-Point Add i Sub	6.9
6. Floating-Point Mul	3.8
7. Floating-Point Div	1.5
8. Fixed-Point Mul	0.6
9. Fixed-Point Div	0.2
10. Pomeranje	4.4
11. Logičke Or, And	1.6
12. Instrukcije kojoe ne koriste registre	5.3
13. Indeksiranje	18.0
Ukupno	100%

Benčmark – mešavina instrukcija -produžetak-

Nedostaci:

- Mešavina instrukcija ignoriše va`nost efekata koje na performanse imaju ulazno-izlazne operacije, hijerarhijska organizacija memorije, itd.
- Savremeni ra~unari imaju u svom repertoaru ve}i broj kompleksnih klasa instrukcija koje se ne mogu u potpunosti uklopiti u *Gibson*-ov model mešavine instrukcija.
- Kod ra~unara novije generacije, vreme izvršenja instrukcije jako zavisi od adresnih na~ina rada, stope keš pogodaka, efikasnosti u proto~noj obradi, i interferencije sa drugim uredjajima prilikom pristupa memoriji
- Ne daje pravu sliku o profilu klasa instrukcija koje se izvršavaju od strane aplikacionih programa

Prednost:

- Mešavina-instrukcija daje kao rezultat jedinstveni broj koji se mo`e koristiti za uporedjivanje sa drugim ra~unarima sli~ne arhitekture

Sintetički benčmark programi

- Dr`e}i se ideje o mešavini-instrukcija analiti~ari su razvili **sinteti~ke ben~mark programe** koji najpribli`nije odslikavaju o~ekivanu ili `eljenu mešavinu-instrukcija
- Ovi programi su vešta~ki programi koji ne izvršavaju koristan posao
- Sinteti~ki programi se ipak ne ponašaju isto kao i realni aplikacioni programi
- Problem je taj što sinteti~ki programi ne modeliraju dovoljno ta~no uticaj na performanse koji imaju redosled izvršenja instrukcija i zavisnosti koje postoje izmedju instrukcija

Mikrobenčmarkovi

- **Mikroben~markovi** (*microbenchmarks*) su mali, specijalno projektovani programi koji se koriste za testiranje specifi~nih delova sistema.

- Ovi programi procenjuju aspekte kakvi su brzina rada CPU-a, brzina odziva memorije, brzina rada ulazno-izlaznog podsistema, performanse operativnog sistema, karakteristike umre`avanja, itd.

- Obi~no mikroben~mark programe kreiraju projektanti koji su veoma dobro upoznati sa mogu}nostima i principom rada komponente koja treba da se testira

Mikrobenčmarkovi - produžetak

- Pored mikroben~mark programa postoje i makroben~mark programi.
- Ovim programima se procenjuju performanse ra~unarskih sistema u celosti.
- Makroben~mark programi ne daju nikakve detalje o tome zbog ~ega sistem dobro ili loše radi.
- Tipi~ni predstavnici mikro i makro ben~mark programa prikazani su

tip	ime	šta se procenjuje
mikrobenčmark	LINPACK	num. izračun. u linearnoj algebri
	STREAM	propusnost memorije
makrobenčmark	SPEC	mešavina benčmark familija
	STAP	signal procesiranje
	TPC	komercijalne aplikacije

Programski kerneli

- **Kernel ben~mark programi** su manji po obimu od realnih aplikacija i u sebi uklju~uju one delove programa koji procenjuju njegove glavne karakteristike.
- Kernel (jezgro ili *nucleus*) je mali program koji je izdvojen iz ve}eg aplikacionog programa.
- Kernel mo`e biti deo kôda tela petlje na koju aplikacioni program troši najve}i deo svog vremena izvršenja.
- Kernel obi~no ~ine desetinu linija kôda, a to ga ~ini lako prenosivim na druge ra~unarske sisteme.

Programski kerneli- produ`etak

- Merenjem performansi kernela na razli~itim sistemima mo`e se dobiti relativno dobra slika o tome kakve }e biti performanse razli~itih sistema kada oni izvršavaju kompletnu aplikaciju.
- Kerneli mogu ta~no da predvide kako }e razli~iti sistemi izvršavati tipove operacije u samom kernelu
- Ovi ben~mark programi ignorišu glavne komponente sistema kakav je recimo operativni sistem.
- Takodje ignorišu realno ponašanje hijerarhijski organizovane memorije
- Kernel programi su korisni, ali samo u ograni~enom smislu.

Aplikacioni benčmark programi

- Da bi se poboljšale ograni~ene mogu}nosti kernela, sinteti~kih i mikroben~mark programa, analiti~ari performansi su standardizovali skupove realnih aplikacionih programa (makroben~markove) koji predstavljaju kolekciju razli~itih **familija aplikacionih ben~mark programa** (*application program benchmark suites*).

- Aplikacije su kompletni, realni programi, koji generišu koristan rezultat.

- Aplikacioni ben~mark programi se obi~no opisuju u zavisnosti od funkcija koje treba da se obave, a one uklju~uju koriš}enje skoro svih resursa sistema kao što su procesor, U/I uredjaji, memorija, sprega sa mre`om, manipulisanje sa bazama podataka, i drugo.

Benčmark strategije

- Najve}i broj tipova ben~mark programa procenu performansi bazira na merenju vremena koje je potrebno da se izvrši taj ben~mark.
- Pored toga postoje i sledeće strategije:
  - Merenje vremena koje je potrebno da se obavi **fiksni iznos izra~unavanja**.
  - Merenje iznosa izra~unavanja koji se obavlja u okviru **fiksnog vremenskog perioda**.
  - **Dozvola promene kako iznosa izra~unavanja koji se obavlja tako i vremena za koje se to izra~unavanje vrši**

Benčmark programi za fiksni iznos izračunavanja

- Definiše se brzina ra~unarskog sistema ili **brzina izvršenja** (*execution rate*) u oznaci R_1 , kao $R_1 = W_1 / T_1$, gde je T_1 vreme potrebno da se obavi izra~unavanje W_1.

- Na `alost, za dati ben~mark program vrednost W_1 nije precizno definisana.

- Da bi kompenzovali ovaj problem, definiše se da vreme izvršenja drugog sistema bude $R_2 = W_2 / T_2$, gde je T_2 vreme potrebno da se obavi izra~unavanje W_2 na tom sistemu.

- Zatim se definiše vrednost **ubrzanje** (*speedup*) S kao $R_1 = S * R_2$. Ova vrednost nam omogu}ava da ka`emo da je brzina izvršenja sistema 1 S puta ve}a od sistema 2.

- Odnos

$$S = \frac{R_1}{R_2} = \frac{W_1 / T_1}{W_2 / T_2}$$

naziva se ubrzanje sistema 1 u odnosu na sistem 2.

Primer :

Projektanti ra~unarskih sistema ~esto koriste termin ubrzanje kada opisuju koliko su se performanse arhitekture promenile nakon što su u~injena neka poboljšanja na toj arhitekturi. Ubrzanje, *S*, u tom slu~aju definiše se kao odnos vremena izvršenja pre i nakon izvršene promene

$$S = \frac{\text{Vreme\_izvrsenja}_{\text{pre}}}{\text{Vreme\_izvrsenja}_{\text{nakon}}}$$

Na primer, ako se program na staroj verziji mašine izvršava za 25s, a na novoj za 15s, tada je ubrzanje

$$S = \frac{25\,\text{s}}{15\,\text{s}} = 1.67$$

Benčmark programi za fiksni iznos izračunavanja- produžetak

- Problem koji nastaje kada se definiše relativno ubrzanje je taj što je teško odrediti aktuelne iznose izra~unavanja, W_1 i W_2, koje obavlja svaka od mašina.
- Kada je reč o MIPS i MFLOPS tada je teško i precizno definisati W_1 i W_2
- Da bi nekako rešili ovaj problem definisa}emo da iznos izra~unavanja koji treba da obave obe mašine bude isti $W = W_1 = W_2$.

- Sada ubrzanje mašine 1 relativno u odnosu na mašinu 2 je

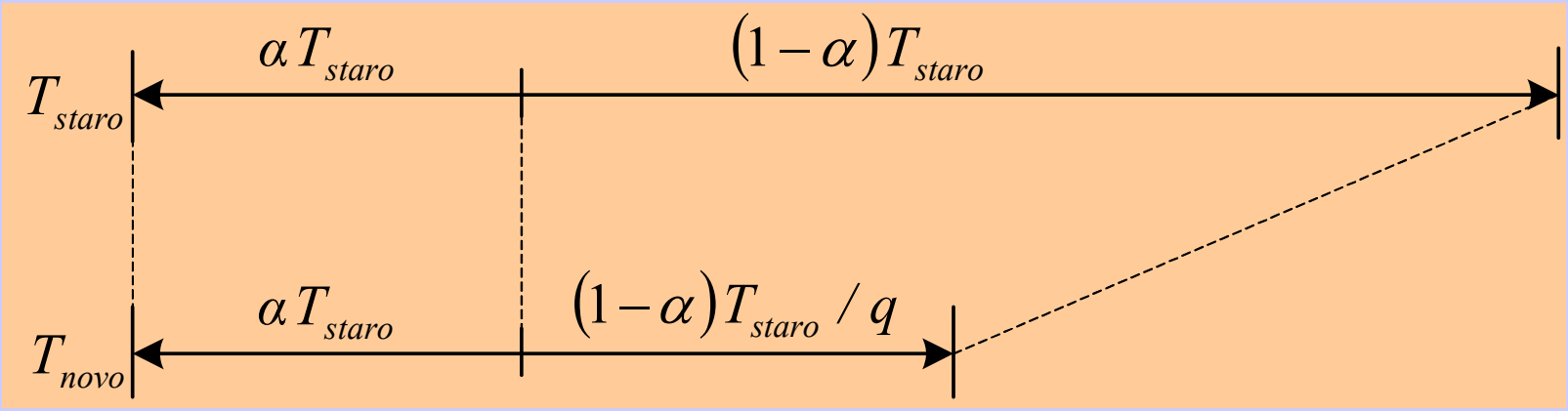
$$S = \frac{R_1}{R_2} = \frac{W / T_1}{W / T_2} = \frac{T_2}{T_1}$$

- Na ovaj na~in fiksira se iznos izra~unavanja koji treba da se obavi od strane ben~mark programa, a koristi se vreme izra~unavanja da bi se procenile performanse.

Amdahl-ov zakon

- Performansna poboljšanja koja se mogu ostvariti koriš}enjem nekog br`eg re`ima izvršenja (rada) ograni~ena su iznosom vremena za koji se taj br`i re`im rada mo`e koristiti.

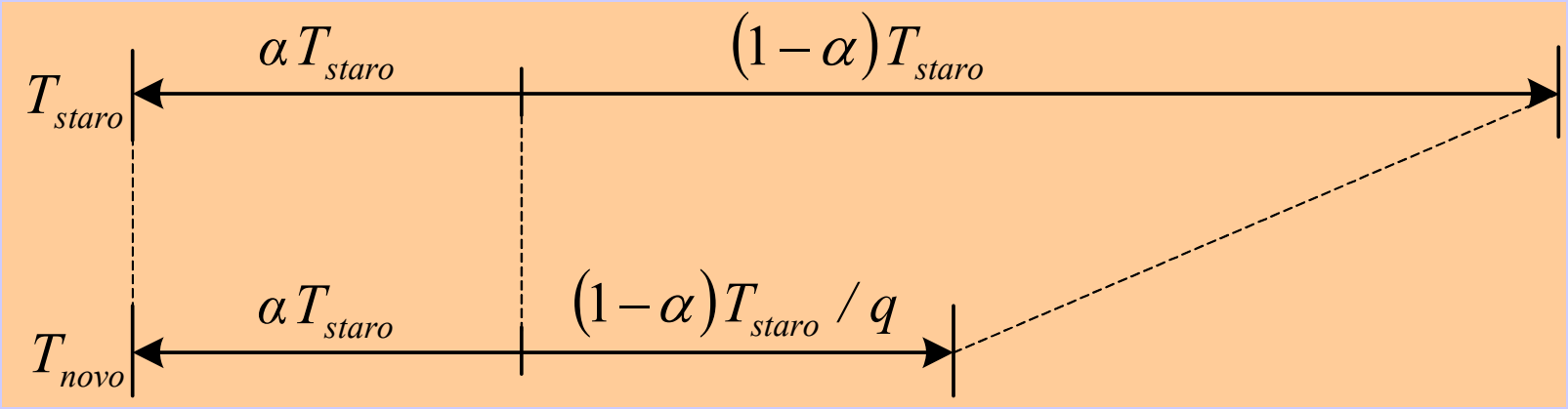
- Razmotri}emo izvršenje programa kao na slici



Vreme izvršenja programa se mo`e podeliti na :

- frakciju α , na koju izvedena sistemska poboljšanja nemaju uticaj, i
- frakciju $1-\alpha$ koja se poboljšava za faktor q .

Amdahl-ov zakon - produžetak



- Gornja linija na slici prikazuje vreme potrebno da se dati program izvrši od strane sistema kod koga izmene nisu izvedene. Ovo vreme nazva}emo T_{staro} .
- Sada usvojimo da su u sistemu izvedene izmene koje redukuju vreme izvršenja nekih operacija u programu za faktor q.
- Program se sada izvršava za vreme T_{novo} , gde je $T_{novo} < T_{staro}$, što je prikazano je na donjoj liniji slike.

Amdahl-ov zakon - produžetak

- Ukupno ubrzanje uzrokovano poboljšanjem odredjuje se na osnovu slede}e relacije:

$$S = \frac{T_{staro}}{T_{novo}} = \frac{T_{staro}}{\alpha T_{staro} + (1-\alpha) T_{staro}/q} = \frac{1}{1/q + \alpha(1-1/q)}$$

- za slučaj kada $q \to \infty$ tada $(1-\alpha)\,T_{staro}/q \to 0$ pa :

$$\lim_{q\to\infty} S = \lim_{q\to\infty} \frac{1}{1/q + \alpha(1-1/q)} = \frac{1}{\alpha}$$

Kao zaključak : Nezavisno od toga koliko je jedan tip operacija u sistemu ostvarenom modifikacijom poboljšan, ipak ukupne performanse sistema ostaju ograni~ene od strane onih operacija koje se i dalje moraju obavljati od dela sistema na koji se ta modifikacija ne odnosi.

Amdahl-ov zakon

Primer :

- U idealnom slu~aju, ubrzanje koje se mo`e ostvariti kod paralelnog ra~unarskog sistema sa ***p*** procesora iznosi ***p***.
- Ako se 10% programa ne mo`e izvršiti paralelno tada maksimalno ubrzanje koje se mo`e posti}i, kada se koristi paralelna mašina, iznosi $1/\alpha = 1/0.1 = 10$, nezavisno od toga koliki je veliki broj procesora, ***p***, koji se ugradjuju u sistem pa čak i ako je beskona~an.
- U konkretnom slu~aju to zna~i, ograni~enje da se 10% od ukupnog programa mora izvršavati sekvencijalno predstavlja faktor koji odredjuje u kom iznosu se mogu ostvariti ukupna performansna poboljšanja

Posledice *Amdahl*-ovog zakon

- Projektant sistema ili programer ako `eli da njegova modifikacija ima najve}i efekat treba da usmeri svoje napore tako što }e izvedenom modifikacijom u~initi najbr`im one aktivnosti koje se naj~eš}e javljaju
- To znači da treba skratiti vreme izvršenja onih operacija koje se naj~eš}e javljaju, tj. operacija koje imaju najve}i uticaj na faktor α .
- Instrukcije koje se naj~eš}e javljaju u toku izvršenja programa su ***Mov***, ***Add***, ***Sub***, ***Branch***, ***Call*** i dr., a one koje se retko javljaju su *FSQRT*, *FSIN*, *FCOS* i dr
- Izvedenom modifikacijom projektanti treba da te`e poboljšanju jednostavnih i ~eš}e koriš}enih, a ne slo`enih i redje pozivanih operacija

Zaključci u vezi benčmark programa za fiksni iznos izračunavanja

- Ben~mark programi koji mere iznos vremena potrebnog da se obavi fiksno izra~unavanje su veoma popularni.

- Razlog za ovo je svakako taj što oni zadovoljavaju našu intuiciju o tome šta poboljšanja u ra~unarskim performansama treba da zna~e.

- Ako kupite sistem koji je br`i, realno je o~ekivati da }e njemu biti potrebno kra}e vreme da izvrši `eljenu aplikaciju.

Benčmark za fiksno vreme izračunavanja

- Alternativa pristupu korišćenja benčmark programa za fiksni iznos izračunavanja predstavlja korišćenje benčmark programa za **fiksno vreme izračunavanja**
- Kada korisnik treba da reši problem velikog obima tada on `eli da za fiksni iznos vremena dobije rešenje.
- Obi~no vreme za koje treba da se dobije rešenje odredjuje se od strane spoljnih faktora.
- Standardno ovakav tip obrade je karakteristi~an za sisteme koji rade u realnom vremenu kakvi su sistemi za prognozu vremena, tj. prognozu o tome kakvo }e vreme biti danas ra~unar treba da izra~una veoma brzo, a ne da to ra~unanje traje do sutra.

Benčmark za fiksno vreme izračunavanja- produžetak

- Umesto da se zadr`ava konstantan iznos izra~unavanja koji se obavlja od strane ben~mark programa, fiksira se prvo vreme za koje se ben~mark izvršava.
- U drugom koraku, menja se iznos izra~unavanja koje treba da se obavi od strane sistema kada on izvršava ben~mark, a meri vreme.
- Na osnovu iznosa izra~unavanja u oba koraka, koji se koriste kao mere za relativnu brzinu razli~itih sistema, procenjuju se performanse sistema.
- SLALOM ben~mark program je tipi~an predstavnik koji implementira strategiju za fiksno-vreme promenljivo-izra~unavanje

Skaliranje *Amdahl*-ovog zakona

- Jedna od glavnih kritika upu}enih *Amdahl*-ovom zakonu bila je ta da on ne pru`a pravu sliku o potencijalu performansi paralelnih ra~unarskih sistema.
- Argument za ovu tvrdnju je taj što kupce paralelnih sistema svrstava u kategoriju korisnika koji `ele da reše obiman problem u okviru dostupnog vremena.
- Slede}i ovu argumentaciju ka`emo da postoje slede}e dve verzije *Amdahl*-ovog zakona:
  - verzija fiksnog vremena
  - verzija skaliranja.

Skaliranje *Amdahl*-ovog zakona - produžetak

- Da bi procenili performanse aplikacije koja se izvršava na paralelnom sistemu uporedi}emo vreme izvršenja programa T_P na sistemu sa p procesora, sa vremenom koje je potrebno da se izvrši ekvivalentna sekvencijalna verzija te aplikacije, T_1 , koriste}i se definicijom ubrzanja $S_P=T_1/T_P$.

- Ako sada sagledamo izvršenje koje se odnosi na fiksno vreme uo~avamo da realno ne postoji jednoprocesorski sistem koji je u stanju da izvrši ekvivalentnu sekvencijalnu verziju paralelne aplikacije.

- **Primera radi:**

  - jednoprocesorski sistem nema dovoljno velike memorije, ili

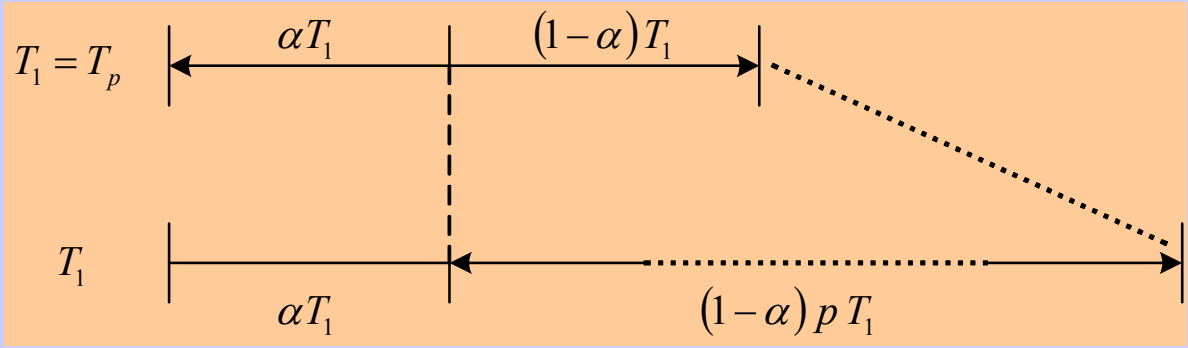
  - vreme koje je potrebno da se izvrši sekvencijalna verzija mo`e biti neprihvatljivo dugo.

Skaliranje *Amdahl*-ovog zakona - produžetak

Da bi se rešio problem vreme paralelnog izvršenja delimo na paralelnu komponentu, **1-α** , i strogo sekvencijalnu komponentu, **α**, što rezultira da vreme izvršenja programa na paralelnom sistemu iznosi

$$T_p = T_1 = \alpha T_1 + (1-\alpha)T_1$$

, gde je T_1 vreme potrebno da se generiše odgovor, tj. aplikaciji je uvek dozvoljeno vreme T_1 za izvršenje



Skaliranje *Amdahl*-ovog zakona - produžetak

S obzirom da ne postoji jednoprocesorski sistem koji je u stanju da izvrši ekvivalentni problem ovakvog obima, usvajamo da se paralelni deo vremena izvršenja pove}ava za faktor ***p*** kada se aplikacija izvršava na hipoteti~kom jednoprocesorskom sistemu.

Ovo rezultira hipoteti~kom sekvencijalnom vremenu izvršenja

$$T_1' = \alpha T_1 + (1-\alpha) p T_1$$

Paralelno ubrzanje se odredjuje sada kao

$$S_p = \frac{T_1'}{T_p} = \frac{\alpha T_1 + (1-\alpha) p T_1}{T_1} = p + \alpha(1-p)$$

Zaključak: Interpretacija *Amdahl*-ovog zakona kada se govori o fiksnom-vremenu izra~unavanja sa promenljivim-obimom problema ukazuje da performanse ra~unarskog sistema nisu ograni~ene od dela koji se izvršava strogo sekvencijalno. Naime, kako sistem postaje ve}i obim problema koji se mo`e rešiti pove}ava se proporcionalno

Benčmarkovi koji se odnose na promenljivo-izračunavanje i promenljivo-vreme

- **Prva strategija:** ben~mark programi koji se odnose na fiksno-izra~unavanje indiciraju nam na koji na~in poboljšanja performansi uti~u na vreme izvršenja aplikacionog programa.

- **Druga strategija:** ben~mark programi koji fiksiraju vreme izvršenja dozvoljavaju da se obim problema skalira kako bi se uskladile mogu}nosti sistema koji se testira sa potrebama aplikacije.

- **Tre}a strategija:** najgeneralnija ben~mark programska strategija ne fiksira kako dostupno vreme tako i iznos izra~unavanja koje treba obaviti. Obi~no to zna~i da se kod ovih ben~markova za procenu performansi parametri kakvi su vreme i obim izra~unavanja naj~eš}e ne uzimaju u obzir.

- **HINT** ben~mark je tipi~an reprezent strategije promenljivo-izra~unavanje promenljivo-vreme, koji ceni kvalitet rešenja.

Popularni benčmark programi

- Veliki broj razli~itih ben~mark programa razvijen je zadnjih nekoliko decenija sa ciljem da se procene performanse razli`itih tipova ra~unarskih sistema za razli~ite domene aplikacija
- Tipični reprezenti su:
  - STREAM
  - SIEVE
  - Livermore loops
  - NAS kerneli
  - LINPACK
  - Whetstone i Dhrystone
  - SPEC benčmark familija

STREAM

- **STREAM** je sinteti~ki ben~mark koji se koristi za merenje memorijske propusnosti (u MB/s) kao i odgovaraju}u brzinu izra~unavanja.

- Razvoj STREAM ben~marka je posledica da su procesori znatno br`i od memorije, i to što je veliki broj programa ograni~en propusnoš}u memorije, a ne brzinom rada procesora.

- Program je predvidjen da radi sa skupovima podataka koji su znatno ve}i od obima dostupne keš memorije.

ime	kôd	bajtova/iteraciji	FLOP/iteraciji
COPY	a(i) = b(i)	16	0
SCALE	a(i) = q* b(i)	16	1
SUM	a(i) = b(i) +c(i)	24	1
TRIAD	a(i) = b(i) +q *c(i)	24	2

- Vektori *a, b* i *c* su strukture podataka tipa jedno-dimenzionalna polja veli~ine 2-3 miliona elemenata, svaki obima 8-bajtova.

SIEVE

- Kernel ben~mark **SIEVE** (sito) se koristi za uporedjivanje mikroprocesora, PC mašina, i HLL-ova.
- Ovaj ben~mark se bazira na algoritmu *Erastoten*-ovo sito, i koristi se za nala`enje svih prostih brojeva manjih od zadatog broja *n*.
- Algoritam se prvo sastoji od upisivanja svih celobrojnih vrednosti po~ev od 1 do *n*, a zatim obele`avanja svih multipla od

$$k = 2,3,\ldots,\sqrt{n}$$

SIEVE – algoritam manuelnog postupka

- Da bi odredili sve proste brojeve od 1 do 20 potrebno je sprovesti slede}e korake:

- Korak1: Upisati sve brojeve od 1 do 20 i ozna~iti ih kao proste

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20

- Korak 2: Izbaciti sve multiple broja 2 iz liste prostih brojeva. Oni koji ostaju su slede}i

1	2	3	5	7	9	11	13	15	17	19

- Korak 3: Naredna celobrojna vrednost u nizu je 3. Izbaciti sve multiple od 3. Oni koji ostaju su slede}i

1	2	3	5	7	11	13	17	19

- Korak 4: Naredna celobrojna vrednost u nizu je 5 koja je ve}a od . Zaklju~ujemo da je sekvenca dobijena u koraku 3 `eljeni niz koji sadr`i sve proste brojeve

Livermore loops

- ***Livermore loops*** ~ine 24 kernel ben~marka, koji su tako odabrani da predstavljaju najkriti~nije sekcije *Fortran*-skih programa koji se veoma ~esto koriste kod numeri~kih aplikacionih programa.
- Svaka petlja je karakteristi~na za sebe, odra`ava neki realni problem koga treba rešiti, a u sebi sadr`i (uklju~uje) kako dobar tako i loš stil programiranja.
- Performansna metrika koja se dobija kao rezultat izvršenja ovog programa je MFLOPS.
- Izvršenje ovih malih kernela je jednostavnije, lakše i br`e u odnosu na izvršenje ukupnog aplikacionog programa, po je ovaj ben~mark veoma popularan kod kompariranja performansi superra~unara.

NAS kerneli

- NAS (*Numerical Aerodinamic Simulation*) kernel ben~mark razvijen je prvenstveno za testiranje performansi vektorskih superra~unara.
- NAS ~ine 7 kernela napisanih na *Fortran*-u koji se koriste za izra~unavanje u oblasti dinamike fluida.
- Ovi kerneli obavljaju operacije tipa 2D FFT sa kompleksnim brojevima, *Cholesky* dekompozicija, matri~no mno`enje, i *Gaussian* eliminacija.
- Performansna metrika je MFLOPS.

LINPACK

- LINPACK je verovatno najpopularniji ben~mark program.
- Izvorno je napisan na *Fortran*-u, ali postoje verzije za druge HLL-ove.
- Ovaj ben~mark koristi LU faktorizaciju i *back*-supstituciju za rešavanj gusto posednutih *NxN* sistema linearnih jedna~ina.
- Performansna metrika koju koristi ova metrika je MFLOPS

Whetstone i Dhrystone

- oba ben~marka su sinteti~ki programi.
- odavno se više ne koriste, ali su interesantni zbog njihove istorijske uloge.

Operacije u ***Whetstone*** ben~mark su odredjene prou~avanjem ponašanja nau~no orijentisanih aplikacionih programa napisanih na *Algol*-u, negde 70-ih godina prošlog veka. Primarni cilj je bio da se mere *FP* performanse, kao što je *FP* sabiranje, *FP* oduzimanje, i nekoliko *FP* trigonometrijskih operacija.

Dhrystone ben~mark je bio namenjen za merenje *integer* performansi koriš}enjem velikog broja operacija tipa kopiranje-karakter-nizova, kao i operacije uporedjivanje-nizova.

SPEC benčmark familija

- Ova familija je razvijena od strane neprofitne organizacije *Standard Performance Evaluation Corporation*.
- SPEC je mnogo bli`i realnim aplikacijama i mnogo bli`e u odnosu na male sinteti~ke kernel programe odra`ava stvarno radno optere}enje.
- prvobitno SPEC familija je razvijena za procenu performansi CPU-a, ali je kasnije proširena na klijent/server izra~unavanje, komercijalne aplikacije, U/I podsisteme, i td.
  - SPEC je razvio slede}e grupe *benchmark*-ova :
    - SPEC 95
    - SPEChpc 96
    - SPECweb 96
    - SDM
    - GPC

SPEC benčmark familija - produžetak

- **SPEC 95** - procenjuje performanse CPU-a, memorijskog sistema, kao i kôd generisan od strane kompilatora.
- **SPEChpc 96** - procenjuje performanse visoko-performansnih ra~unarskih sistema koji izvršavaju aplikacije tipi~ne za potrebe industrije. Teku}e sadr`i dva ben~marka, *SPECseis 96* koji se koristi za seizmi~ka procesiranja, i *SPECchem 96* koji je namenjen za izra~unavanja tipi~na kod hemijskih reakcija.
- **SPECweb 96** - procenjuje performanse Web servera.
- **SDM** (*System Development Multitasking*) - procenjuje kako se sistem ponaša kada radi u okru`enju sa ve}im brojem korisnika pod Unix operativnim sistemom.
- **GPC** (*Graphic Performance Characterization*) - procenjuje grafi~ke performanse sistema.

SPEC 95

- Svakako najpoznatija i najšire prihva}ena ben~mark grupa je *SPEC 95* koja sadr`i ben~mark programe za testiranje performansi CPU-a.
- Ovi ben~mark programi se koriste za procenu brzine rada CPU-a, keš/memorijskog sistema, kao i kvalitet generisanog koda od strane kompilatora.
- Vreme koje procesor troši na izvršenje rutina operativnog sistema i U/I funkcija je zanemarljivo.
- *SPEC 95* se sastoji od:
  - CINT 95 - skup od osam programa koji manipulišu sa celobrojnim vrednostima (*integer*), i
  - CFP 95 - skup od deset programa koji manipulišu sa brojevima u pokretnom zarezu (*floating point*).

Prezentacija podataka

III poglavlje

- Osnovni pojmovi-

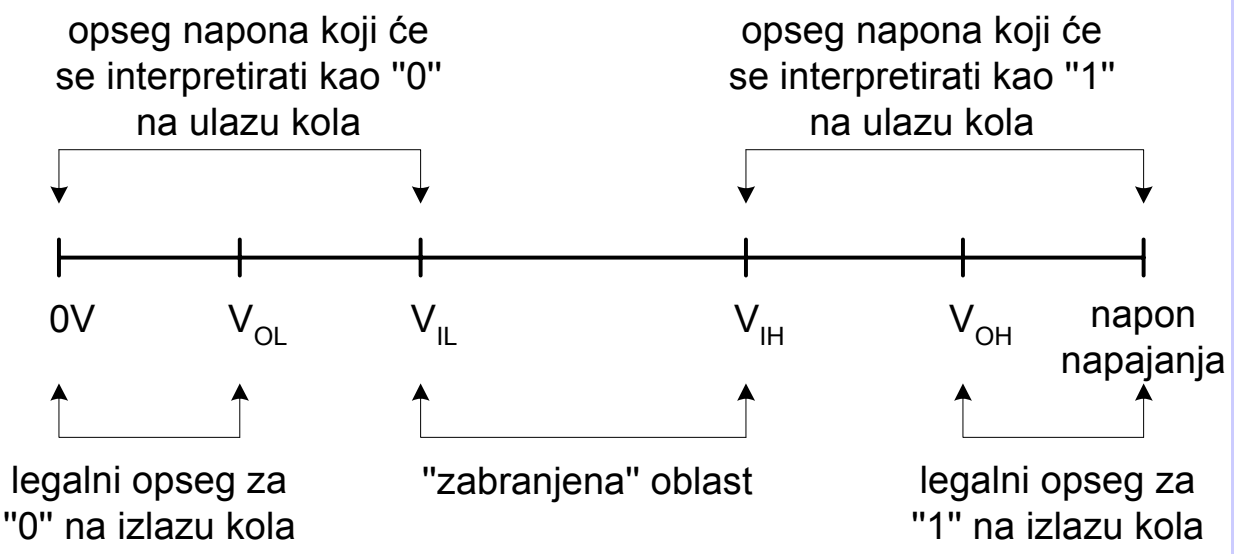
Definicija pojmova i tehnike kodiranja

- Standardna aktivnost koja se kod ra~unarskih sistema koristi za predstavljanje podataka naziva se **prezentacija podataka** (*data presentation)* ili **kodiranje podataka** (*data encoding*).
- Najjednostavnije i najvažnije tehnike kodiranja su:
  - kodiranje ozna~enih i neozna~enih brojeva (*integer*);
  - kodiranje realnih brojeva (*floating point*); i
  - kodiranje karaktera (*characters*) koji se mogu štampati
- Razlozi za dobro razumevanje tehnika kodiranja su:
  - način izvodjenja aritmetičkih operacija koje zavise od kodiranja
  - način prezentacije podataka kojim se direktno odredjuje obim memorije potreban za memorisanje svakog podatka

Bitovi kao jedinice informacija

- Pojam **konvencija-signaliziranja** odredjuje kako se analogni elektri~ni signali interpretiraju kao digitalne vrednosti.

- Preslikavanje naponskih nivoa u bitove se vrši shodno slici :



- Sistemi koji preslikavaju svaki elektri~ni signal u dve vrednosti zovu se **binarni sistemi.** Informacija koju nosi svaki od signala zove se **bit** (*Binary Digit*).

Prezentacija podataka

III poglavlje

- Tipovi podataka -

Tipovi podataka

- Ista vrednost se mo`e predstaviti na ve}i broj na~ina. Tako na primer, prezentacija broja pet kod razli~itih notacija je prikazana kao

tip notacije	prezentacija
standardna decimalna	5
rimska	V
unarna	11111
binarna	00000101

- Za konkretnu prezentaciju podataka ka`emo da je **tip podataka** (*data type*) ako u ra~unaru postoje operacije koje mogu da operišu nad informacijom koja je kodirana u toj prezentaciji. Svaka ISA poseduje svoji sopstveni skup tipova podataka i sopstveni skup instrukcija sa kojima operiše nad tim tipovima podataka.

Brojni sistemi

- Najčešće korišćeni brojni sistemi su decimalni (osnove 10), binarni (osnove 2), oktalni (osnove 8), i heksadecimalni (osnove 16)

- Mašina najlakše manipuliše sa brojnom osnovom 2, a čovek sa brojnom osnovom 10

- Sa prakti~ne ta~ke gledišta, razli~ite aplikacije koriste razli~ite brojne osnove. U principu, brojne osnove 8 i 16 pogodnije su za prezentaciju bit-oblika (grafi~ki prikazi), dok je brojna osnova 10 korisnija za razumevanje vrednosti memorisanih brojeva koji se koriste u raznim ra~unskim transakcijama.

Pozicioni koeficijenti i težine

- Naj~eš}e koriš}eni brojni sistemi zasnovani su na **pozicionim koeficijentima** i **te`inama**.

- Vrednost cifre na svakoj poziciji u broju odgovara njegovom pozicionom koeficijentu. Po~ev od desne strane prema levoj te`ini svake cifre sukcesivno se dodeljuje ve}i stepen osnove brojnog sistema. Tako na primer, vrednost *Q* u brojnom sistemu osnove *r* (osnova se naziva *radix*) se izra~unava kao:

$$Q = x_n w_n + x_{n-1} w_{n-1} + \ldots + x_1 w_1 + x_0 w_0 + \{x_{-1} w_{-1} + \ldots + x_{-m} w_{-m}\}$$

gde se viti~astim zagradama predstavlja razlomljeni deo vrednosti *Q*, dok

$$w_i = r^i$$

odgovara *te`ini*, a *r* predstavlja *brojnu osnovu* ili *bazu*, pri ~emu va`i da je $0 \le x_i \le r-1$, gde je x_i pozicioni koeficijent.

Prezentacija broja 154 u brojnim osnovama 2, 8, 10, i 16

$$154 = 1*10^2 + 5*10^1 + 4*10^0 \text{.....osnova10}$$

$$= 1*2^7 + 0*2^6 + 0*2^5 + 1*2^4 + 1*2^3 + 0*2^2 + 1*2^1 + 0*2^0 \text{...osnova2}$$

$$= 2*8^2 + 3*8^1 + 2*8^0 \text{.....osnova8}$$

$$= 9*16^1 + A*16^0 \text{.....osnova16}$$

- Kada se govori o prezentaciji brojeva osnove 16, notacije koje se standardno koriste u ra~unarskoj tehnici su *0X* ili *0x* (od strane C kompajlera) ili sufiks *H* ili *h* (obi~no u dokumentaciji kompanije Intel), kao na primer

$$154_{10} = 9A_{16} = 0x9A = 0X9A = 9Ah = 9AH$$

Binarne i heksadecimalne prezentacije

- Rad tranzistorsko zasnovanih elektronskih komponenti, kao osnovnih gradivnih blokova integrisanih kola koje se ugradjuju u ra~unarske sisteme, je po prirodi binarni

- S obzirom da duge nizove nula i jedinica ~ovek teško pihvata i razume, najve}i broj ljudi koji se profesionalno bavi ra~unarstvom rutinski koristi bazu 16, a redje 8.

- Razli~ita softverska sredstva, kakvi su asembleri, kompajleri, linkeri itd., vrše konverziju ovih vrednosti u binarne.

Konverzija binarnih brojeva u brojeve osnove 8 ili 16

- Konverzija binarnih brojeva u brojeve baze 8 ili 16 se vrši jednostavno. Počev od desne strane prema levoj izdvajaju se grupe od po 3 bita (za oktalni) ili 4 bita (za heksadecimalni), nakon toga se vrši zamena svake binarne grupe odgovarajućom cifrom u novoj osnovi

01001110001_{2} osnova2

$010\ 011\ 100\ 001_{2} = 2341_{8}$ osnova8

$0100\ 1110\ 0001_{2} = 4E1_{16}$ osnova16

Prezentacija decimalnih vrednosti u binarnoj, oktalnoj i heksadecimal noj notaciji

decimalni	binarni	oktalni	heksadecimalni
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	20

Proširenje bit reprezentacije broja

- Jedna od standardnih konverzija koja se veoma ~esto koristi kod manipulisanja sa celobrojnim vrednostima ~iji je obim re~i razli~it, zadr`avaju}i pri tome istu numeri~ku vrednost, predstavlja proširenje bit reprezentacije broja. U toku sprovodjenje postupka konverzije moramo biti svesni slede}ih dveju ~injenica:

  - konverzija nije mogu}a kada je odredišni tip podataka suviše mali da bi predstavio `eljenu vrednost, i
  - konverzija sa manjeg na ve}i tip treba da je uvek mogu}a.

Proširenje neoznačenih brojeva

- Konverzija neozna~enog broja u podatak ve}eg obima obavlja se jednostavnim dopisivanjem vode}ih nula reprezentaciji broja. Ova operacija se naziva **proširenje nulom** (*zero extension*).

- Neoznačeni broj

$$[x_{n-1}, x_{n-2}, \cdots, x_0]$$

se proširuje na sledeći način

$$[0, \ldots, 0, x_{n-1}, x_{n-2}, \cdots, x_0]$$

Proširenje označenih brojeva u dvojičnom komplementu

- U postupku konverzije broja predstavljenog u dvoji~nom komplementu u podatak ve}eg obima potrebno je pridr`avati se pravila koje se odnosi na **proširenje znaka** (*sign extension*), tj. dopisivati MS bit reprezentaciji broja.

- Označeni broj u dvojičnom komplementu

$$[x_{n-1}, x_{n-2}, \dots, x_0]$$

se proširuje na sledeći način

$$[x_{n-1}, x_{n-1}, \dots, x_{n-1}, x_{n-2}, \dots, x_0]$$

Odsecanje brojeva

- **Odsecanje broja** (*truncating number*) je operacija koja se obavlja redukcijom broja bitova kojima se predstavlja broj i po prirodi je suprotna operaciji proširenja.

- Kod odsecanja ***n***-bitnog broja

$$x = [x_{n-1}, x_{n-2}, \dots, x_0]$$

na ***k***-to bitni broj zadr`avamo ($n - k$)-LS bitova

- Rezultantni bit vektor je oblika

$$x' = [x_{k-1}, x_{k-2}, \dots, x_0]$$

Tipovi podataka sa kojima manipuliše računar

- Za mikroprocesor se ka`e da podr`ava odredjeni tip podataka samo ako je u stanju da sa prihvatljivom efikasnoš}u manipuliše sa operandima tog tipa.
- Za efikasnu podršku va`na su slede}a dva faktora:
  - instrukcije podržavaju manipulisanje nad formatima podataka sa kojima se manipuliše, i
  - adresni na~ini rada koji obezbedjuju jednostavan pristup podacima sa kojima se manipuliše.

Obim podataka

- Memoriju ra~unara ~ini skup uzastopno numerisanih (adresibilnih) registara, pri ~emu svaki od registara obično ~uva po jedan **bajt**.

- Svaki registar ima svoju adresu koja se naziva **memorijska lokacija**, tj. za memoriju ka`emo da je adresibilna na nivou bajta.

- Pojam **nibl** (*nibble* ili *nybble*) se odnosi na kolekciju od ~etiri susedna bita.

- U ra~unarstvu, zna~enje termina "bit", "bajt", i "nibl" je standardizovano i ne zavisi od specifi~nosti arhitekture, dok zna~enje termina **re~** (*word*) zavisi od tipa procesora. Standardni obimi tipa re~ su 4-, 8-, 16-, 32-, 64- i 128-bitova.

- Danas se uobičajeno smatra da je re~ obima 32-bita, 16-bitna re~ se naziva **polu-re~** (*half-word*), a 64-bitna **dupla-re~** (*double word*).

Podaci kojima manipulišu mikroprocesori

- Saglasno obimu podataka sa kojima munipulišu, mikroprocesore delimo na 4-, 8-, 16-, 32-, 64- i 128-bitne.
- Veličina 4, 8, 16, 32, 64 i 128 odnosi se na obim podatka kojim mikroprocesor manipuliše.
- Pri ovome razlikujemo dva slu~aja:
  - obim podatka predstavlja maksimalan broj bitova koji se mo`e preneti izmedju CPU-a i memorije u jednom mašinskom ciklusu,
  - obim podataka je odredjen maksimalnim obimom operanda kojim mo`e da manipuliše ALU jedinica CPU-a.
- Prvi slu~aj odnosi se na spoljni put, a drugi na interni

Organizacija memorije sa aspekta obima podataka

- Svi mikroprocesori poseduju memorijski podsistem koji je organizovan kao linearno polje osnovnih memorijskih elemenata.
- Svakom elementu pridru`ena jedinstvena adresa.
- Obim osnovnih memorijskih elemenata je tako izabran da zadovolji kompromis izmedju slede}a dva nekompatibilna zahteva:
  - kompaktna prezentacija podataka,
  - efikasan pristup podacima koji se ~uvaju u memoriji.
- Postoje implementacije kod kojih je dopustivo:
  - **adresiranje na nivou bita**,
  - **na nivou fiksnih re~i**,
  - **na nivou bit-polja**.

Adresiranje na nivou bita

- Podacima se dodeljuje niz kontinualnih bitova
- Glavni problem bit adresne šeme je slede}i:

  - selektovani memorijski bit treba, u zavisnosti od razlike koja postoji izmedju memorijske blok-adrese i bit adrese koju definiše CPU, prenositi izmedju memorije i CPU-a po razli~itim fizi~kim linijama. Ovakav pristup zahteva **ugradnju multipleksera** na memorijskim plo~ama.

  - alternativno rešenje je da se obavi emulacija bit adresiranja pomo}u mašine koja pristupa ve}im memorijskim blokovima. Kod ovog rešenja, memorija se adresira u ve}im blokovima, a izdvajanje bit polja obavlja se od strane CPU-a kod koga se, standardno, ugradjuje poseban hardver za tu namenu, kao što je **barel-pomera~** (*barrel-shifter*).

Adresiranje na nivou fiksnih reči

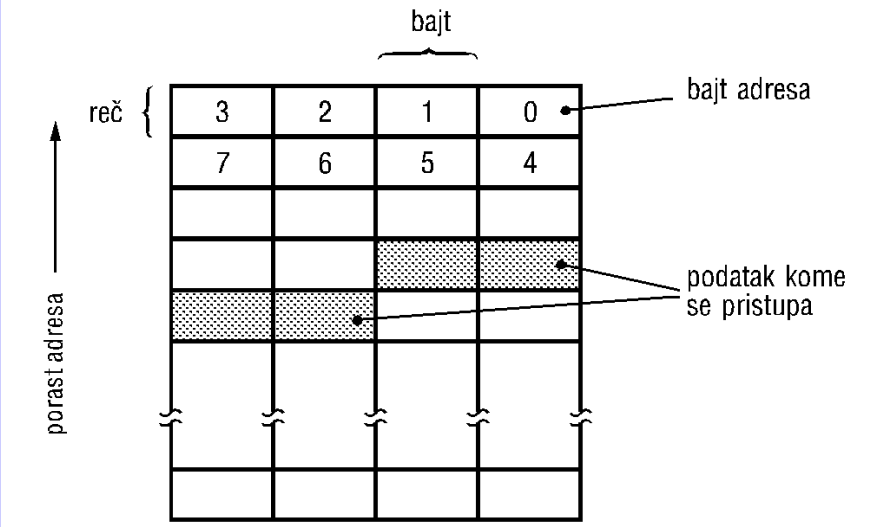
- Skoro kod svih 32-bitnih mikroprocesora memorija je organizovana u blokovima obima 32 bita.
- Svaki blok se deli na ~etiri bajta.
- Najmanja adresibilna memorijska jedinica je bajt.
- Adrese koje se predaju od strane CPU-a su bajt-adresibilne.
- Podatku ~iji je obim ve}i od jednog bajta pristupa se slanjem adrese prvog bajta.
- Adresi se pridru`uju upravlja~ki signali koji ukazuju na obim podatka koji se ~ita.

Adresiranje na nivou fiksnih reči – (produžetak)

- Kod 32-bitnih mikroprocesora naj~eš}e koriš}eni podaci, kojima se pristupa u jednom mašinskom ciklusu, su obima:
  - re~ - 32-bitni podatak,
  - polure~ - 16-bitni podatak,
  - bajt - 8-bitni podatak, i
  - dupla re~ - 64-bitni podatak

Poravnanje

- Bajt adresiranje i organizacija memorije na nivou fiksnih re~i, kao tehnike izvodjenja, ne rešavaju problem pristupa podacima kada su oni locirani na granici dve re~i



- Rešenje ovog problema se sastoji u sledećem :
  - izbegava se dodela (alokacija) podataka (kao na slici)
  - ovakve situacije se po odredjenom automatizmu obradjuju od strane procesora.

Poravnanje za različiti obim reči

- Pristup podatku obima ***s*** bajtova na adresi ***A*** je poravnan (obavlja se u jednom mašinskom ciklusu) ako je ***A mod s = 0***

Adresirani objekat	Poravnjanost na bajt ofset adresi	Neporavnjanost na bajt ofset adresi
bajt	0, 1, 2, 3, 4, 5, 6, 7	nikad
polu reč	0, 2, 4, 6	1, 3, 5, 7
reč	0, 4	1, 2, 3, 5, 6, 7
dupla reč	0	1, 2, 3, 4, 5, 6, 7

Bit i bit-polje

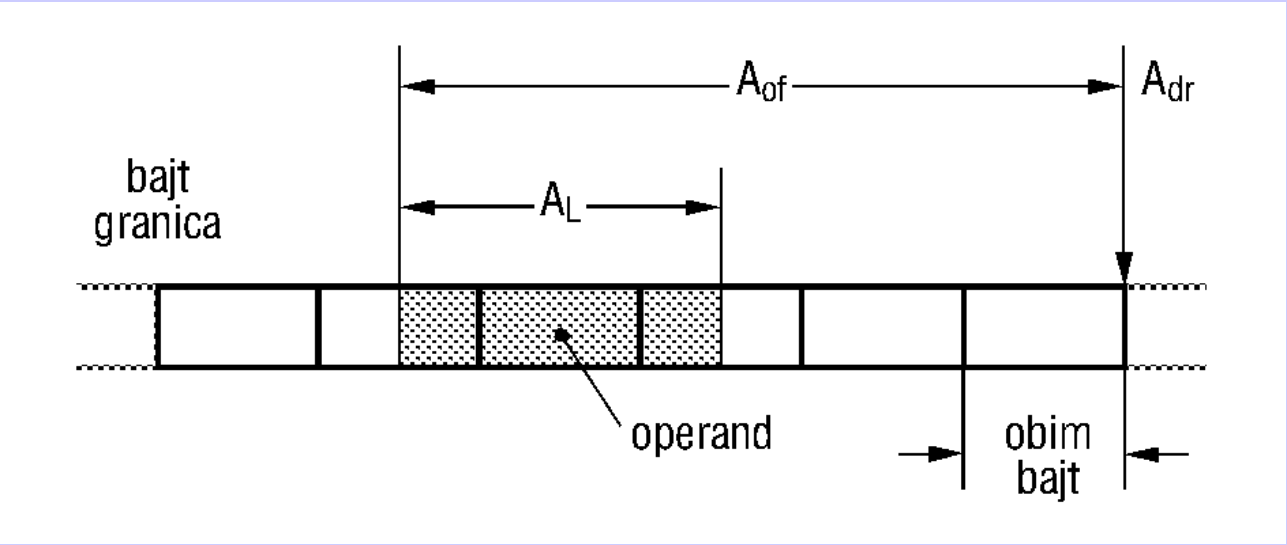
- Bit polje predstavlja skup kontinualnih bitova u memoriji, pri čemu ne postoji specijalna interpretacija koja se pridružuje njegovom sadržaju.

- Mašine koje podržavaju rad sa ovakvim tipom podataka poseduju instrukcije koje mogu da obave jednu ili veći broj od sledećih aktivnosti:

  - kopiranje bit polja u memorijsku lokaciju ili registar,
  - slaganje (pakovanje) rezultata, i
  - proširenje znaka, kada se to instrukcijom eksplicitno zahteva.

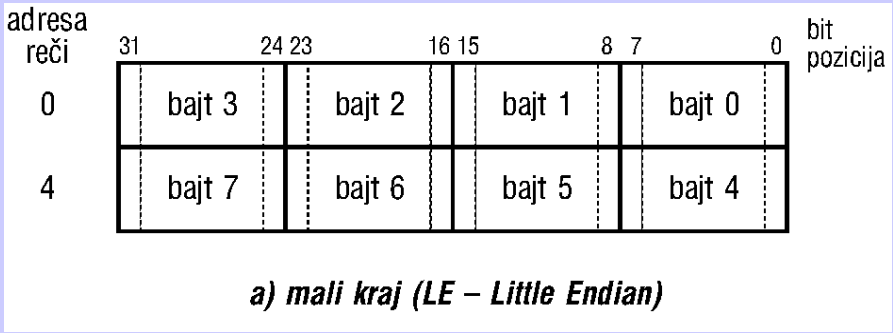
Definicija bit-polja

- Da bi se definisalo bit-polje potrebne su slede}e informacije:
  - adresa bajta, A_{dr}
  - adresa pomeraja (ofset adresa), A_{of}, i
  - du`ina polja, A_L.

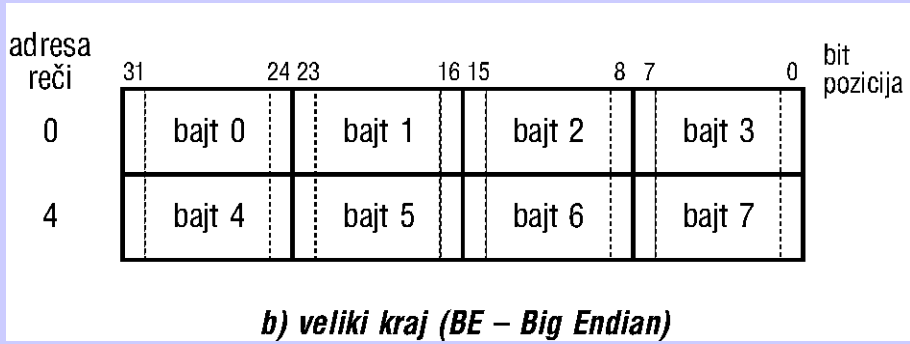


Uredjenost bajtova u okviru reči

- Više-bajtne re~i su smeštene u memoriji kao niz bajtova, a adresiraju se bajtom u re~i koji ima najni`u adresu
- Postoje slede}a dva izbora koja se odnose na redosled pam}enja bajtova u memoriji:
  - bajt najve}e te`ine (MS bajt) na najni`oj adresi, ova uredjenost se naziva **veliki-kraj** (*big-endian*, **BE**), i
  - bajt najmanje te`ine (LS bajt) na najni`oj adresi, ovo uredjenje se naziva **mali-kraj** (*little-endian*, **LE**).



a) mali kraj (LE – Little Endian)



b) veliki kraj (BE – Big Endian)

Uredjenost bajtova kod mikroprocesora

- Oba na~ina uredjenosti bajtova, u okviru re~i, sre}emo kod današnjih mikroprocesora, a karakteristi~na su ~etiri pristupa u radu:

Tip mikroprocesora	korišćena uredjenost
Intel iAPX x86	LE bajt adrsiranje
MotorolaMC 68xxx	BE bajt adresiranje
MIPS x000	u trenutku reseta odredjuje da li }e se koristiti LE ili BE bajt adresiranje, ono se u toku rada ne mo`e promeniti
Intel i860	postoji softverska instrukcija pomoću koje u toku izvršenja programa može da se vrši prelaz sa BE na LE bajt adresiranje i obratno

Klasifikacija podataka

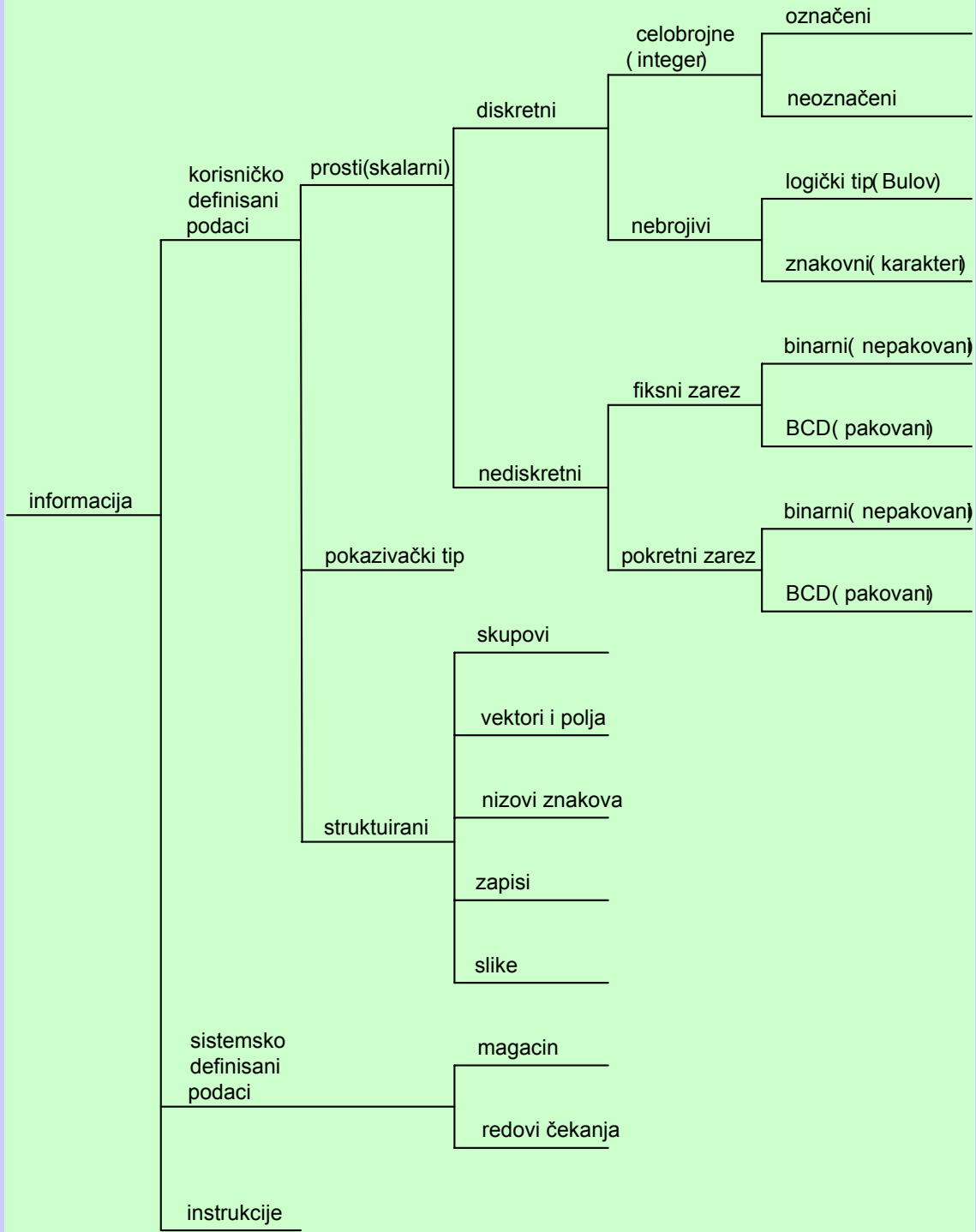
- Tipove podataka (informacija) sa kojima ra~unar manipuliše delimo u tri kategorije:

  - **korisni~ko definisani podaci** - korisnik ih eksplicitno specificira u programu. Struktura ovih podataka je uglavnom odredjena mogu}nostima programskog jezika.

  - **sistemsko definisani podaci** - u toku izvršenja programa implicitno se generišu od strane operativnog sistema.

  - **instrukcije** - program koji se izvršava mo`e se posmatrati kao kompozicija podataka koji imaju svoju sopstvenu strukturu i osobine.

Podela tipova podataka



Korisničko definisani podaci

- Korisničko definisane podatke delimo na:
  - proste (skalarne)
  - struktuirane
  - pokazivačkog tipa

Korisničko definisani podaci

Prosti (skalarni) tipovi podataka

- Svaki tip podatka iz ove grupe ima slede}e tri osobine:
  - skup vrednosti koje podatak mo`e da uzima (domen);
  - operacije koje su nad tim podatkom dozvoljene; i
  - na~in predstavljanja u memoriji ra~unara.
- **Domen** karakterišu dva aspekta:
  - **opseg** - broj vrednosti koje podatak mo`e da uzima; i
  - **preciznost** - rastojanje izmedju sukcesivnih vrednosti podataka

Označeni i neoznačeni brojevi

Informacija

Korisničko def. podaci

Skalarni

Diskretni

Integer

neoznačeni

$$A_{US} = \sum_{i=0}^{n-1} a_i \cdot 2^i$$

označeni

znak moduo

$$A_{SM} = (-1)^{a_{n-1}} \cdot (\sum_{i=0}^{n-2} a_i \cdot 2^i)$$

jedinični komplement

$$A_{1C} = -a_{n-1} \cdot (2^{n-1} - 1) + \sum_{i=0}^{n-2} a_i \cdot 2^i$$

dvojični komplement

$$A_{2C} = -a_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} a_i \cdot 2^i$$

Neoznačeni celi brojevi

- **Celobrojne vrednosti** (*integers*) se normalno koriste:
  - za brojanje,
  - kao indeksi elemenata tipa polja,
  - za predstavljanje memorijskih adresa i dr.

- Kod ra~unarskih sistema njihov **opseg** (*range*) je, s obzirom na broj bitova koji se koriste za prezentaciju, ograni~en od strane arhitekture ra~unara.

Predstavljanje i opseg neoznačenih brojeva

- Vrednost A predstavlja se kao niz od *n* bitova ($a_{n-1},...,a_0$) ~ija je vrednost

$$A = \sum_{i=0}^{n-1} a_i \cdot 2^i$$

- Kod ove prezentacije opseg dozvoljenih vrednosti je

$$0 \leq A \leq 2^n - 1$$

a ove brojeve zovemo **neozna~eni celi brojevi**.

Označeni celi brojevi

- Najkarakteristi~nije su slede}e metode: znak-moduo, jedini~ni komplement, dvoji~ni komplement, i notacija viška

decimalni	neoznačeni	znak moduo	jedinični komplement	dvojični komplement	višak 4
7	111	-	-	-	-
6	110	-	-	-	-
5	101	-	-	-	-
4	100	-	-	-	-
3	011	011	011	011	111
2	010	010	010	010	110
1	001	001	001	001	101
+0 / -0	000	000/100	000 / 111	000	100
-1	-	101	110	111	011
-2	-	110	101	110	010
-3	-	111	100	101	001
-4	-	-	-	100	000

Znak moduo

- Kod ove binarne prezentacije bit najve}e te`ine (MSB - *Most Significiant Bit*) odredjuje znak broja. Ako MSB ima vrednost 0 broj je pozitivan, a kada je 1 broj je negativan. Ostatak bitova sadr`i apsolutnu vrednost broja. Tako na primer, u 8-bitnoj prezentaciji znak-moduo brojevi +12 i -12 imaju slede}i format:

$$(+12)_{10} = (00001100)_2 \qquad (-12)_{10} = (10001100)_2$$

- Negativan broj se dobija kada se MSB pozitivnog broja promeni sa 0 na 1.

- Postoji pozitivna i negativna prezentacija nule: 00000000 i 10000000, tj. +0 i -0 su isti broj. Na nivou fizi~ke implementacije logi~kih i aritmeti~kih kola dva na~ina prezentacije nule stvaraju posebne teško}e (karakteristi~no za operaciju kompariranja brojeva)

Jedinični komplement

- Operacija jedini~ni komplement je trivijalna, a obavlja se konverzijom svih jedinica u nule i svih nula u jedinice. Kod pozitivnih brojeva MSB je 0, a kod negativnih 1.

- Posmatrajmo ponovo $(+12)_{10}$ i $(-12)_{10}$ u 8-bitnom formatu koriste}i reprezentaciju jedini~ni komplement.

$$(+12)_{10} = (00001100)_2$$

$$(-12)_{10} = (11110011)_2$$

- Sada ponovo postoje dve reprezentacije nule i to +0 i -0 , koje su 00000000 i 11111111, respektivno.

- Jedini~ni komplement se retko koristi, prvenstveno zbog teško}a koje se javljaju kod izvodjenja operacija uporedjivanja brojeva, a posledice su dvostruke prezentacije nule.

Dvojični komplement

- Formira se na sličan način kao i jedinični komplement: komplementiraju se svi bitovi broja, zatim se dodaje 1, i ako se kao rezultat sabiranja javi prenos na mesto MSB-a on se zanemaruje. Kada je MSB broj u dvojičnom komplementu 0 tada je broj pozitivan, a u slušaju da je 1 broj je negativan. Prezentacija 0 je jedinstvena. Naglasimo da se bit prenosa (*carry-out*) na mestu najveće težine kod operacije sabiranja u dvojičnom komplementu zanemaruje sa izuzetkom kada se detektuje premašaj (*overflow*).

- Razmotrimo sada ponovo 8-bitni format brojeva $(+12)_{10}$ i $(-12)_{10}$ u reprezentaciji dvojičnog komplementa.

$$(+12)_{10} = (00001100)_2$$

$$\begin{array}{rl} & (+12)_{10} = (00001100)_2 \\ \hline & (-12)_{10} = (11110011)_2 \quad 1\text{-komplement} \\ + & (+1)_{10} = (00000001)_2 \\ \hline & (-12)_{10} = (11110100)_2 \quad 2\text{-komplement} \end{array}$$

Reprezentacija viška

- Kod reprezentacije viška (*excess* ili *baised representation*) broj se tretira kao neoznačeni, ali je "pomeren" za vrednost viška (ofseta) u odnosu na njegovu početnu vrednost. Koncept konverzije se sastoji u tome da se broju dodaje takva vrednost da najnegativniji broj postane nula, a svi ostali bit oblici se povećavaju za taj iznos. Da bi ukazali na proces konverzije razmatraćemo ponovo reprezentaciju i u 8-bitnom formatu, ali sada koristeći višak 128. Broj sa viškom od 128 formira se dodavanjem 128 prvobitnoj vrednosti (prezentaciji u dvojičnom komplementu) a zatim kreiranjem njene neoznačene binarne verzije. U konkretnom slučaju:

$(+12)_{10} = (10001100)_2$ se izračunava kao $(128 + 12 = 140)_{10}$

$(-12)_{10} = (01110100)_2$ se izračunava kao $(128 + (-12) = 116)_{10}$

Logički tipovi

- Logi~ki tipovi su najjednostavniji tipovi **nenumerisanih** podataka koji imaju samo dve vrednosti.

- Kod najve}eg broja HLL-ova definišu se kao:

```
type Boolean = (False,True);
```

Prezentacija podataka

III poglavlje

- Kodovi: ASCII, EBCDIC, *Unicode* -

ASCII kôd

ASCII is a 7-bit code, commonly stored in 8-bit bytes.

"A" is at 41_{16}. To convert upper case letters to lower case letters, add 20_{16}. Thus "a" is at $41_{16} + 20_{16} = 61_{16}$.

The character "5" at position 35_{16} is different than the number 5. To convert character-numbers into number-numbers, subtract 30_{16}: $35_{16} - 30_{16} = 5$.

00	NUL	10	DLE	20	SP	30	0	40	@	50	P	60	`	70	p
01	SOH	11	DC1	21	!	31	1	41	A	51	Q	61	a	71	q
02	STX	12	DC2	22	"	32	2	42	B	52	R	62	b	72	r
03	ETX	13	DC3	23	#	33	3	43	C	53	S	63	c	73	s
04	EOT	14	DC4	24	$	34	4	44	D	54	T	64	d	74	t
05	ENQ	15	NAK	25	%	35	5	45	E	55	U	65	e	75	u
06	ACK	16	SYN	26	&	36	6	46	F	56	V	66	f	76	v
07	BEL	17	ETB	27	'	37	7	47	G	57	W	67	g	77	w
08	BS	18	CAN	28	(	38	8	48	H	58	X	68	h	78	x
09	HT	19	EM	29	)	39	9	49	I	59	Y	69	i	79	y
0A	LF	1A	SUB	2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
0B	VT	1B	ESC	2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
0C	FF	1C	FS	2C	´	3C	<	4C	L	5C	\	6C	l	7C	|
0D	CR	1D	GS	2D	-	3D	=	4D	M	5D	]	6D	m	7D	}
0E	SO	1E	RS	2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
0F	SI	1F	US	2F	/	3F	?	4F	O	5F	_	6F	o	7F	DEL

NUL	Null	FF	Form feed	CAN	Cancel
SOH	Start of heading	CR	Carriage return	EM	End of medium
STX	Start of text	SO	Shift out	SUB	Substitute
ETX	End of text	SI	Shift in	ESC	Escape
EOT	End of transmission	DLE	Data link escape	FS	File separator
ENQ	Enquiry	DC1	Device control 1	GS	Group separator
ACK	Acknowledge	DC2	Device control 2	RS	Record separator
BEL	Bell	DC3	Device control 3	US	Unit separator
BS	Backspace	DC4	Device control 4	SP	Space
HT	Horizontal tab	NAK	Negative acknowledge	DEL	Delete
LF	Line feed	SYN	Synchronous idle		
VT	Vertical tab	ETB	End of transmission block		

EBCDIC kôd

EBCDIC Character Code

- **EBCDIC is an 8-bit code.**

00	NUL	20	DS	40	SP	60	–	80		A0		C0	{	E0	\
01	SOH	21	SOS	41		61	/	81	a	A1	~	C1	A	E1	
02	STX	22	FS	42		62		82	b	A2	s	C2	B	E2	S
03	ETX	23		43		63		83	c	A3	t	C3	C	E3	T
04	PF	24	BYP	44		64		84	d	A4	u	C4	D	E4	U
05	HT	25	LF	45		65		85	e	A5	v	C5	E	E5	V
06	LC	26	ETB	46		66		86	f	A6	w	C6	F	E6	W
07	DEL	27	ESC	47		67		87	g	A7	x	C7	G	E7	X
08		28		48		68		88	h	A8	y	C8	H	E8	Y
09		29		49		69		89	i	A9	z	C9	I	E9	Z
0A	SMM	2A	SM	4A	¢	6A	‘	8A		AA		CA		EA	
0B	VT	2B	CU2	4B		6B	,	8B		AB		CB		EB	
0C	FF	2C		4C	<	6C	%	8C		AC		CC		EC	
0D	CR	2D	ENQ	4D	(	6D	_	8D		AD		CD		ED	
0E	SO	2E	ACK	4E	+	6E	>	8E		AE		CE		EE	
0F	SI	2F	BEL	4F	|	6F	?	8F		AF		CF		EF	
10	DLE	30		50	&	70		90		B0		D0	}	F0	0
11	DC1	31		51		71		91	j	B1		D1	J	F1	1
12	DC2	32	SYN	52		72		92	k	B2		D2	K	F2	2
13	TM	33		53		73		93	l	B3		D3	L	F3	3
14	RES	34	PN	54		74		94	m	B4		D4	M	F4	4
15	NL	35	RS	55		75		95	n	B5		D5	N	F5	5
16	BS	36	UC	56		76		96	o	B6		D6	O	F6	6
17	IL	37	EOT	57		77		97	p	B7		D7	P	F7	7
18	CAN	38		58		78		98	q	B8		D8	Q	F8	8
19	EM	39		59		79		99	r	B9		D9	R	F9	9
1A	CC	3A		5A	!	7A	:	9A		BA		DA		FA	|
1B	CU1	3B	CU3	5B	$	7B	#	9B		BB		DB		FB	
1C	IFS	3C	DC4	5C	·	7C	@	9C		BC		DC		FC	
1D	IGS	3D	NAK	5D	)	7D	'	9D		BD		DD		FD	
1E	IRS	3E		5E	;	7E	=	9E		BE		DE		FE	
1F	IUS	3F	SUB	5F	¬	7F	"	9F		BF		DF		FF	

STX	Start of text	RS	Reader Stop	DC
DLE	Data Link Escape	PF	Punch Off	DC
BS	Backspace	DS	Digit Select	DC
ACK	Acknowledge	PN	Punch On	CU
SOH	Start of Heading	SM	Set Mode	CU
ENQ	Enquiry	LC	Lower Case	CU
ESC	Escape	CC	Cursor Control	SY
BYP	Bypass	CR	Carriage Return	IFS
CAN	Cancel	EM	End of Medium	EC
RES	Restore	FF	Form Feed	ET
SI	Shift In	TM	Tape Mark	NA
SO	Shift Out	UC	Upper Case	SM
DEL	Delete	FS	Field Separator	SC
SUB	Substitute	HT	Horizontal Tab	IG
NL	New Line	VT	Vertical Tab	IR
LF	Line Feed	UC	Upper Case	IU

Unicode karakter kôd

Unicode is a 16-bit kôd.

0000	NUL	0020	SP	0040	@	0060	`	0080	Ctrl	00A0	NBS	00C0	À	00E0	à
0001	SOH	0021	!	0041	A	0061	a	0081	Ctrl	00A1	¡	00C1	Á	00E1	á
0002	STX	0022	"	0042	B	0062	b	0082	Ctrl	00A2	¢	00C2	Â	00E2	â
0003	ETX	0023	#	0043	C	0063	c	0083	Ctrl	00A3	£	00C3	Ã	00E3	ã
0004	EOT	0024	$	0044	D	0064	d	0084	Ctrl	00A4	¤	00C4	Ä	00E4	ä
0005	ENQ	0025	%	0045	E	0065	e	0085	Ctrl	00A5	¥	00C5	Å	00E5	å
0006	ACK	0026	&	0046	F	0066	f	0086	Ctrl	00A6	¦	00C6	Æ	00E6	æ
0007	BEL	0027	'	0047	G	0067	g	0087	Ctrl	00A7	§	00C7	Ç	00E7	ç
0008	BS	0028	(	0048	H	0068	h	0088	Ctrl	00A8	¨	00C8	È	00E8	è
0009	HT	0029	)	0049	I	0069	i	0089	Ctrl	00A9	©	00C9	É	00E9	é
000A	LF	002A	*	004A	J	006A	j	008A	Ctrl	00AA	ª	00CA	Ê	00EA	ê
000B	VT	002B	+	004B	K	006B	k	008B	Ctrl	00AB	«	00CB	Ë	00EB	ë
000C	FF	002C	´	004C	L	006C	l	008C	Ctrl	00AC	¬	00CC	Ì	00EC	ì
000D	CR	002D	-	004D	M	006D	m	008D	Ctrl	00AD	–	00CD	Í	00ED	í
000E	SO	002E	.	004E	N	006E	n	008E	Ctrl	00AE	®	00CE	Î	00EE	î
000F	SI	002F	/	004F	O	006F	o	008F	Ctrl	00AF	¯	00CF	Ï	00EF	ï
0010	DLE	0030	0	0050	P	0070	p	0090	Ctrl	00B0	°	00D0	Ð	00F0	¶
0011	DC1	0031	1	0051	Q	0071	q	0091	Ctrl	00B1	±	00D1	Ñ	00F1	ñ
0012	DC2	0032	2	0052	R	0072	r	0092	Ctrl	00B2	²	00D2	Ò	00F2	ò
0013	DC3	0033	3	0053	S	0073	s	0093	Ctrl	00B3	³	00D3	Ó	00F3	ó
0014	DC4	0034	4	0054	T	0074	t	0094	Ctrl	00B4	´	00D4	Ô	00F4	ô
0015	NAK	0035	5	0055	U	0075	u	0095	Ctrl	00B5	µ	00D5	Õ	00F5	õ
0016	SYN	0036	6	0056	V	0076	v	0096	Ctrl	00B6	¶	00D6	Ö	00F6	ö
0017	ETB	0037	7	0057	W	0077	w	0097	Ctrl	00B7	·	00D7	×	00F7	÷
0018	CAN	0038	8	0058	X	0078	x	0098	Ctrl	00B8	¸	00D8	Ø	00F8	ø
0019	EM	0039	9	0059	Y	0079	y	0099	Ctrl	00B9	¹	00D9	Ù	00F9	ù
001A	SUB	003A	:	005A	Z	007A	z	009A	Ctrl	00BA	º	00DA	Ú	00FA	ú
001B	ESC	003B	;	005B	[	007B	{	009B	Ctrl	00BB	»	00DB	Û	00FB	û
001C	FS	003C	<	005C	\	007C	|	009C	Ctrl	00BC	1/4	00DC	Ü	00FC	ü
001D	GS	003D	=	005D	]	007D	}	009D	Ctrl	00BD	1/2	00DD	Ý	00FD	Þ
001E	RS	003E	>	005E	^	007E	~	009E	Ctrl	00BE	3/4	00DE	ý	00FE	þ
001F	US	003F	?	005F	_	007F	DEL	009F	Ctrl	00BF	¿	00DF	ß	00FF	ÿ

NUL	Null	SOH	Start of heading	CAN	Cancel	SP	Space
STX	Start of text	EOT	End of transmission	EM	End of medium	DEL	Delete
ETX	End of text	DC1	Device control 1	SUB	Substitute	Ctrl	Control
ENQ	Enquiry	DC2	Device control 2	ESC	Escape	FF	Form feed
ACK	Acknowledge	DC3	Device control 3	FS	File separator	CR	Carriage return
BEL	Bell	DC4	Device control 4	GS	Group separator	SO	Shift out
BS	Backspace	NAK	Negative acknowledge	RS	Record separator	SI	Shift in
HT	Horizontal tab	NBS	Non-breaking space	US	Unit separator	DLE	Data link escape
LF	Line feed	ETB	End of transmission block	SYN	Synchronous idle	VT	Vertical tab

Prezentacija podataka

III poglavlje

- Fiksni i pokretni zarez -

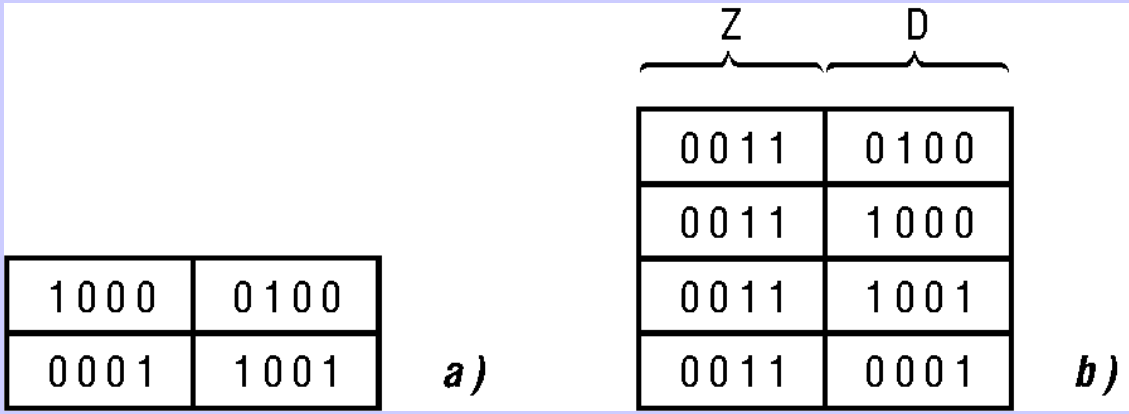
Brojevi u fiksnom zarezu

- Reprezentaciju brojeva u fiksnom zarezu karakteriše:
  - **opseg** (*range*) brojeva koji se mo`e predstaviti kao razlika izmedju najve}eg i najmanjeg broja;
  - **preciznost** (*precision*) - odnosi se na rastojanje izmedju dva susedna broja na brojnoj osi;
  - **greška** (*error*) odgovara 1/2 razlike izmedju dva susedna broja.
- Brojevi u fiksnom zarezu širu primenu nalaze u knjigovodstvu i bankarstvu

Binarno kodirani decimalni brojevi - BCD

- Cifra brojne osnove 10 zauzima ~etiri bit pozicije i naziva se **binarno kodirana decimalna cifra** (*binary coded decimal digit - BCD digit*).

- Postoje slede}a dva formata pakovanja BCD brojeva:
  - **pakovani BCD format** - svaki bajt se deli na dva ~etvorobitna polja, od kojih svako sadr`i po jednu BCD cifru .
  - **nepakovani BCD format** - svaki bajt se deli na dva ~etvorobitna polja, zona (Z) i polje cifre (D), i specificira jednu cifru.



pakovani BCD ASCII kodirani nepakovani BCD

Brojevi u pokretnom zarezu

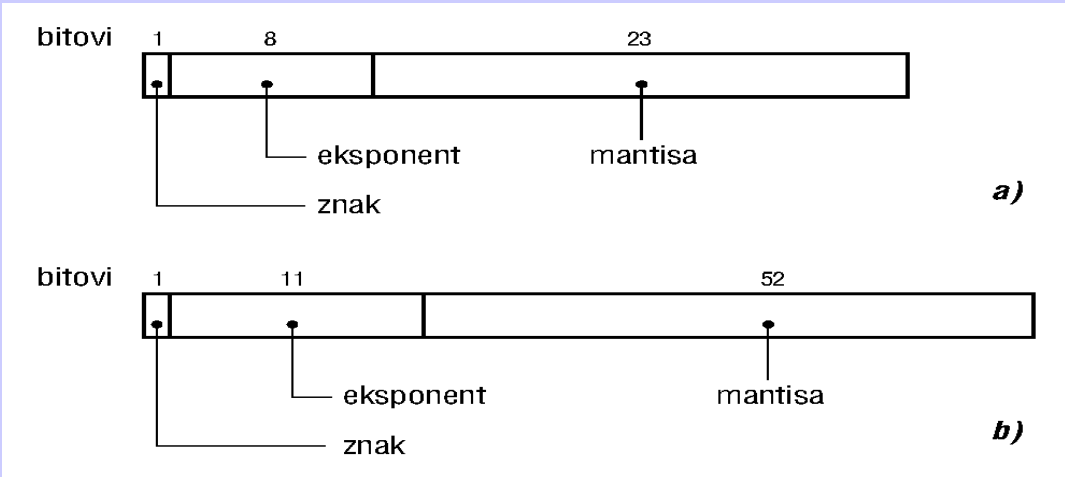
- Vrednost *V* se mo`e izraziti kao

$$V = (-1)^S * F * R^E$$

gde je: *S* - znak, *F* - frakcija ili mantisa, *R* - brojna osnova, a *E* - eksponent.

- FP broj se mo`e smestiti u binarnoj re~i sa tri polja: (a) znak *S*, plus ili minus, (b) mantisa *F*, i (c) eksponent *E*.

- Postoje dva formata kod prezentacije *FP* brojeva:
  - obična preciznost →
  - dupla preciznost →

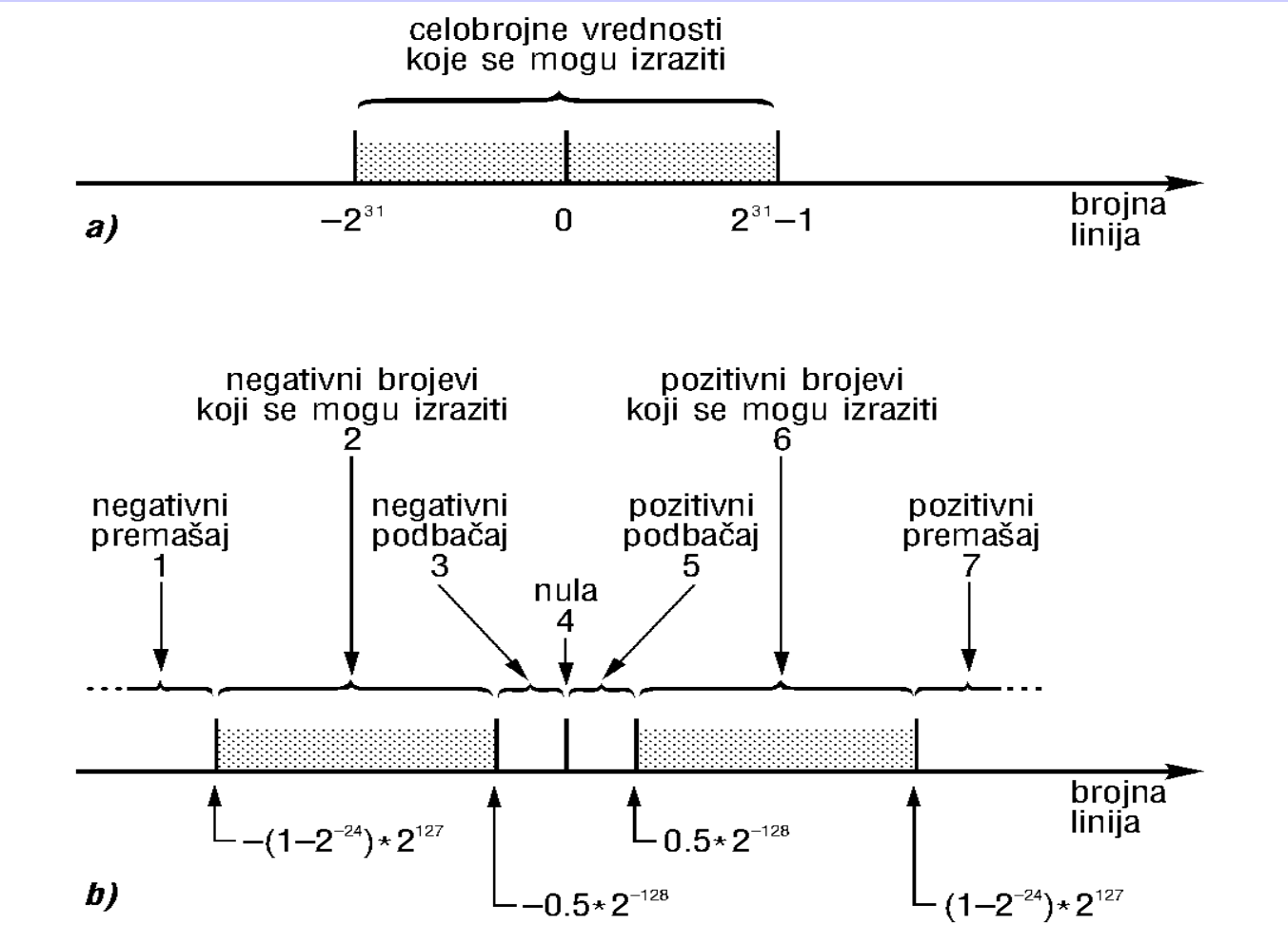


ANSI/IEEE 754 –*FP* brojevi

	obična preciznost	dupla preciznost
znak	1 bit	1 bit
eksponent	8 bitova	11 bita
mantisa	23 bita	52 bita
modifikacija za eksponent	127	1023
minimalna vrednost	$1.175 * 10^{-38}$	$2.225 * 10^{-308}$
maksimalna vrednost	$3.403 * 10^{+38}$	$1.798 * 10^{+308}$
preciznost-binarna	24 bita	53 bita
preciznost-decimalna	6 dec. cifara	15 dec. cifara

Izračunavanje brojeva kod 32-bitnog formata

- dvojični komplement
- *FP* broj



Prezentacija podataka

III poglavlje

- Struktuirani tipovi podataka -

Struktuirani tipovi podataka

Jednodimezionaln nizovi (vektori)

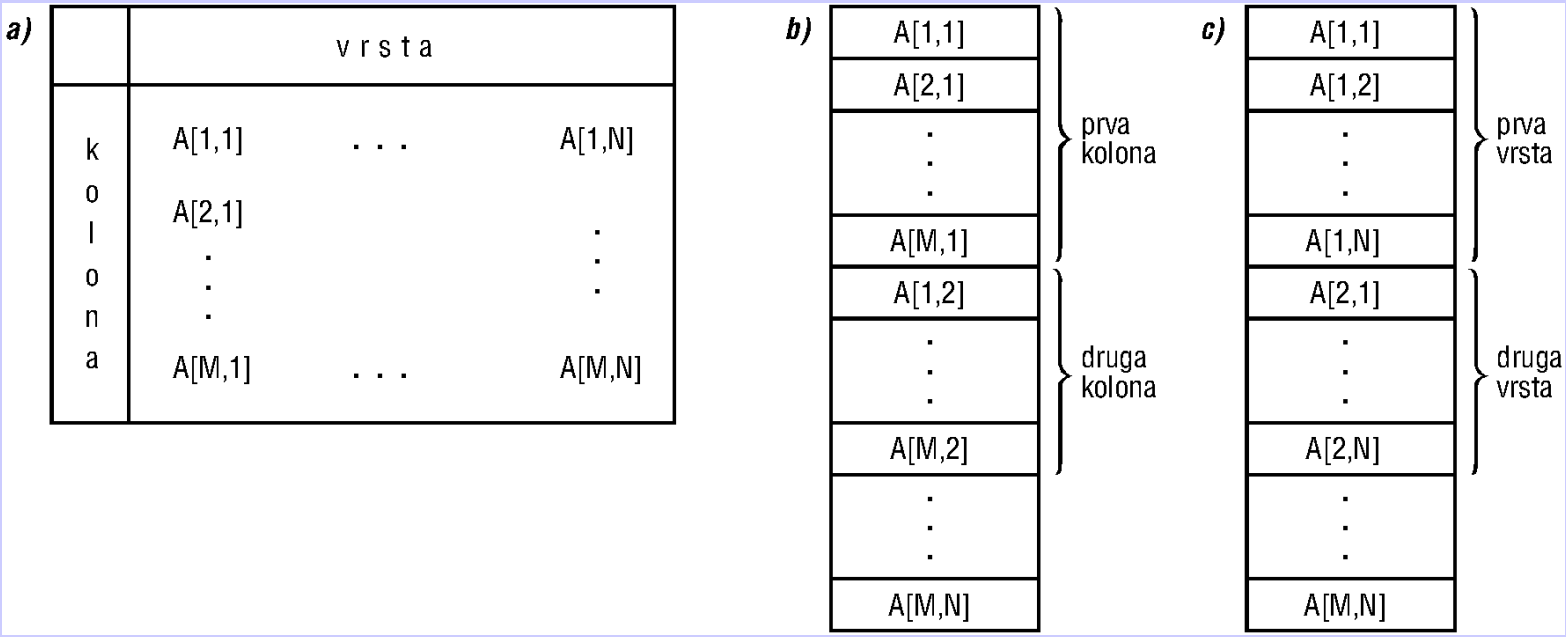
- Jednodimenzionalne nizove nazivamo **vektorima**.
- Vektor je organizovan kao niz elemenata jedno-indeksnih promenljivih istog tipa.
- Svakom od elemenata mogu} je individualni pristup pomo}u indeksa ~ija se vrednost mo`e izra~unati.
- Vrednost elementa vektora mo`e se promeniti naredbom dodeljivanja ili naredbom ulaza.
- Broj elemenata vektora je unapred poznat i ne mo`e se menjati.
- Indeks vektora ima svoju gornju i donju granicu.

Polja (višedimenzionalni nizovi)

- **Polje** je višedimenzionalni skup elementata istog tipa.
- Svakom elementu polja se mo`e pojedina~no pristupiti koriste}i ve}i broj indeksnih vrednosti.
- Indeksi imaju donje i gornje granice.
- Dvodimenzionalni nizovi se obi~no nazivaju matrice.
- Kompilator preslikava višedimenzionalne strukture podataka u jednodimenzionalno memorijsko polje bajtova
- Preslikavanje se obavlja koriš}enjem *n* bajtova po elementu, gde je vrednost *n* odredjena tipom elementa

Polja (višedimenzionalni nizovi) - produžetak

- Elementi polja mogu se smeštati u memoriju koriste}i šemu:
  - uredjenosti_po_kolonama (*column-major ordering*)
  - uredjenosti_po_vrstama (*row-major ordering*)



Prezentacija podataka

III poglavlje

- Kodiranje podataka slike -

Podaci o slici

- Alfanumeri~ki podaci se danas tradicionalno koriste u ra~unarskoj tehnici
- Savremene ra~unarske tehnologije i *Web* sve ve}i zna~aj pridaju va`nosti slika kao nosiocu informacije.
- Slike se mogu generisati u razli~itim oblicima, bojama, efektima senki, ubacivanjem tekstualnih informacija, itd.
- Slike koje se koriste u okviru ra~unarskih sistema mogu se svrstati u slede}e dve kategorije:
  - bit-mapirane
  - vektorske

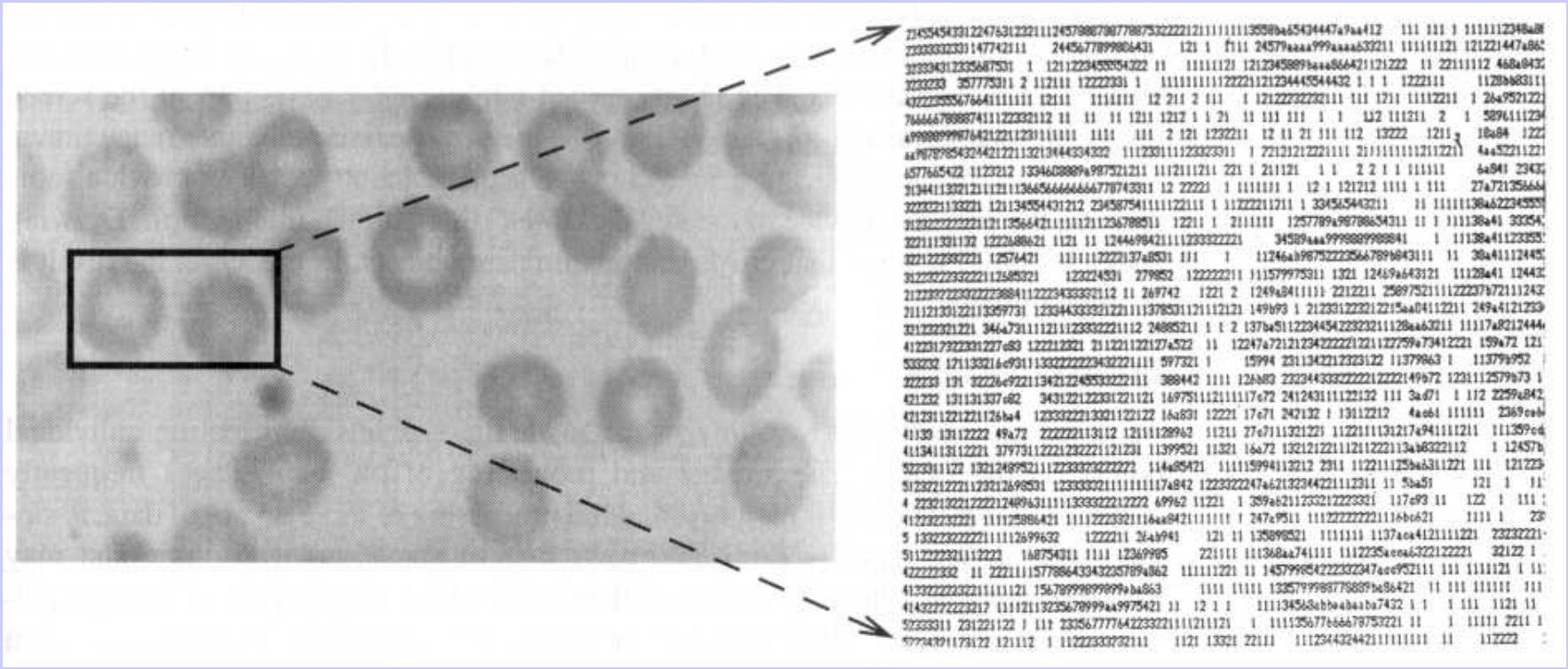
Bit-mapirane slike

- Nazivaju se **raster slike** (*bit map images* ili *raster images*).
- Princip generisanja ovih slika sli~an je kao i onaj koji se koristi u televiziji.
- Sliku ~ini ve}i broj horizontalnih linija,
- U okviru svake linije mogu}e je prikazivati veliki broj osnovnih elemenata slike (*pixel*-a).
- Svaki elemenat slike odgovara jednoj ta~ki na ekranu.
- Ta~ka se mo`e karakterisati ve}im brojem nijansi sivog ili biti obojena.

Primer

- Primer slike kod koje se svakoj ta~ki dodeljuje ~etvorobitni kôd što odgovara jednom od 16 nivoa sivog.
- Heksadecimalna vrednost F odgovara crnoj, a hesadecimalna vrednost 0 beloj boji.
- Svakom *pixel*-u dodeljuje se jedinstvena *x-y* koordinata (adrese vrste i kolone).

Ovakav na~in adresiranja omogu}ava lakše memorisanje



Vektorske slike

- Često se nazivaju i **objektne slike** (*object images*).
- Sliku ~ine grafi~ki oblici kakvi su geometrijski definisane linije i krive.
- Oblici koji se iscrtavaju mogu biti veoma slo`eni i nazivaju se **grafi~ki objekti** (*grphical object*).
- Da bi se generisala slika dovoljno je zapamtiti geometrijsku informaciju o svakom objektu i relativnu poziciju jednog objekta u odnosu na drugi.

Bit mapirane slike – standardni prikaz

- Bit mapirane slike ~ine *pixel*-i koji reprezentuju individualne ta~ke slike.
- Memorisanje i obrada bit mapiranih slika ~esto zahteva veliki iznos memorije, i obradu velike koli~ine podataka.
- Neka slika u boji sadr`i 600 linija sa po 800 *pixel*-a po liniji, i svakom *pixel*-u se dodeljuju po tri bajta (24 bita) za pam}enje intenziteta osnovnih boja (crvena, zelena i plava). Za memorisanje jedne takve slike potrebno je 1, 44 MB.

Bit mapirane slike – alternativni prikaz

- Alternativni metod reprezentacije, kojim se u zna~ajnoj meri redukuju zahtevi za kapacitetom memorisane slike, zasniva se na kodiranju svakog *pixel*-a, a ne upisu njegove stvarne kolor vrednosti.

- Kôd svakog *pixel*-a se prevodi u aktuelnu kolor vrednost koriš}enjem kolor tabele prevodjenja poznate kao **paleta**.

Definicija osnovnih pojmova o prikazu

- Slike koje se prikazuju na displeju ra~unara predstavljaju samo aproksimaciju originalne slike jer imaju konačni broj boja i nivoa osvetljaja
- Originalna slika predstavlja kontinualni opseg intenziteta, a takodje i boja
- Fino}a slike koja se prikazuje zavisi od veli~ine *pixel*-a, broja nivoa osvetljaja, i broja boja.
- Smanjenjem veli~ine *pixel*-a pove}ava se **rezolucija**, ili sposobnost prikazivanja sitnih detalja slike.
- Pove}anjem broja nivoa po *pixel*-u pove}ava se broj nivoa sivog ili boja za opis slike ~ime se pove}ava ukupna vrednost prikaza nivoa sivog i boja u slici.

Kompromis izmedju memrijskog zahteva i vremena procesiranja

- Mora se u~initi kompromis izmedju memorijskih zahteva i vremena procesiranja/prenosa sa jedne strane i kvaliteta prikazane slike sa druge strane.
- Bit mapirane reprezentacije su korisne kada u slici postoji veliki broj detalja, ali su zahtevi za obradom slike relativno mali.
- Tipi~na procesiranja su:
  - memorisanje i prikazivanje
  - izbacivanje i lepljenje delova slike
  - jednostavne tranformacije slike kakve su promena kontrasta, dimenzija, ili boja.

Objektne slike

- Kada sliku ~ine geometrijsko definisani oblici s njom se mo`e lako i efikasno manipulisati.
- Ove slike je mogu}e memorisati u kompaktnoj formi.
- Dijagrami, grafikoni, nacrti u tehnici i dr. su tipi~ni predstavnici objektnih slika.

Objektne slike - produžetak

- Objektne slike komponovane su od ve}eg broja osnovnih elemenata kakve su prave linije, zakrivljene linije poznate kao *Bezier*-ove krive, krugovi i delovi odse~aka kruga, ovali, i dr.
- Svaki od ovih elemenata se mo`e matemati~ki definisati pomo}u manjeg broja parametara.
- Krug se mo`e definisati pomo}u slede}a tri parametra, *X* i *Y*, kao koordinate centra kruga, plus radijus kruga.
- Za definisanje prave linije potrebne su *X* i *Y* koordinate krajnjih ta~aka, ili alternativno, po~etna ta~ka, du`ina linije i smer.

Objektne slike - produžetak

- S obzirom da se objekti definišu matemati~ki, oni se mogu lako premeštati po ekranu, skalirati, i rotirati bez opasnosti da se izgubi identitet o njihovom obliku.
- Zatvoreni objekti se mogu sen~ati ili farbati razli~itim bojama.
- Objektni elementi se mogu kombinovati ili povezivati radi formiranja slo`enijih elemenata.

Prednosti:

- za memorisnje potrebno je znatno manje prostora.
- znatno lakše se može manipulisati, a da pri tome ne dodje do gubitka njihovog identiteta.

Kompresija podataka

- Enormna koli~ina multimedija podataka, posebno ona koja se odnosi na istovremeni prenos slike visoke rezolucije kao i tona visokog kvaliteta, ~esto ~ini nemogu}im ili neprakti~nim da se proces memorisanja, prenosa i manipulisanja podacima realizuje u svojoj normalnoj formi (bez ograni~enja).
- Umesto toga po`eljno je, a u nekim slu~ajevima i neophodno, obaviti kompresiju podataka.
- **Kompresija** je od posebnog interesa kada se simultani prenos signala slike i tona vrši preko Interneta putem modemskih veza.

Kompresija podataka - produžetak

- Algoritmi za kompresiju podataka, se dele u slede}e dve kategorije:
  - **bez-gubitaka** (*lossless*) i
  - **sa-gubicima** (*lossy*).
- Algoritmi za kompresiju bez-gubitaka vrše kompresiju na takav na~in da se primenom inverznog algoritma vrši obnavljanje komprimovanih podataka u prvobitnoj formi.
- Algoritmi za kompresiju podataka sa-gubicima rade pod pretpostavkom da korisnik mo`e da prihvati odredjeni iznos degradacije podataka kako bi se uštedelo na kriti~nim resursima kakvi su oni koji vrše memorisanje i prenos podataka.
- Algoritmi za kompresiju podataka bez-gubitaka jedino prihvatljivo rešenje kada originalne podatke treba sa~uvati, dok se algoritmi sa-gubicima ~esto apliciraju kod multimedia aplikacija.

Prezentacija podataka

III poglavlje

- Sistemsko struktuirani tipovi podataka -

Sistemsko struktuirani tipovi podataka - magacin

- **Magacin** predstavlja memorijski blok u kome se ~uvaju neki podaci. Na **vrh magacina** (*top_of_stack* - TOS) ukazuje **pokaziva~ magacina** (*stack pointer* - SP)

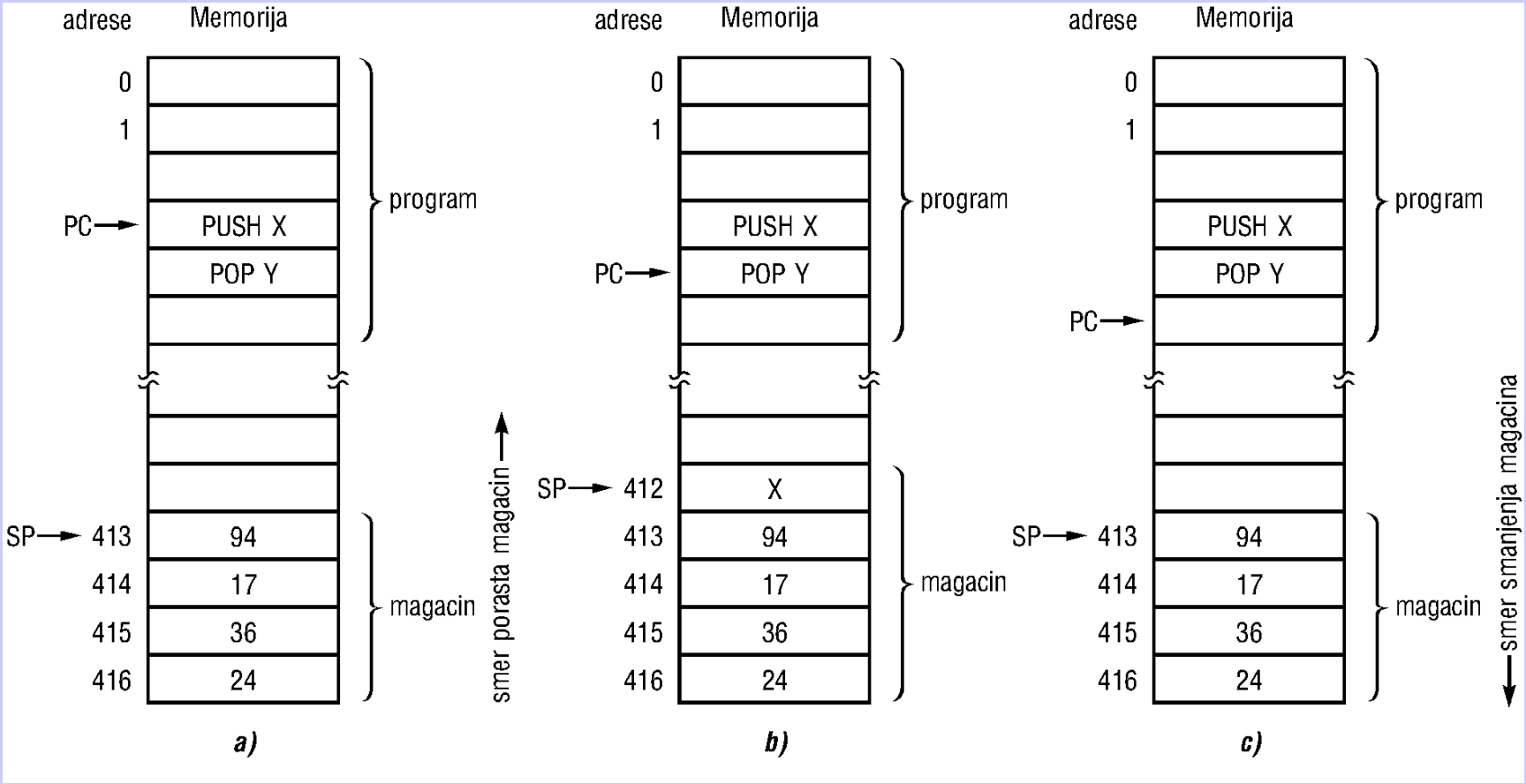
- Kada se obavljaju operacije nad magacinom, SP se inkrementira ili dekrementira, dok svi elementi ostaju na istim memorijskim lokacijama

- Za manipulisanje sa magacinom definisano je nekoliko operacija. Medju njima dve najva`nije su **PUSH** i **POP**

- Operacija PUSH X, prvo dekrementira SP, a nakon toga smešta X u lokaciju na koju ukazuje SP. PUSH pove}ava obim magacina za jednu stavku.

- POP Y smanjuje obim magacina smeštaju}i element sa vrha magacina na lokaciju Y, a zatim inkrementira SP za jednu stavku

Manipulisanje sa magacinom



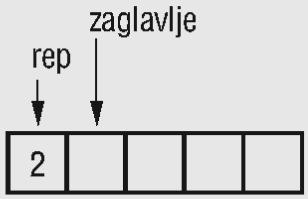
Magacin je memorija tipa LIFO (*Last In First Out*)

Redovi čekanja (queue)

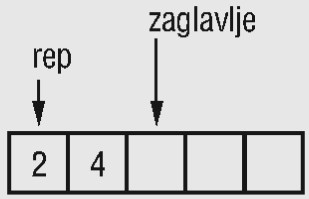
- Po svojoj koncepciji to je **FIFO** (***First In First Out***) bafer. Podatak koji se prvi smešta u red ~ekanja prvi se izbavlja.
- Od strane operativnog sistema redovi ~ekanja se koriste za smeštanje opisa zadataka koga treba izvršiti.
- Nasuprot magacinu, kod reda ~ekanja potrebna su dva pokaziva~a: **zaglavlje** (*head*) i **rep** (*tail*).
- Zaglavlje pokazuje na prvu slobodnu lokaciju koja sledi nakon zadnje unetog podatka, a rep na lokaciju prvog unetog podatka. Kada zaglavlje i rep pokazuju na istu lokaciju red ~ekanja je prazan.

Način unošenja (*enqueue*) i izbavljanja (*dequeue*) podataka u/iz reda čekanja

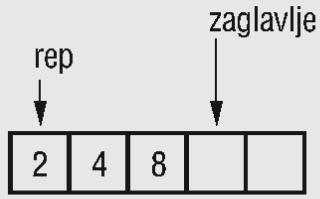
a) red čekanja prazan



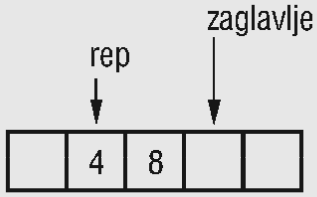
b) u red čekanja se unosi 2



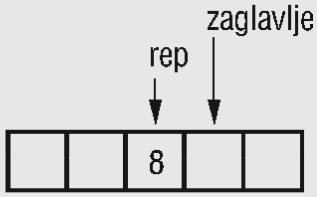
c) u red čekanja se unosi 4



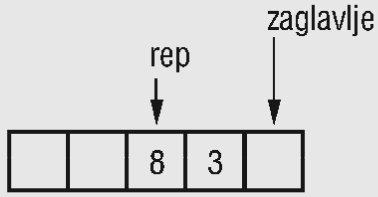
d) u red čekanja se unosi 8



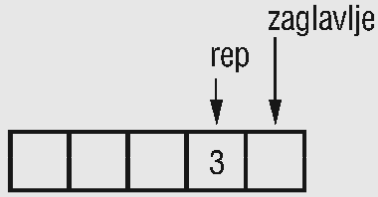
e) iz reda čekanja se izbavlja 2



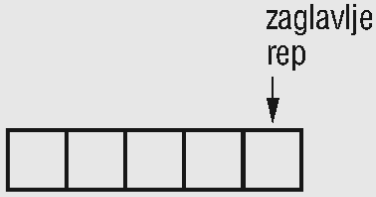
f) iz reda čekanja se izbavlja 4



g) u red čekanja se unosi 3

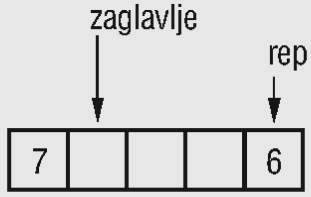


h) iz reda čekanja se izbavlja 8

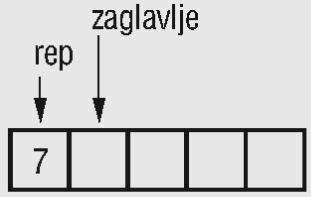


i) iz reda čekanja se izbavlja 3

j) u red čekanja se unosi 6



k) u red čekanja se unosi 7



l) iz reda čekanja se izbavlja 6

Predstavljanje instrukcija i načini adresiranja

IV poglavlje

- predstavljanje instrukcija -

Predstavljanje instrukcija

- Na mašinskom nivou svaka instrukcija se predstavlja sekvencom bitova
- Kodiranjem ovih bitova formira se informacija koja je potrebna radi upravljanja izvršenjem instrukcije
- Ovom informacijom treba da se definiše:
  - tip operacije koja treba da se izvrši
  - implicitna ili eksplicitna specifikacija jednog ili većeg broja operanada nad kojima se izvršava operacija tj. specificiraju se izvorni operandi
  - adresa gde treba da se smesti rezultat obavljene operacije, tj. specificira se odredišni operand, i
  - adresa naredne instrukcije koju treba izvršiti nakon što se obavi tekuća

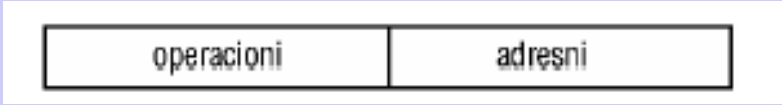
What Must an Instruction Specify?

Data Flow ←

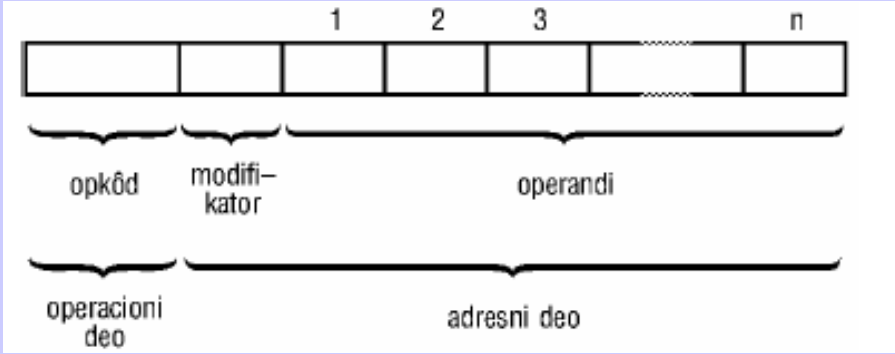
- Which operation to perform: add r0, r1, r3
  - Ans: Op code: add, load, branch, etc.
- Where to find the operand or operands add r0, r1, r3
  - In CPU registers, memory cells, I/O locations, or part of instruction
- Place to store result add r0, r1, r3
  - Again CPU register or memory cell
- Location of next instruction add r0, r1, r3
  br endloop
- The default is usually memory cell pointed to by program counter—PC: the next instruction in sequence
- Sometimes there *is* no operand, or no result, or no next instruction. Can you think of examples?

Predstavljanje instrukcija- (produžetak)

- Instrukciju čine sledeća dva globalna dela:
  - operacioni
  - adresni



- Operacioni deo se zove operacioni kôd, tj. opkôd
- Adresni deo sadrži informaciju o izvornim operandima, odredišnom operandu i adresu naredne instrukcije
- Binarni nizovi kojima se kodiraju pojedine vrste informacija u instrukciji smešataju se u odgovarajuće delove koje zovemo **polja.**
- Izgled (*layout*) instrukcije zove se **format instrukcije**



Adresni deo instrukcije

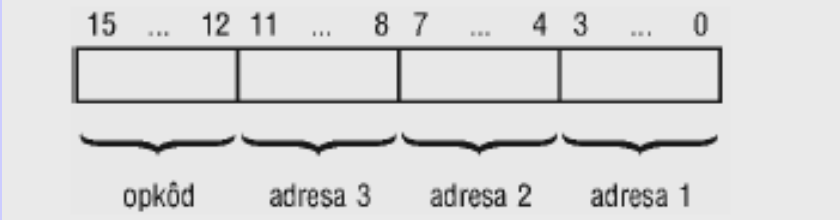
- Sastavni delovi adresnog dela instrukcije su polja operandi i modifikatori
- Modifikator sadrži informaciju kojom se:
  - opisuje način adresiranja
  - ukazuje o dodatnim uslovima koji se odnose na pristup podacima ili na način izvršenja operacije
- Najveći broj CPU-ova se izvodi kao jedno-, dvo-, ili tro-adresna mašina, pri čemu se adresa naredne instrukcije dobija implicitno preko programskog brojača (PC)

Operacioni deo instrukcije

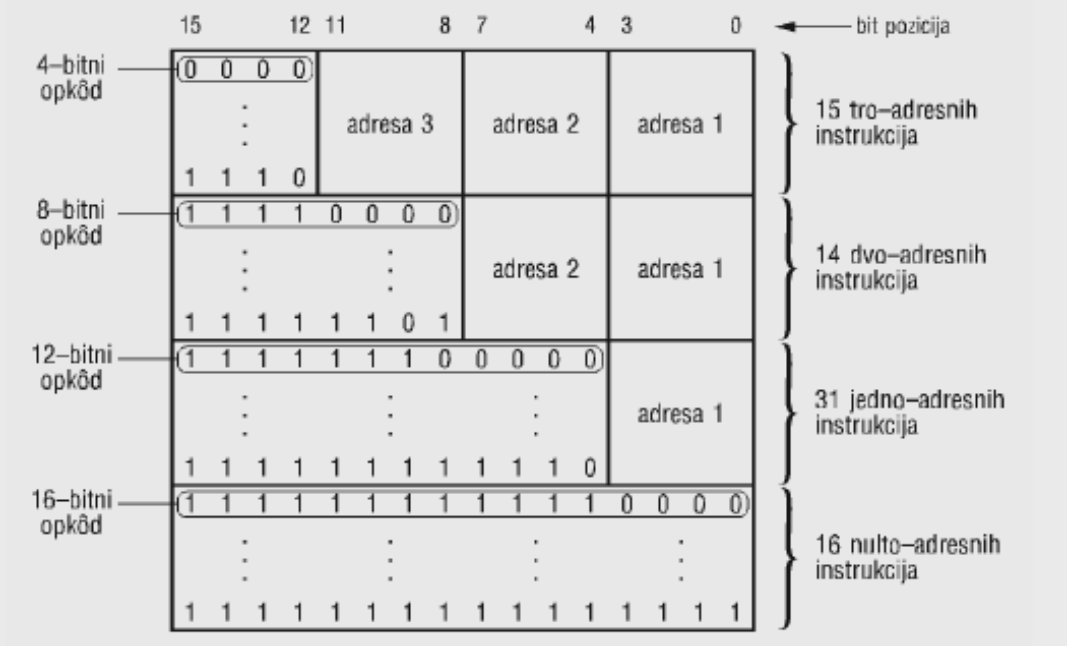
- Neka instrukciju čine (n+k) bitova , od kojih se k bitova dodeljuje opkôdu, a n bitova jedinstvenoj adresi.
- Sa k bitova dobija se 2^k različitih opkôdova.
- Alternativno, istih (n+k) je moguće podeliti tako da se (k-1) bit dodeli opkôdu, a (n+1) bit adresi, što znači da je broj instrukcija prepolovljen, ali je broj memorijskih lokacija udvostručen
- Tehnika kodiranja koja se bazira na proširenju broja bitova opkôda naziva se *kodiranje sa proširenjem*

Primer: Kodiranje instrukcija sa proširenjem

- Mašina koristi fiksni 16-to bitni format i ima 16 registara sa kojima manipulišu sve aritmetičke i logičke instrukcije



- Ako je sa istim brojem bitova potrebno ostvariti 15 tro-adresnih, 14 dvo-adresnih, 31 jedno-adresnih i 16 nulto-adresnih instrukcija kako izgleda kodiranje instrukcija



Diskusije u vezi izbora formata

- Projektovanje formata instrukcija je kompleksan zadatak
- Mora se uzeti u obzir veliki broj faktora kakvi su:
  - obim i organizacija memorije
  - struktura magistrale
  - kompleksnost CPU-a i brzina rada
- Mora da se napravi dobar kompromis izmedju želja i mogućnosti
- Programeri uvek preferiraju ka:
  - većem broju opkôdova
  - većem broju operanada
  - većem broju adresnih načina rada
  - većem adresnom opsegu

Diskusije u vezi izbora formata - produžetak

- Veći broj opkôdova i veći broj adresnih načina rada pruža programeru mogućnost da napiše svoj program kraće

- Sa većim brojem adresnih načina rada lakše je implementirati odredjene funkcije kao što je manipulisanje sa tabelama i višestruko grananje

- Želja je da se manipuliše sa većim adresnim prostorom što neminovno dovodi da se specificira duža adresa

- Sve nabrojane stavke (opkôdovi, operandi, adresni načini rada, i adresni opseg) ukazuju da je potrebno koristiti veći broj bitova po instrukciji što znači da će instrukcija biti duža

Diskusije u vezi izbora formata - produžetak

- Duže instrukcije zauzimaju veći memorijski prostor i za duže vreme se pribavljaju iz memorije, a to znači da ih CPU izvršava sporije

- Za memoriju kažemo da predstavlja "usko grlo" sistema jer u principu CPU može da izvršava instrukcije brže nego što ih može pribaviti

- Rešenje se sastoji u korišćenju :
  - keš memorije
  - kraćih instrukcija

- Za zadatu dužinu instrukcije neophodno je napraviti kompromis izmedju broja opkôdova i adresnih mogućnosti

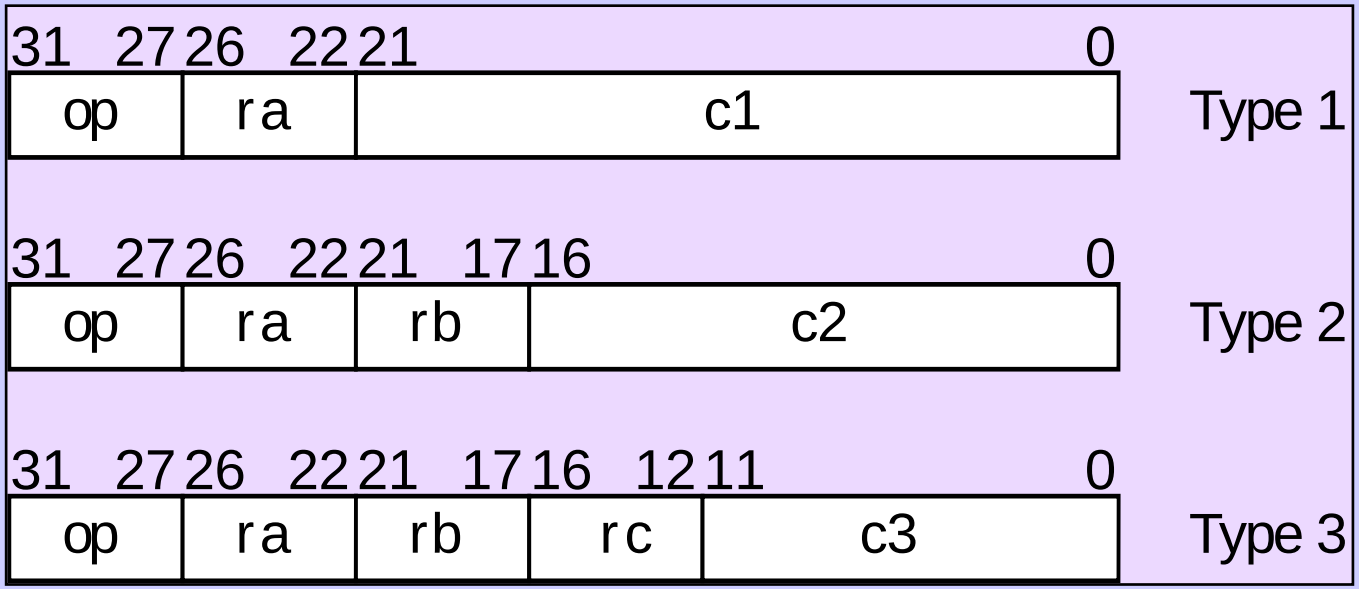
Diskusije u vezi izbora formata - produžetak

- Faktori koji imaju direktni uticaj na dužinu polja za adresiranje su:
  - broj adresnih načina rada
  - broj operanada
  - specifikacija internih registara u odnosu na memoriju
  - adresni opseg
  - adresna granularnost

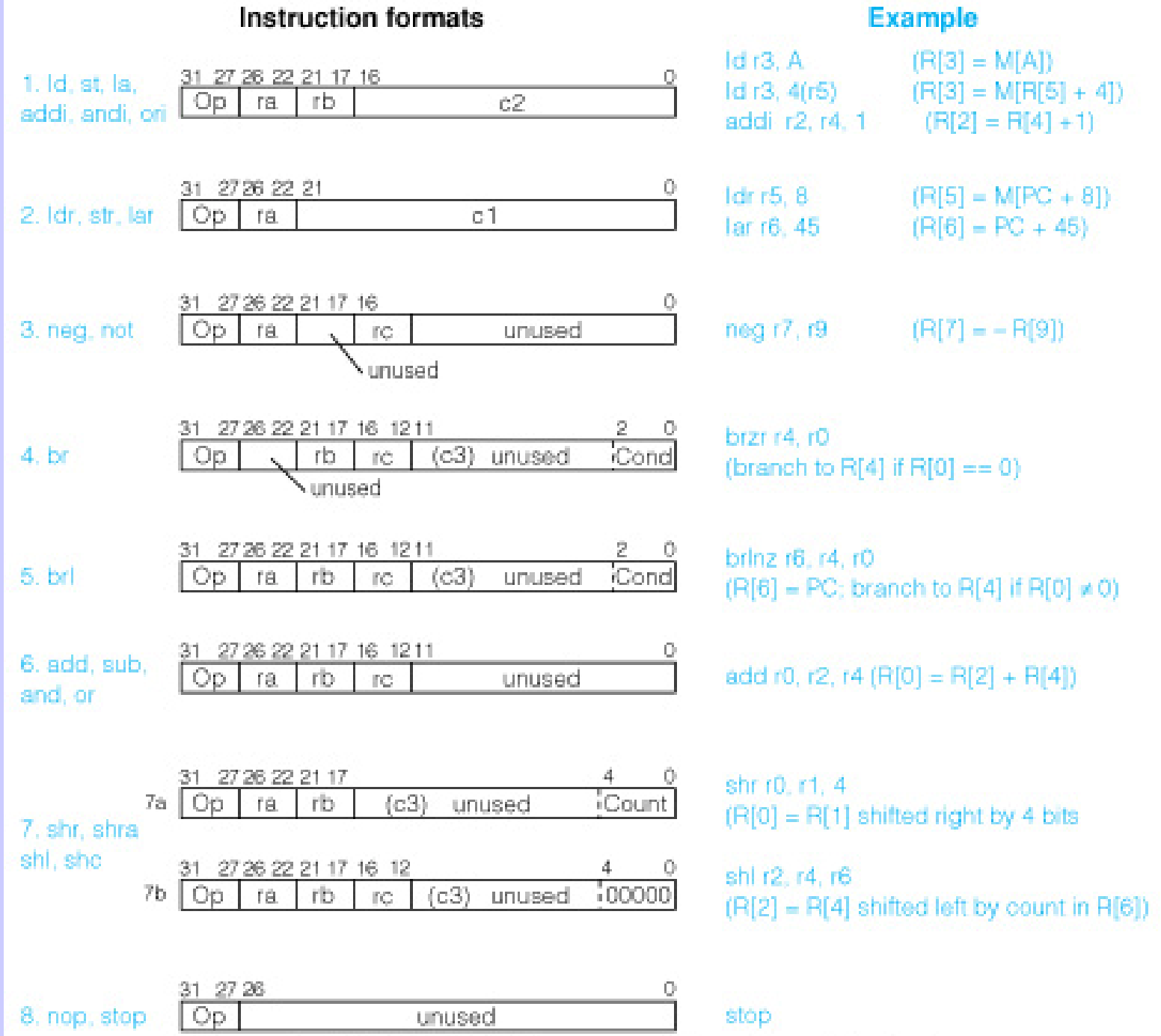
Simple RISC Computer (SRC)

- Basic Instruction Formats -

- There are three basic instruction format types
- The number of register specifier fields and length of the constant field vary
- Other formats result from unused fields or parts

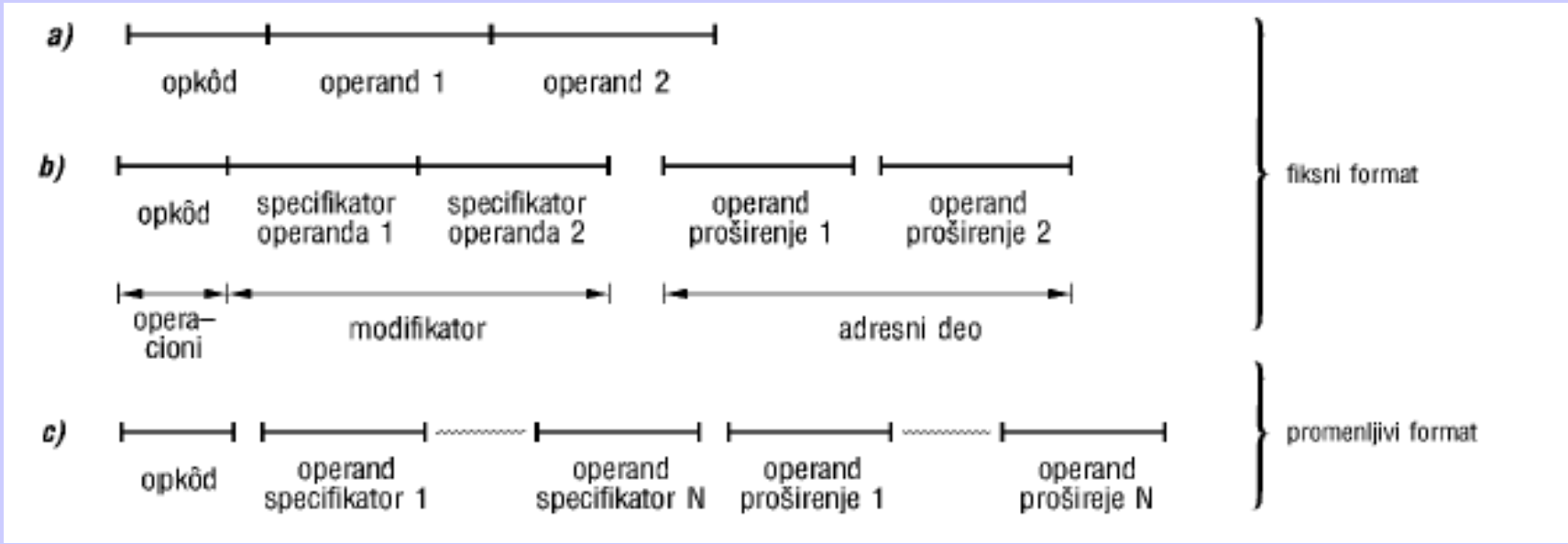


Programmer's Model of SRC



Fiksni i promenljivi format instrukcije

- Razlikujemo arhitekture koje koriste sledeće formate instrukcija :
  - fiksni
  - promenljivi



Broj adresa

- Jedan od standardnih načina za opis arhitekture uzima u obzir koliko je eksplicitnih adresa operanada uključeno u instrukcije

- Najveći broj CPU-ova izvodi se kao jedno-, dvo-, ili tro-adresna mašina, pri čemu se adresa naredne instrukcije dobija implicitno (obično preko programskog brojača (PC))

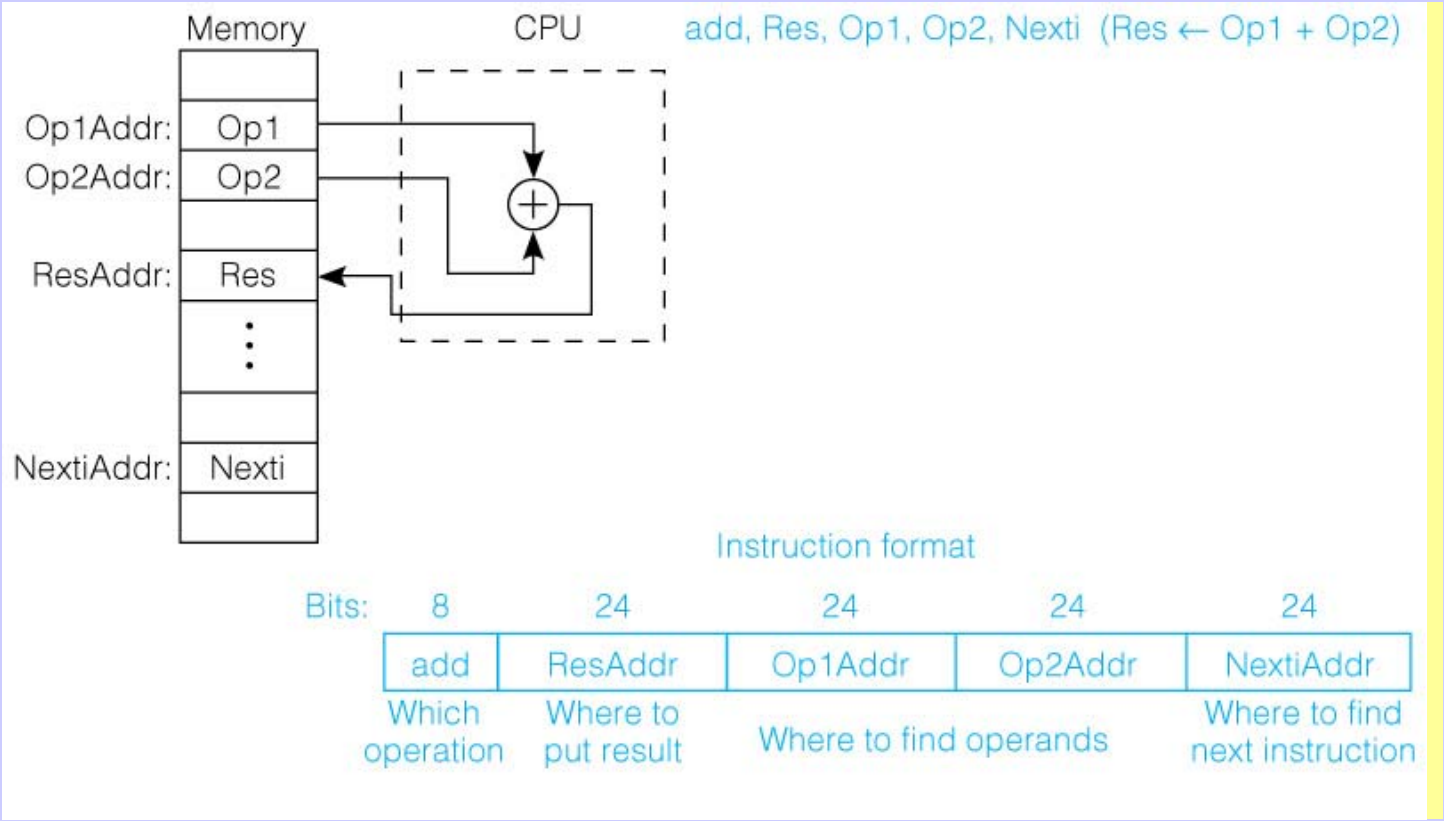
- U praksi četvoro-adresne instrukcije veoma retko srećemo

- Neki računari su projektovani tako da najveći broj instrukcija ne sadrži eksplicitne adrese, pa se zbog toga ove mašine zovu nulto-adresne mašine.

3, 2, 1, & 0 Address Instructions

- The classification is based on arithmetic instructions that have two operands and one result
- The key issue is "how many of these are specified by memory addresses, as opposed to being specified implicitly"
- A 3 address instruction specifies memory addresses for both operands and the result: R ← Op1 op Op2
- A 2 address instruction overwrites one operand in memory with the result: Op2 ← Op1 op Op2
- A 1 address instruction has a register, called the **accumulator register** to hold one operand & the result (no address needed):
  Acc ← Acc op Op1
- A 0 address + uses a CPU register stack to hold both operands and the result: TOS ← TOS op SOS where TOS is Top Of Stack, SOS is Second On Stack)
- The 4-address instruction, hardly ever seen, also allows the address of the next instruction to specified explicitly.

The 4 Address Instruction



- Explicit addresses for operands, result & next instruction
- Example assumes 24-bit addresses
  - Discuss: size of instruction in bytes

Broj adresa – tro-adresna mašina

- Kod tro-adresnih instrukcija specificiraju se lokacije sva tri operanda, dva izvorišna, i jedan odredišni .
- Ove instrukcije zahtevaju relativno dug format.
- Tro-adresni format instrukcija obično srećemo kod 32-bitnih RISC procesora

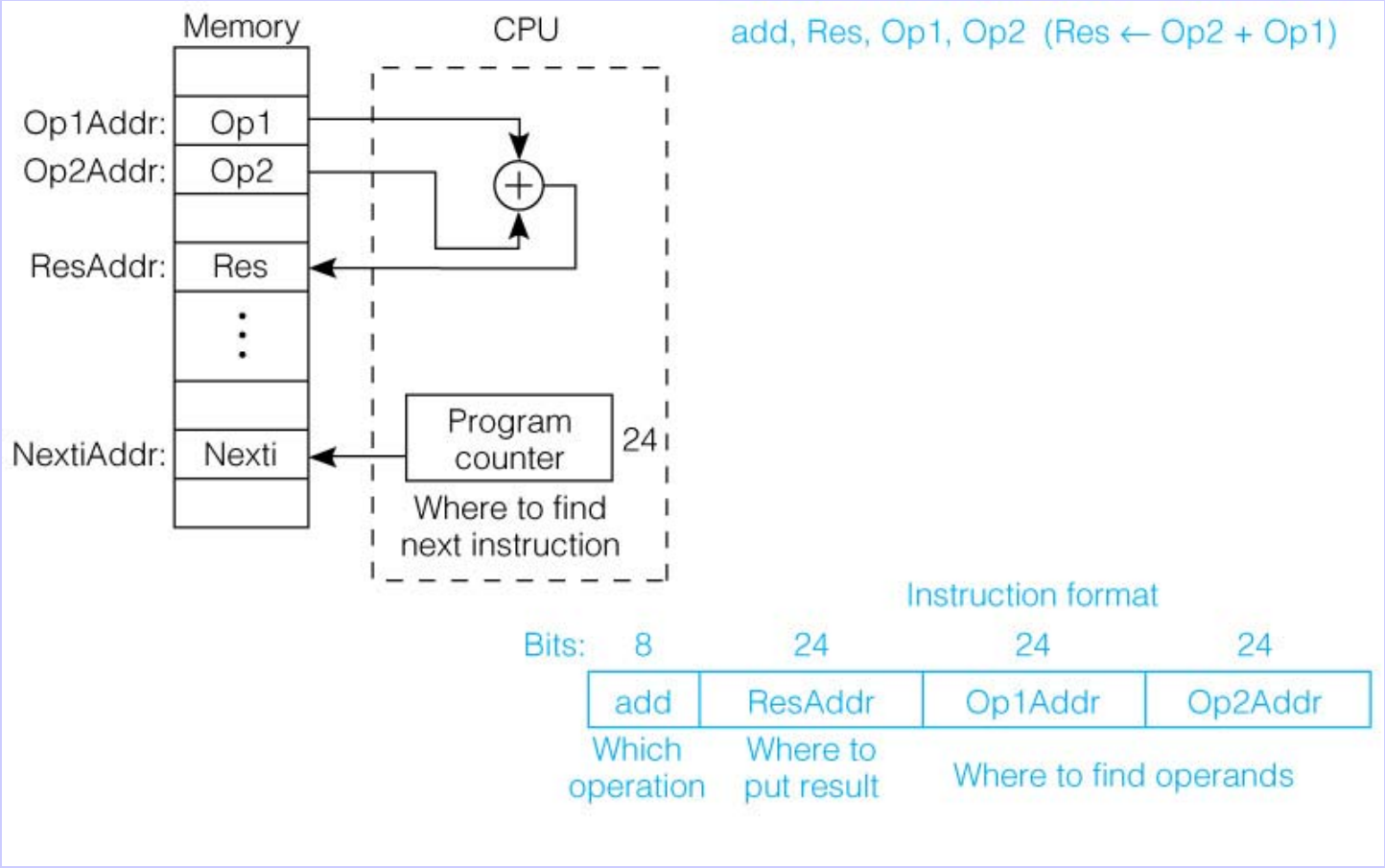
Primer:

- HLL aritmetički iskaz **Y:= (A-B) / (C+D*E)** se izvršava pomoću sledeće sekvence

a) *tro-adresne instrukcije*

instrukcija	komentar
`Sub Y,A,B`	Y←A–B
`Mpy T,D,E`	T←D*E
`Add T,T,C`	T←T+C
`Div Y,Y,T`	Y←Y/T

The 3 Address Instruction



- Address of next instruction kept in a processor state register—the PC (Except for explicit Branches/Jumps)
- Rest of addresses in instruction
  - Discuss: savings in instruction word size

Broj adresa – dvo-adresna mašina

- Kod dvo-adresnih instrukcija specificiraju se dva operanda pri čemu je jedan od izvorišnih operanada istovremeno i odredišni
- Ovaj način adresiranja se veoma često koristi kod 8-, 16- i 32-bitnih mikroprocesora tipa CISC

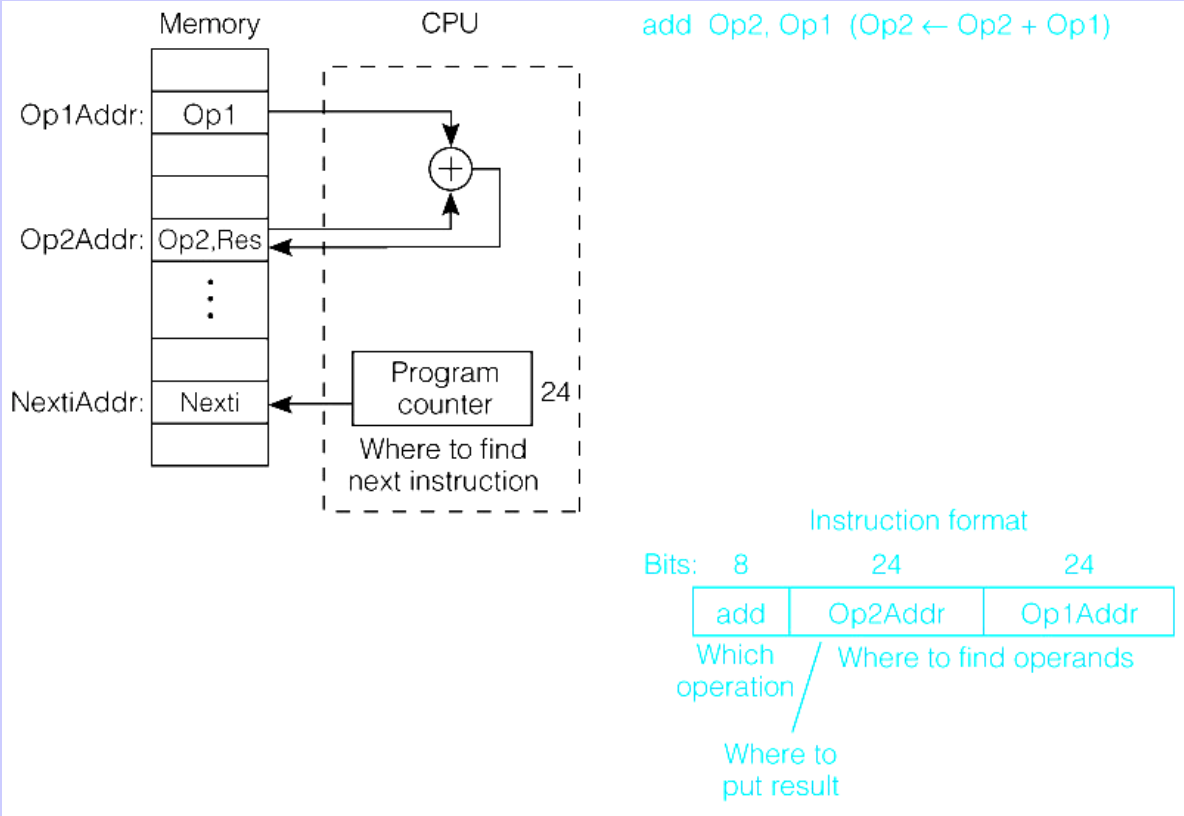
Primer:

- HLL aritmetički iskaz **Y:= (A-B) / (C+D*E)** se izvršava pomoću sledeće sekvence

b) *dvo-adresne instrukcije*

instrukcija	komentar
`Move Y,A`	Y←A
`Sub Y,B`	Y←Y–B
`Move T,D`	T←D
`Mpy T,E`	T←T*E
`Add T,C`	T←T+C
`Div Y,T`	Y←Y/T

The 2 Address Instruction



- Be aware of the difference between address, Op1Addr, and data stored at that address, Op1.
- Result overwrites Operand 2, Op2, with result, Res
- This format needs only 2 addresses in the instruction but there is less choice in placing data

Broj adresa – jedno-adresna mašina

- Kod jedno-adresnih mašina specificira se jedan operand i to izvorišni, drugi izvorišni operand je istovremeno i odredišni i implicitno se podrazumeva
- Ovaj pristup je bio korišćen kod ranijih mašina pri čemu se implicitna adresa odnosila na CPU-ov registar poznat kao akumulator (*ACC*)

Primer:

- HLL aritmetički iskaz **Y:= (A-B) / (C+D*E)** se izvršava pomoću sledeće sekvence

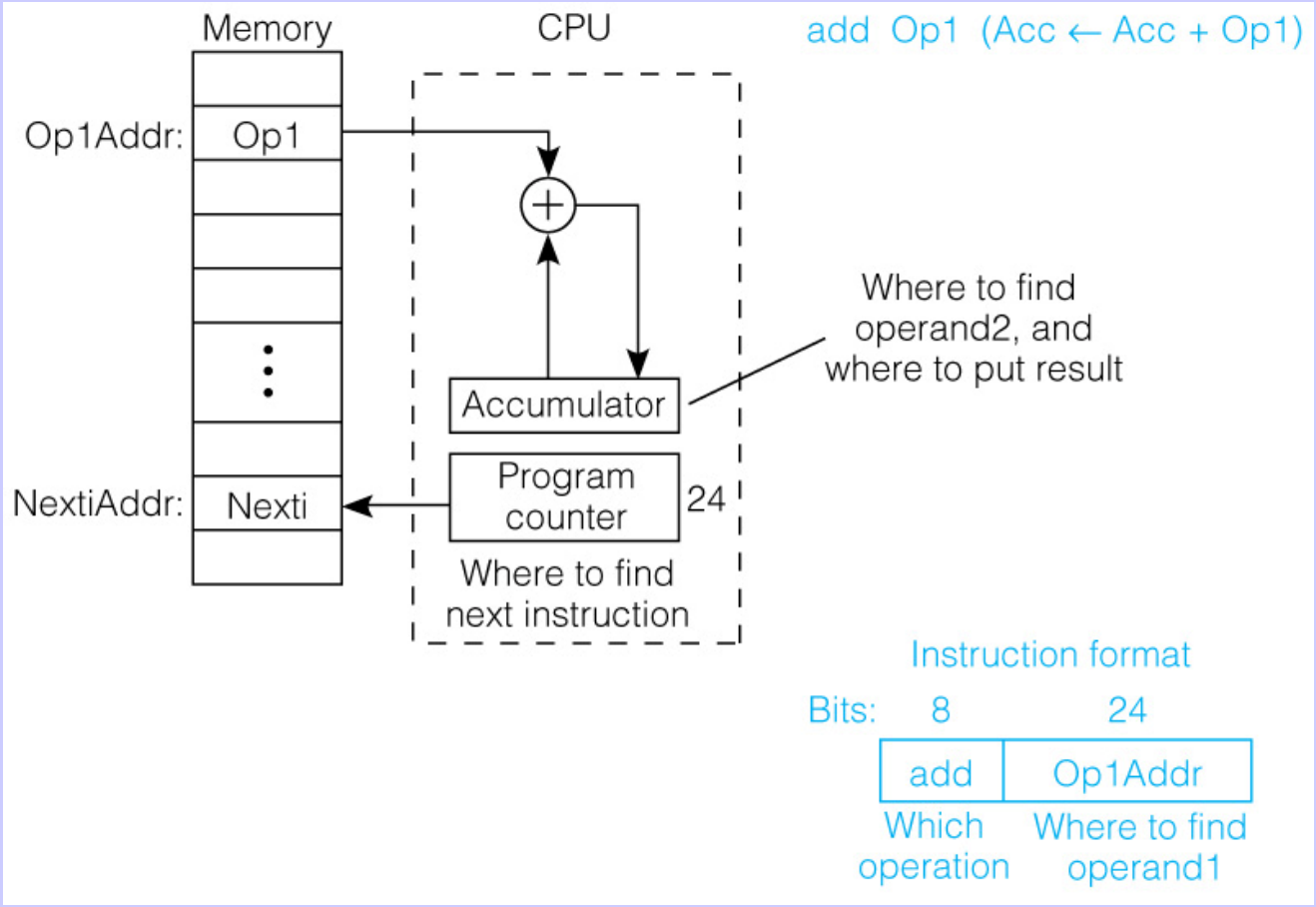
c) *jedno-adresne instrukcije*

instrukcija	komentar
Load D	ACC←D
Mpy E	ACC← ACC*E
Add C	ACC← ACC+C
Store Y	Y←ACC
Load A	ACC←A
Sub B	ACC← ACC–B
Div Y	ACC← ACC/Y
Store Y	Y←ACC

1 Address Instructions

We now need instructions to load and store operands:

LDA OpAddr
STA OpAddr



- Special CPU register, the accumulator, supplies 1 operand and stores result
- One memory address used for other operand

Broj adresa – nulto-adresna mašina

- Kod aritmetičko-logičkih operacija ne specificira se eksplicitno adresa operanda, jer se eksplicitno podrazumeva gde se nalazi operand
- Svi operandi koji se koriste potrebno je da budu na vrhu magacina
- Na vrh magacina ukazuje pokazivač *SP* (*Stack Pointer*)
- Za izbavljanje podatka sa vrha magacina koristi se operacija *Pop*, a za smeštanje podatka na vrh magacina opeacija *Push*, koje su jedine operacije za komuniciranje sa memorijom

Primer:

- HLL aritmetički iskaz

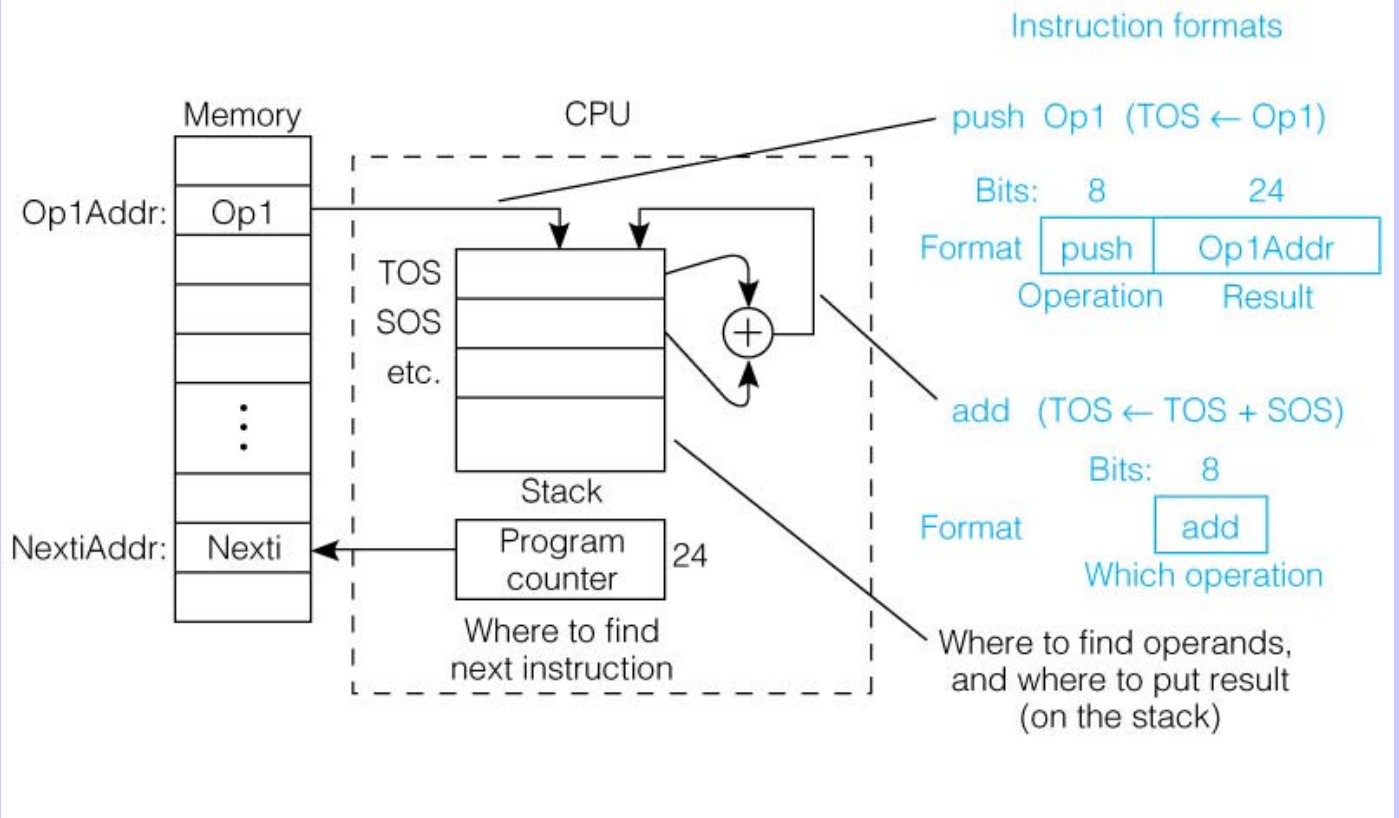
  Y:= A*B+C*C

  se izvršava pomoću sledeće sekvence

instrukcija	komentar
PUSH A	TOS←A
PUSH B	TOS←B
MPY	TOS←A*B
PUSH C	TOS←C
PUSH C	TOS←C
MPY	TOS←C*C
ADD	TOS←A*B+C*C
POP X	X←TOS

- *TOS* označava vrh magacina

The 0 Address Instruction



- Uses a push down stack in CPU
- Arithmetic uses stack for both operands. The result replaces them on the TOS
- Computer must have a 1 address instruction to push and pop operands to and from the stack

Expression evaluation for 3-0 address instructions.

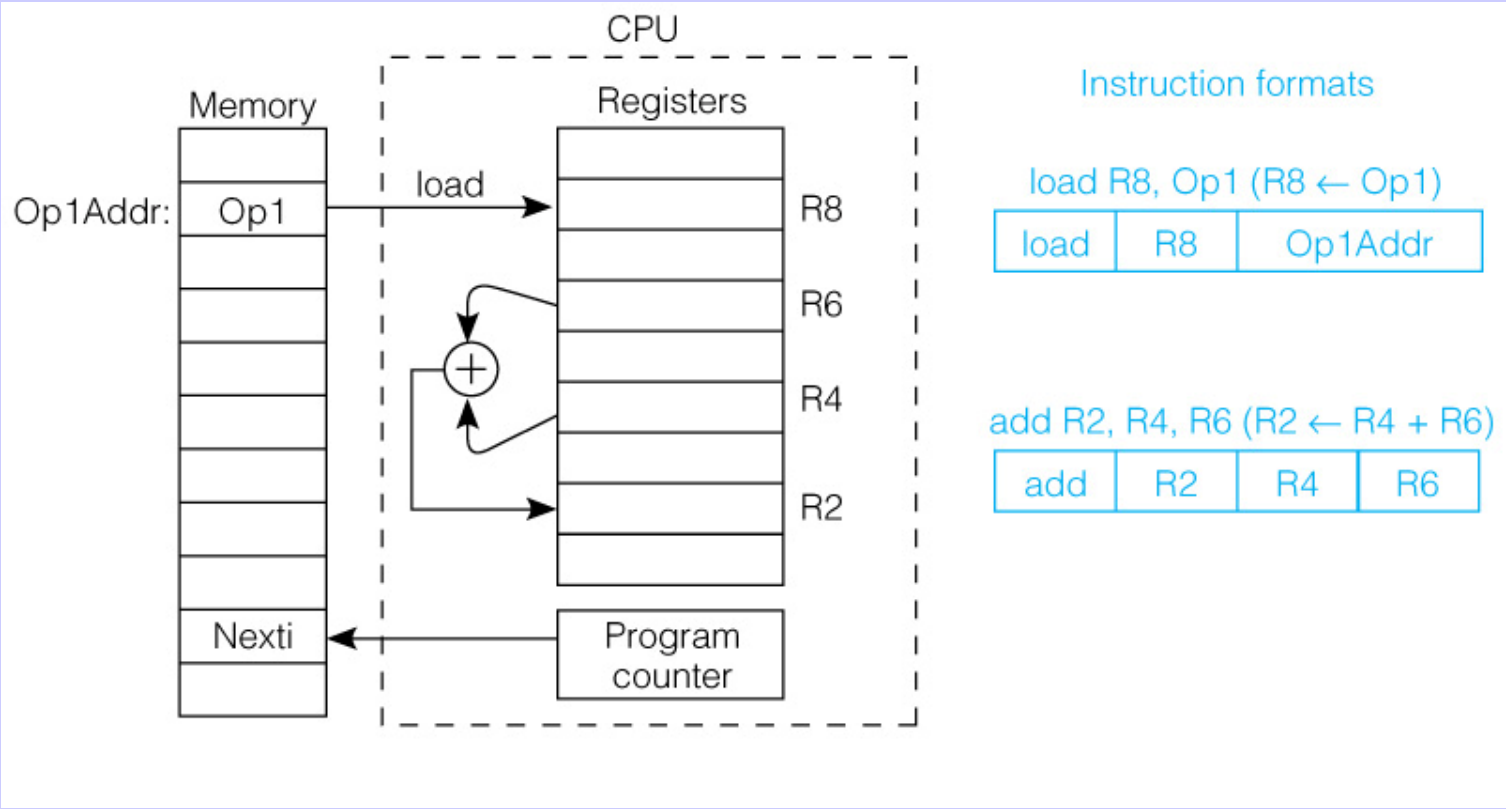
Evaluate a = (b+c)*d-e for 3- 2- 1- and 0-address machines.

3-Address	2-Address	Accumulator	Stack

add a,b,c
mpy a,a,d
sub a,a,e
	load a,b
	add a,c
	mpy a,d
	sub a,e
		lda b
		add c
		mpy d
		sub e
		sta a
			push b
			push c
			add
			push d
			mpy
			push e
			sub
			pop a

- # of instructions & # of addresses both vary
- Discuss as examples: size of code in each case

General Register Machines

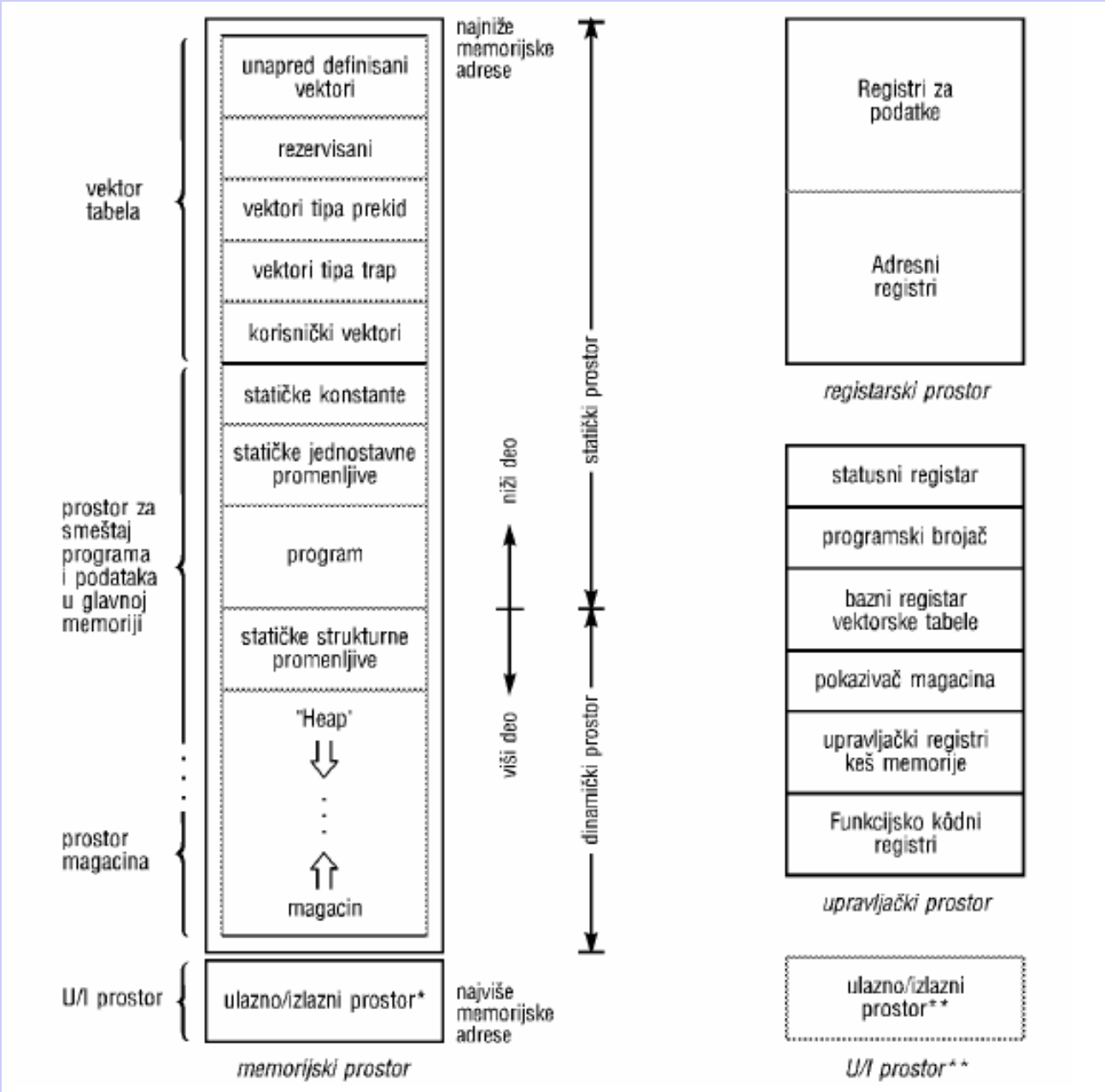


- It is the most common choice in today's general purpose computers
- *Which* register is specified by small "address" (3 to 6 bits for 8 to 64 registers)
- Load and store have one long & one short address: 1 1/2 addresses
- 2-Operand arithmetic instruction has 3 "half" addresses

Adresni prostori

- Kod najvećeg broja računara postoji veći broj različitih adresnih prostora
- Svaki prostor ima odredjenu namenu, a najvažnije su:
  - registarski prostor – čine ga registri opšte i specifične namene
  - prostor glavne memorije – koristi se za čuvanje programa i podataka
  - U/I prostor – namenjen za adresiranje periferala
  - prostor magacina – realizuje se kao memorija tipa LIFO
  - upravljački prostor – čine ga *PSW*, *SR*, registri *MMU*-a i dr.

Načini izvodjenja – adresnog prostora



Predstavljanje instrukcija i načini adresiranja

IV poglavlje

- načini adresiranja -

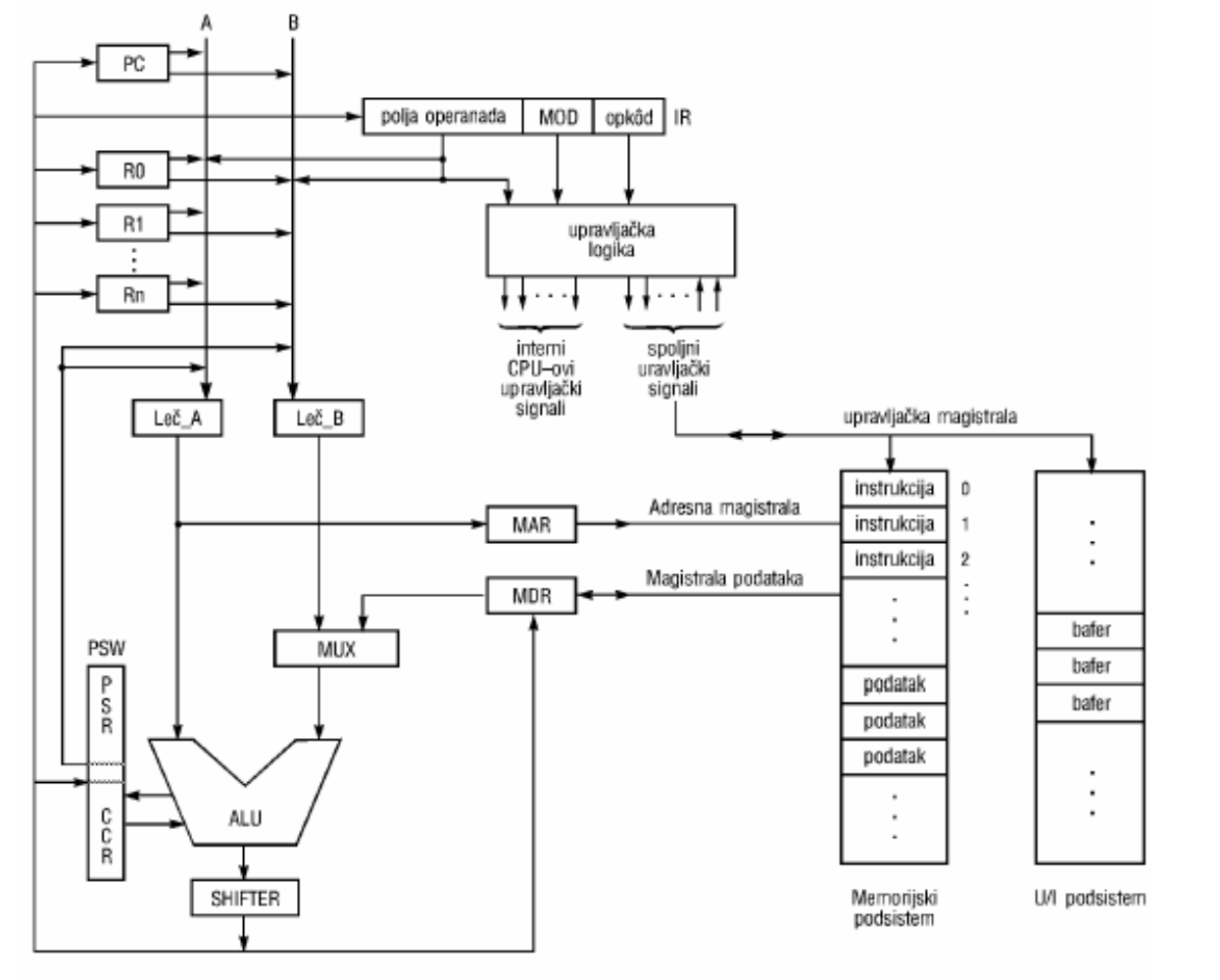
Načini adresiranja

- Svakom operandu instrukcije pridružen je odgovarajući podatak *X*.
- Podatak može biti lociran u memorijskom, U/I podsistemu, CPU-ovim registrima, ili da predstavlja deo instrukcije.
- Da bi izvršio instrukciju procesor treba da pribavi tekuću vrednost *X*-a.
- Ova vrednost se može specificirati na nekoliko načina koje zovemo **adresni načini rada** (*addressing modes*) ili **adresne tehnike** (*addressing techniques*)

Adresni načini rada su sredstva koja se koriste za preslikavanje načina vidjenja adresibilnog prostora mašine na pogled mašine

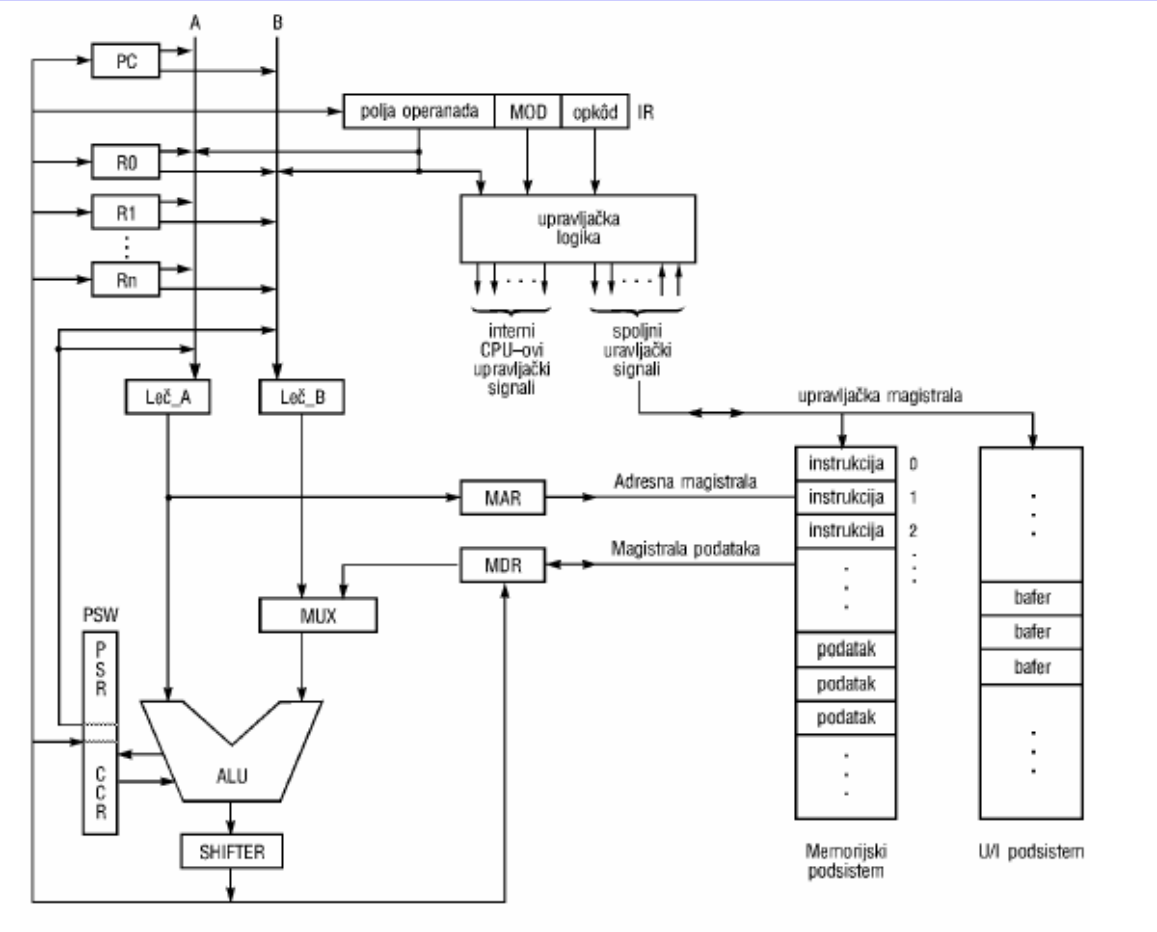
Princip rada jedne mikroarhitekture – aktivnosti CPU-a

Korak 1. U PC-u se nalazi adresa naredne instrukcije u programu koju treba pribaviti. Ova adresa se smešta prvo u *leč_A*, a nakon toga kopira u *MAR* i postavlja na adresnu magistralu



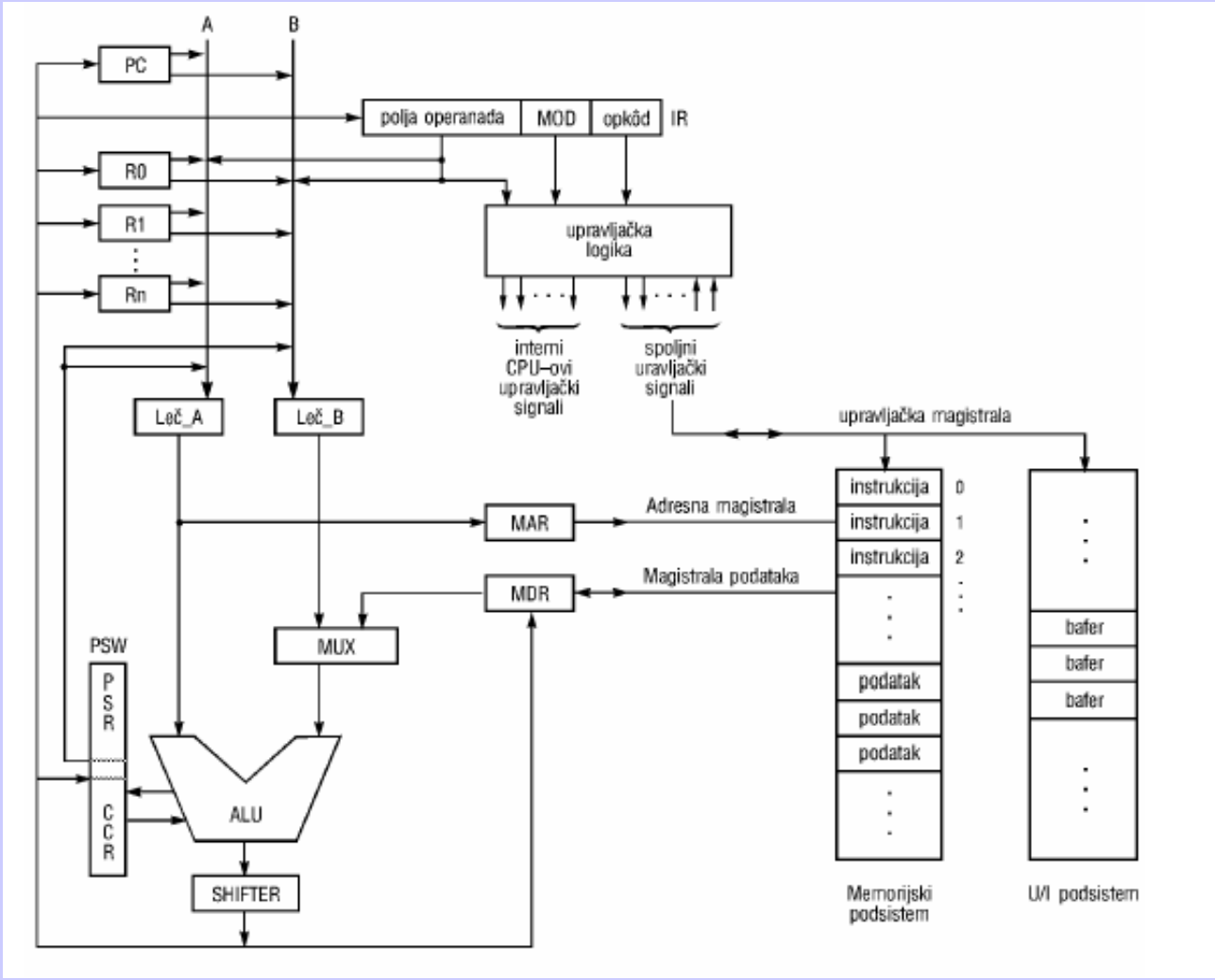
Princip rada jedne mikroarhitekture – aktivnosti CPU-a

Korak 2. Upravljačka logika generiše signal *Mem-Read*, čita se memorija, a podatak koji je prisutan na magistrali za podatke se kopira u registar *MDR*. Nakon toga, preko multipleksera, *MUX*, aritmetičko-logičke jedinice, *ALU*, i pomerača, *SHIFTER*, sadržaj registra *MDR* se kopira u registar *IR*



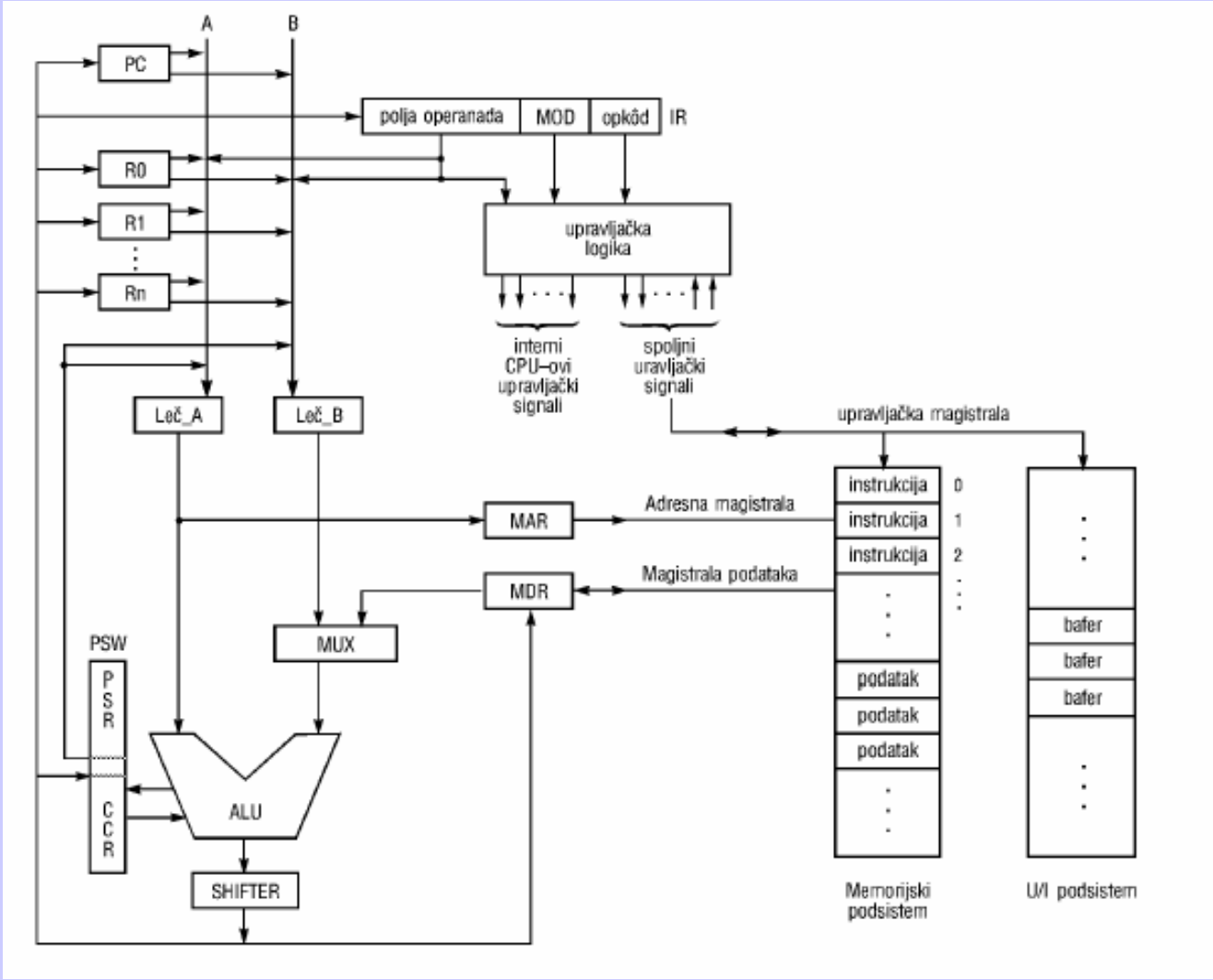
Princip rada jedne mikroarhitekture – aktivnosti CPU-a

Korak 3. *PC* se zatim inkrementira za 1, kako bi ukazao na adresu naredne instrukcije/adresu podatka koju/koga treba pribaviti



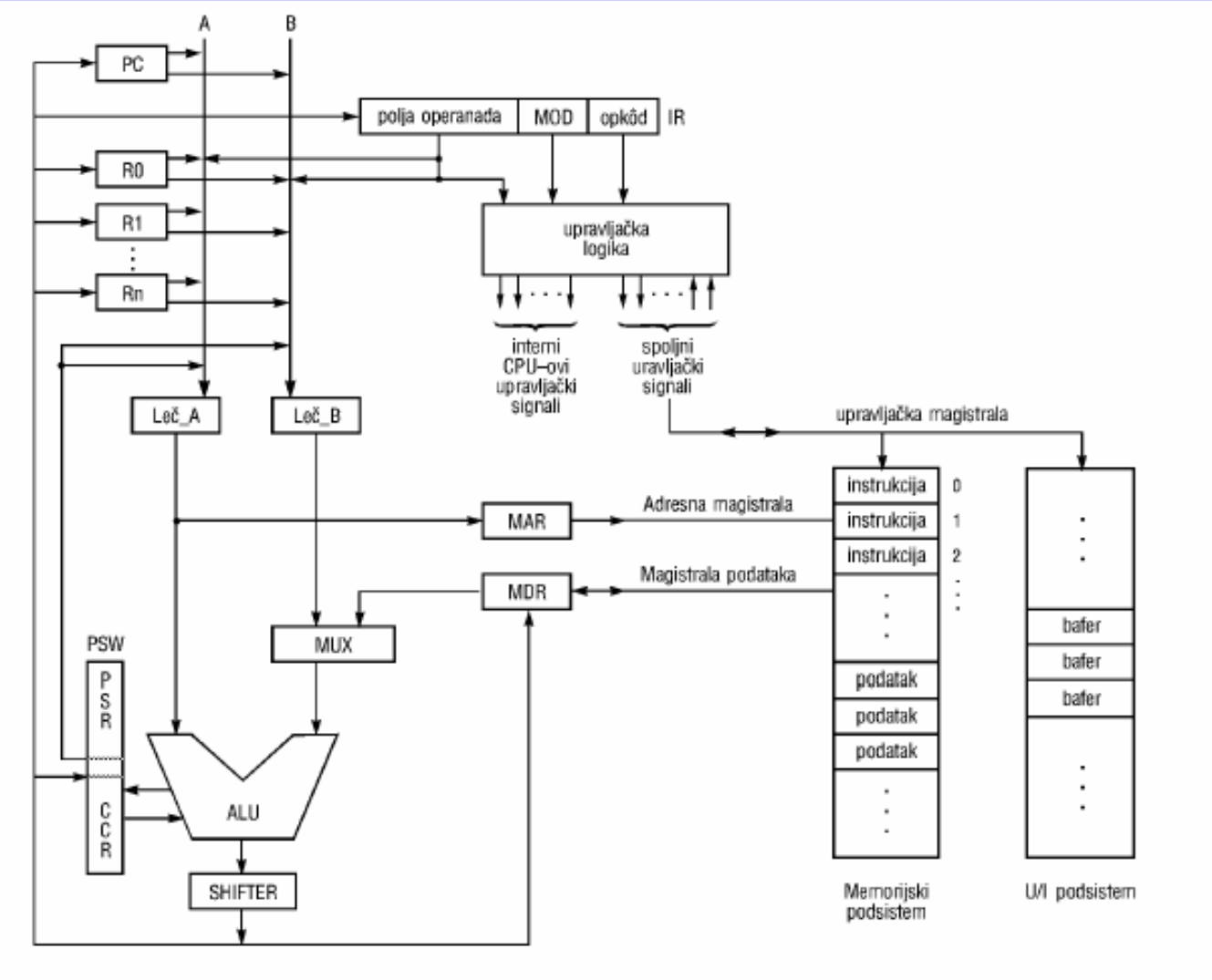
Princip rada jedne mikroarhitekture – aktivnosti CPU-a

Korak 4. Sadržaj *IR*-a se dekodira od strane upravljačke logike i odredjuje se tip instrukcije koju treba izvršiti



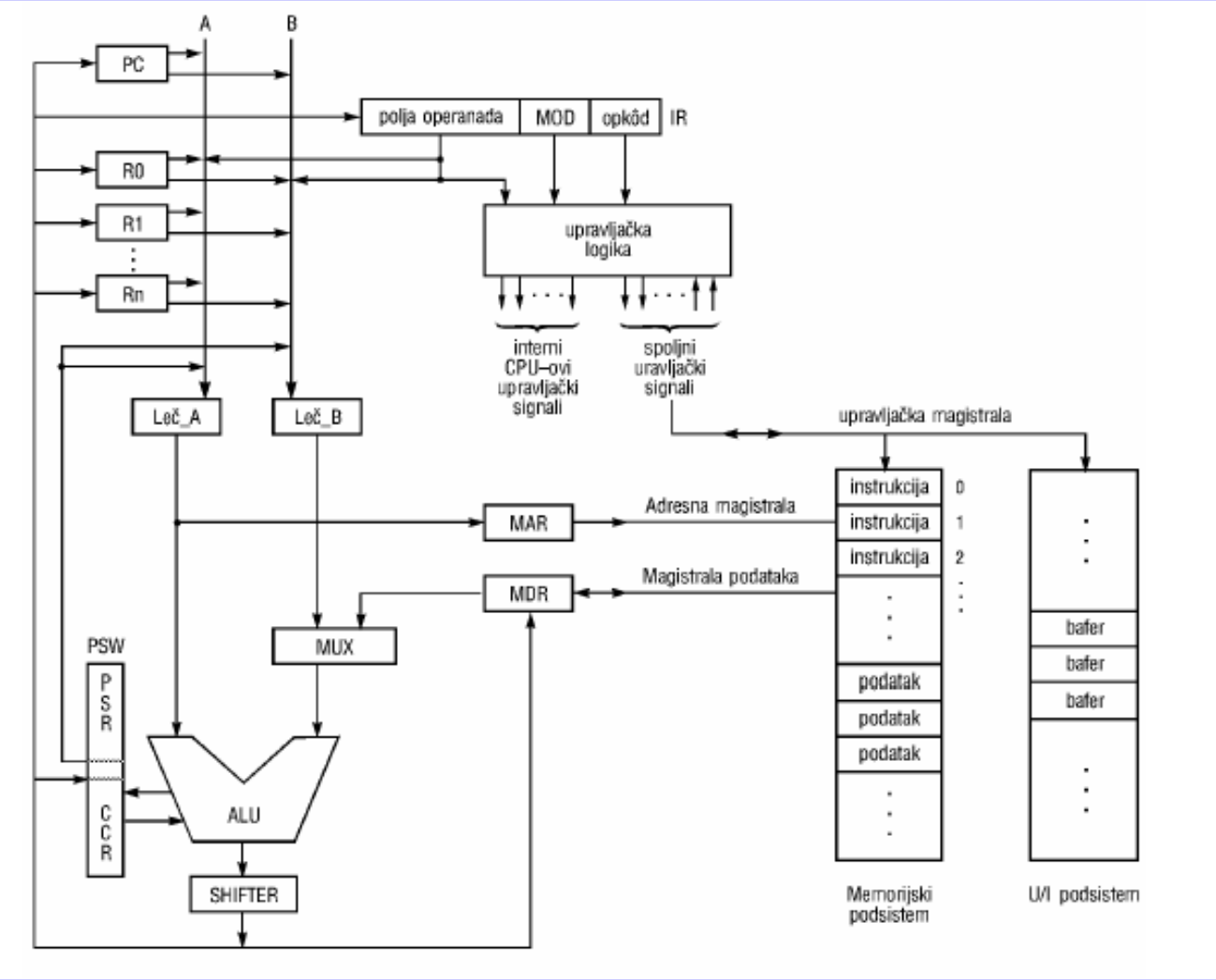
Princip rada jedne mikroarhitekture – aktivnosti CPU-a

Korak 5. Ako instrukcija koristi memorijske operande, odredjuje se gde su oni smešteni u memoriji



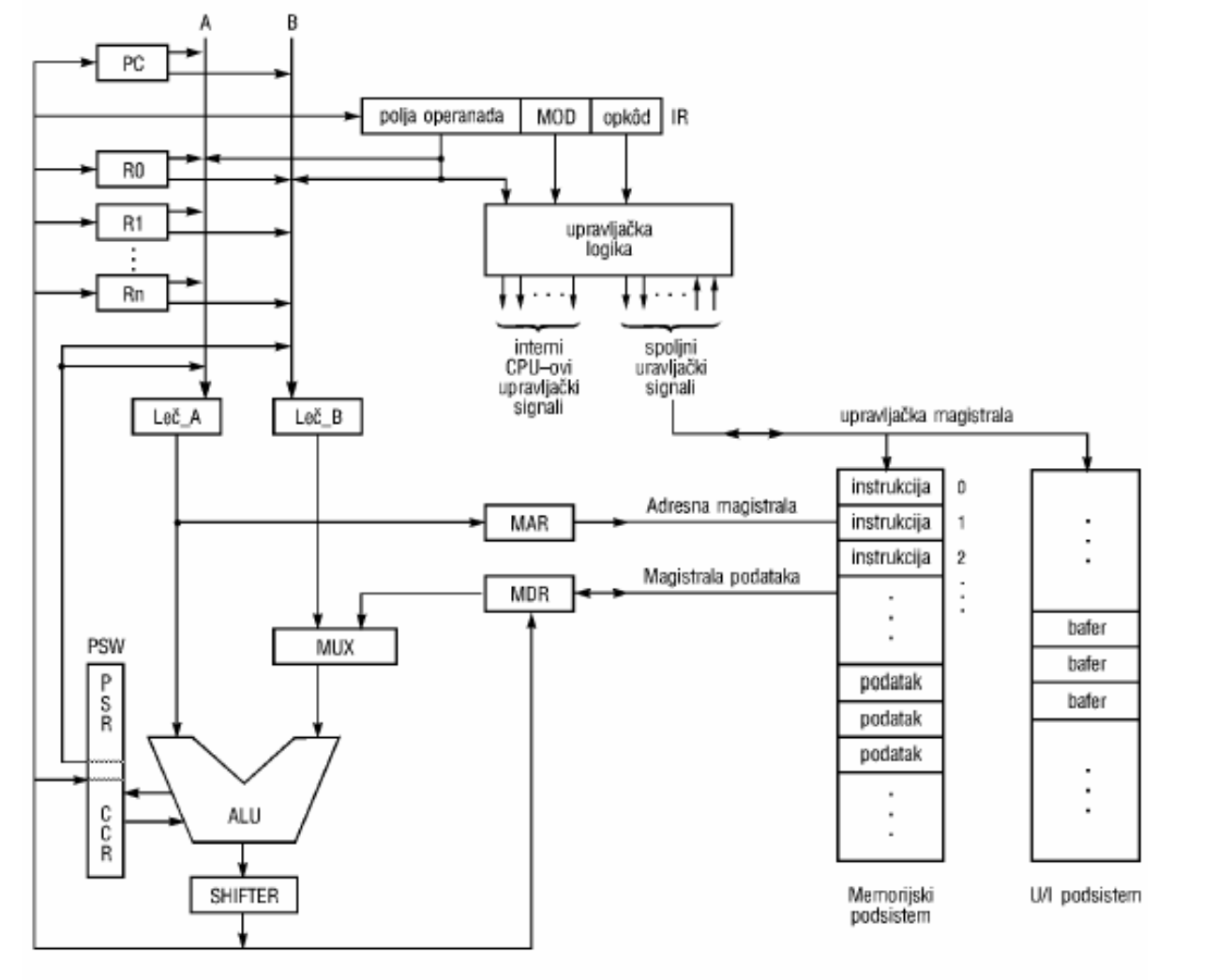
Princip rada jedne mikroarhitekture – aktivnosti CPU-a

Korak 6. Ako je specificirano instrukcijom, podaci koji se pribavljaju iz memorije smeštaju se u interne registre CPU-a



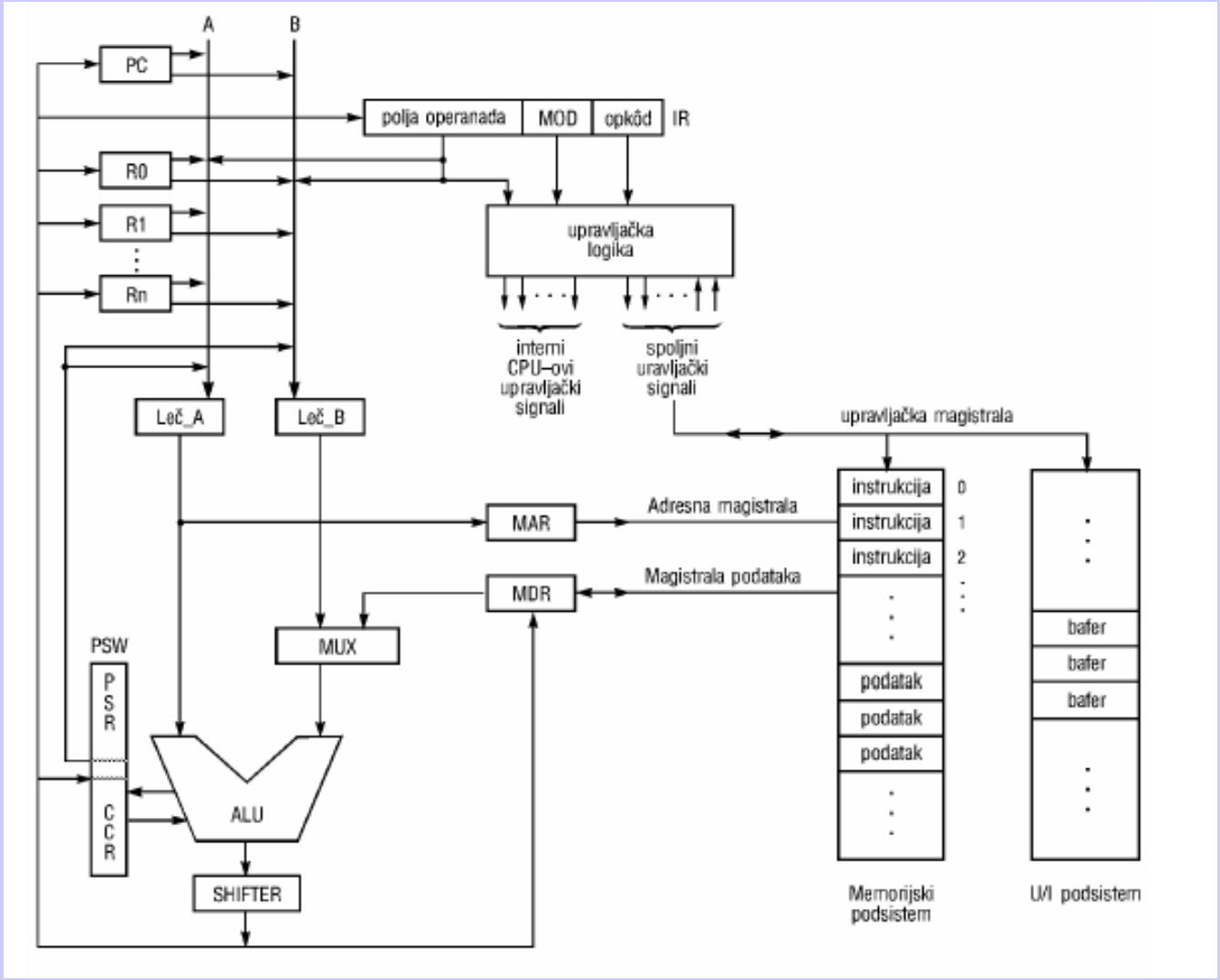
Princip rada jedne mikroarhitekture – aktivnosti CPU-a

Korak 7. Izvršava se instrukcija



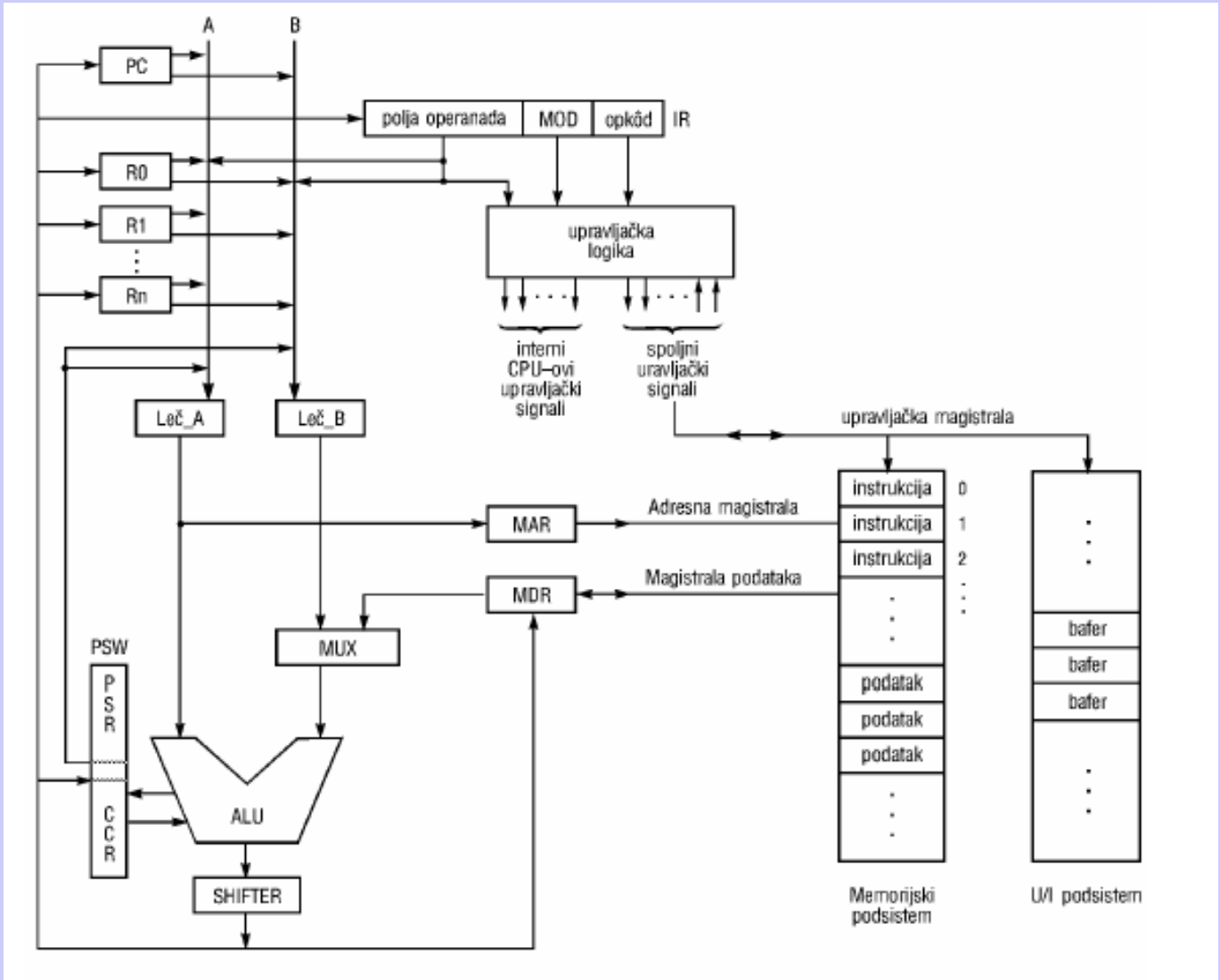
Princip rada jedne mikroarhitekture – aktivnosti CPU-a

Korak 8. Rezultat izračunavanja smešta se na specificiranu lokaciju (memoriju ili registar)



Princip rada jedne mikroarhitekture – aktivnosti CPU-a

Korak 9. Prelazi se na korak (1) da bi se počelo sa izvršavanjem naredne instrukcije u programu



Adresni načini rada

Odredjuju gde se precizno nalazi željeni podatak tj. gde je njegova **efektivna adresa, *EA***.

Specifikacija adresnog načina rada, koja je sadržana u okviru instrukcije, odredjuje kako procesor treba da koristi i interpretira bitove instrukcije i ostalu informaciju da bi izračunao efektivnu adresu.

Adresni načini rada - produžetak

- Da bi odredio efektivnu adresu CPU koristi usluge sledećih operacija:

  - selekcija – bira se operand koji može biti registarski ili memorijski
  - korišćenje baze – odnosi se na sabiranje vrednosti pomoću kojih se formira adresa
  - skaliranje – umnožavanje vrednosti nekog od operanda (obično umnožavanje sa faktorom 1, 2, 4, 8, ...)
  - indeksiranje – odnosi se na sabiranje i noženje celobrojne vrednosti
  - indirekcija – čitanje vrednosti sa zadate adrese
  - inkrementiranje – dodavanje konstante registru ili mem. lokaciji
  - dekrementiranje – oduzimanje konstante registru ili mem. lokaciji

Načini adresiranja

- U zavisnosti od toga kako se vrši formiranje efektivne adrese razlikujemo sledeće načine adresiranja :

Apsolutno adresiranje

- Ako adresa u instrukciji specificira (memorijsku) lokaciju operanda bez korišćenja bilo koje druge dodatne informacije tada taj tip efektivne adrese nazivamo apsolutna adresa.

- Ovaj tip apsolutne efektivne adrese moguće je specificirati na više načina pa u zavisnosti od toga razlikujemo sledeće načine apsolutnog adresiranja:

  - neposredno adresiranje – vrednost operanda je sastavni deo instrukcije
  - direktno adresiranje – odredjeni bitovi instrukcije sadrže kompletnu *EA*
  - implicitno – sama instrukcija ne sadrži adresnu informaciju u eksplicitnoj formi, već se *EA* formira na osnovu opkôda

Indirektno adresiranje

- Indirektnom adresom nazivamo onu adresu koja ne specificira memorijsku lokaciju operanda već memorijsku lokaciju u kojoj se nalazi adresa operanda.
- Razlikujemo sledeća dva načina indirektnog adresiranja:
  - registarsko indirektno – procesor interpretira sadržaj specificiranog registra kao (direktnu) memorijsku lokaciju.
  - indirektno adresiranje preko memorije – indirektna memorijska adresa specificira direktno memorijsku adresu gde se nalazi adresa operanda

Relativno adresiranje

- Kod relativnog adresiranja polje operanda sadrži relativnu adresu ili razmeštaj, a kompletna adresa operanada formira se od strane CPU-a

- Instrukcijom se takodje aksplicitno ili implicitno identifikuju ostale memorijske lokacije koje sadrže dodatnu adresnu informaciju.

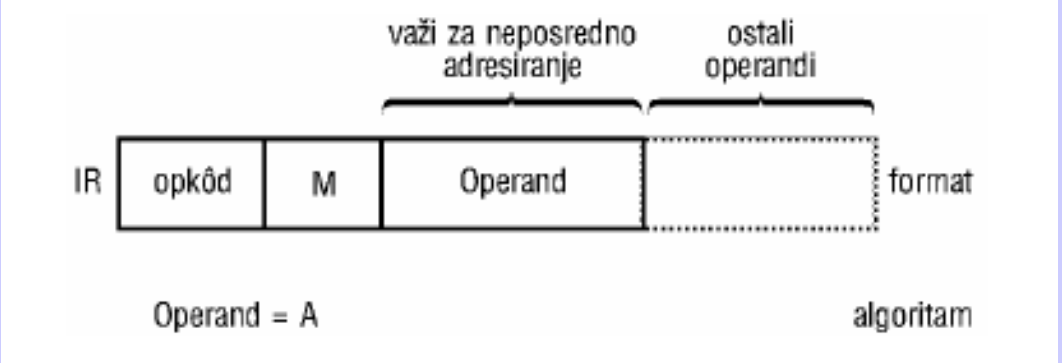
Relativno adresiranje - produžetak

- Kod relativnog adresiranja postoje sledeće varijante:

  - PC-relativno - adresa operanda je specificirana relativno u odnosu na sadržaj programskog brojača
  - bazno-registarsko - u registru kome se obraćamo čuva se memorijska adresa koju sabiramo sa razmeštajem (specificiran u samoj instrukciji), pa se na taj način formira EA.
  - indeksno-adresiranje - EA se izračunava sabiranjem sadržaja selektovanog registra sa adresom specificiranom u okviru instrukcije.
  - autoindeksno - u toku izvršenja instrukcije vrši se automatsko inkrementiranje ili dekrementiranje indeksnog registra
  - preko magacina - pristup magacinu reguliše se pomoću pokazivača magacina.

Neposredno adresiranje

- Najjednostavniji oblik apsolutnog adresiranja je neposredni (*immediate addressing*)
- Adresni deo instrukcije specificira vrednost operanda, tj. operand je konstanta i sastavni je deo instrukcije



Karakteristike:

- koristi se da definiše konstante
- operand se pribavlja u CPU u trenutku kada se pribalja instrukcija
- obim podatka koji se pribavlja ograničen je obimom adresnog polja

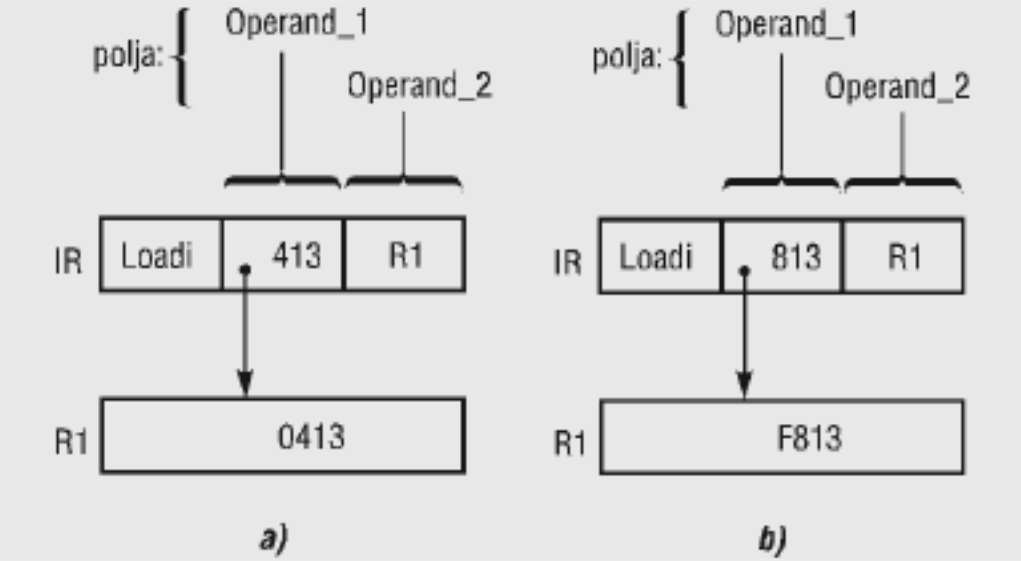
Neposredno adresiranje

Primer:

Primer 4.5

Neka je obim polja neposrednog operanda (#data) instrukcije `Load #data,R1` dvanaestobitni. Ukazati na koju vrednost će se napuniti registar R1, nakon izvršenja instrukcija `Loadi #413,R1`, i `Loadi #813,R1`, ako se nakon punjenja R1 vrši proširenje znaka, a R1 je 16-bitni registar.

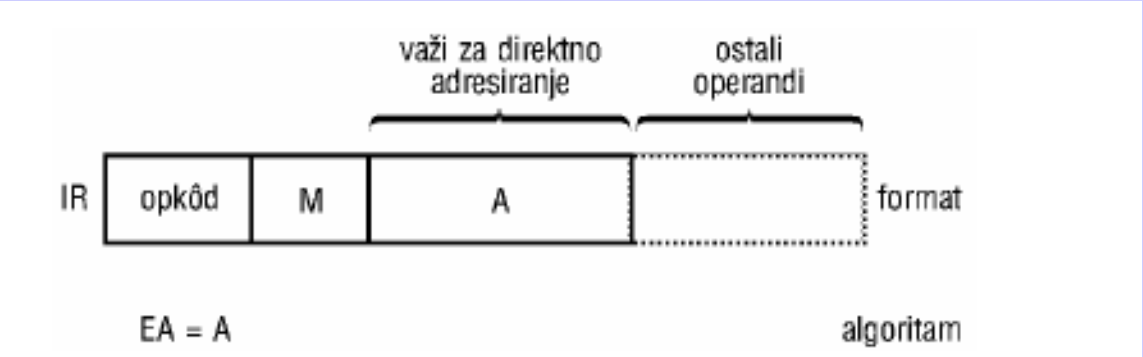
Rešenje



Napomena: Opkod `Loadi` specificira neposredno adresiranje. Specifikacija neposredne vrednosti odnosi se samo na Operand_1, a ne i Operand_2.

Direktno adresiranje

- Kod direktnog načina adresiranja u odgovarajuće polje operanda (polje *operand_A*) smeštena je adresa memorijske lokacije *A* koja odgovara podatku (operandu) *X* kome se pristupa, tj. adresni deo instrukcije sam predstavlja adresu na osnovi koje se vrši obraćanje memoriji, pa se zbog toga ova adresa zove direktna adresa



Karakteristike:

- prednost ovog načina rada adresiranja je jednostavnost izvodjenja, a nedostatak ograničeni adresni opseg

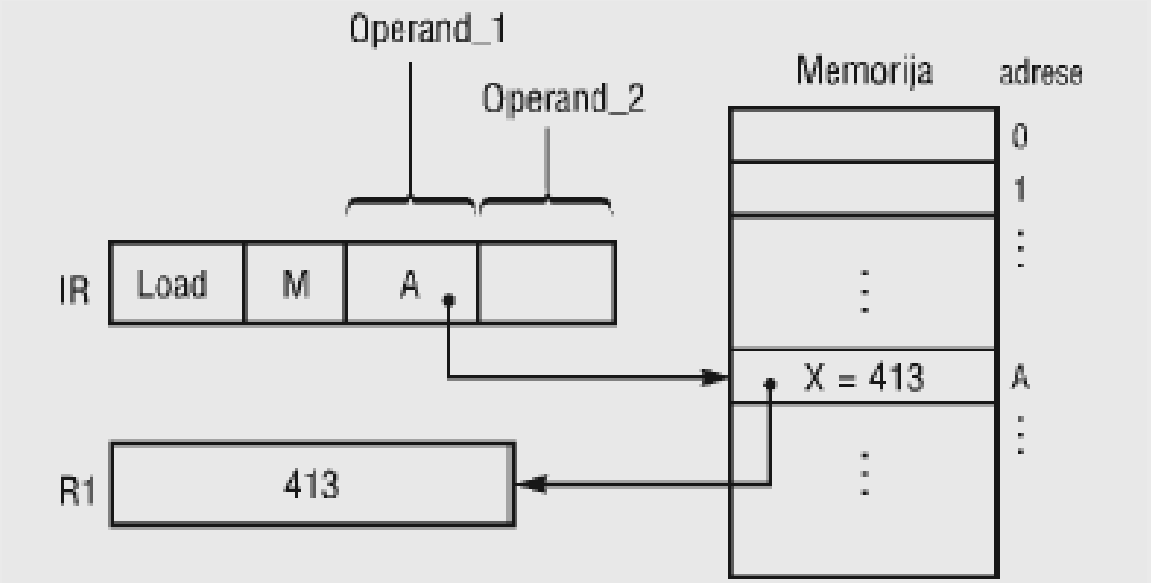
Direktno adresiranje

Primer:

Primer 4.6

Neka je sadržaj memorijske lokacije A jednak 413, tj. (A)=X=413. Koja će vrednost biti upisana u R1 nakon izvršenja instrukcije `Load A,R1`.

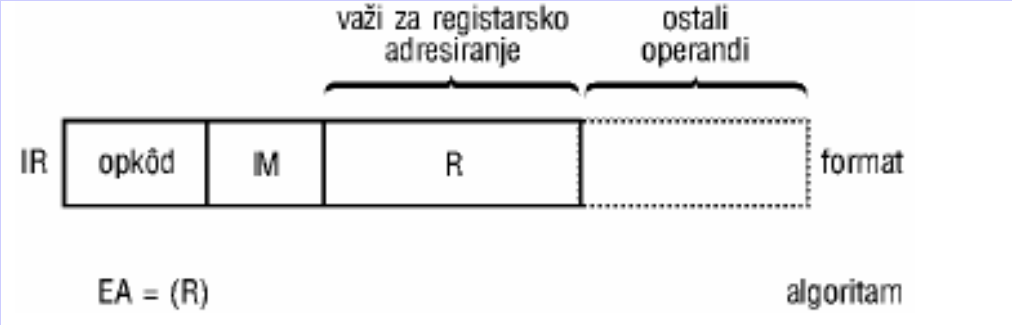
Rešenje



Napomena: Opkod `Load` specificira direktno adresiranje. Specifikacija direktnog adresiranja odnosi se samo na Operand_1, a ne i Operand_2.

Registarsko adresiranje

- Slično je direktnom adresiranju. Jedina razlika je u tome što se adresno polje odnosi na specifikaciju registra, a ne adresu glavne memorije



Karakteristike:

- adresno polje instrukcije kojim se specificira registar je mali i obima nekoliko bitova
- ne vrši se obraćanje memoriji
- instrukcije koje manipulišu sa sadržajem registra brzo se izvršavaju
- ograničen adresni prostor

Registarsko adresiranje

Primer:

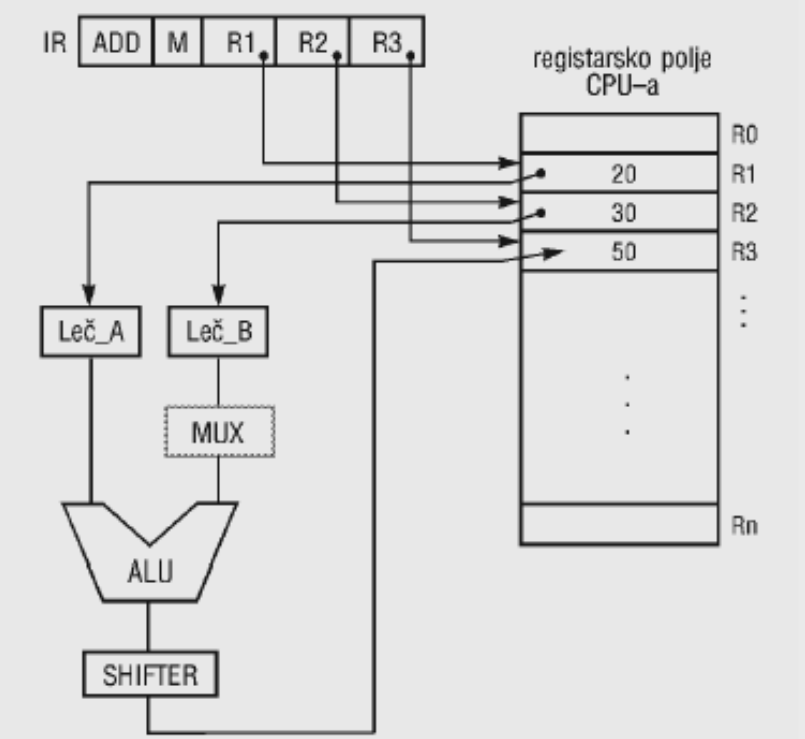
Primer 4.7

Neka je kod mašine sadržaj registra R1 jednak 20 heksadecimalno, a registra R2 30 heksadecimalno. Nakon izvršenja instrukcije

```
Add R1,R2,R3     ; R1+R2→R3
```

odrediti sadržaj registra R3 i ukazati koji način adresiranja se koristi za specifikaciju prvog, drugog i trećeg operanda.

Rešenje



Nakon izvršenja instrukcije sadržaj registra R3 je 50 hexa. Za specifikaciju prvog, drugog i trećeg operanda, u ovom slučaju, koristi se registarsko adresiranje.

Implicitno adresiranje

- Za implicitnu adresu operanda kažemo da je ona koja se u adresnom delu naredbe ne specificira posebno, već je mesto operanda unapred poznato i zavisi od opkôda polja.

Karakteristike:

- ovakav način rada adresiranja koristimo kod postavljanja vrednosti markera registra CCR

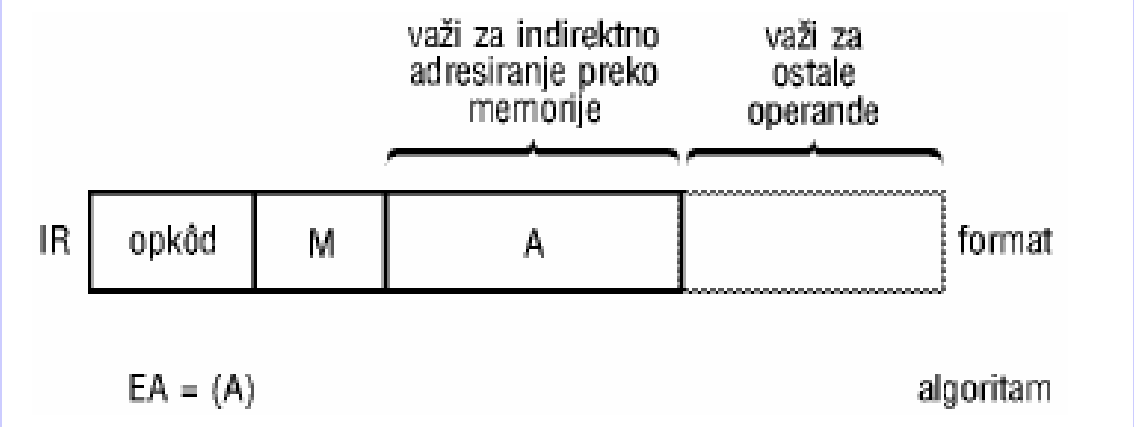
Primer:

Tipični primeri su instrukcije

```
Clc   ;marker prenosa C se postavlja na nulu
Stc   ;marker prenosa C se postavlja na jedan
```

Indirektno adresiranje preko memorije

- Kod indirektnog adresiranja, adresom se specificira memorijska lokacija u kojoj se ne čuva operand nego adresa operanda



Karakteristike:

- potrebne su dve operacije pribavljanja
- mogućnost pristupa većem adresnom prostoru
- veći broj obraćanja memoriji

Razlika izmedju direktnog i indirektnog adresiranje

Primer:

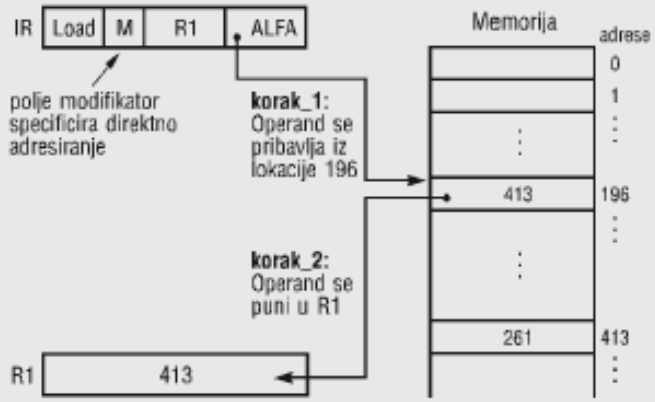
Da bi jasnije ukazali na razliku izmedju direktnog i indirektnog adresiranja memorije analizirajmo izvršenje sledeće dve instrukcije

```
Load   196,R1        ; direktno adresiranje
Loadn  196,R1        ; indirektno adresiranje
```

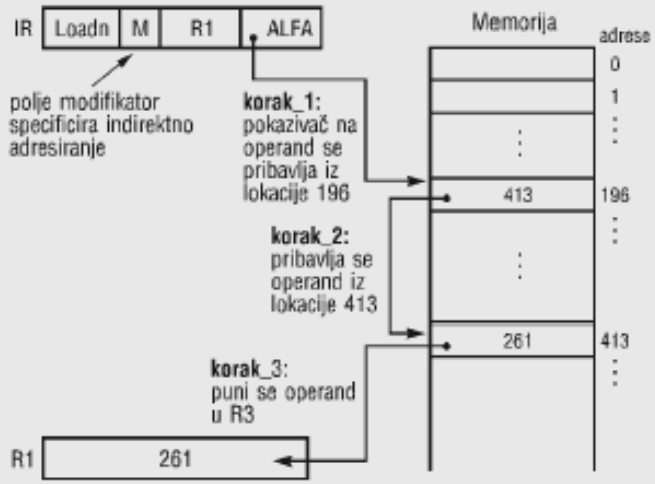
Uslovno usvajamo da je sadržaj lokacije (196)=413 hexa, a sadržaj lokacije 413 hexa 261 hexa.

Rešenje

Direktno adresiranje



Indirektno adresiranje preko memorije



Broj na lokaciji 413 nije operand nego se koristi da "pokaže" na operand, pa se zbog toga zove **pokazivač** (pointer).

Registarsko indirektno adresiranje

- Kao što je registarsko adresiranje analogno sa direktnim adresiranjem, tako je i registarsko indirektno adresiranje analogno sa indirektnim adresiranjem.
- U oba slučaja razlika se ogleda u sledećem:
- Adresno polje instrukcije kod indirektnog adresiranja ukazuje na memorijsku lokaciju, a kod registarsko indirektnorg adresiranja na registar

Registarsko indirektno adresiranje

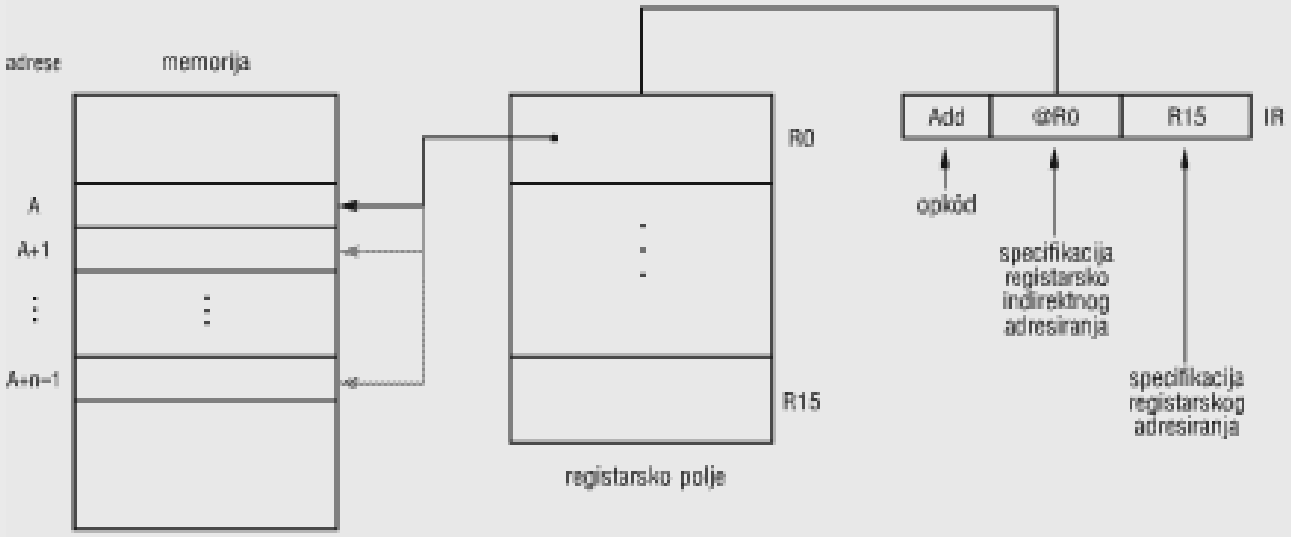
Primer:

Primer 4.9

Neka je dat skup (vektor) od *n* brojeva lociranih u susednim lokacijama glavne memorije. Brojeve treba sabirati a rezultat smestiti u R15. Početna adresa brojeva, A smeštena je u R0, a R15 je inicijaliziran na nulu. Neka simbol *@R0* ukazuje da se registar *R0* koristi kao izvor adrese a ne operanda, a simbol *#K* označava da je *K* neposredni operand. Ako sekvencu instrukcija

```
Add @R0,R15
Inc R0,#1
```

ponovimo *n* puta sabraćemo svih *n* brojeva koji su smešteni u glavnoj memoriji na lokacijama A, A + 1, ..., A + n - 1, a rezultat upisati u R15.



Relativno adresiranje

- Kada se deo adresne informacije sadrži u instrukciji, a kompletna adresa formira od strane CPU-a, za ovaj način adresiranja kažemo da se zove relativno.
- Ovo adresiranje se karakteriše time da polje operanda sadrži relativnu adresu ili razmeštaj (*displacement* ili *offset*) – neki autori za termin razmeštaj koriste pojam pomeraj
- Tri najčešće korišćene tehnike relativnog adresiranja su:
  - PC-relativno
  - bazno-registarsko
  - indeksno

PC - relativno adresiranje

- Kod ovog adresiranja registar kome se implicitno obraćamo je programski brojač (PC)
- Tekuća adresa instrukcije sabira se sa razmeštajem i na taj način formira se *EA*. Kod ove operacije razmeštaj se tretira kao broj u dvojičnom komplementu.
- Kao rezultat, *EA* predstavlja relativni razmeštaj u odnosu na adresu instrukcije
- PC relativno adresiranje koristi koncept lokalnosti jer najveći broj obraćanja memoriji je relativno blizu instrukciji koja se tekuće izvršava.
- Efektivna adresa se izračunava na sledeći način:

$$\mathbf{EA = (PC) + D}$$

gde je:

D – razmeštaj specificiran instrukcijom

PC - relativno adresiranje

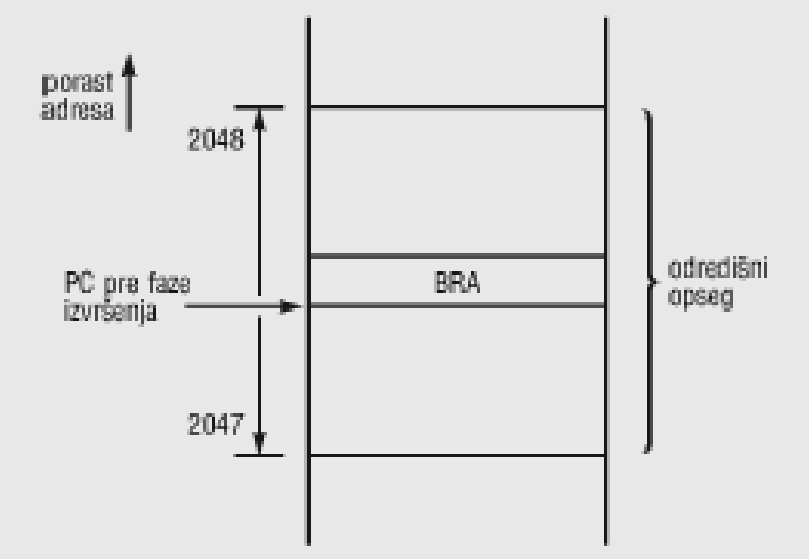
Primer:

Primer 4.10

Kod mikroprocesora MC68020 srećemo PC relativno adresiranje. Tipičan primer je instrukcija

```
BRA(CONST_12,PC)
```

gde je CONST_12 12-bitna celobrojna vrednost koja specificira razmeštaj od 2047 lokacija prema višim adresama i 2047 lokacija prema nižim. Opseg mogućih odredišta grananja kod PC relativnog grananja sa 12-bitnim razmeštajem prikazan je na slici

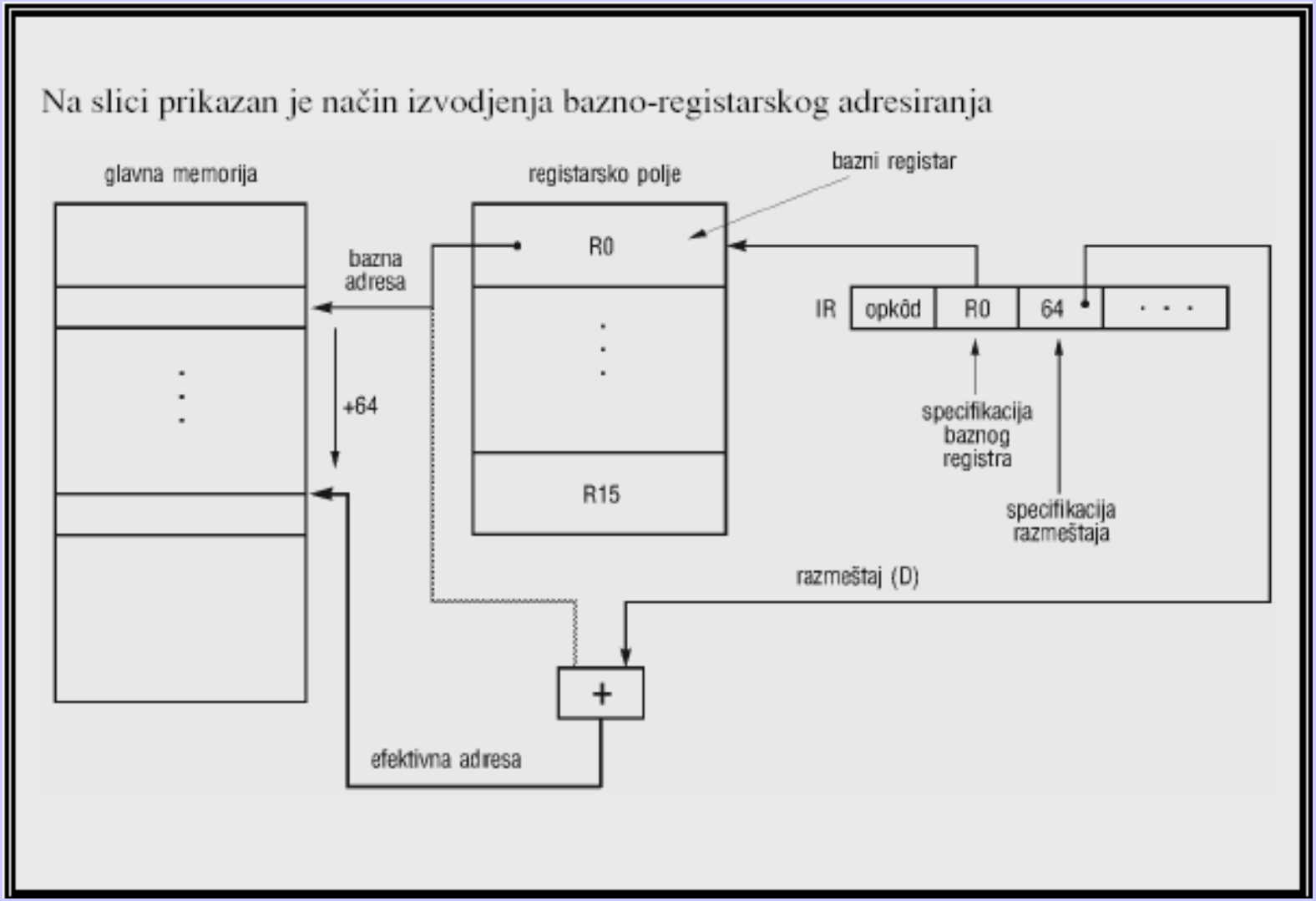


Bazno-registarsko adresiranje

- Interpretacija bazno registarskog adresiranja je sledeća:
- Registar kome se obraćamo, *R*, čuva memorijsku adresu, a adresno polje sadrži razmeštaj, *D*, (obično je to neoznačena celobrojna vrednost) u odnosu na memorijsku lokaciju.
- Kada se registar *R* koristi na ovaj način kažemo da je bazni registar, a njegov sadržaj se zove bazna adresa
- Obraćanje baznom registru može biti eksplicitno ili implicitno
- Bazno-registarsko adresiranje podržava relokatibilnost programa i podataka, tj. mogućnost računarskog sistema da u različito vreme programu ili segmentima podataka dodeljuje različite oblasti glavne memorije

Bazno-registarsko adresiranje

Primer:



Indeksno adresiranje

- Kod indeksnog adresiranja interpretacija je sledeća:
- Adresno polje, *A*, sadrži adresu glavne memorije, a u registru *R*, u kome se obraćamo smešten je pozitivni razmeštaj u odnosu na tu adresu.
- Za registar *R* kažemo da je indeksni registar
- Korišćenje ovog načina adresiranja je suprotan interpretaciji bazno-registarskog adresiranja.
- Nezavisno od interpretacije, kod oba načina adresiranja (bazno-registarski i indeksni) metod izračunavanje *EA* je isti,

$$EA=R+D$$

- Indeksiranjem se obezbedjuje efikasan mehanizam za obavljanje iterativnih operacija

Indeksno adresiranje

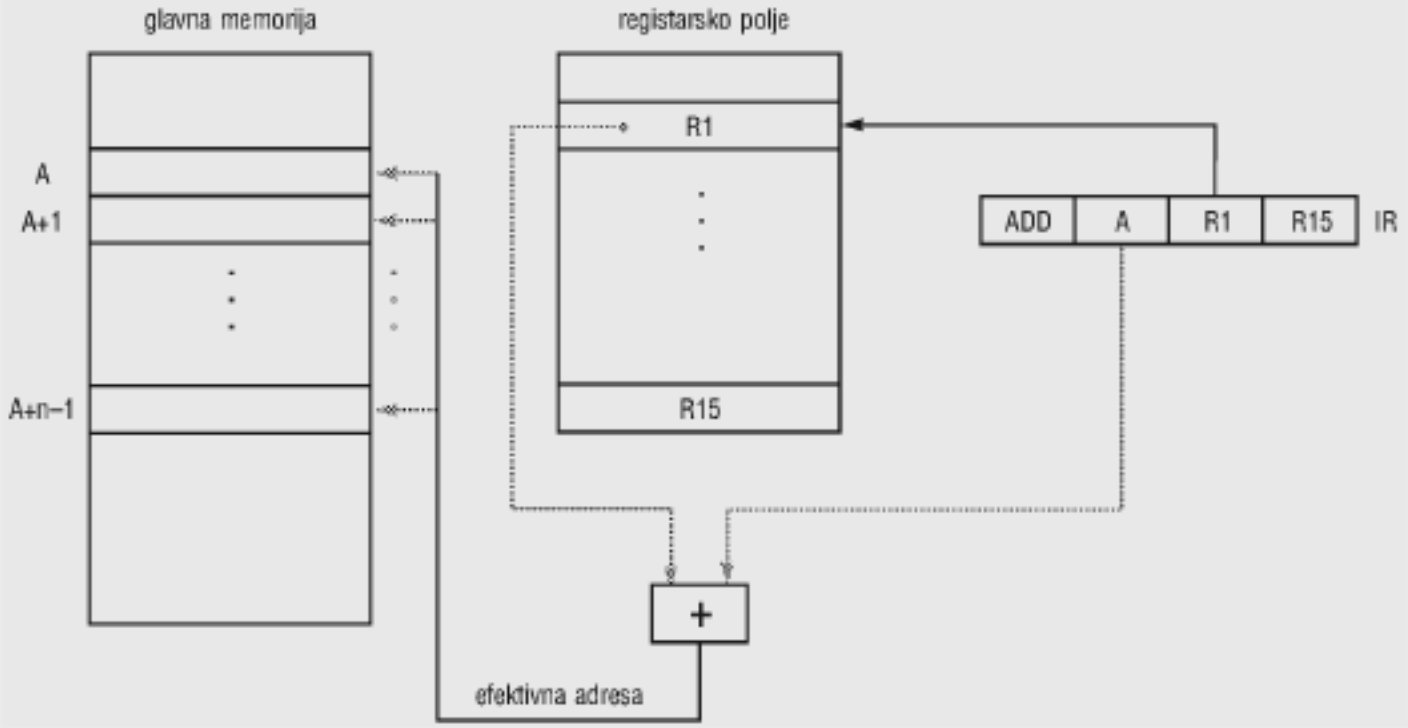
Primer:

Pristup elementima skupa od *n* brojeva, osim korišćenja tehnike zasnovane na indeksno-registarskom adresiranju, može se ostvariti i preko indeksiranog adresiranja. Neka je početna adresa polja A specificirana kao konstanta, a razmeštaj *i*-tog elementa (i=0, 1, ..., n-1) čuva se u indeksnom registru R1. Na početku registri R1 i R15 se inicijaliziraju na nuli. Ako se sledeća sekvenca instrukcija

```
Add #A,R1,R15
Inc R1,#1
```

izvrši *n* puta, u registru R15 dobiće se suma svih elemenata skupa. Instrukcijom `Inc R1,#1` povećava se sadržaj indeksnog registra R1 za jedan. Na slici prikazan je način izvodjenja indeksiranog adresiranja na primeru izvršenja instrukcije `Add #A,R1,R15`.

Imajući u vidu da registar R1 pripada skupu registara opšte namene u instrukciji `Add #A,R1,R15` on je eksplicitno specificiran.



Autoindeksiranje

- Indeksni registar se često koristi za obavljanje iterativnih zadataka
- Kod nekih arhitektura, nakon obavljene iterativne operacije, ugradjuje se onaj deo aktivnosti instrukcije koji automatski vrši inkrementiranje ili dekrementiranje indeksnog registra
- Rezultantni adresno modifukujući proces se zove **autoindeksiranje**
- Postoje sledeće varijante autoindeksiranja :
  - postinkrementiranje
  - postdekrementiranje
  - preinkementiranje
  - predekrementiranje
  - kombinacija post-/pre-inrementiranje/dekrementiranje.

Autoindeksiranje - postinkrementiranje

Primer:

Instrukcija

```
Move.W    (A1)+,(A2)+
```

ekvivalentna je sledećoj sekvenci

```
Move.W    (A1),(A2)    ; kopira se 16-to bitni podatak
Add.L      #2 , A1     ; A1:=A1+2
Add.L      #2 , A2     ; A2:=A2+2
```

Autoindeksiranje - postdekrementiranje

Primer:

Instrukcija

```
Move.W    (A1)-,(A2)-
```

ekvivalentna je sledećoj sekvenci

```
Move.W    (A1),(A2)    ; kopira se 16-to bitni podatak
Sub.L      #2 , A1     ; A1:=A1-2
Sub.L      #2 , A2     ; A2:=A2-2
```

Autoindeksiranje - preinkrementiranje

Primer:

Instrukcija

```
Move.W    + (A1),+(A2)
```

ekvivalentna je sledećoj sekvenci

```
Add.L       #2 , A1     ; A1:=A1+2
Add.L       #2 , A2     ; A2:=A2+2
Move.W    (A1),(A2)     ; kopira se 16-to bitni podatak
```

Autoindeksiranje - predekrementiranje

Primer:

Instrukcija

```
Move.W    - (A1),-(A2)
```

ekvivalentna je sledećoj sekvenci

```
Sub.L       #2 , A1     ; A1:=A1-2
Sub.L       #2 , A2     ; A2:=A2-2
Move.W    (A1),(A2)     ; kopira se 16-to bitni podatak
```

Autoindeksiranje – kombinovano preinkrementiranje i postdekrementiranje

Primer:

Instrukcija

```
Move.W    + (A1), (A2)-
```

ekvivalentna je sledećoj sekvenci

```
Add.L      #2 , A1      ; A1:=A1+2
Move.W    (A1),(A2)     ; kopira se 16-to bitni podatak
Sub.L      #2 , A2      ; A2:=A2-2
```

Adresiranje magacina

- Magacin predstavlja linearno polje lokacija glavne memorije
- Magacinu se može pristupiti samo sa jednog kraja koji se naziva vrh-magacina (*TOS- Top of Stack*)
- Pristup magacinu reguliše se registrom ***SP*** koji se naziva pokazivač magacina (*stack pointer*)
- Operacijom upis u magacin, nazvanom ***Push*** smešta se novi podatak na vrh magacina
- Operacija čitanje, nazvana ***Pop*** izbavlja podatak sa vrha magacina
- Operacija *Push/Pop* menja se položaj vrha magacina za odgovarajuću dužinu operanda koji se smešta/izbavlja u/iz magacina

Korišćenje *Push i Pop* operacija

Primer 1:

```
Push  R1        ; SP ← SP- 4,  (R1) → M [SP]
                ; registar R1 je 32-bitni
                ; magacin kako se puni raste naniže
```

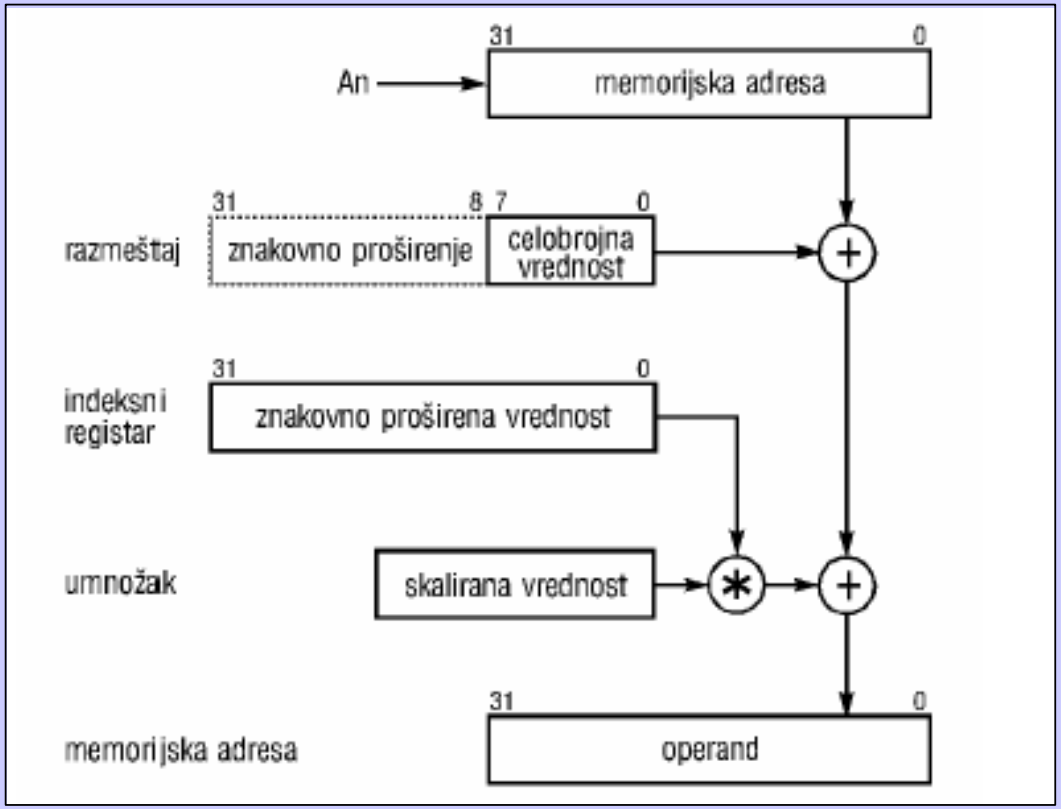
Primer 2:

```
Pop  R1         ; (R1)←  M [SP],   SP ← SP+4
                ; registar R1 je 32-bitni
                ; magacin kako se prazni smanjuje se naviše
```

Indirektno sa indeksiranjem kod skaliranja i baznim razmeštajem

- Efektivna adresa se izračunava kao

$$EA = (A_n) + (X_i).[S] * SCALE + < disp >$$



Primer:

```
MOVE.W  2(A1,D1.W), D2
```

Indirektno sa indeksiranjem kod skaliranja i baznim razmeštajem

- Efektivna adresa se izračunava kao

$$EA = (A_n) + (X_i).[S] * SCALE + < disp >$$

gde je:

- Xi indeksni registar i može biti bilo koji An ili Dn;
- imenovanje (Xi).S dozvoljava da je [S] = W ili L tj. 16-bitna ili 32-bitna informacija koja se uvek znakovno proširuje na 32-bita u toku adresnog izračunavanja.
- vrednost umnožavanja SCALE omogućava skaliranje za 1, 2, 4 ili 8.
- vrednost <disp> mpže biti 8-, 16- ili 32-bitna.

Primer 2:

Kod instrukcije

```
MOVE.L   ($900,A3,D2.W*4), D1
```

sa (A3) = $00020000 i (D2) = $00001000, adresira se heksadecimalna lokacija

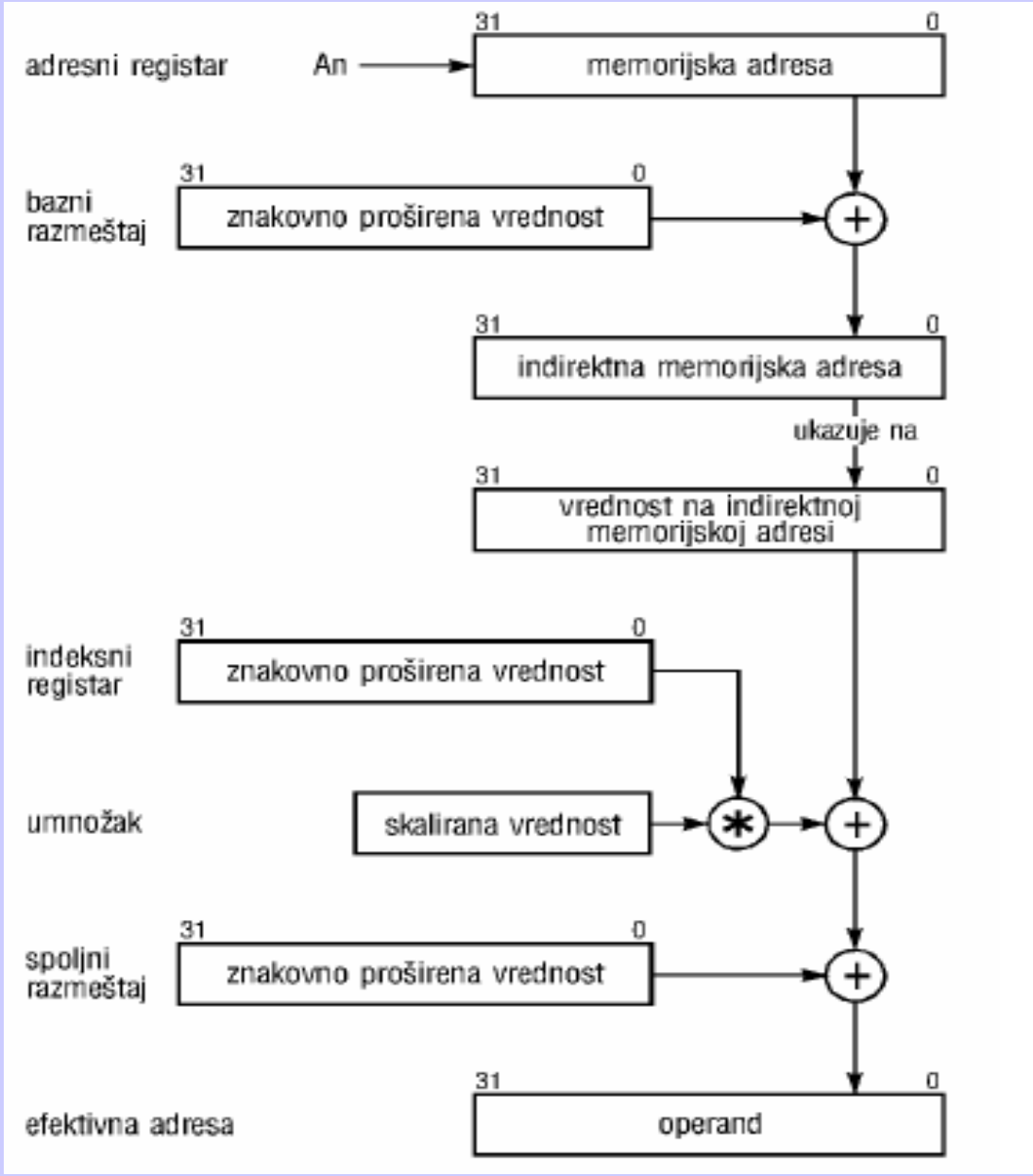
20000	baza
4000	indeks, skaliranje
900	bazni razmeštaj
24900	

i kopira 32-bitnu vrednost sa lokacije M [24900] u D1

Memorijsko indirektno sa postindeksiranjem

- Efektivna adresa se izračunava kao

$$EA = [< bd > + An] + (X_n).[S] * SCALE + < od >$$



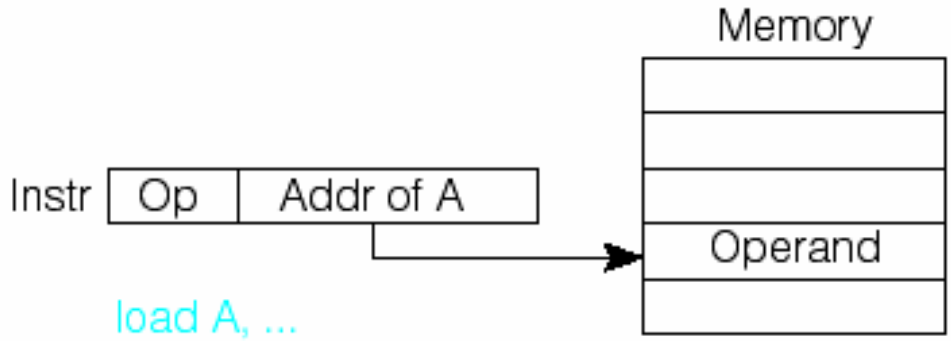
Common Addressing Modes

(a) Immediate addressing:

instruction contains
the operand

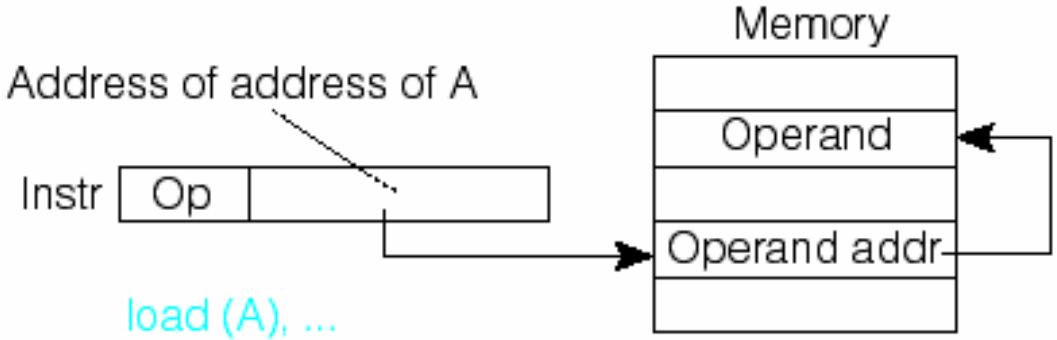
(b) Direct addressing:

instruction contains
address of operand



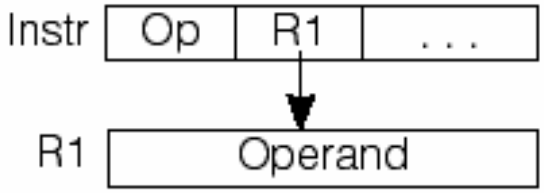
(c) Indirect addressing:

instruction contains
address of address
of operand



(d) Register direct addressing:

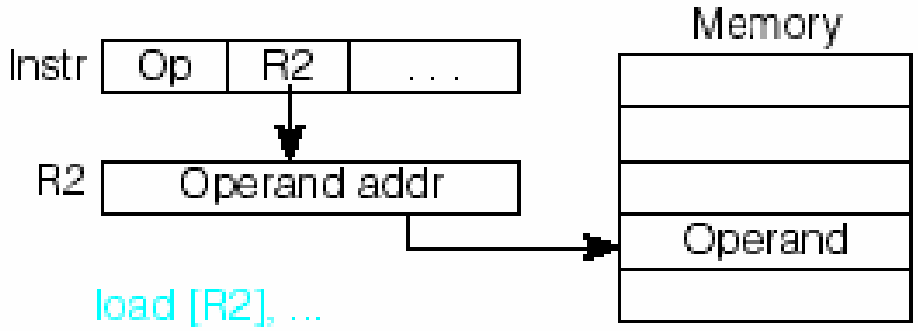
register contains operand



Common Addressing Modes

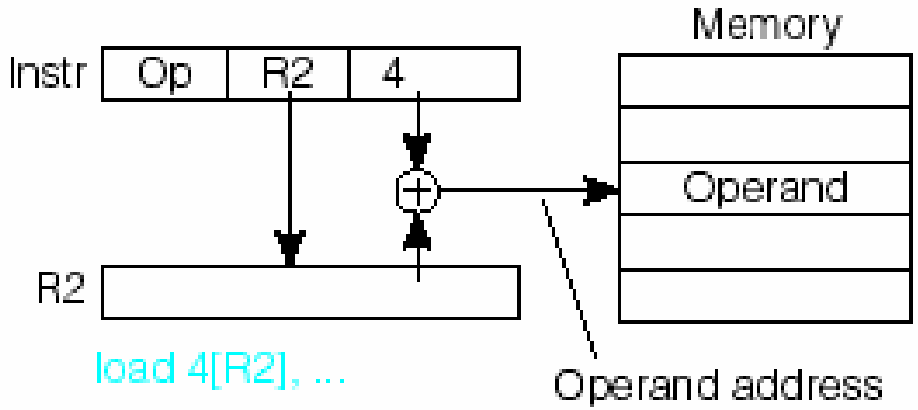
(e) Register indirect addressing:

register contains address of operand



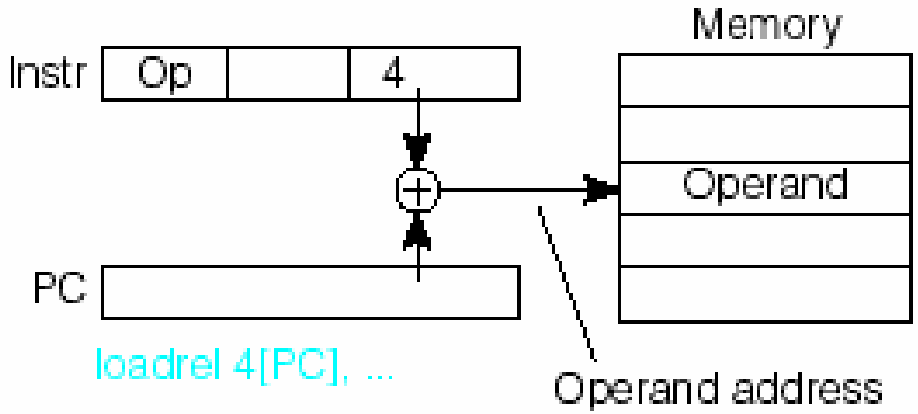
(f) Displacement (based or indexed) addressing:

address of operand = register + constant



(g) Relative addressing:

address of operand = PC + constant

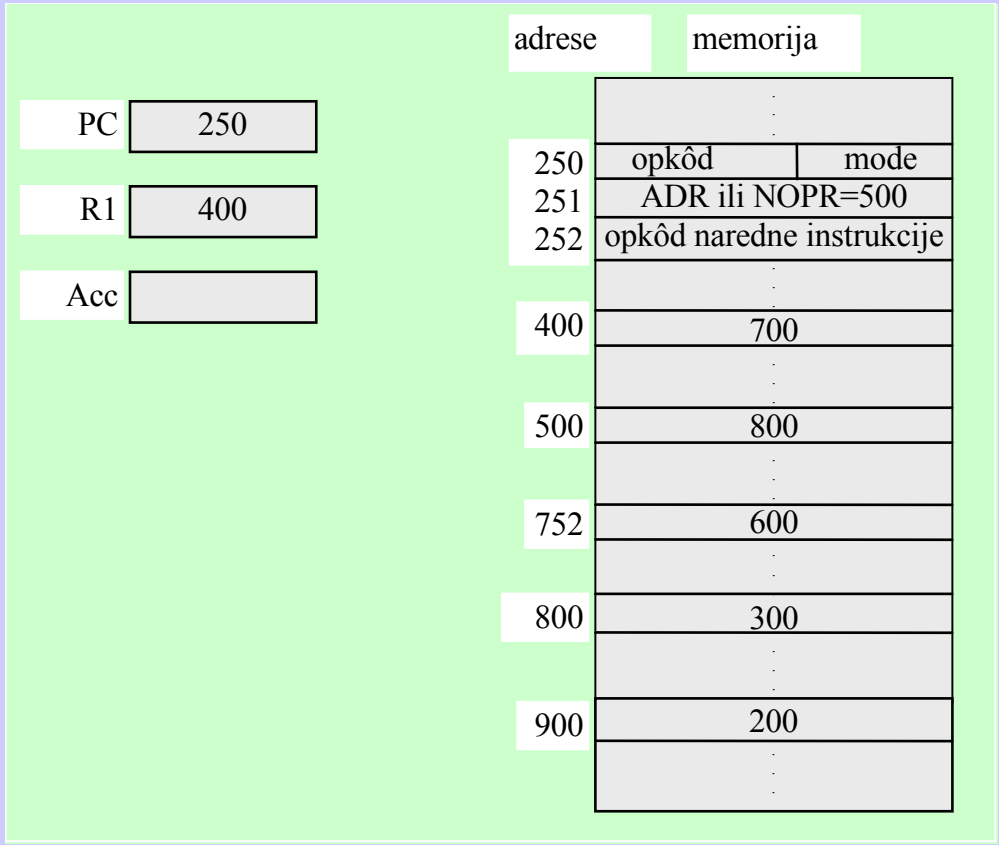


Zadatak 1.

Instrukcija koja je sme{tena na adresama 250 i 251 je tipa "*Load u Acc*" kod koje je adresno polje *ADR* (ili neposredni operando NOPR) jednak 500. Vrednost *PC*-a pre pribavljanja ove instrukcije je 250 a nakon pribavljanja 252 (ukazuje na narednu instrukciju). Sadr`aj registra *R1* je 400, a u *Acc* se sme{ta rezultat koji se dobija izvr{enjem instrukcije.

Odrediti sadr`aj registra *Acc* i efektivnu adresu za slede}e adresne na~ine rada ove instrukcije:

- **direktni**
- **neposredni**
- **indirektni**
- **relativni**
- **indeksni**
- **registarski**
- **registarsko indirektni**



Izgled memorije i stanje *PC*-a, *R1* i *Acc* pre izvršenja instrukcije *Lda*

Odgovor

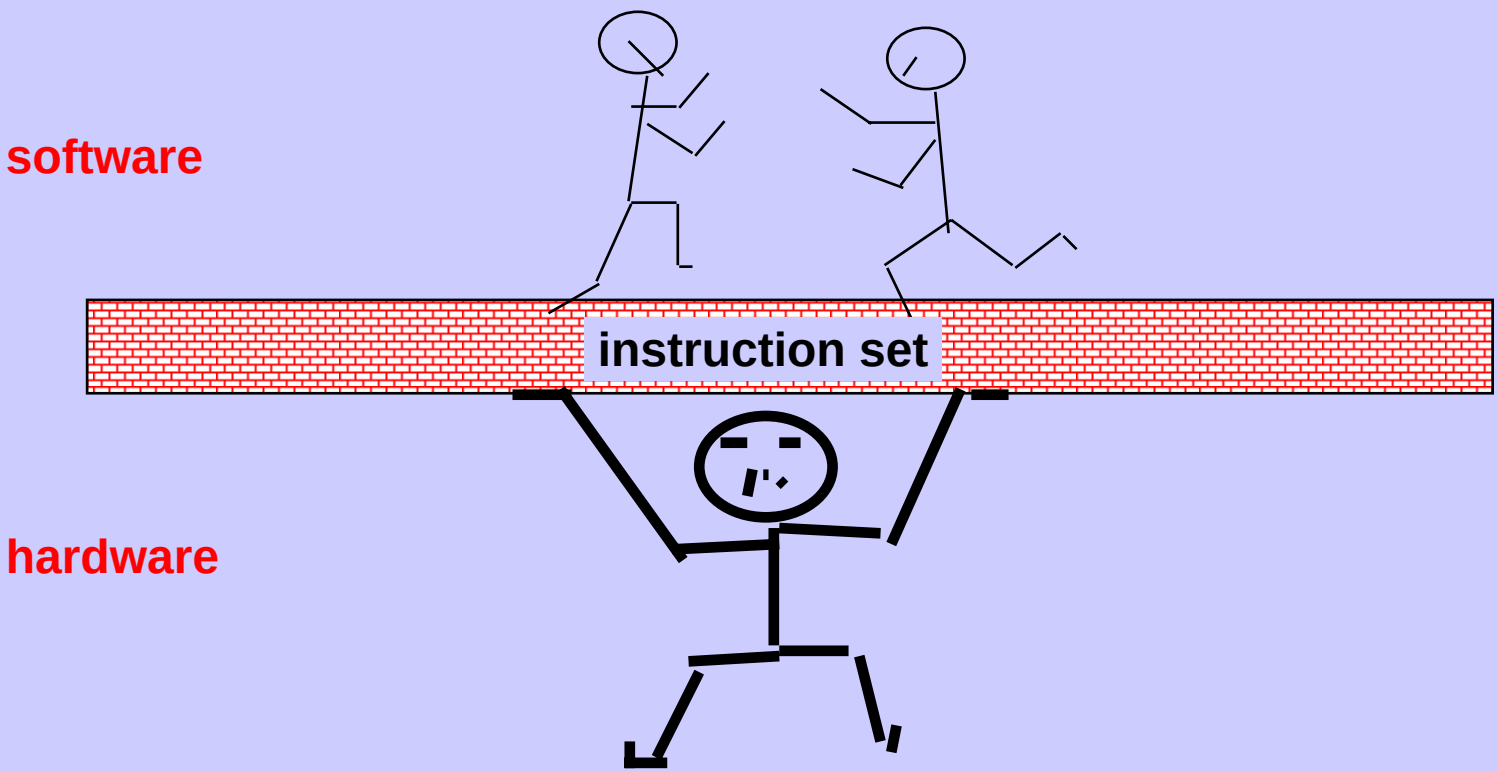
adresni na~in rada	simboli~ko obele`avanje	registarski prenos	efektivna adresa	sadr`aj Acc-a
direktni	`Lda ADR`	Acc ← M[ADR]	500	800
neposredni	`Lda #NOPR`	Acc ← NOPR	251	500
indirektni	`Lda [ADR]`	Acc ← M[M[ADR]]	800	300
relativni	`Lda $ADR`	Acc ← M[ADR+PC]	752	600
indeksni	`Lda (R1)ADR`	Acc ← M[ADR+R1]	900	200
registarski	`Lda R1`	Acc ← R1	-	400
registarsko-indirektni	`Lda (R1)`	Acc ← M[R1]	400	700

Tipovi instrukcija

V poglavlje

- Tipovi instrukcija -

The Instruction Set: a Critical Interface



Tipovi instrukcija

- Skup instrukcija predstavlja jednu od ključnih osobina arhitekture računara jer opisuje mogućnosti računarskog sistema.

- Skup instrukcija čini specifičan skup operacija koje dati sistem može da obavi.

Zahtevi koje treba da ispuni skup instrukcija na mašinskom jeziku

- **kompletan** – uz korišćenje razumljivog iznosa memorijskog prostora treba biti u stanju izračunati bilo koju funkciju
- **efikasan** – funkcije koje se često zahtevaju trebaju da se izvrše brzo
- **regularan** – skup instrukcija poseduje ortogonalne osobine
- **kompatibilan** – saglasan sa instrukcijama prethodnih mašina iz njegove familije
- **simetričan** – isti skup adresnih načina rada se može primeniti za pristup operandima u svim instrukcijama

Grupe mašinskih instrukcija

- Mašinske instrukcije možemo podeliti u sledeće grupe :
  - **prenos podataka**– vrše kopiranje informacije iz jedne lokacije u drugu
  - **aritmetičke**– obavljaju aritmetičke operacije nad numeričkim podacima
  - **logičke**– uključuju *Boole*-ove i druge nenumeričke operacije
  - **programsko upravljanje**- menjaju sekvencu programskog izvršenja
  - **ulazno-izlazne** – uzrokuju prenos informacija izmedju CPU-a ili glavne memorije i spoljnih U/I uredjaja
  - **konverzija** – koriste se za promenu formata podataka
  - **sistemsko-upravljačke** – koriste se od strane operativnog sistema

Top 10 80x86 Instructions

° Rank	instruction	Integer Average Percent total executed
1	load	22%
2	conditional branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	move register-register	4%
9	call	1%
10	return	1%
	Total	96%

° Simple instructions dominate instruction frequency

Instrukcije za prenos podataka

- Najosnovniji tip mašinskih instrukcija :
  - specificiraju lokacije izvornog i odredišnog operanda (lokacija može biti memorijska, registarska ili vrh magacina)
  - naznačava se dužina podatka kojim se manipuliše
  - specificira se način adresiranja za svaki operand

Prenos podataka

Move	prenos reči ili bloka od izvorišta ka odredištu
Store	prenos reči od CPU-a ka spoljnoj memoriji
Load	prenos reči iz spoljne memorije ka CPU-u
Exchg	izvorni i odredišni operand uzajamno promene sadržaj
Clr	u odredišni operand se upisuje "0"
Set	u odredišni operand se upisuje "1"
Push	prenos reči od izvorišta na vrh magacina
Pop	prenos reči sa vrha magacina u odredište

Examples of Data Movement Instructions

Instruct.	Meaning	Machine
MOV A, B	Move 16 bits from mem. loc. A to loc. B	VAX11
lwz R3, A	Move 32 bits from mem. loc. A to reg. R3	PPC601
li $3, 455	Load the 32 bit integer 455 into reg. 3	MIPS R3000
mov R4, dout	Move 16 bits from R4 to out port dout	DEC PDP11
IN, AL, KBD	Load a byte from in port KBD to accum.	Intel Pentium
LEA.L (A0), A2	Load address pointed to by A0 into A2	M68000

- Lots of variation, even with one instruction type
- Notice differences in direction of data flow left-to-right or right-to-left

Example Load & Store Instructions: Memory Addressing for SRC

- Address can be constant, constant+register, or constant+PC
- Memory contents or address itself can be loaded

Instruction	op	ra	rb	c1	Meaning	Addressing Mode
ld r1, 32	1	1	0	32	R[1] ← M[32]	Direct
ld r22, 24(r4)	1	22	4	24	R[22] ← M[24+R[4]]	Displacement
st r4, 0(r9)	3	4	9	0	M[R[9]] ← R[4]	Register indirect
la r7, 32	5	7	0	32	R[7] ← 32	Immediate
ldr r12, -48	2	12	–	-48	R[12] ← M[PC -48]	Relative
lar r3, 0	6	3	–	0	R[3] ← PC	Register (!)

(note use of la to load a constant)

Aritmetičke instrukcije

- Dvo-operandske (*dyadic*) – sabiranje, oduzimanje, množenje i deljenje
- Jedno-operandske (*monadic*) – apsolutna vrednost, negacija, inkrementiranje i dekrementiranje
- izvršavaju se od strane :
  - **ALU** (*Aritmetic-Logical Unit*)
  - **FPU** (*Floating Point Unit*)
  - **množača/delitelja** – specijalno projektovanog hardvera

Aritmetičke

Add	izračunava zbir dva operanda
Sub	izračunava razliku dva operanda
Mul	izračunava proizvod dva operanda
Div	izračunava količnik dva operanda
Abs	zamenjuje operand sa njegovom apsolutnom vrednošću
Neg	menja znak operanda
Inc	dodaje 1 operandu
Dec	oduzima 1 od operanda
FAdd	izračunava zbir dva FP operanda
FMul	izračunava proizvod dva FP operanda

Examples of ALU (Arithmetic and Logic Unit) Instructions

Instruction	Meaning	Machine
`MULF A,B,C`	multiply the 32-bit floating point values at mem loc'ns. A and B, store at C	VAX11
`nabs r3,r1`	Store abs value of r1 in r3	PPC601
`ori $2,$1,255`	Store logical OR of reg $ 1 with 255 into reg $2	MIPS R3000
`DEC R2`	Decrement the 16-bit value stored in reg R2	DEC PDP11
`SHL AX, 4`	Shift the 16-bit value in reg AX left by 4 bits	Intel 8086

- Notice again the complete dissimilarity of both syntax and semantics

Logičke i pomeračke

And **Or** **Not** **Xor**	obavljaju se specificirane logičke operacije I, ILI, NEGACIJA i ISKLJUČIVO ILI respektivno, nad svakom bit pozicijom
Shl **Shr**	Pomeranje operanda levo (SHL) ili desno (SHR) sa ubacivanjem konstante (obično "0“) na kraju
Rol **Ror**	kružno rotiranje operanda levo (ROL) ili desno (ROR)

Logičke Pomeračke

Assembly Language Forms of Arithmetic and Logic Instructions for SRC

Format	Example	Meaning
neg ra, rc	neg r1, r2	;Negate (r1 = -r2)
not ra, rc	not r2, r3	;Not (r2 = r3´)
add ra, rb, rc	add r2, r3, r4	;2's complement addition
sub ra, rb, rc		;2's complement subtraction
and ra, rb, rc		;Logical and
or ra, rb, rc		;Logical or
addi ra, rb, c2	addi r1, r3, 1	;Immediate 2's complement add
andi ra, rb, c2		;Immediate logical and
ori ra, rb, c2		;Immediate logical or

- Immediate subtract not needed since constant in addi may be negative

Instrukcije za promenu toka programskog izvršenja

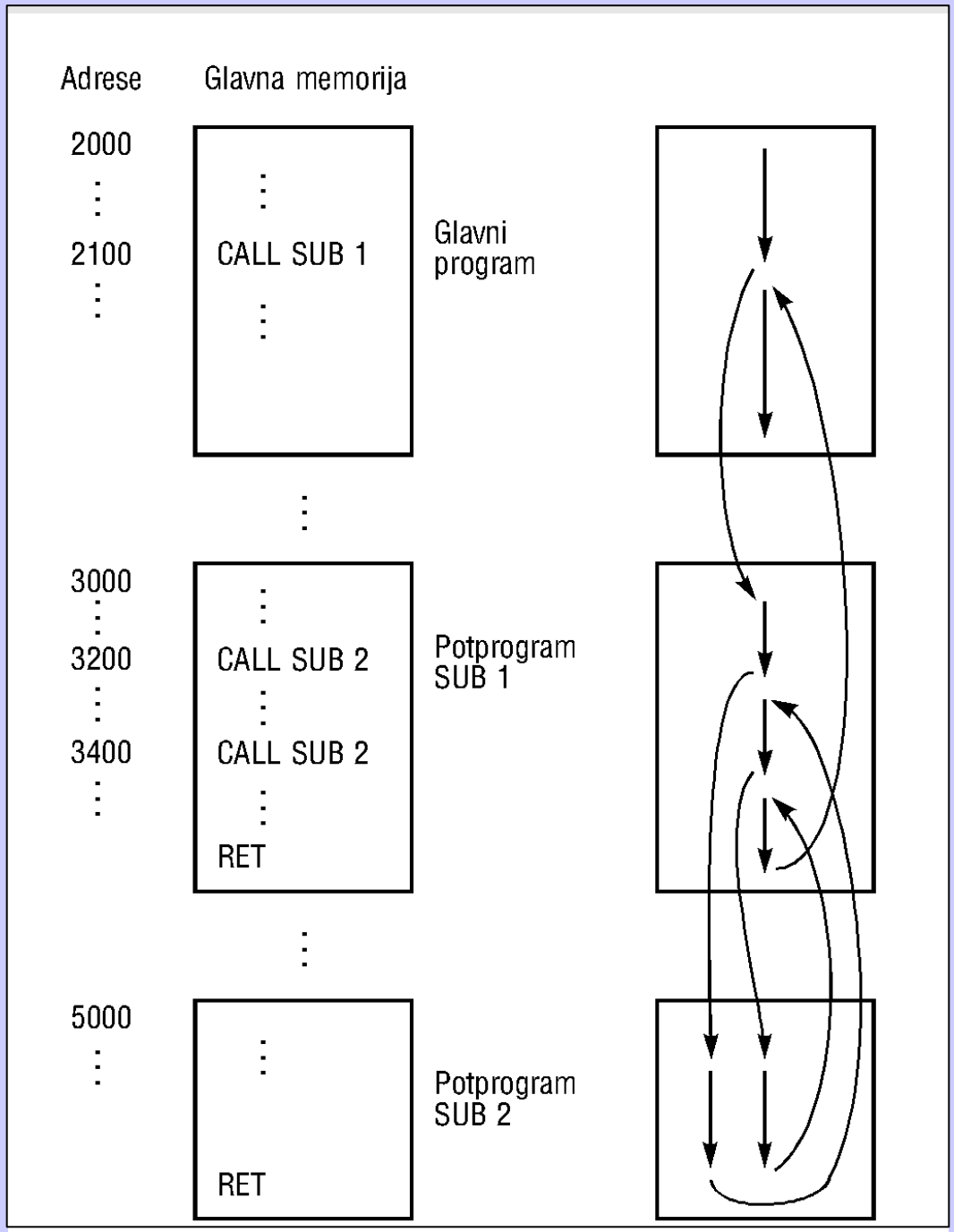
- **instrukcija grananja**
  - mogu biti bezuslovne (***Jump***) i uslovne (***Jump Conditional***)
- **instrukcija preskakanja**
  - ukazuje da se izvršenje jedne instrukcije u programskom toku mora preskočiti (***Branch Increment and Skip-if-Zero*** **-BRIZ**)
- **instrukcija za poziv potprograma**
  - poziv se realizuje sa **Call**, a povratak sa **Return**

Efekat gneždjenja kod poziva potprograma

- Mehanizam za poziv potprograma uključuje dve osnovne instrukcije:
  - instrukciju tipa poziva (*Call*) kojom se obavlja grananje sa tekuće lokacije na početnu lokaciju potprograma
  - instrukcija tipa povratak (*Ret*) kojom se vrši povratak iz potprograma na mesto odakle je taj potprogram bio pozvan

- Postoje tri mesta gde se mogu čuvati povratne adrese :
  - u registrima
  - na početku potprograma
  - na vrhu magacina

Efekat gneždjenja kod poziva potprograma



Prenos upravljanja

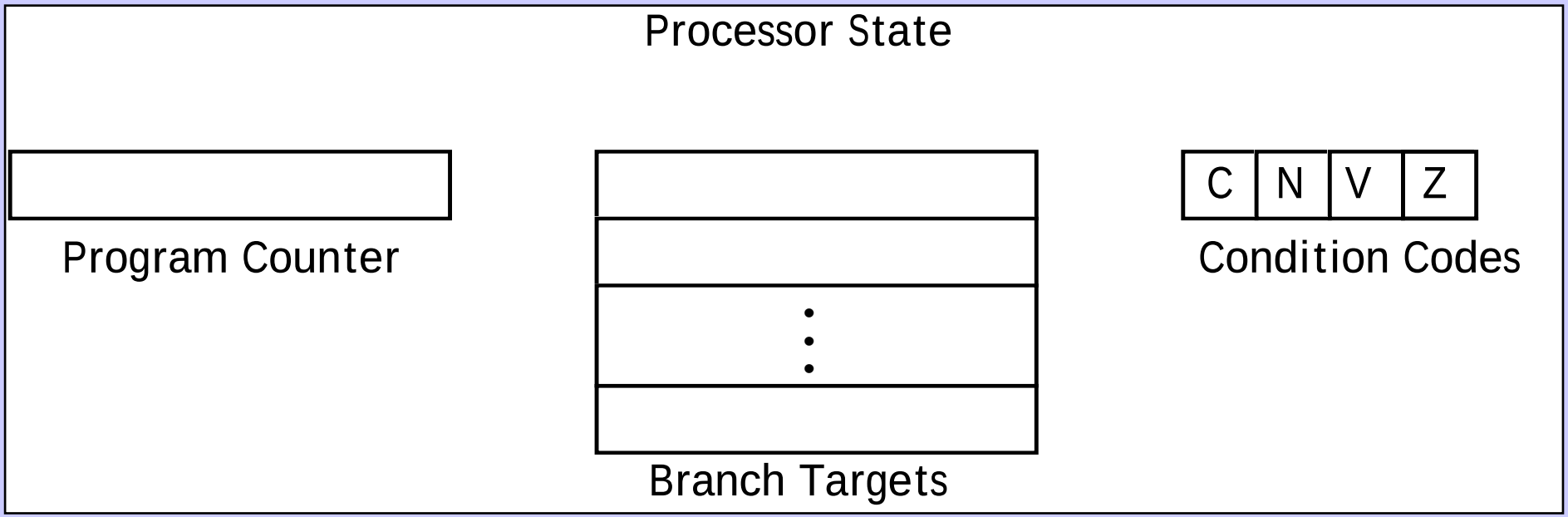
Jmp	bezuslovni skok; postavlja PC na specificiranu adresu
Jcond	testira specificirani uslov; u zavisnosti od uslova postavlja PC na specificiranu adresu ili prelazi na izvršenje naredne instrukcije
Call	postavlja tekuću programsku upravljačku informaciju (PC, statusni registar PSW i dr.) na poznatu lokaciju, tj. na vrh magacina
Bra	inkrementira PC sa ciljem da se preskoči izvršenje naredne instrukcije u programu
BrCond	testira specificirani uslov; postavlja marker(e) u zavisnosti od rezultata testa
Cmp	obavlja logičko ili aritmetičko poredjenje dva ili većeg broja operanada; postavlja marker(e) u zavisnosti od rezultata testa
Stc	naredba privilegovanog tipa; postavlja markere za dozvolu /zabranu prihvatanja zahteva za prekid, dozvolu rada tajmera
Hlt	stopira izvršenje programa
Wait	stopira izvršenje programa, neprekidno testira specificirani uslov, kada je uslov zadovoljen, nastavlja sa izvršenjem programa
Nop	operacija bez efekta, ali program nastavlja sa izvršenjem

Examples of Branch Instructions

Instruction	Meaning	Machine
BLSS A, Tgt	Branch to address Tgt if the least significant bit of mem loc'n. A is set (i.e. = 1)	VAX11
bun r2	Branch to location in R2 if result of previous floating point computation was Not a Number (NAN)	PPC601
beq $2, $1, 32	Branch to location (PC + 4 + 32) if contents of $1 and $2 are equal	MIPS R3000
SOB R4, Loop	Decrement R4 and branch to Loop if R4 $\neq 0$	DEC PDP11
JCXZ Addr	Jump to Addr if contents of register CX $= 0$.	Intel 8086

CPU Registers Associated with Flow of Control—Branch Insts.

- Program counter usually contains the address of, or "points to" the next instruction
- Condition codes may control branch
- Branch targets may be contained in separate registers



HLL Conditionals Implemented by Control Flow Change

- Conditions are computed by arithmetic instructions
- Program counter is changed to execute only instructions associated with true conditions

C language	Assembly language	
	CMP.W #5, NUM	;the comparison
if NUM==5 then SET=7	BNE L1	;conditional branch
	MOV.W #7, SET	;action if true
L1	...	;action if false

Branch Instruction Format for SRC

There are actually only two branch op codes:

```
br rb, rc, c3<2..0>          ; branch to R[rb] if R[rc] meets
                             ; the condition defined by c3<2..0>
brl ra, rb,  rc, c3<2..0>    ; R[ra] ← PC; branch as above
```

- **It is c3<2..0>, the 3 lsbs of c3, that governs what the branch condition is:**

lsbs	condition	Assy language form	Example
000	never	brlnv	brlnv r6
001	always	br, brl	br r5, brl r5
010	if rc = 0	brzr, brlzr	brzr r2, r4
011	if rc ≠ 0	brnz, brlnz	
100	if rc ≥ 0	brpl, brlpl	
101	if rc < 0	brmi, brlmi	

- **Note that branch target address is always in register R[rb].**
- **It must be placed there explicitly by a previous instruction.**

Branch Instructions—Example for SRC

C: goto Label3

SRC:

```
        lar  r0,  Label3    ; put branch target address into tgt reg.
        br r0               ; and branch
        • • •

Label3  • • •
```

Example of conditional branch

```
in C:   #define Cost 125
        if (X<0) then X = -X;
in SRC:
Cost    .equ    125         ;define symbolic constant
        .org    1000        ;next word will be loaded at address 1000(10)
X:      .dw     1           ;reserve 1 word for variable X
        .org    5000        ;program will be loaded at location 5000(10)
        lar     r31, Over   ;load address of "false" jump location
        ld      r1, X       ;load value of X into r1
        brpl    r31, r1     ;branch to Else if r1≥0
        neg     r1, r1      ;negate value
Over:   • • •               ;continue
```

Ulaz-izlaz

In	prenos podataka sa specificiranog U/I porta ili uredjaja u odredište tj. glavnu memoriju ili CPU-ov registar
Out	prenos podataka sa specificiranog izvorišta na U/I port ili uredjaj
Stio	prenos naredbe ka U/I procesoru sa ciljem da se inicira U/I operacija
Tstio	prenos statusne informacije iz U/I sistema u specificirano odredište

Konverzija

Tran	prevodjenje vrednosti u nekom delu memorije na osnovu tabele o korespondenciji
Conv	konverzija sadržaja reči iz jednog oblika u drugi (decimalno pakovani u binarni)

Upravljanje sistemom

Oscall	uzrokuje prekid koji prepušta upravljanje CPU-a operativnom sistemu

Akcije CPU-a u toku trazličitih tipova instrukcija

Prenos podatak	Prenos podatak iz jednog registra u drugi Ako je u prenos uključena memorija: – odredjuje se adresa memorijske lokacije – obavlja se virtuelno-fizička memorijska adresna transformacija – proverava keš memoriju – inicira čitanje/upis memorije
Aritmetičke	Moguće je prenos podataka pre ili posle: - obavljanja ALU i FPU funkcija - postavljanje uslovnih kôdova ili markera
Logičke	Iste aktivnosti kao i aritmetičke
Konverzija	Slične su aritmetičkim i logičkim instrukcijama. Može da postoji specijalna logika za obavljanje konverzije
Prenos upravljanja	Ažuriraju PC. Kod naredbe za poziv i povratak iz potprograma (CALL i RET) upravljaju prenosom parametara i povezivanjem
Ulaz-izlaz	Koriste kojmande za U/I module. Kada je U/I memorijsko-preslikanog tipa odredjuju adresu memorijskog preslikavanja

Uvod u asemblerski jezik

VI poglavlje

- Karakteristike -

Uvod u asemblerski jezik

Skup instrukcija mikroprocesora predstavlja skup binarnih ulaza koji generišu odredjene akcije u toku ciklusa jedne instrukcije

Primer :

Verzija instrukcije na mašinskom jeziku za mikroprocesor MC 68000 koja glasi:

"Napuni neposredni decimalni operand 2001 na adresu na koju ukazuje registar A“

ima sledeći 32-bitni format:

0010 0000 0111 1000 0000 0111 1101 0001

Asemblerski jezik

Primer :

Za mikroprocesor MC 68000 sekvenca :

"Izvrši sabiranje sadržaja dve memorijske lokacije 6000_{16} i 6002_{16}, a rezultat smesti u memorijsku lokaciju 6004_{16}"

imaće sledeći oblik:

0011 0000 0011 1000

0110 0000 0000 0000

1101 0000 0111 1000

0110 0000 0000 0000

0011 0001 1100 0000

0110 0000 0000 0100

Ovo predstavlja **mašinski jezik**, ili **objektni (*object*)** kôd (program) i njega će mašina direktno izvršavati, tj. ovakav program koga čini sekvenca 0 i 1 mašina jedino i razume

Asemblerski jezik - osobine

- Kreiranje programa na mašinskom jeziku se može iskazati kroz sledeća zapažanja:
  - programi su teški za razumevanje, veoma teško se nalaze greške, a iste nakon toga koriguju
  - instrukcije se teško unose u mašinu, kuca se bit po bit, a zatim se cela reč unosi u paralelnoj formi
  - programi ne poseduju komentar koji će programeru olakšati rad
  - programi su dugački i zamorni za analizu
  - programer često iz nehata čini greške koje je veoma teško locirati, a nakon toga korigovati

Asemblerski jezik – binarna forma

Preglednost programa se može poboljšati ako se binarana forma

```
0011 0000 0011 1000
0110 0000 0000 0000
1101 0000 0111 1000
0110 0000 0000 0010
0011 0001 1100 0000
0110 0000 0000 0100
```

napiše u heksadecimalnoj brojnoj notaciji

Asemblerski jezik – heksadecimalna forma

```
3 0 3 8
6 0 0 0
C 0 7 8
6 0 0 2
3 1 C 0
6 0 0 4
```

Program koji konvertuje heksadecimalne brojeve u binarne i smešta u memoriju se naziva "heksadecimalni punilac“ – (*hexadecimal loader*)

Asemblerski jezik - mnemonici

- Programsko poboljšanje se postiže dodeljivanjem imena svakom instrukcionom kôdu
- Ime kôda instrukcije zove se **mnemonik**
- Mnemonik opisuje sa minimalnim brojem znakova šta instrukcija radi

Primer :

`Add`	za sabiranje
`Sub`	za oduzimanje

Programska sekvenca za MC 68000, kada se koriste mnemonici, ima sledeći oblik

```
Move.W      $6000, D0
Add.W       $6000, D0
Move.W      D0,$6004
```

} program napisan na asemblerskom jeziku

Proces asembliranja

Za programsku sekvencu

```
Move.W          $6000, D0
Add.W           $6000, D0
Move.W          D0,$6004
```

kažemo da je napisana na asemblerskom jeziku

Asemblerski program (asembler) vrši prevodjenje korisničkog programa ili izvornog programa, napisanog pomoću mnemonika, u program na mašinskom jeziku, ili objektni program, koga će mikroračunar izvršavati

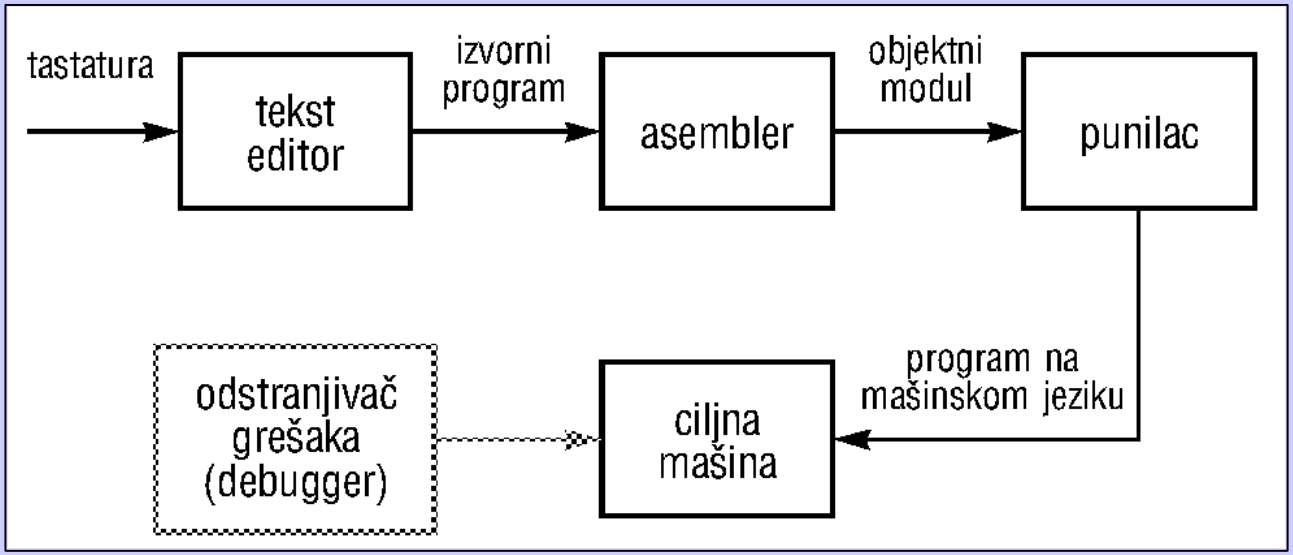
Ulaz u asembler je izvorni program

Izlaz iz asemblera je objektni (izvršivi) program

Proces razvoja programa na asemblerskom jeziku

- Programer koristi usluge tekst editora za kreiranje **tekst datoteke** (*text file*) koja sadrži program napisan na asemblerskom jeziku.

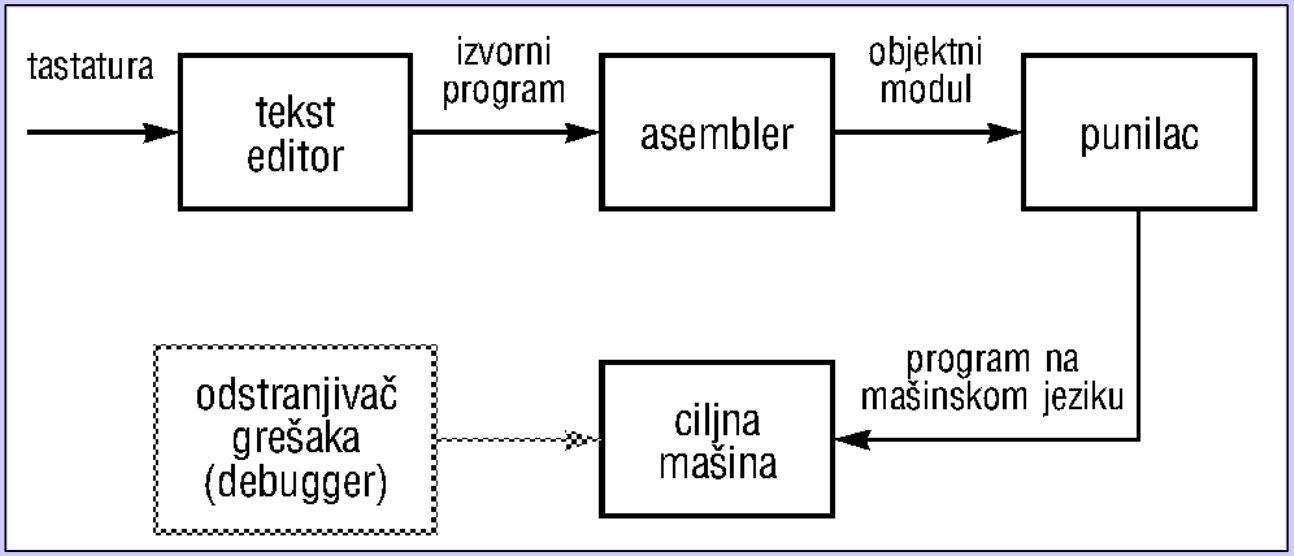
- Imena **izvorna datoteka** (*source file*) i **izvorni program** (*source program*) se često koriste u asemblerskom jeziku i jezicima višeg nivoa da označe tekst datoteke.



- Asembler prihvata izvorni program kao ulaz, vrši proveru formata zbog grešaka u pisanju i proizvodi objektni modul koji sadrži program napisan na mašinskom jeziku

Proces razvoja programa na asemblerskom jeziku

- **Punilac** (*loader*) smešta objektni modul u memoriju na kojoj će se program izvršavati (**ciljna mašina** - *target machine*)



- Na ciljnoj mašini program se testira pomoću programa za **odstranjivanje grešaka** (*debugger*)

Kros-asembler

- Punilac, odstranjiva~ grešaka i program na mašinskom jeziku moraju da se izvršavaju na ciljnoj mašini, dok tekst editor i asembler mogu se izvršavati na drugoj (razli~itoj) mašini.
- Asembler koji se izvršava na jednoj mašini, a proizvodi (generiše) objektne module za drugu mašinu zove se **kros-asembler** (*cross-assembler*).
- Proces **punjenja** (*loading*) objektnog modula sa mašine na kojoj je izvršeno kros asembliranje u ciljnu mašinu (*down load*) obi~no se izvodi serijskom komunikacionom vezom.

Karakteristike asemblerskog jezika

- Kada programer vrši programiranje na asemblerskom jeziku, on mora strogo da zna :
  - koji }e mikrora~unar koristiti.
  - koje registre i instrukcije doti~ni mikrora~unar poseduje,
  - koje adresne na~ine rada koristi, i puno drugih detalja.
- Asembler nije prenosiv
- Svaki mikrora~unar ima svoj sopstveni asemblerski jezik koji odra`ava njegovu sopstvenu arhitekturu.
- Asebmlerski program napisan za MC 68020 ne}e raditi na mikrora~unaru zasnovanom na iAPX486 ili bilo kom drugom mikroprocesoru

Karakteristike asemblerskog jezika

- produžetak-

- asemblerski jezik je jezik kod koga svaka instrukcija generiše ta~no jednu mašinsku instrukciju.
- postoji korespondencija jedan-prema-jedan izmedju mašinskih instrukcija i instrukcija na asemblerskom programu.
- postoji onoliko asemberskih jezika koliko postoji i razli~itih mikroprocesora

Karakteristike asemblerskog jezika

- nekonzistentnost-

- Za odredjivanje vrednosti algebarskog izraza $X=Y+Z$ kod jednih procesora se koristi instrukcija

```
Add X,Y,Z
```

a kod drugih (mikroprocesora) instrukcija

```
Add Y,Z,X
```

- Sagledavaju}i ograman broj nedostataka programiranja na asemblerskom jeziku udru`enje IEEE oformila je radnu grupu *IEEE P 694* koja je imala zadatak da standardizuje mnemonike asemblerskog jezika.

- Na `alost preporuke ove grupe nisu široko prihva}ene od strane industrije.

Karakteristike asemblerskog jezika

- pokušaj standardizacije-

- Nezavisno od toga što su pokušaji standardizacije propali ostali su tragovi tog rada za koje se ~ini da su pre}utno šire prihva}eni od strane proizvodja~a RISC procesora a koji se odnose na:

  - specifikaciju imena mnemonika
  - definiciju tipa modifikatora
  - uslove grananja
  - dogovor o koriš}enju formata ALU instrukcija

Karakteristike asemblerskog jezika

imena najčešće korišćenih mnemonika

mnemonik	operacija
Add	sabiranje
Sub	oduzimanje
Mul	množenje
Div	deljenje
Inc	inkrementiranje
Dec	dekrementiranje
Cmp	kompariranje
Move	kopiranje
Clr	brisanje (sadržaja registra)
Ld	punjenje (registra iz memorije)
St	smeštaj (sadržaja registra u memoriju)
Lda	punjenje adreses u registar
Ldm	punjenje većeg broja registara
Stm	smeštaj većeg broja registara
Shl	pomeranje u levo
Shr	pomeranje u desno
Br	bezuslovno grananje
Bc	uslovno grananje
Bal	grananje i povezivanje
Bal R	grananje na registar i povezivanje

Karakteristike asemblerskog jezika -modifikatori za tip podataka

- Instrukcijom:

```
Move.W   R1,R2     ;kopira sa sadr`aj 32 bitnog registra R2 u R1.
```

- Modifikatori koji prate odredjeni tip podataka mogu biti oblika:

modifikator	tip podatka
B	bajt (8-bitova)
UB	neoznačeni bajt (8-bitova)
H	polu-reč (16 bitova)
UH	neoznačena polu-reč (16-bitova)
W	reč (32 bita)
UW	neoznačena reč (32-bita)
F	pokretni zarez (32-bita)
D	dupla preciznost pokretni zarez (64-bita)
C	znak ili decimalni broj u 8-bitnom formatu
P	decimalni broj u pakovanom (4-bitnom) formatu

Karakteristike asemblerskog jezika

uslovi grananja

CC	uslovi
T	istinit
F	pogrešan
V	premašaj
C	prenos ili pozajmljivanje
PE	parna parnost
PO	neparna parnost
LE	manji od ili jednak
LT	manji od
EQ	jednak
NE	nije jednak
GE	veći od ili jednak
GT	veći od

Karakteristike asemblerskog jezika

korišćeni formati ALU instrukcija

slučaj 1	OP.X odredište, izvorište 1, izvorište 2 (tro-operandski format)
slučaj 2	OP.X odredište, izvorište (samo kada se specificiraju dva operanda)
slučaj 3	OP.X odredište/izvorište 1, izvorište 2 (u ovom slučaju OP koristi dva izvorna operanda, a upisuje rezultat na mesto operanda izvorište 1)

Formati asemblerskih instrukcija

- Struktura instrukcija na asemblerskom jeziku dosta verno odslikava strukturu mašinskih instrukcija koje on predstavlja
- Asemblerski jezici razli~itih procesora medjusobno su dosta sli~ni
- Ova osobina dozvoljava da se o njima u izvesnoj meri diskutira uopšteno
- Kada se dobro poznaje jedan asemblerski jezik lako se u~i drugi.

Formati asemblerskih instrukcija

- Svaki iskaz na asemblerskom jeziku ~ine slede}a ~etiri polja:
  - oznaka (labela)
  - opkôd ili direktiva
  - operand (i)
  - komentar

Polja			
oznaka	opkôd i direktiva (pseudoinstrukcija)	operand (i)	komentar
[< labela >]	<opkôd> ili <direktiva >	[<operand 1>[, <operand 2>]]	[<komentar>]

Formati asemblerskih instrukcija

- polje oznaka -

- **Oznaka** - za najve}i broj instrukcija i direktiva ovo polje je i opciono

  polje oznaka predstavlja simboli~ko ime za memorijsku adresu.

```
Izraz: Move.L  B,D0
```

- **Izraz** definiše lokaciju instrukcije u memoriji nakon što je program napunjen (loadovan).
- Oznaka se mo`e koristiti da definiše po~etak programskog segmenta radi kasnijeg obra}anja.
- Oznaka se koristi kod dodela memorije pseudoinstrukcijama jer omogu}ava da memorisani podaci budu dostupni preko simboli~kih imena.

Formati asemblerskih instrukcija

- polje operacioni kôd ili direktiva -

- Polje **operacioni kôd** ili **direktiva** sadrži:
  - mnemonik instrukcije- simboli~ku skra}enicu za opkôd ako iskaz predstavlja mašinsku instrukciju ili
  - asemblersku direktivu – tj. pseudoinstrukciju ako je iskaz komanda asembleru
  - izbor imena za mnemonike i direktive razlikuje se od jednog asemblera do drugog

Primer :

```
Add R1,R2,R3     ;instrukcija
Alfa EQU 50      ;direktiva
```

Formati asemblerskih instrukcija

- polje operandi -

- **Operandi** - polja operandi u iskazima asemblerskog jezika namenjena su da specificiraju adrese i registre koji se koriste kao operandi u mašinskoj instrukciji.
- Polje operand(i) kod instrukcije `Add` ukazuje šta sa ~ime treba sabrati.
- Polje operand kod `Jump (Branch)` instrukcije ukazuje gde treba sko~iti.
- Kod pseudoinstrukcija ovo polje ukazuje koliko memorijskog prostora treba rezervisati.
- Asembleri prepoznaju i odredjene simbole u polju operand (i). Simbolu mo`e da se dodeli apsolutna adresa, neposredna vrednost, ili bilo koji drugi va`e}i operand.

Formati asemblerskih instrukcija

- polje operandi - produžetak

- **Izraz** (*expression*) predstavlja kombinaciju simbola, konstanti (literali), algebarskih operatora i zagrada, a koristi se od strane asemblera za odredjivanje adresa ili vrednosti operanada.

Primer :

- Instrukcijom

```
Move.W  # $ 2000, D1
```

definiše se 16-bitna heksadecimalna vrednost 2000 kao neposredni izvorni operand.

- Konstante za koje asembler automatski rezerviše memorijski prostor zovemo **literali**

Asemblerske direktive

- Alternativno ime je asemblerske pseudoinstrukcije
- Najve}i broj asemblera pru`a pomo} programeru na razli~ite na~ine putem koriš}enja **asemberskih direktiva**

Direktive su instrukcije namenjene asembleru, a ne procesoru

- Akcija koja se preuzima od strane direktiva egzistira samo u toku asembliranja izvornog programa, tj. direktive su neizvršive asemblerske instrukcije

tip	opkôd	komentar
upravljanje asembliranjem	Org<izraz>	početak programske sekvence
	End	kraj programa koji se asemblira
definicija simbola	<oznaka> Equ<izraz>	izjednačava simboličko ime (na poziciji oznaka) sa vrednošću operanda
definisanje podataka i memorijskog prostora	[<oznaka>] DC.<I> <vrednost(i)>	definiše konstantu(e) tj. memoriše vrednosti operanda kao konstante
	[<oznaka>] DS.<I> <vrednost(i)>	rezerviše memorijski prostor
makro definicije	Macro	početak makro definicije
	Endm	kraj makro definicije
upravljanje listingom	Lien	dužina linije
	List	listanje
	NoList	nema listanja
	Spc<N>	<N> blanko linija
	Page	naredna stranica
ostale	Title	koristi operand kao naslov na svakoj stranici listinga
	If	početak asembliranja uslovnog bloka instrukcije samo ako je specificirani uslov ispunjen
	Endif	kraj uslovnog bloka

Proces asembliranja

- Proces asembliranja, kao aktivnost, je dovoljno sličan i može se kao postupak opisati da važi za sve mašine, tj. bude generalni
- Asembliranje možemo da zamislimo na sledeći način:
  - program na asemblerskom jeziku predstavlja niz iskaza
  - asembler u datom trenutku čita jedan iskaz, prevodi ga u mašinski jezik, a na svom izlazu generiše dva tipa informacija:
    - ✓ odnosi se na generisani mašinski kôd i pridružuje se datoteci u kojoj se čuva izvršivi mašinski kôd,
    - ✓ dodaje se odgovarajućem delu listing datoteke koja se koristi za potrebe štampanja.
- Proces se ponavlja dok se ne prevede ceo program.

Proces asembliranja - produžetak

- Na prvi pogled veoma logičan postupak koji na `alost nije korektan. Zbog ~ega?

Analizirajmo slede}u situaciju:

- Neka prvi iskaz u programu koji se prevodi bude `Jump Adr_1`. Asembler ne}e biti u stanju da asemblira ovaj iskaz sve dok nezna adresu *Adr_1*. Skok mo`e biti na kraju programa koji se prevodi, što zna~i da asembler ne}e biti u stanju da pronadje adresu ako se prethodno ne pro~ita ceo program.

- Ovaj problem je poznat kao **obra}anje unapred** (*forward reference*) a javlja se zbog toga što se simbol *Adr_1* javlja pre nego što je definisan, tj. obra}amo se simbolu ~ija definicja sledi kasnije.

Problem obraćanja unapred – prvi pristup

- Asembler ~ita izvorni program dva puta. Pri ovome svako ~itanje izvornog programa zovemo **prolaz** (*pass*). Bilo koji prevodilac koji ~ita ulazni program dva puta zove se **dvo-prolazni prevodilac**.

  - U toku prvog prolaza, kod dvo prolaznog asemblera, skupljaju se i sme{taju u tabeli definicije simbola i oznake (labele) iskaza

  - U toku drugog prolaza vrednosti svih simbola su poznate, problem obra}anja unapred nije vi{e kriti~an, svaki iskaz se ~ita, asemblira i generi{e njegov izlaz.

Problem obraćanja unapred – drugi pristup

- Drugi pristup se zasniva na jednom prolazu. Ideja je slede}a:
- Kada se naidje na iskaz koji se ne mo`e asemblirati zbog toga {to postoji obra}anje unapred ne generi{e se izlaz. Umesto toga kreira se ulaz u (jednoj) tabeli ~iji sadr`aj ukazuje to da iskaz kod koga se javlja obra}anje unapred nije asembliran. Na kraju procesa asembliranja svi iskazi u neasembliranoj tabeli se asembliraju.

Nedostaci:

- dvo prolaznog asemblera ogleda se u tome {to zahteva ne{to du`e vreme prevodjenja
- jedno prolaznog: (a) dodatni memorijski prostor, koji mo`e biti veliki ako u programu postoji veliki broj obra}anja unapred; (b) slo`enija struktura; i (c) razumljivost listinga objektnog koda nije uvek najjasnija. Zbog nabrojanih razloga najve}i broj asemblera je dvo prolazni.

Prvi prolaz

Glavna aktivnost koja se obavlja u toku prvog prolaza u vezi je sa kreiranjem **tabele simbola**. U tabeli se ~uvaju vrednosti svih simbola. Simbol mo`e biti: (a) oznaka (labela); i (b) vrednost kojoj je pomo}u direktiva dodeljeno simboli~no ime, kao na primer:

```
DISK_1   Equ   18
```

Kada simbol pripada polju Oznaka (slika 6.7) asembler mora da zna, koju adresu treba da dodeli toj instrukciji u toku izvr{enja programa. Da bi sa~uvao adresni trag o svim asembliranim instrukcija asembler mora u svakom ulazu tabele simbola da pamti vrednost jedne promenljive koja se zove **lokacioni broja~ instrukcija** (*instruction location counter - ILC*). Na po~etku prvog prolaza asembler postavi ILC na vrednost 0. Nakon obrade svake instrukcije ILC se inkrementira za du`inu te instrukcije.

Tabela simbola

- **Tabela simbola** - ima po jedan ulaz za svaki simbol
- Simboli su definisani bilo kao oznake (labele) ili preko definicija (Januar Equ 1)
- Svaka tabela simbola sadr`i sam simbol (ili pokaziva~ na taj simbol), njegovu numeri~ku vrednost i dodatnu informaciju koja se odnosi na:
  - du`inu polja podatka koje prati taj simbol,
  - relokacione bitove, tj. da li simbol menja vrednost ako se program puni na neku drugu adresu u odnosu na onu koja je specificirana asemblerom
  - da li je ili ne simbol dostupan van granica procedure.

Opkôd tabela

- **Opkôd tabela** - sadr`i najmanje jedan ulaz za svaki mnemonik asemblerskog jezika
- Svaki ulaz opkôd tabele sadr`i:
  - simboli~ki opkôd (mnemonik)
  - dva operanda
  - du`inu instrukcije
  - klasu instrukcije koja deli opkodove u grupe u zavisnosti od broja i tipova operanada

Drugi prolaz

- U ovom prolazu obavljaju se slede}e aktivnosti:
  - generi{e se objektni program
  - {tampa se listing na asemblerskom jeziku ako se to zahteva
  - generi{e se dodatna informacija koja je potrebna programu za povezivanje (linkeru) radi povezivanja procedura koje se asembliraju u razli~itim vremenskim trenucima

Makro naredbe i potprogrami

- Programeri ~esto imaju potrebu da nekoliko puta u okviru programa ponavljaju odredjene sekvence
- Najjednostavnije ovoj problem se re{ava tako {to se sekvenca instrukcija kad god zatreba prepi{e
- Ako je sekvenca duga~ka, ili se ~esto ponavlja, ovaj posao postaje zaista dosadan
- Kao alternative se nude sledeća dva re{enja koja se baziraju na grupisanju instrukcija u jedinstvenu celinu :

  - **makro naredbe**
  - **potprogrami (procedure)**

Makro naredbe i potprogrami - produžetak

- Obe strategije ima svoje prednosti i nedostatke
- Kod poziva procedure izvr{avaju instrukcije `Call` i `Return`
- Ako je procedura kratka, a poziv procedure ~esti, izvr{enje instrukcija `Call` i `Return` u zna~ajnoj meri usporava brzinu izvr{enja programa
- Makro naredbe obezbedjuju efikasnije re{enje problema u~estalog izvr{enja iste ili skoro iste sekvence instrukcija
- Nedostatak makro naredbe se ogleda u pove}anju obima programa.

Makro naredbe

- Veliki broj asemblerskih jezika je pro{irljiv
- Asembleri dozvoljavaju korisniku da defini{e nove instrukcione mnemonike koje zovemo **makro naredbe**
- Makro naredbe se mogu smatrati kao sastavni deo jezika, pa se kao takve i mogu pozivati
- Novo definisane instrukcije mogu se koristiti potpuno ravnopravno sa svim ostalim instrukcijama iz skupa instrukcija na asemlerskom jeziku
- Makro naredba predstavlja sekvencu ma{inskih instrukcija koja se defini{e jedanput, a mo`e se koristiti po nekoliko puta u toku izvornog programa.

Makro naredbe - produžetak

- Definicija makro naredbe li~i na definiciju procedure
- Procedure i makro naredbe nisu isto
- Razlika se ogleda u slede}em:
  - kori{}enjem makro naredbi ne {tedi se memorijski prostor
  - na mestima na kojima se poziva marko naredba ubacije se sekvenca ma{inskih instrukcija, koje ~ine makro naredbe, ~ime se obim programa pove}ava.
  - makro naredbe nije potrebno povezivati, tj. linkovati.
  - makro naredbe su pogodnije za primenu kada se u toku programa javljaju kratke celine po nekoliko puta, a procedure kada su celine du`e.

Definicija makro naredbe

- Definicija makro naredbe predstavlja metod dodele imena delu programa
- Ra`li~iti asembleri koriste razli~itu notaciju kod definicje makro naredbi
- Nezavisno od detalja svi imaju slede}e tri osnovne celine:
  - zaglavlje u kome se javlja ime makro naredbe koja se defini{e,
  - tekst koji ~ini telo makro naredbe,
  - direktiva koja se odnosi na kraj definicije.

Tri celine kod definicije makro naredbe

ime `Makro` operand, . . . , operand

.

. } telo makro naredbe

.

`Endm`

Makro naredbe -produžetak

- Kada se ime makro naredbe koristi kao opkôd tada je ono poznato kao **makro poziv** a njegova ekspanzija telom makro naredbe zove se **makro ekspanzija**

- Definicija makroa uklju~uje zadavanje **formalnih** parametara, a makro pozivom se predaju **aktuelni** parametri.

Procedure

Procedura je osnovna programska jedinica kod najve}eg broja *HLL*-ova.

- Kori{}enjem procedure mo`e da pobolj{a ~itljivost, pisanje, i odr`avanje velikih programa
- Procedure takodje smanjuju veli~inu programa jer je njihov kôd deljiv, tj. zajedni~ki za ve}i broj programa
- Upotreba procedura je takodje svojstvena kod uvodjenja novih programskih tehnika kao {to je rekurzivno programiranje ~ime se u zna~ajnoj meri smanjuju napori programera kod realizacije odredjenih algoritama
- Lako prepoznajemo kori{}enje procedura u skoro svim programima koje danas sre}emo.

Procedure - produžetak

- implementacija procedura podrazumeva upravljanje nekim strukturama podataka koje zbog poziva procedura unose dodatno vremensko prekora~enje, tj. usporavaju brzinu izvr{enja programa

- savremeni mikroprocesori poseduju specijalne instrukcije koje izvode ove aktivnosti, a njihov efekat se ogleda u pobolj{anju performansi

Aktivnosti kod poziva procedura

- Prenos parametara proceduri (preko magacina, registara, itd)
- Predaja upravljanja `eljenoj proceduri
- Dodela prostora radi memorisanja lokalno promenljivih,
- Pam}enje konteksta programa iz koga je izvr{en poziv,
- Pristup parametrima u toku izvr{enja procedure,
- Obnavljanje konteksta programa iz koga je izvr{en poziv,
- Oslobadjanje prostora koji je bio rezervisan za memorisanje lokalno promenljivih,
- Izbavljanje parametara (iz magacina, registara, itd),
- Vra}anje upravljanja programu iz koga je izvr{en poziv.

Primer kori{}enje procedura kod Pascala

```
procedure P(X : integer; var y : integer);
        var z : integer;
        begin
        .
        y:=z+x
        .
        end;
        program (glavni)
        .
        P(A,B);
        .
        end
```

Prenos upravljanja izmedju programa i potprograma

- Metod prenosa upravljanje izmedju programa iz koga je izvr{en poziv i potprograma zove se **povezivanje potprograma** (*subroutine linkage*)
- Informacija koja je potrebna potprogramu defini{e se u zavisnosti od **parametara**
- Definicaija procedure uklju~uje **formalne** ili **fiktivne parametre**, a pozivom procedure se predaju **aktuelni** ili **stvarni parametri**
- Alternativno formalne parametre zovemo **parametri**, a za stvarne parametre se koristi termin **argumenti**

Tehnike za prenos vrednosti ili adresa

tip	opis	komentar
registarski	rutina iz koje se vrši poziv puni unapred definisane registre vrednostima ili adresama	• broj parametara je ograničen • predaja je dinamička
magacin	adresa iz koje se vrši poziv smešta vrednosti ili adrese magacina	• adresa povratka u toku obrade mora biti sačuvana, a obnovljena pre povrataka • predaja dinamička
parametarske oblasti	definišu se oblasti u memoriji u kojima se čuvaju vrednosti ili adrese	• predaja statička , ako su oblasti definisane u toku asembliranja, • predaja je dinamička ako se bazna adresa oblasti predaje u registru
in line	vrednosti ili adrese se memorišu nakon instrukcije za poziv potprograma. Potprogram izračunava lokacije parametara	• predaja je statička

Tipovi argumenata i parametri koji se prenose

- Argumenti koji se predaju potprogramu zove se **ulazni parametri**
- Dobijeni rezultati u formi vrednosti ili adrese odgovaraju **izlaznim parametrima** potprograma
- Parametri koji se prenose izmedju pozivnog i pozvanog programa mogu se predavati kao:
  - **vrednosti** (*by value*) - pozvanom programu se predaje stvarna vrednost
  - **adrese** (*by reference*) - stvarni parametar se predaje kao adresa

Registarski prenos

- Najjednostavniji metod za prenos argumenata
- Podaci se predaju preko internih registara CPU-a
- Brzi prenos podataka
- Broj argumenata koji se mo`e preneti je ograni~en brojem dostupnih registara
- Projektanti oba programa moraju prethodno da se dogovore preko kojih registara }e se prenosti argumenti

Registarski prenos

Primer :

Slede}om sekvencom instrukcija (va`i za MC 68020)

```
        .
Move.L      vred_1,D1          ; prenos vrednosti
Move.L      adr_x,A1           ; prenos adrese
Lea         zaglav, A2         ; adresa zaglavlja
Jsr         PP_1               ; poziv potprograma
        .
```

postavlja se 32-bitna vrednost u D1, adresa pokaziva~a na lokaciju *adrese* u A1, a adresa *zaglav* u A2. Potprogram PP_1 mo`e kada obavlja svoju funkciju direktno da pristupa registrima

Prenos preko magacina

- Argumenti se sme{taju u magacin pre nego {to se izvr{i poziv potprograma

- Kod sme{tanja podataka u magacin registar SP se prvo dekrementira, za vrednost koja je odredjena obimom podatka koga treba zapamtiti, a nakon toga se memori{e podatak

Prenos preko magacina

Primer :

Argumenti se sme{taju u magacin pre nego {to se izvr{i poziv potprograma na slede}i na~in:

```
        .
Move.L      vred_1,-(SP)     ; sme{ta se prvi podatak
Move.W      vred_2,-(SP)     ; sme{ta se drugi podatak
Jsr         PP_A             ; poziv potprograma PP_A
        .
```

- na vrh magacina pokazuje registar SP
- registar SP uvek pokazuje na punu lokaciju

Prenos preko magacina

Primer :

Sekvenca instrukcija, mikroprocesora MC68020, koja obavlja ovu aktivnost ima slede}i oblik:

```
PP_A:   Move.L   (SP)+,A1; privremeno sa~uvaj (PC) u A1
        Move.W   (SP)+,D1      ; smesti vred_2 u D1
        Move.L   (SP)+,D2      ; smesti vred_1 u D1
                 .
                 .                    }   telo potprograma PP_A
                 .
        Move.L   (A1),-(SP)    ;smesti (PC) u magacin
        Rts                    ;povratak iz potprograma
```

Prenos preko magacina- produžetak

- **Prednosti :**
  - broj argumenata koji se prenosi mo`e biti veliki
  - metoda je univerzalna jer magacin poseduju svi procesori
  - ova metoda prenosa argumenata koristi se od strane velikog broja kompilatora, kakvi su C, Pascal, i dr.
- **Nedostaci :**
  - s obzirom da se pristupa memorijskim lokacijama prenos je sporiji u odnosu na registarski
  - ako je broj poziva potprograma ili broj podataka koji se prenosi veliki, a dodeljen prostor magacinu mali, tada mo`e da dodje do prekora~enja opsega magacina. Ovo neminovno dovodi do gubitaka podataka, a time i do kraha rada sistema.

Prenos podataka preko memorijskih lokacija

- Kada je broj parametara koji se prenosi veliki, parametarsku oblast je bolje postaviti u memoriji
- Ova oblast sadr`i, po unapred odredjenoj sekvenci, vrednosti ili adrese kojima se pristupa od strane potprograma nakon {to je njemu predata po~etna adresa ove oblasti
- Ista oblast se mo`e koristiti od strane nekoliko potprograma pod uslovom da je ona dovoljno velika da mo`e ~uvati sve podatke
- Potprogram mo`e da pristupa vrednostima koriste}i indirektno adresiranje sa razme{tajem

Prenos podataka preko memorijskih lokacija

Primer :

Sekvenca za prenos pet podataka preko memorijskih lokacija, za mikroprocesor MC 68020, ima}e oblik

```
                .
Move.L    vred_1, memo_ob          ; sme{taj prvog podatka
Move.W    vred_2, memo_ob+4        ; sme{tj drugog podatka
Move.W    vred_3, memo_ob+6        ; sme{taj tre}eg podatka
Move.L    vred_4, memo_ob+8        ; sme{taj ~etvrtog podatka
Move.W    vred_5, memo_ob+12       ; sme{taj petog podatka
Lea       memo_ob, A1          ; smesti adresu oblasti u A1
Jsr       PP_2                     ; poziv potprograma
```

Primer -produ`etak:

```
            .
memo_ob:    DS.L 1              ; rezervacija prostora
            DS.W 2
            DS.L 1
            DS.W 1
            End
```

instrukcijom

```
        Move.L (8,A1),D1
```

vr{i se prenos *vred_4* u D1

definisanje parametarske oblasti se mo`e obaviti na ve}i broj na~ina

"*In-line*" kodiranje

- Metod za prenos vrednosti potprogramu zasniva se na kôdiranju vrednosti koja sledi nakon poziva potprograma
- Ovaj metod se zove ***in-line* kodiranje** i defini{e vrednosti argumenata koje su konstante i ne menjaju se nakon asembliranja

"*In-line*" kodiranje

Primer :

Sekvencom instrukcija:

```
Jsr     PP_3        ; poziv potprograma PP_3
DC.W  1             ; in line argument
```

32-bitna vrednost PC-a koja se sme{ta u sistemski magacin nakon poziva potprograma ukazuje na lokaciju argumenta u instrukcionoj sekvenci.

"*In-line*" kodiranje

Primer :

Slede}om sekvencom koja je deo potprograma PP_3, sme{tamo argumet u LS deo registra D1, a zatim u magacin sme{tamo povratnu adresu koja se nalazi iza konstante:

```
Move.L      (SP), A0        ; izbavi (PC) i smesti ga u A0
Move.W      (A0)+, D1       ; pribavi argumet i inkrementiraj A0
Move.L       A0, (SP)       ; smesti novu povratnu adresu
.
.                            } telo potprograma PP_3
.
Rts                          ; povratak iz potprograma
```

Projektovanje modularnih programa i dokumentacija

- Kreiranje i dokumentovanje programa je od izuzetne va`nosti za programere
- Promena funkcionalnih zahteva nekog programskog modula iziskuje modifikaciju prethodno kreiranog kôda
- Dobro dokumentovani program mo`e da u{tedi mnogo vremena koje je potrebno za analizu postoje}eg kôda
- Svaka organizacija na jedinstven na~in formira svoj sopstveni standard o vodjenju dokumentacije

Projektovanje modularnih programa i dokumentacija- produžetak

- Klju~ne komponente, minimalnog standarda koga treba ispo{tovati, su :
  - funkcionalni opis
  - algoritamski opis
  - tabela o kori{}enju registara
  - *in line* dokumentacija

Funkcionalni opis

- **Funkcionalni opis**: opisuje **{ta** odredjena funkcija obavlja, ali ne daje informaciju **kako se** ta funkcija izvr{ava
- Funkcionalni opis mora da objasni na koji na~in se argumenti prenose toj funkciji, i na koji na~in se rezultati vra}aju (ako postoje)

Primer :

Funkcionalni opis

Hexout (R1 : vrednost)

32-bitna binarna vrednost se prenosi funkciji u registru R1, a heksadecimalna vrednost se {tampa sa desnim poravnjanjem.

Decout (R1 : vrednost)

32-bitna binarna vrednost se prenosi funkciji u registru R1, a decimalna vrednost se {tampa sa desnim poravnjanjem.

Decin (R1 : vrednost, R2 : status)

^ita se niz decimalnih cifara sa tastature i vra}a 32-bitni binarni ekvivalenat u registar R1. Ako niz ne predstavlja na korektan na~in decimalni broj tada se vrednost "1" kao tip statusne gre{ke vra}a u registar R2, ina~e, kada je decimalni broj va`e}i, u registar R2 se vra}a vrednost "0".

Hexin (& niz, vrednost)

Analizira se niz ASCII karaktera koji predstavlja heksadecimalni broj i vra}a se 32-bitna binarna ekvivalentna vrednost u magacin na lokaciji Mem[SP-4]. Pokaziva~ na niz se prenosi funkciji preko magacina na lokaciji Mem[SP]. Nakon povratka, pokaziva~ u Mem[SP] pokaziva}e na bajt koji sledi nakon zadnje heksadecimalne cifre u nizu.

Algoritamski opis

Pseudokôd: obja{njava na koji na~in se implementira funkcija

📖 svaki programer kome se dodeljuje zadatak da modifikuje kôd izuzetno je zainteresovan za logi~ku strukturu postoje}eg kôda

📖 obi~no logi~ku strukturu kôda lak{e je razumeti ako se koristi neka HLL notacija

📖 kori{}enje HL pseudokôda od izuzetne je va`nosti u toku inicijalnog razvoja kôdnog modula kao i od velike pomo}i programeru koji se bavi odr`avanjem

📖 pseudokôd olak{ava saradnju individua zadu`enih za modifikacije, a takodje i olak{ava debagiranje

📖 na osnovu svega izlo`enog, preporu~uje se da, pseudokôd bude sastavni deo dokumentacije programa na asemblerskom jeziku

Algoritamski opis

Primer :

Jedan tipi~an primer koji uklju~uje funkcionalni opis kao i opis algoritma na nivou pseudokôda imao bi slede}i oblik:

```
;
#############################################################################
; Ime programa: Suma integer vrednosti
; Programer: Mika Rus
; Datum zadnje modifikacije: 12.01.2004.
;
#############################################################################
```

Algoritamski opis

```
##########################################################################
; Funkcionalni opis:
; Program odredjuje sumu celobrojnih vrednosti (integer) od 1 do N, gde je N vrednost
; koja se prihvata (~ita) sa tastature
; ##########################################################################
; Opis algoritma na pseudokod nivou
; main: cout << "\n Uneti ulaznu vrednost za N =  "
        cin >> v0
        if (v0 > 0)
        {t0 = 0;
        while (v0 > 0) do
        {t0 = t0 + v0;
        v0 = v0 - 1}
        cout << "Zbir integer-a od 1 do N je", t0;
        go to main;
        }
        else
        cout << "\n Ovo je kraj opisa"
; ##########################################################################
```

Tabela o korišćenju registara

- **Tabela o korišćenju registara**: U okviru granica algoritma ovom tabelom se definiše način korišćenja svakog od registara procesora i premošćava se jaz izmedju opisnih imena promenljivih i odgovarajućih registara procesora

Tabela o korišćenju registara

Primer :

Jedan tipičan primer zaglavlja glavnog programa koji koristi tabelu o korišćenju registara bio bi sledećeg oblika:

```
;####################Primer zaglavlja glavnog programa#####################
; Program 1: <opisno ime>
; Programer: <ime i prezime>
; Dan kreiranja: dan, mesec, godina
; Zadnja modifikacija: dan, mesec, godina
; ###########################################################################
; Funkcionalni opis programa
; Sledi kratak opis programa
; ###########################################################################
; Kori{}enje registara u glavnom programu
; R1 = po~etna adresa niza .....
; R2 = vrednost koja se dobija sabiranjem .....
; ###########################################################################
; Opis pseudokoda
; ###########################################################################
```

Tabela o korišćenju registara

Primer :

Drugi tipi~ni primer koji bi se odnosio na opis funkcije bio bi slede}eg oblika:

```
; ################### Primer zaglavlja funkcije ###################
; Ime funkcije: <opisno ime>
; Programer: <ime i prezime>
; Dan kreiranja: dan, mesec, godina
; Zadnja modifikacija: dan, mesec, godina
; #################################################################
; Funkcionalni opis:
; Sledi kratak opis:
; #################################################################
; Obja{njenje koji se parametri prenose funkciji i na koji na~in
; R1 = pokaziva~ niza u memoriji
; Obja{njenje koje se vrednosti vra}aju od strane funkcije i kako
; R2 = binarna vrednost rezultata izra~unavanja
; R3 = vrednost statusa, "1" - rezultat ima korektnu vrednost, "0" - rezultat nema
; korektnu vrednost
; Primer pozivne sekvence:
; <prikaz prenosa parametara preko registara, magacina, .....>
; JSR xxxx      poziv potprograma
; <povratak na ovo mesto .....>
; #################################################################
; Na~in kori{}enja registara u funkciji:
; R5 = adresa lokacije .....
; R6 = privremena suma
; #################################################################
; Algoritamski opis na nivou pseudokoda:
; sledi opis
; #################################################################
```

"In line" dokumentacija

- **In line dokumentacija**: Ovaj tip dokumentacije je od velike koristi u fazi identifikacije koju aktivnost svaki od blokova na asemblerskom jeziku obavlja

"In line" dokumentacija

Primer :

Jedan tipičan primer kori{}enja *in line* dokumentacije je slede}i:

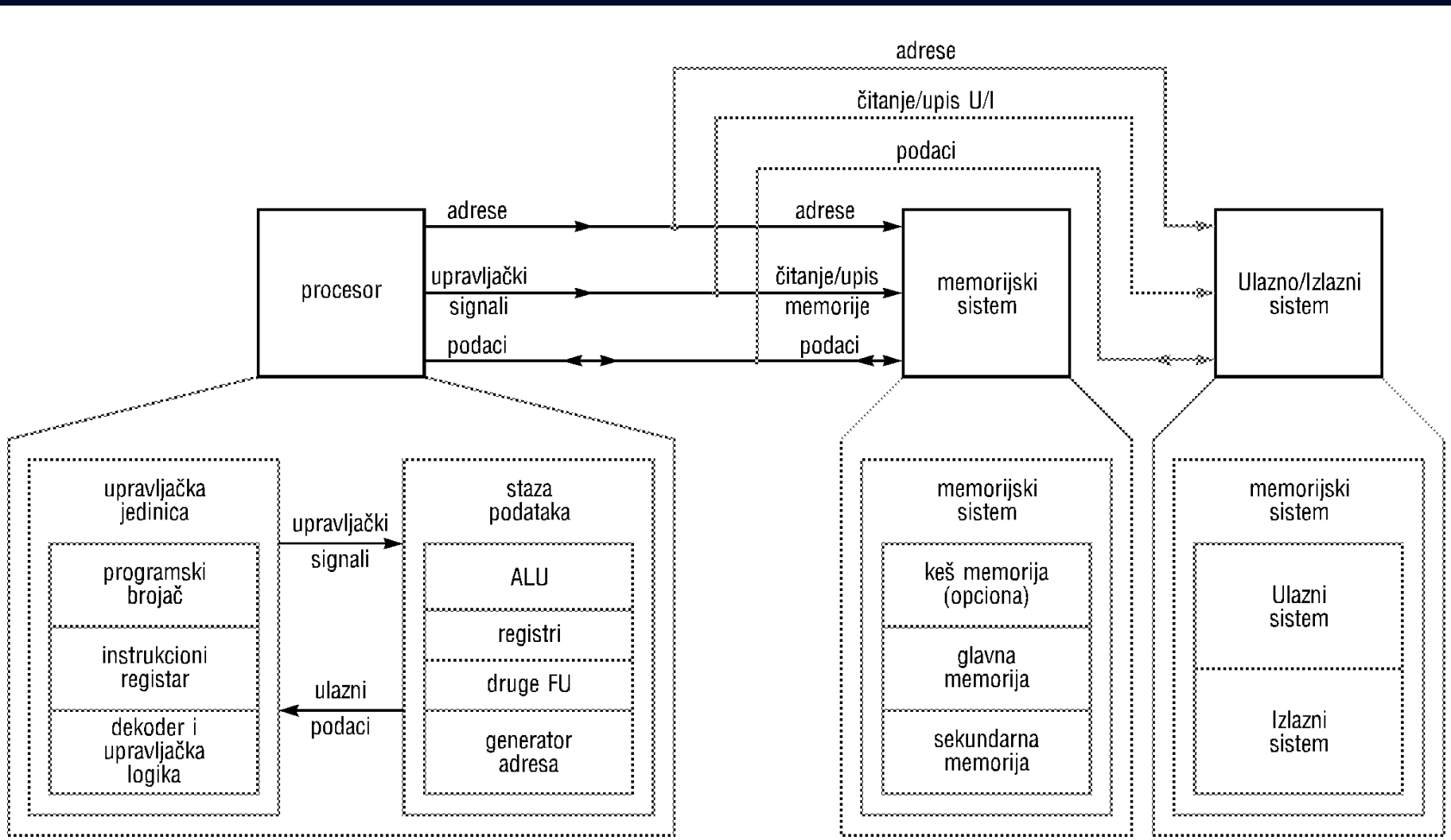
```
Andi R3,R1,1      ; maskiraj LSB registra R1 i smesti vrednost u
registar R3
Beqz R3,parno     ; ako je LSB=0 sko~i na parno
Addi R2,R2,1      ; inkrementiraj broja~ neparno za 1
parno: ......
```

Mikroprocesorski sistemi

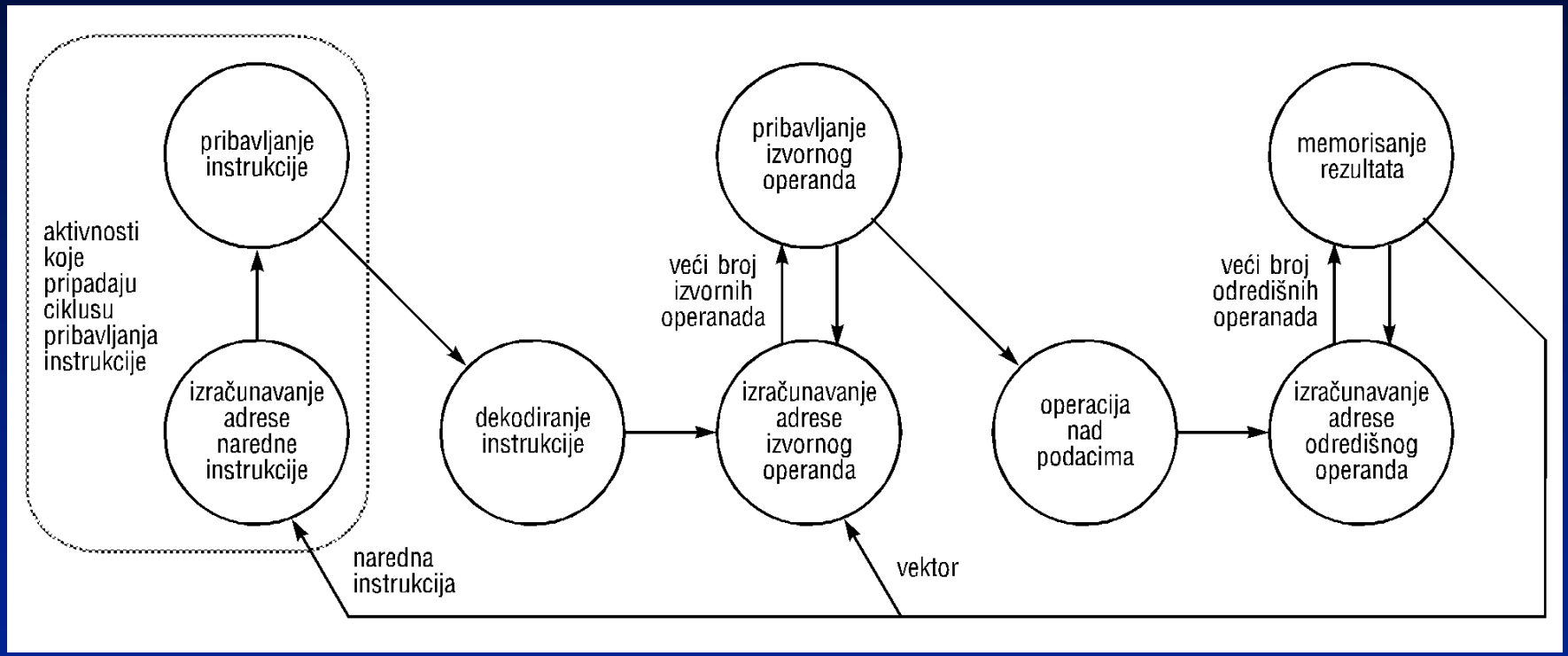
II deo

Struktura jednostavnog procesora

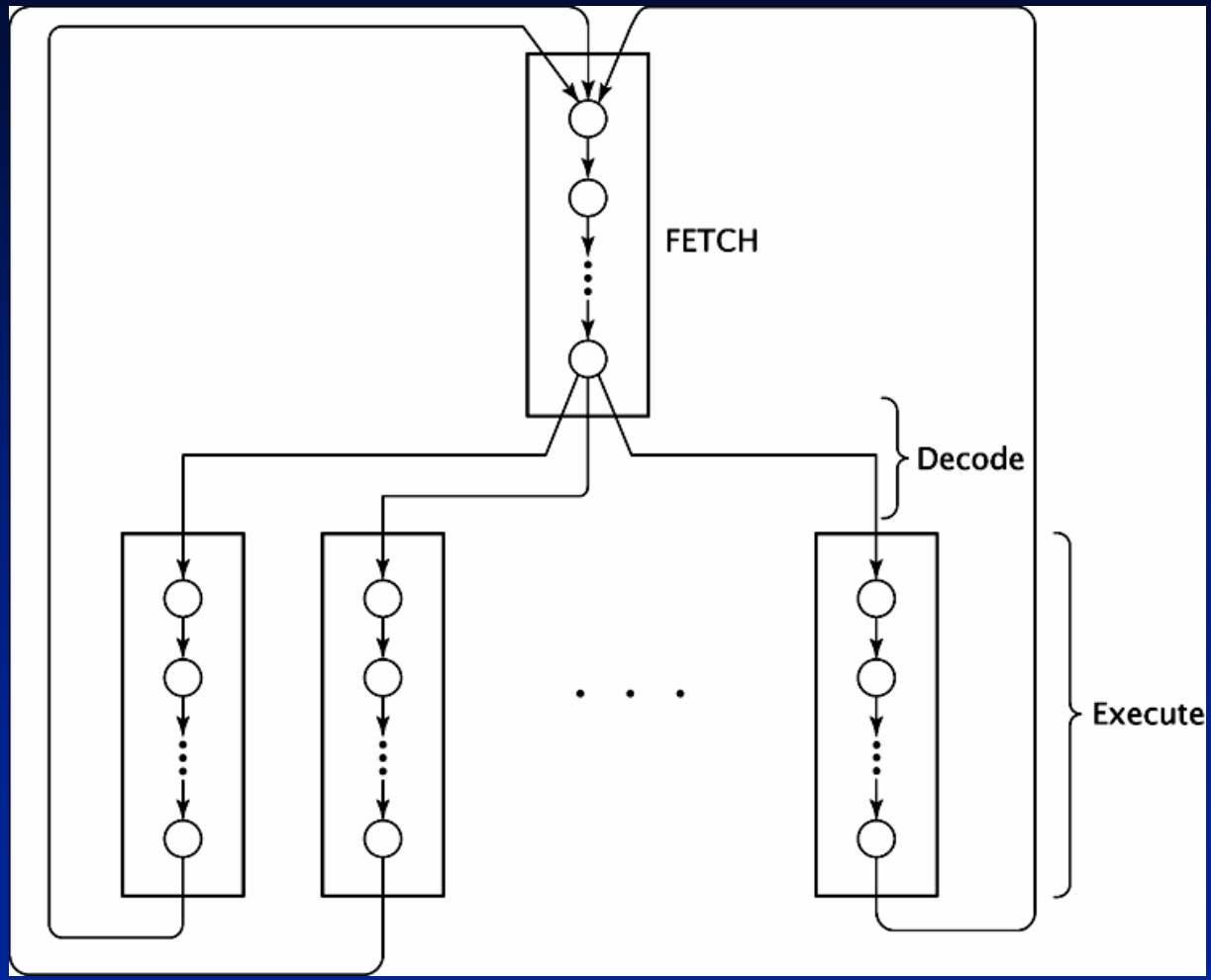
Globalna struktura računara



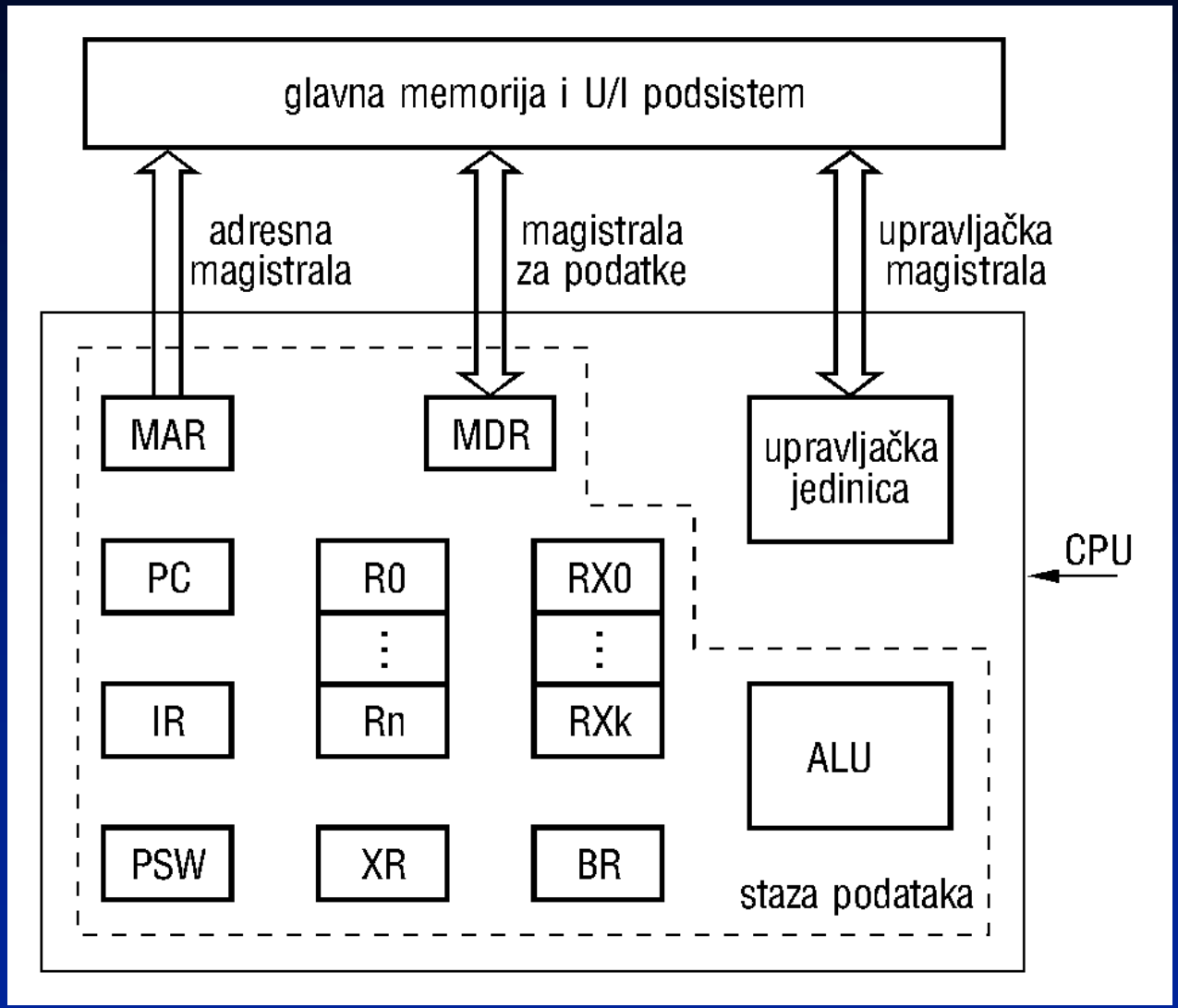
Aktivnosti u toku ciklusa instrukcije



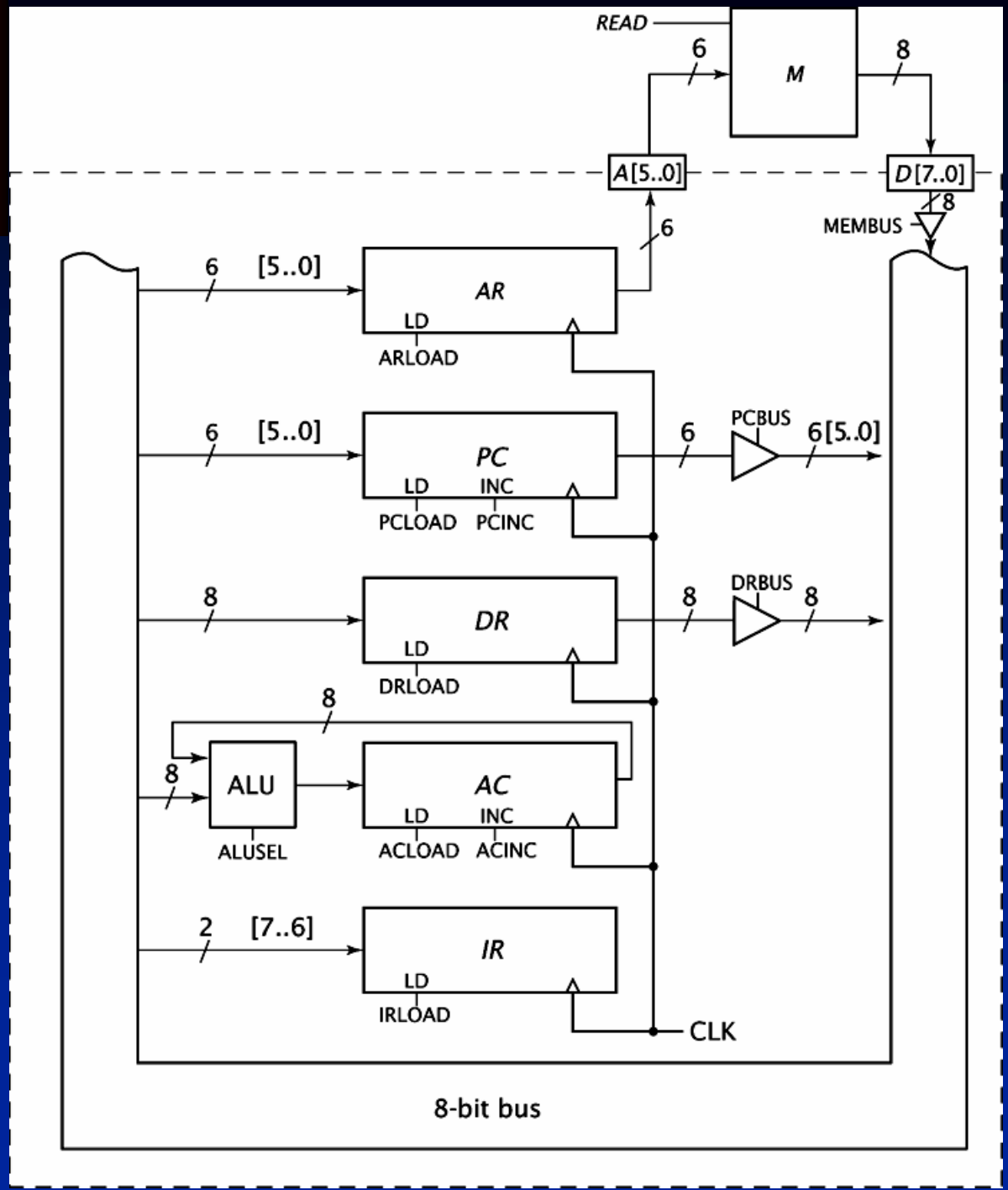
Opšti dijagram stanja CPU-a



Povezivanje CPU-a



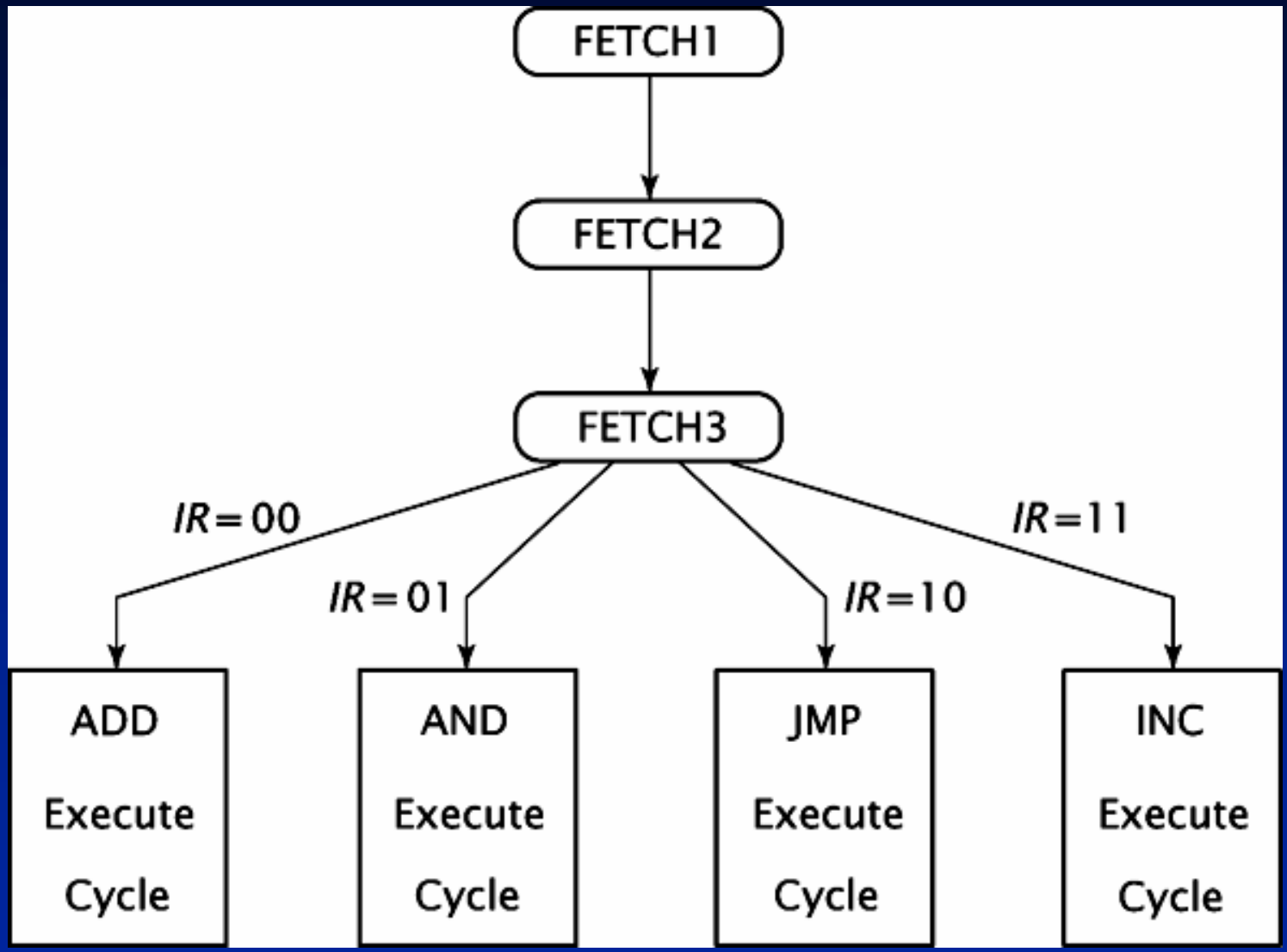
Struktura jednog jednostavnog 8-bitnog CPU-a



Skup instrukcija za jednostavan procesor

Instruction	Instruction Code	Operation
ADD	00AAAAAA	AC←AC + M[AAAAAA]
AND	01AAAAAA	AC←AC ^ M[AAAAAA]
JMP	10AAAAAA	GOTO AAAAAA
INC	11XXXXXX	AC←AC + 1

Fetch i Decode ciklusi za jednostavni 8-bitni CPU



Tipovi operacija u svakom od stanja CPU-a

FETCH1: $AR \leftarrow PC$

FETCH2: $DR \leftarrow M$, $PC \leftarrow PC + 1$

FETCH3: $IR \leftarrow DR[7..6]$, $AR \leftarrow DR[5..0]$

ADD1: $DR \leftarrow M$

ADD2: $AC \leftarrow AC + DR$

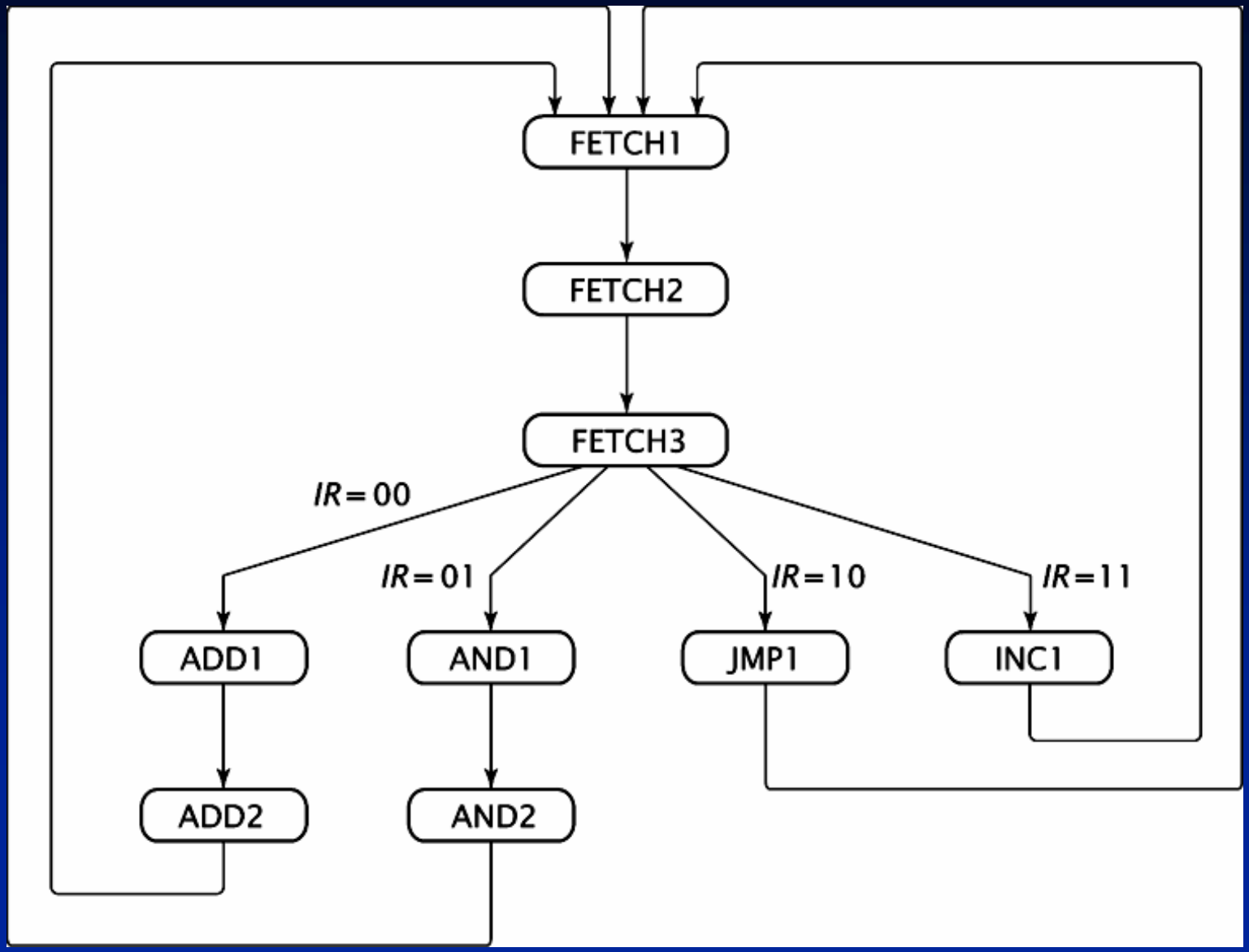
AND1: $DR \leftarrow M$

AND2: $AC \leftarrow AC \wedge DR$

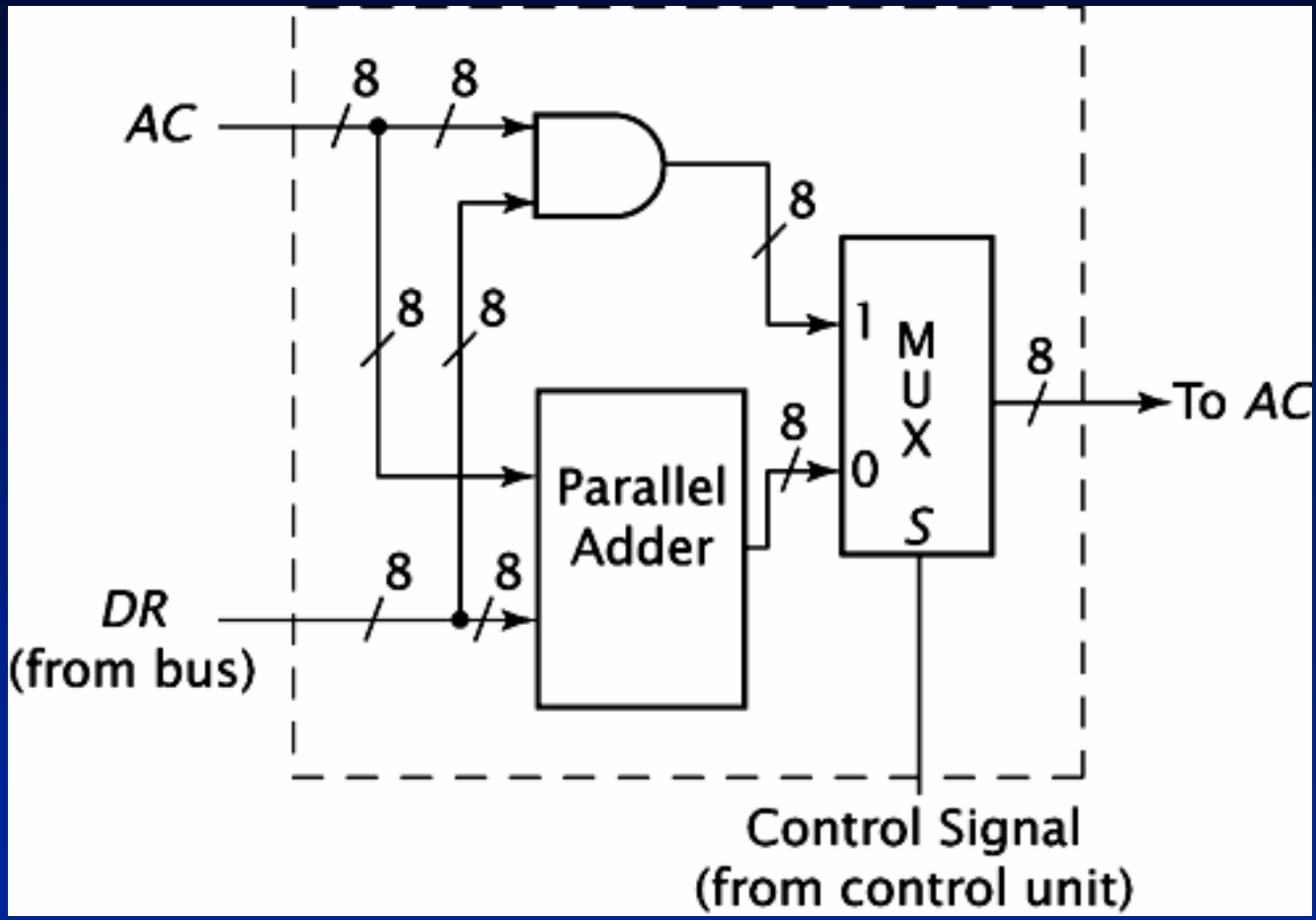
JMP1: $PC \leftarrow DR[5..0]$

INC1: $AC \leftarrow AC + 1$

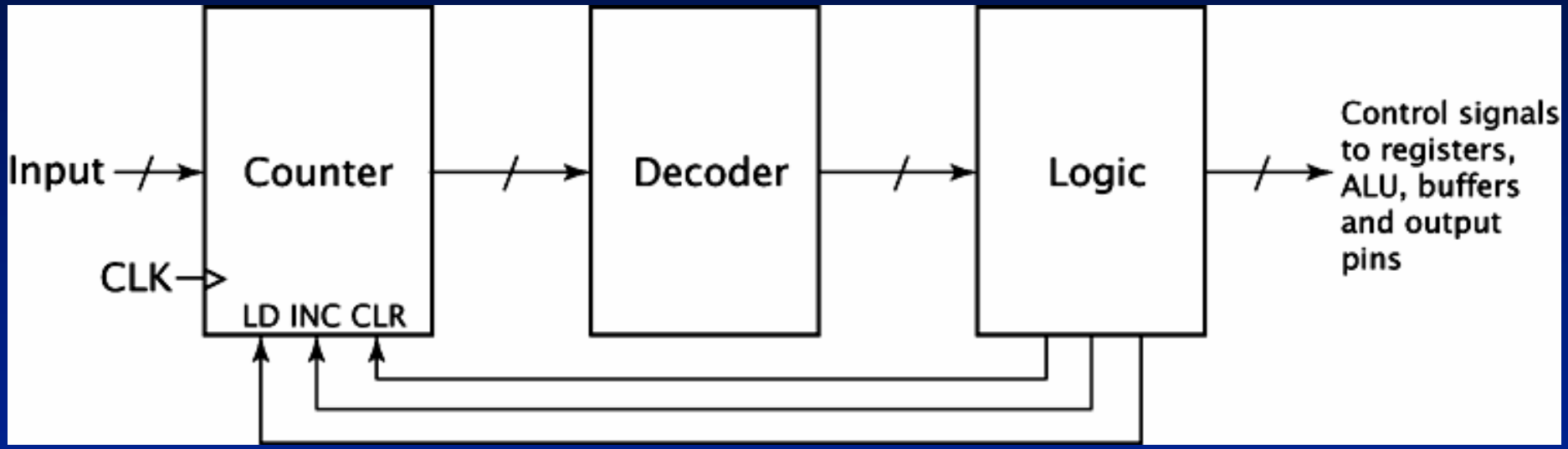
Kompletni dijagram stanja jednostavnog 8-bitnog CPU-a



Struktura ALU-a



Struktura upravljačke jedinice



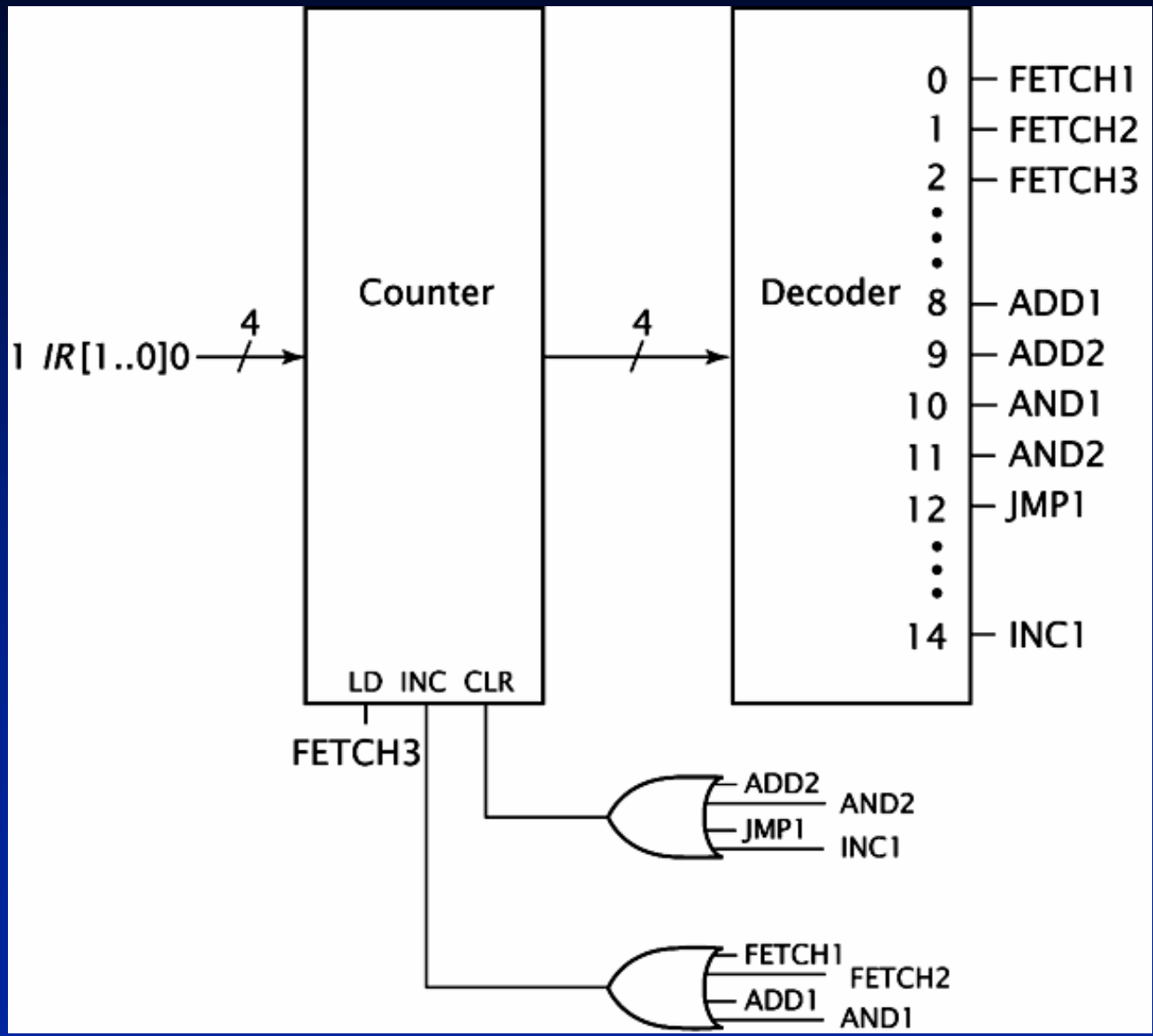
Instrukcije, prva stanja, i opkodovi za jednostavni 8-bitni CPU

Instruction	First State	*IR*
ADD	ADD1	00
AND	AND1	01
JMP	JMP1	10
INC	INC1	11

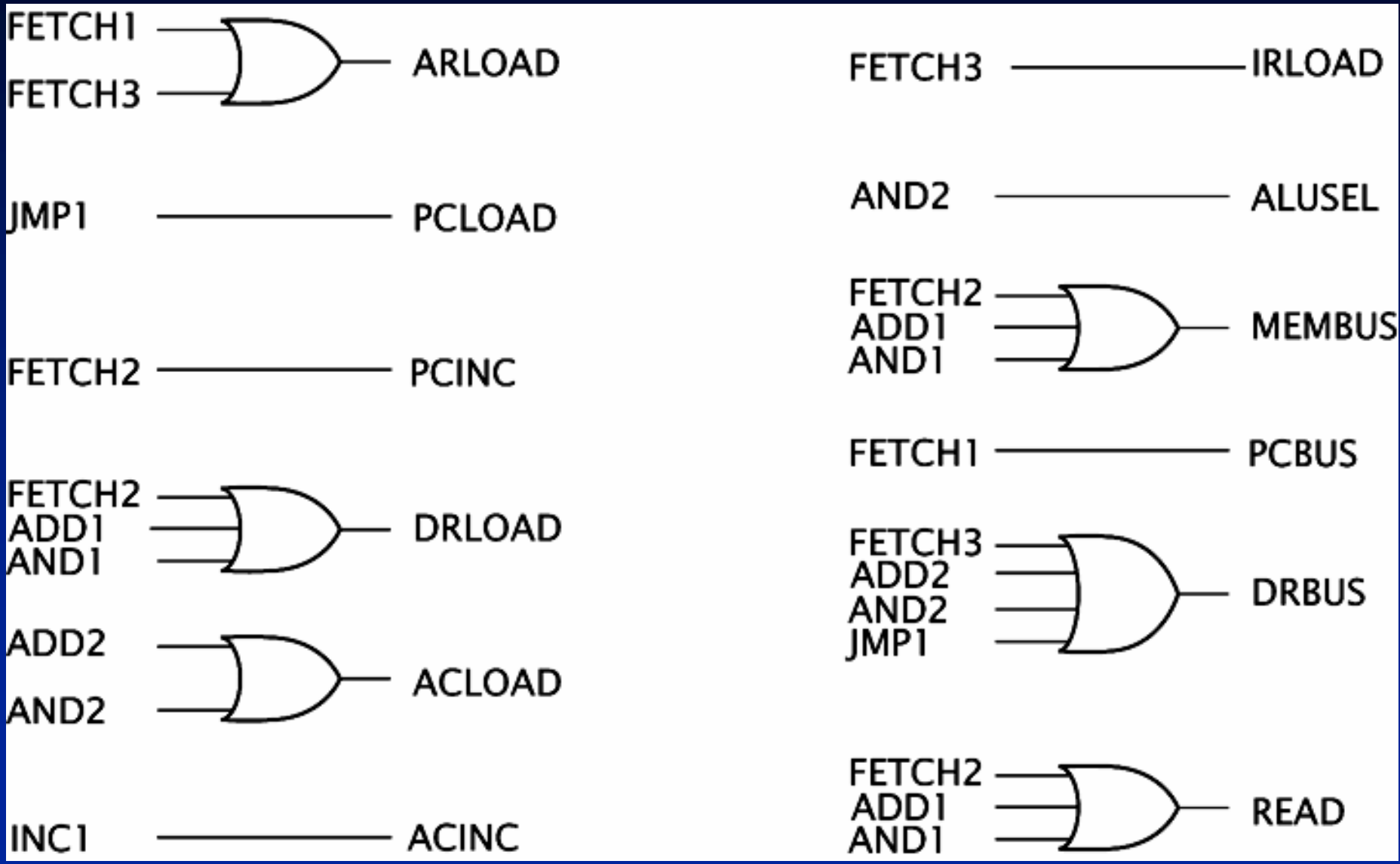
Vrednosti brojača za predloženu funkciju preslikavanja

IR[1..0]	Counter Value	State
00	1000 (8)	ADD1
01	1001 (9)	AND1
10	1010 (10)	JMP1
11	1011 (11)	INC1

Upravljačka jedinica jednostavnog 8-bitnog CPU-a



Generisanje upravljačkih signala



Izvršenje jednostavne sekvence

```
0:  ADD4
1:  AND5
2:  INC
3:  JMP 0
4:  27H
5:  39H
```

ADD4: FETCH1→FETCH2→FETCH3→ADD1→ADD2

AND5: FETCH1→FETCH2→FETCH3→AND1→AND2

INC: FETCH1→FETCH2→FETCH3→INC1

JMP 0: FETCH1→FETCH2→FETCH3→JMP1

Trasa izvršenja jednostavne sekvence - prod.

Instruction	State	Active Signals	Operations Performed	Next State
ADD 4	FETCH1	PCBUS, ARLOAD	*AR*←0	FETCH2
	FETCH2	READ, MEMBUS, DRLOAD, PCINC	*DR*←04H, *PC*←1	FETCH3
	FETCH3	DRBUS, ARLOAD, IRLOAD	*IR*←00, *AR*←04H	ADD1
	ADD1	READ, MEMBUS, DRLOAD	*DR*←27H	ADD2
	ADD2	DRBUS, ACLOAD	*AC*←0 + 27H = 27H	FETCH1
AND 5	FETCH1	PCBUS, ARLOAD	*AR*←1	FETCH2
	FETCH2	READ, MEMBUS, DRLOAD, PCINC	*DR*←45H, *PC*←2	FETCH3
	FETCH3	DRBUS, ARLOAD, IRLOAD	*IR*←01, *AR*←05H	AND1
	AND1	READ, MEMBUS, DRLOAD	*DR*←39H	AND2
	AND2	DRBUS, ALUSEL, ACLOAD	*AC*←27H ^ 39H = 31H	FETCH1
INC	FETCH1	PCBUS, ARLOAD	*AR*←2	FETCH2
	FETCH2	READ, MEMBUS, DRLOAD, PCINC	*DR*←C0H, *PC*←3	FETCH3
	FETCH3	DRBUS, ARLOAD, IRLOAD	*IR*←11, *AR*←00H	INC1
	INC1	ACINC	*AC*←21H + 1 = 22H	FETCH1
JMP 0	FETCH1	PCBUS, ARLOAD	*AR*←3	FETCH2
	FETCH2	READ, MEMBUS, DRLOAD, PCINC	*DR*←80H, *PC*←4	FETCH3
	FETCH3	DRBUS, ARLOAD, IRLOAD	*IR*←10, *AR*←00H	JMP1
	JMP1	DRBUS, PCLOAD	*PC*←0	FETCH1

Organizacija oko jedinstvene magistrale

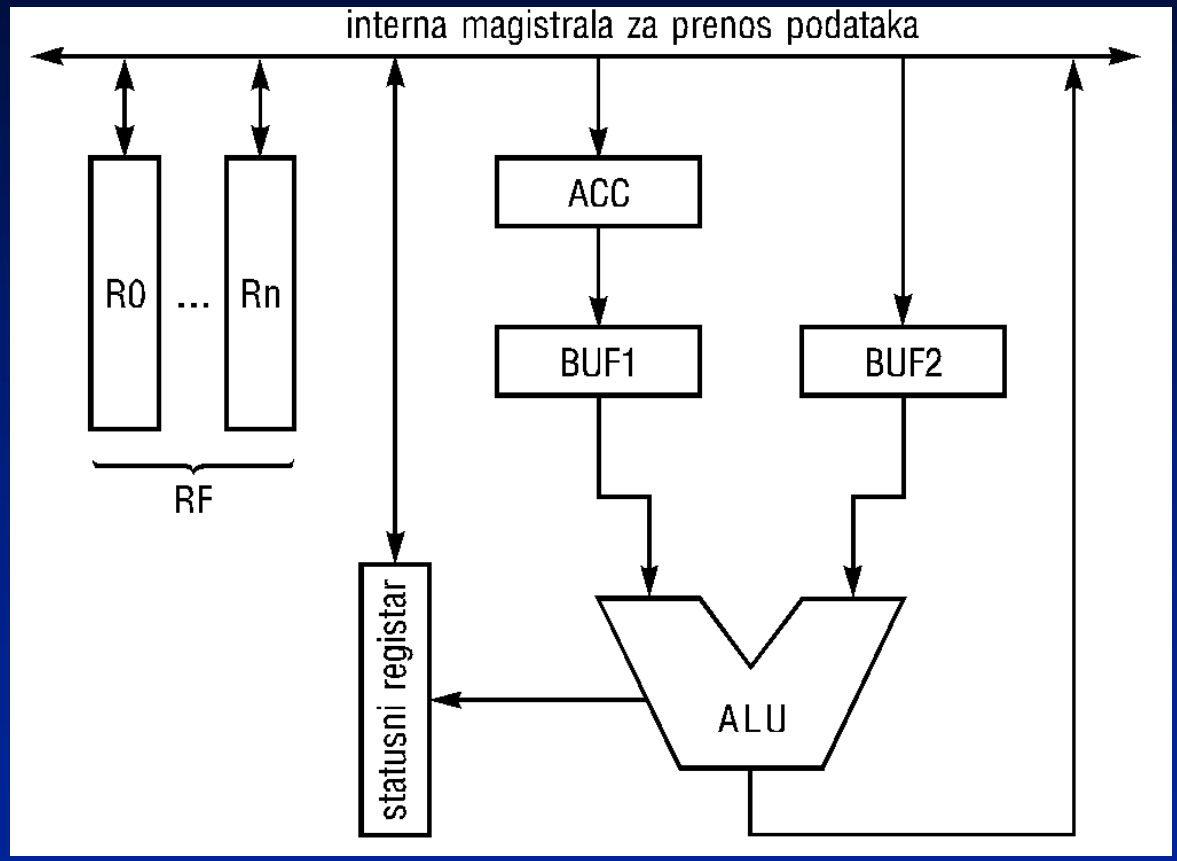
(R0) ←(R0) + (R1)

(BUF2) ←(R0)

(ACC) ←(R1)

(BUF1) ←(ACC)

(R0) ←(BUF1) + (BUF2)

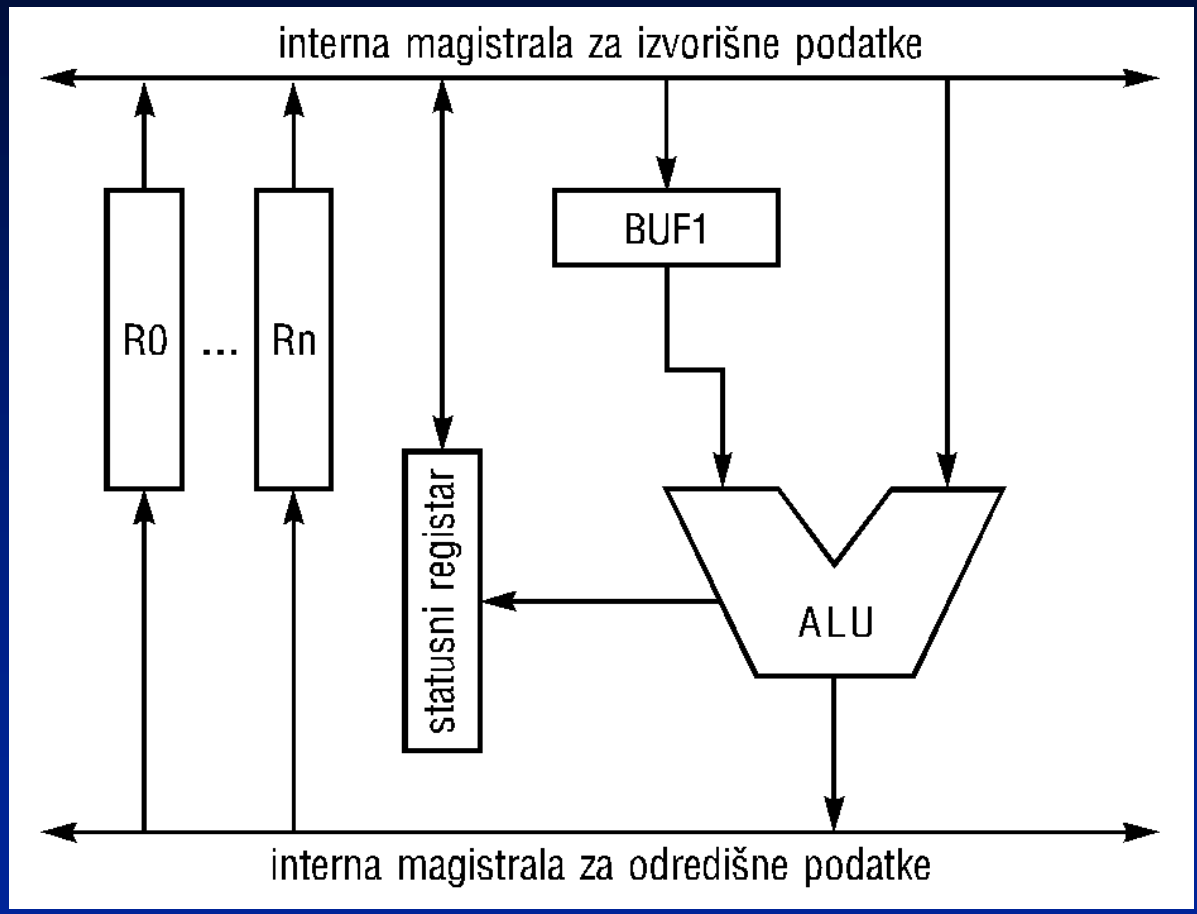


Organizacija oko dve magistrale

(R0) ←(R0) + (R1)

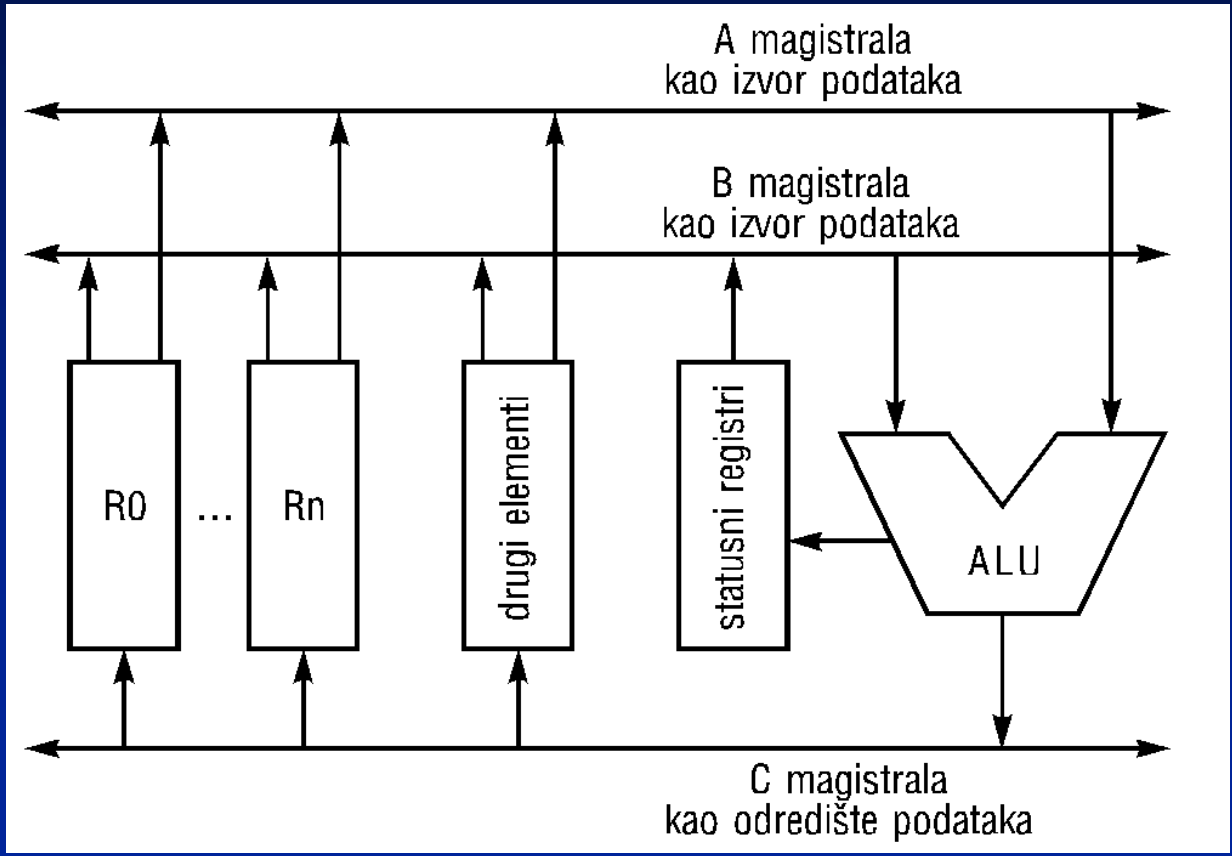
(BUF1) ←(R0)

(R0) ←(BUF1) + (R1)



Organizacija oko tri magistrale

(R2) ←(R0) + (R1)

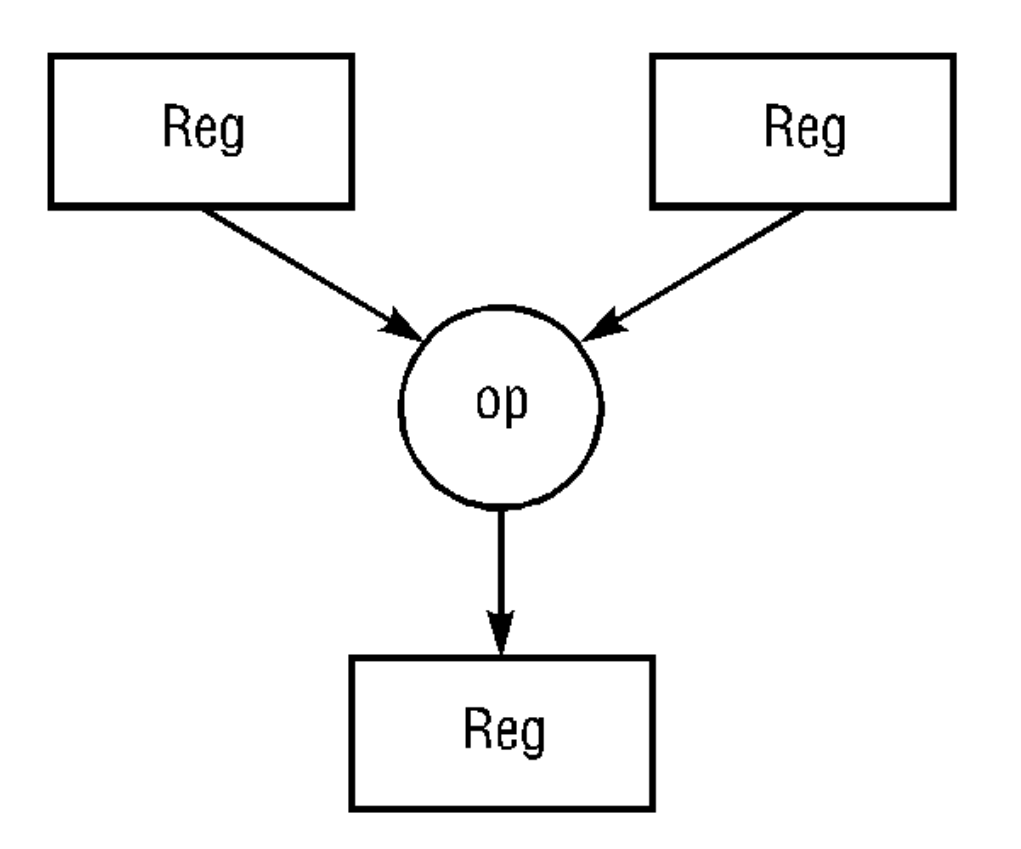


Podela računara

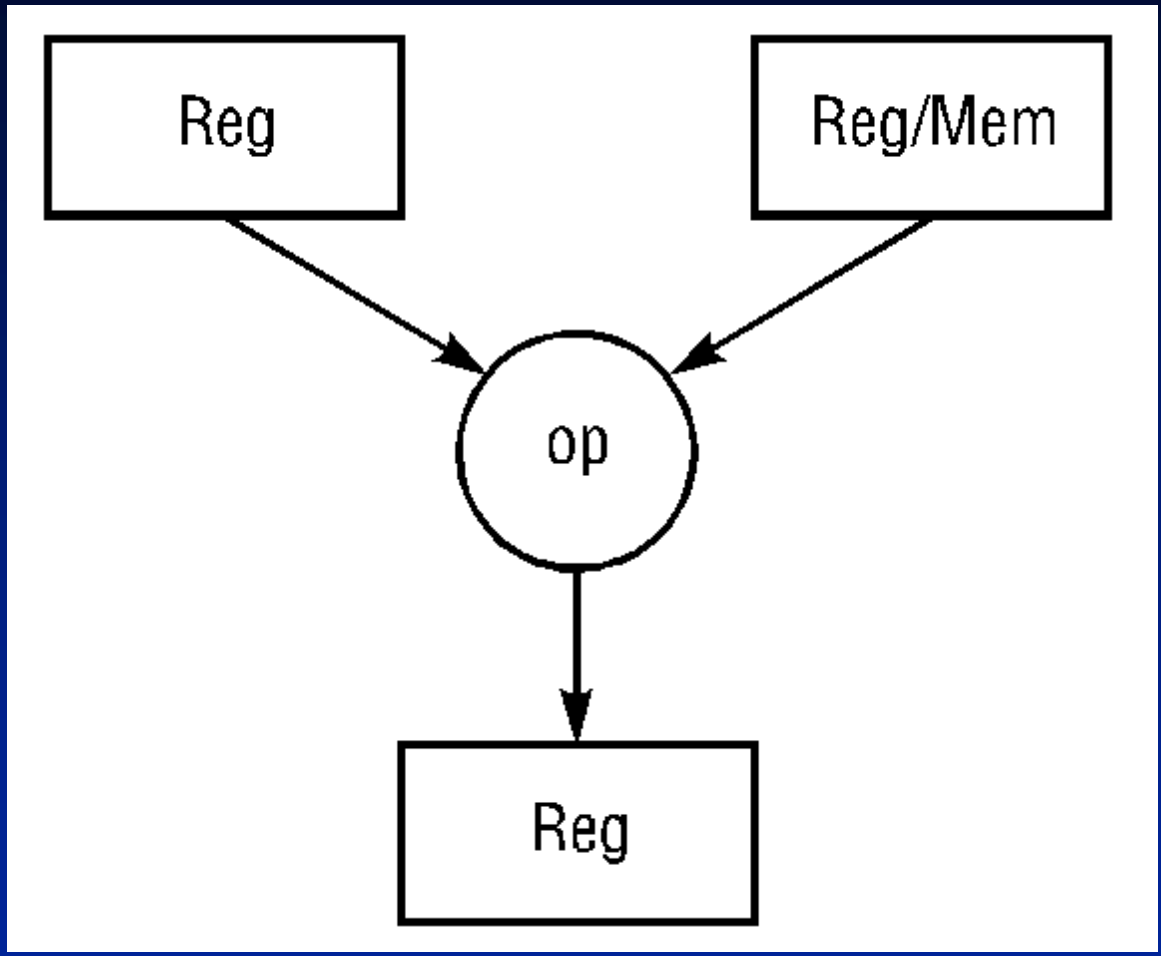
- U odnosu na to kako se skupom instrukcija vrši specifikacija operanada instrukcije računare možemo podeliti na sledeće tipove:

  - L/S – Load/Store arhitekture
  - R/M – Registar-Memorija arhitektura
  - R + M – Registar–plus-Memorija arhitekture

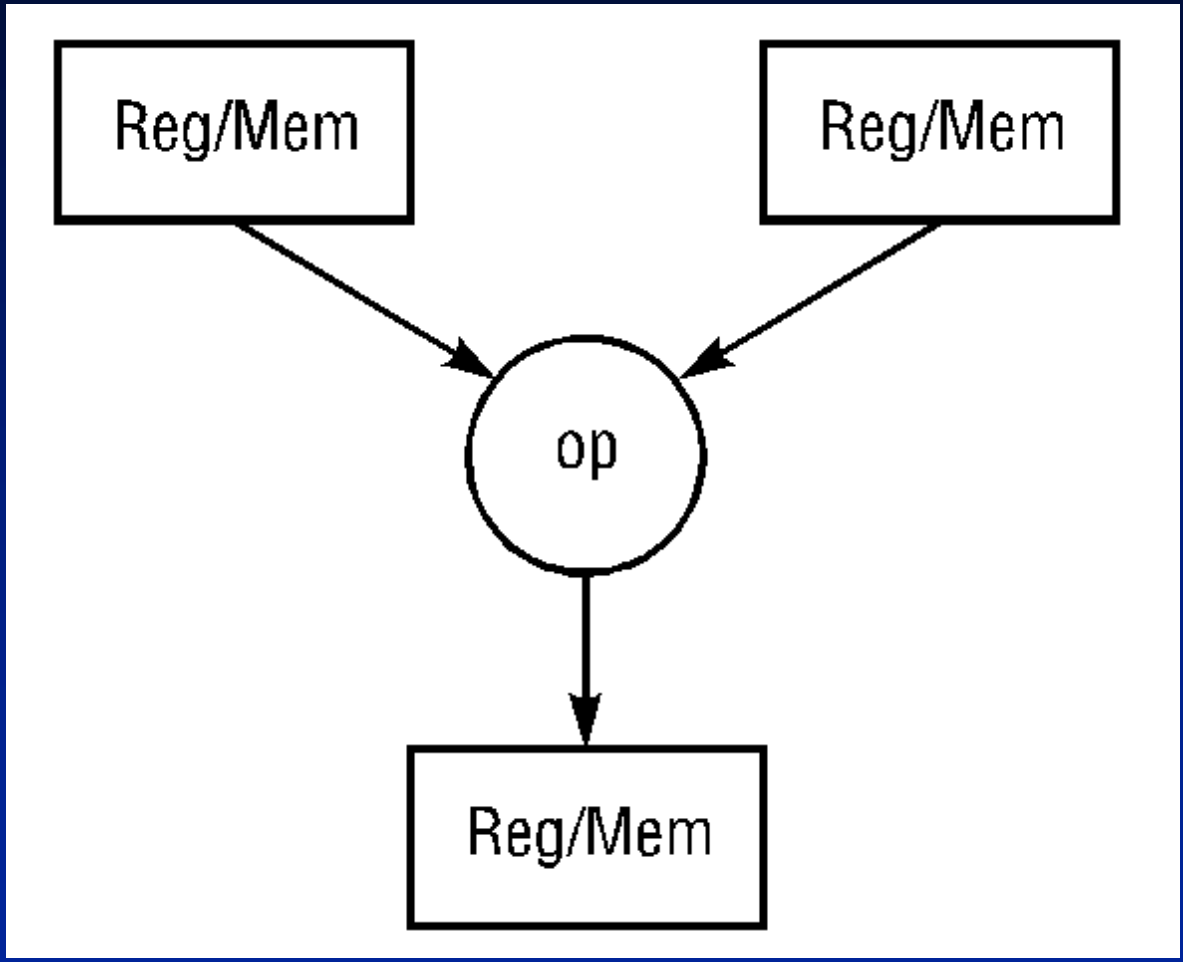
Load/Store arhitektura



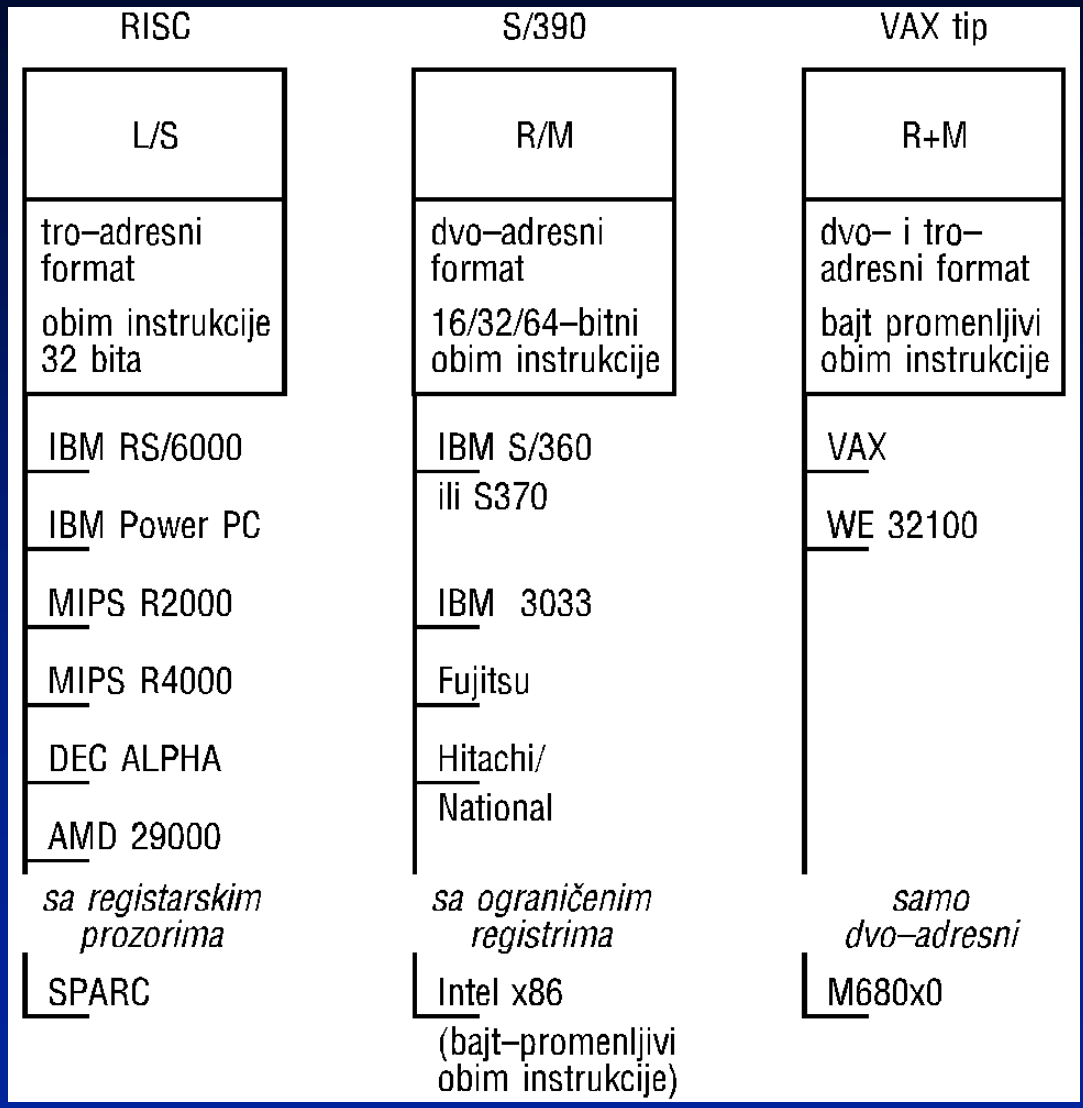
Registar/Memroija arhitektura



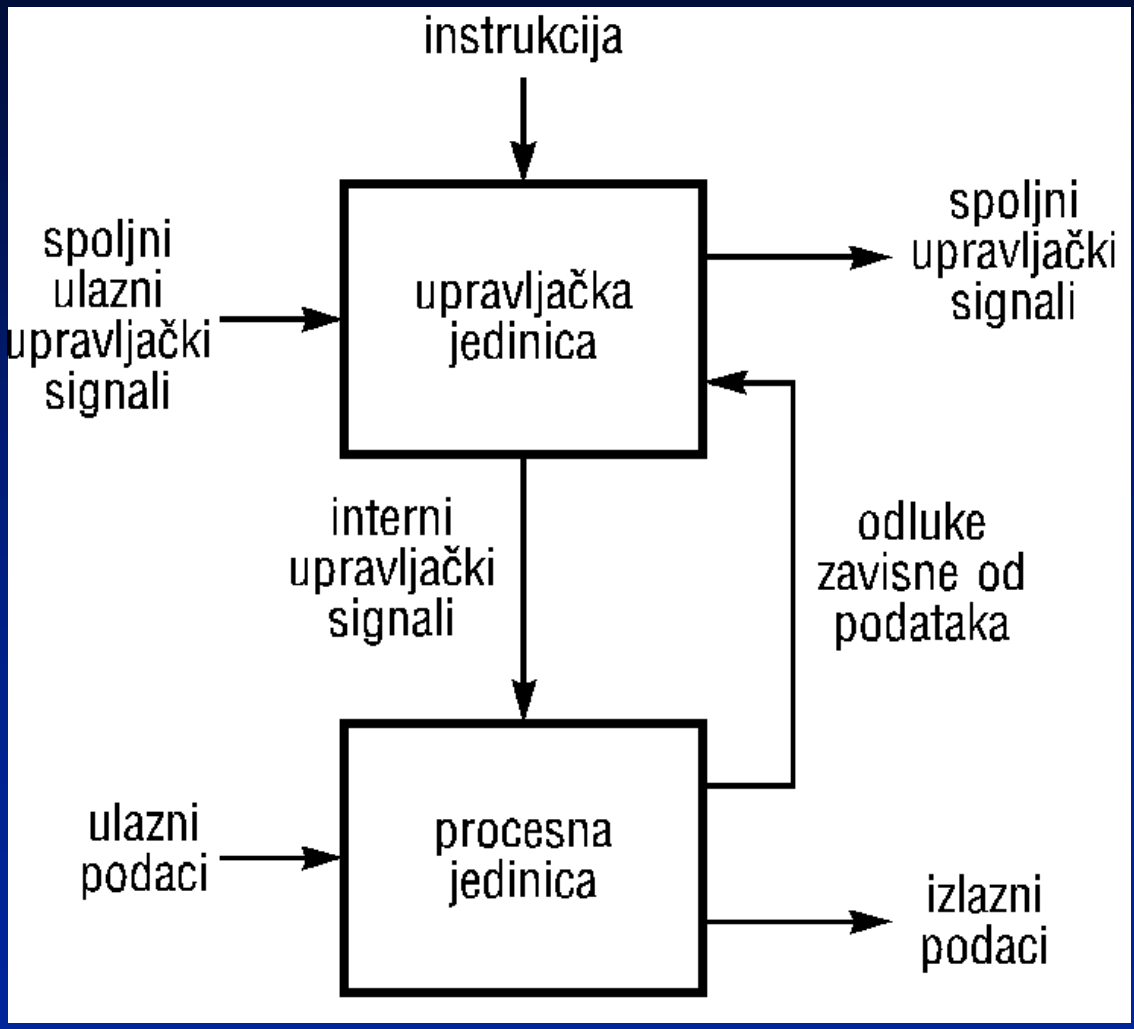
Registar-plus-Memorija arhitektura



LS, R/M, i R+M arhitekture



Upravljačka jedinica - osnovna strategija upravljanja



Tipovi upravljačkih jedinica

- **Struktura savremenih upravljačkih jedinica se zasniva na jednom od sledeća dva pristupa:**
  - **direktno upravljanje (hardwired control)**
  - **mikroprogramsko upravljanje (microprogrammed control)**

Karakteristike direktnog upravljanja

- Upravljačka linija se realizuje kao sekvencijalno logičko kolo
- U cilju postizanja maksimalne brzine u radu upravljačka jedinica je projetovana sa minimalnim brojem komponenata
- Kada je jedanput projektovano, male promene iziskuju potpuni redizajn kola
- Glavna prednost je brzina rada i to je razlog zašto se ova tehnika koristi za realizaciju brzih proocesora

Karakteristike mikroprogramskog upravljanja

- Skup upravljačkih signala zajedno sa sekvencirajućom informacijom smešta se u memoriju

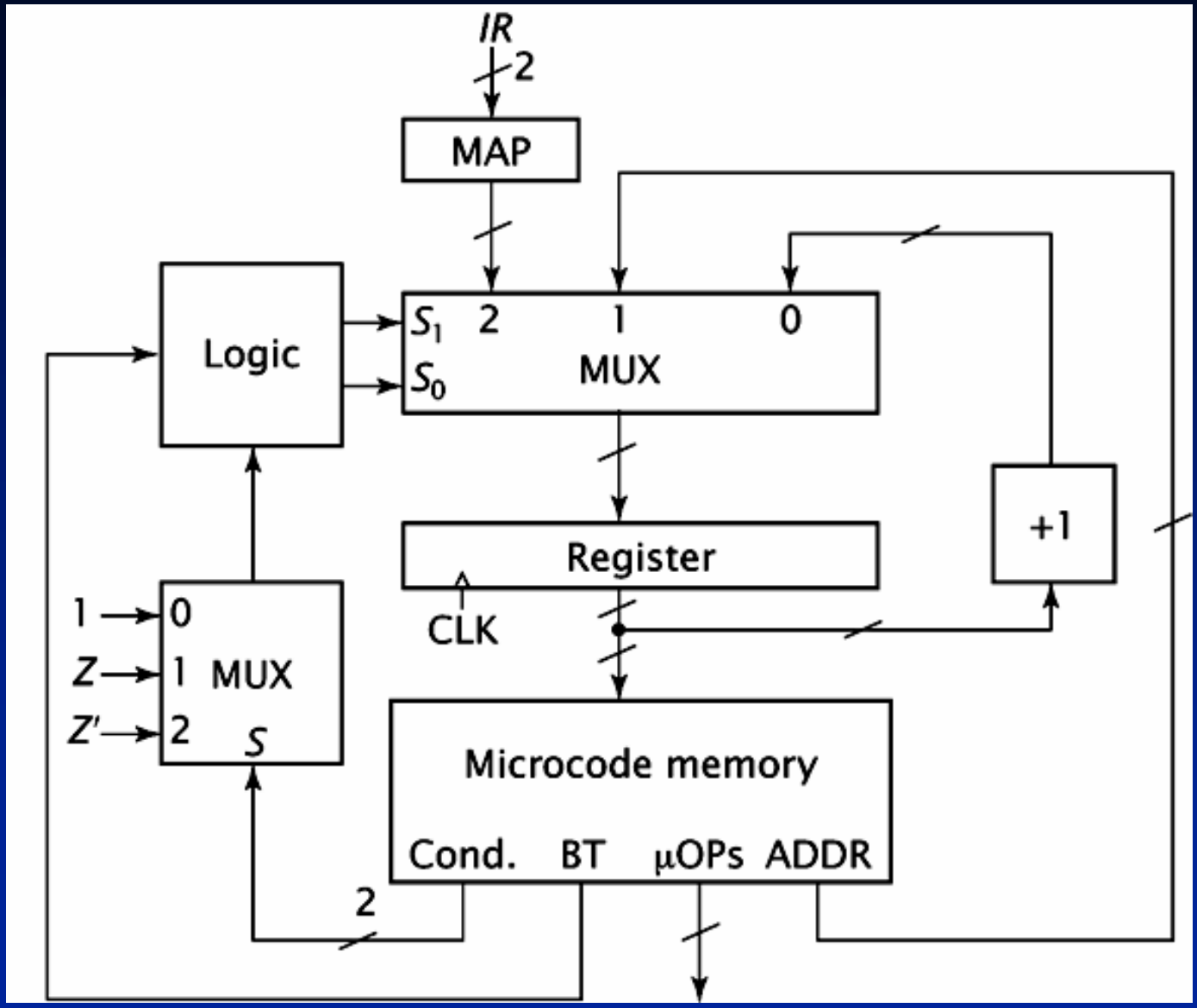
- Memorija može biti ROM ili RAM tipa i naziva se mikroprogramska memorija (microprogramme memory ili control memory)

- Mikroprogramsko upravaljanje se realizuje pomoću sekvence mikroinstrukcija

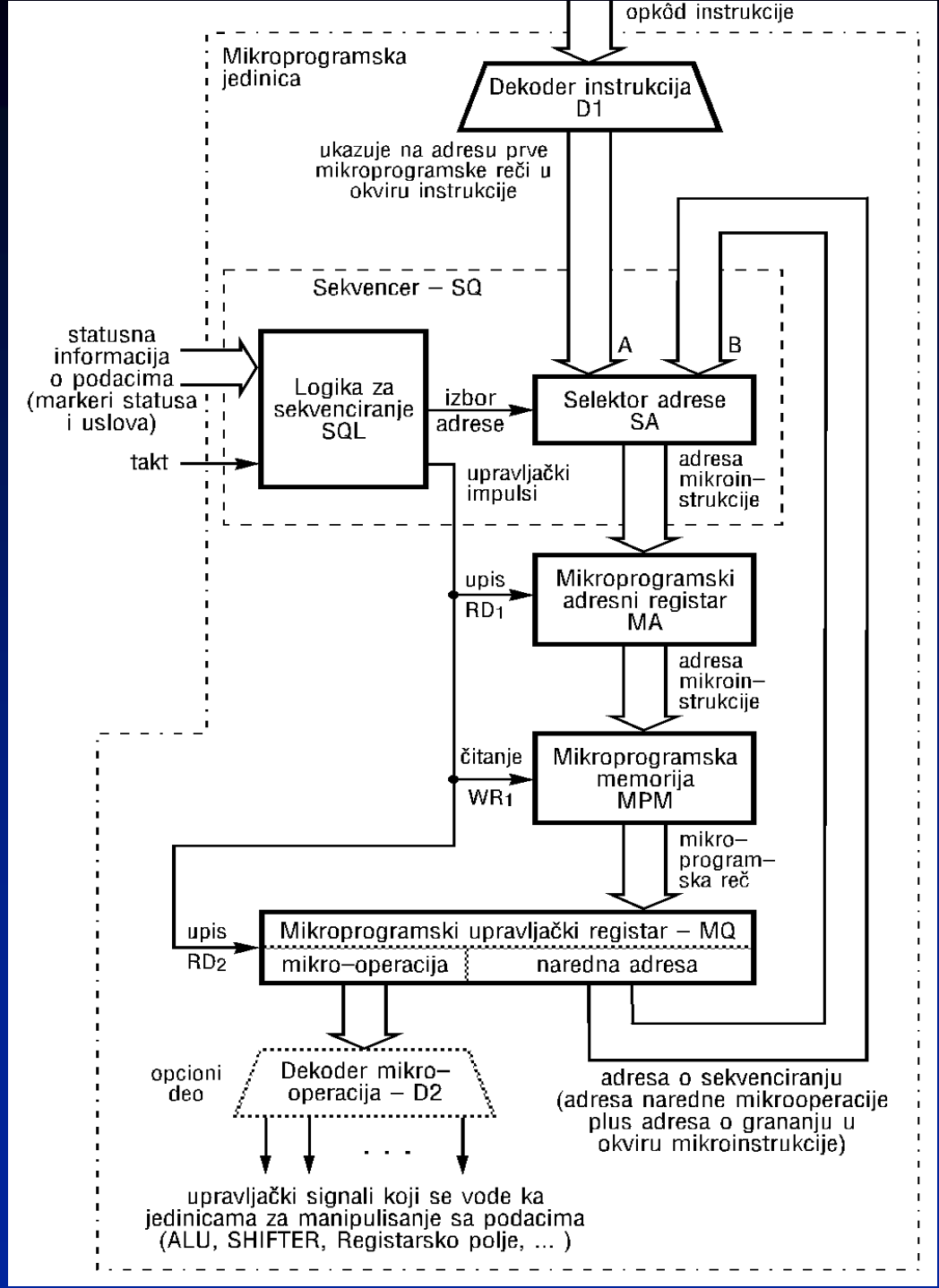
Prednosti mikroprogramskog upravljanja

- Izmene se izvode brže
- Greške se lakše otklanjaju
- Rutine na asembleru ili HLL-u se mogu implementirati na osnovnom mašinskom nivou koristeći mikroprograme
- Projektovanje i održavanje računara je pojednostavljeno
- Koristeći mikroprogramsku emulaciju moguće je ostvariti kompatibilnost sa drugom mašinom
- Nedostatak je sporije izvršenje instrukcija

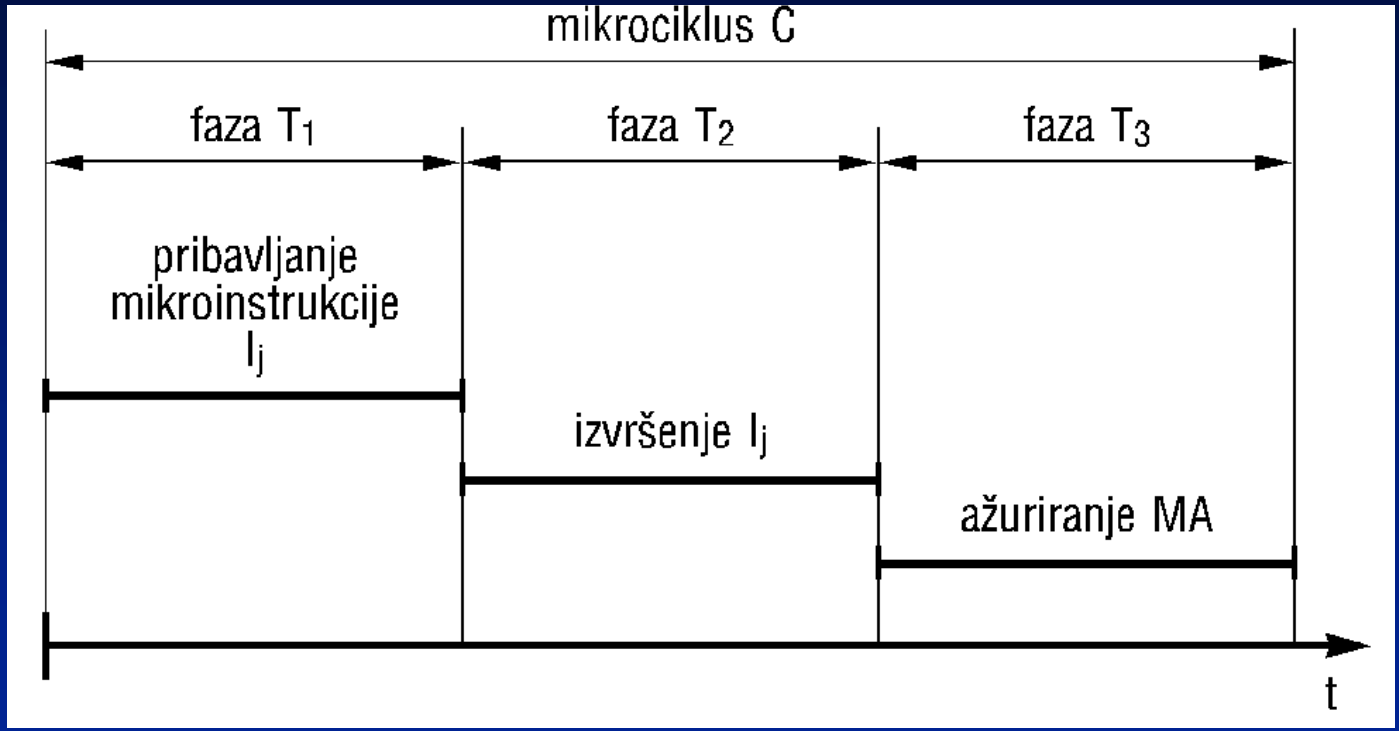
Struktura jednostavne mikroprogramske jedinice



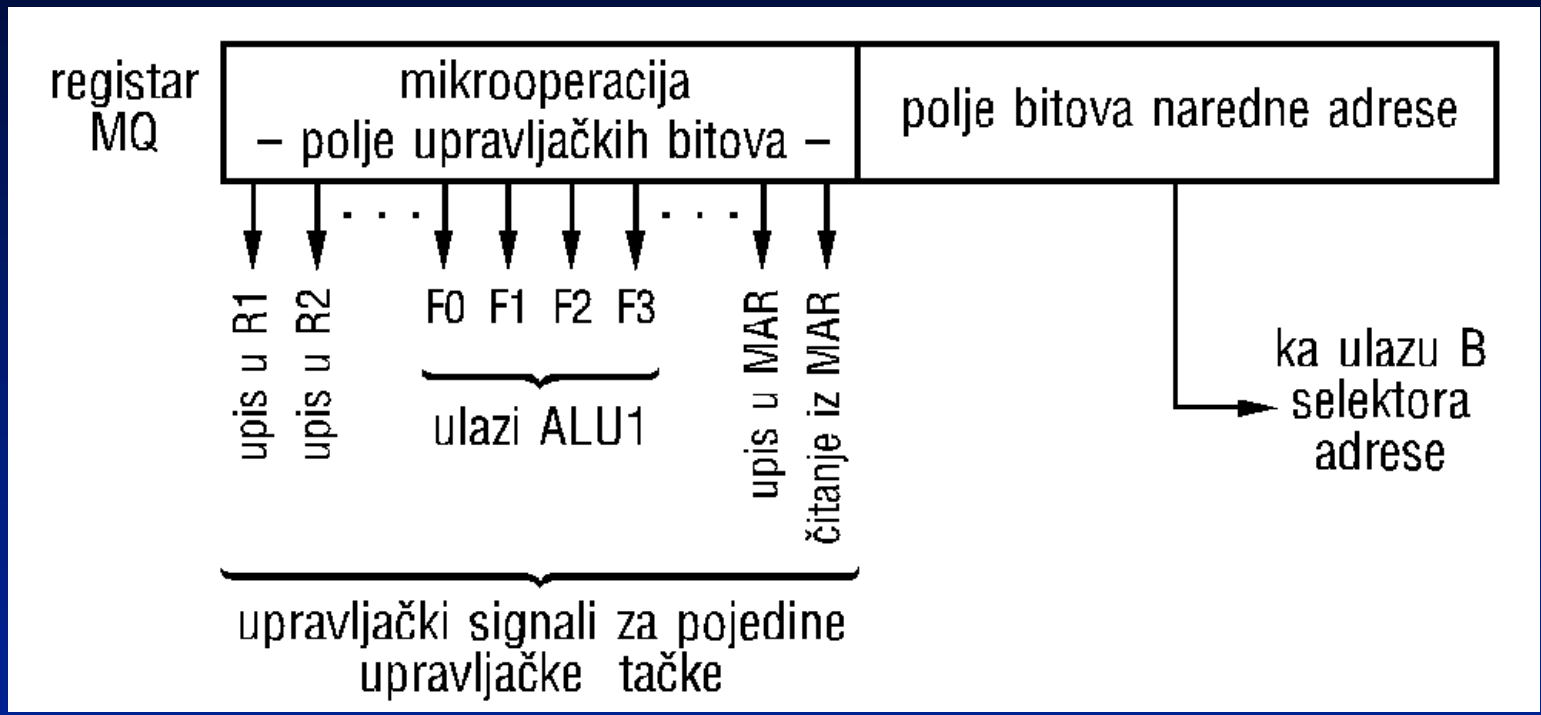
Struktura mikroprogramske jedinice



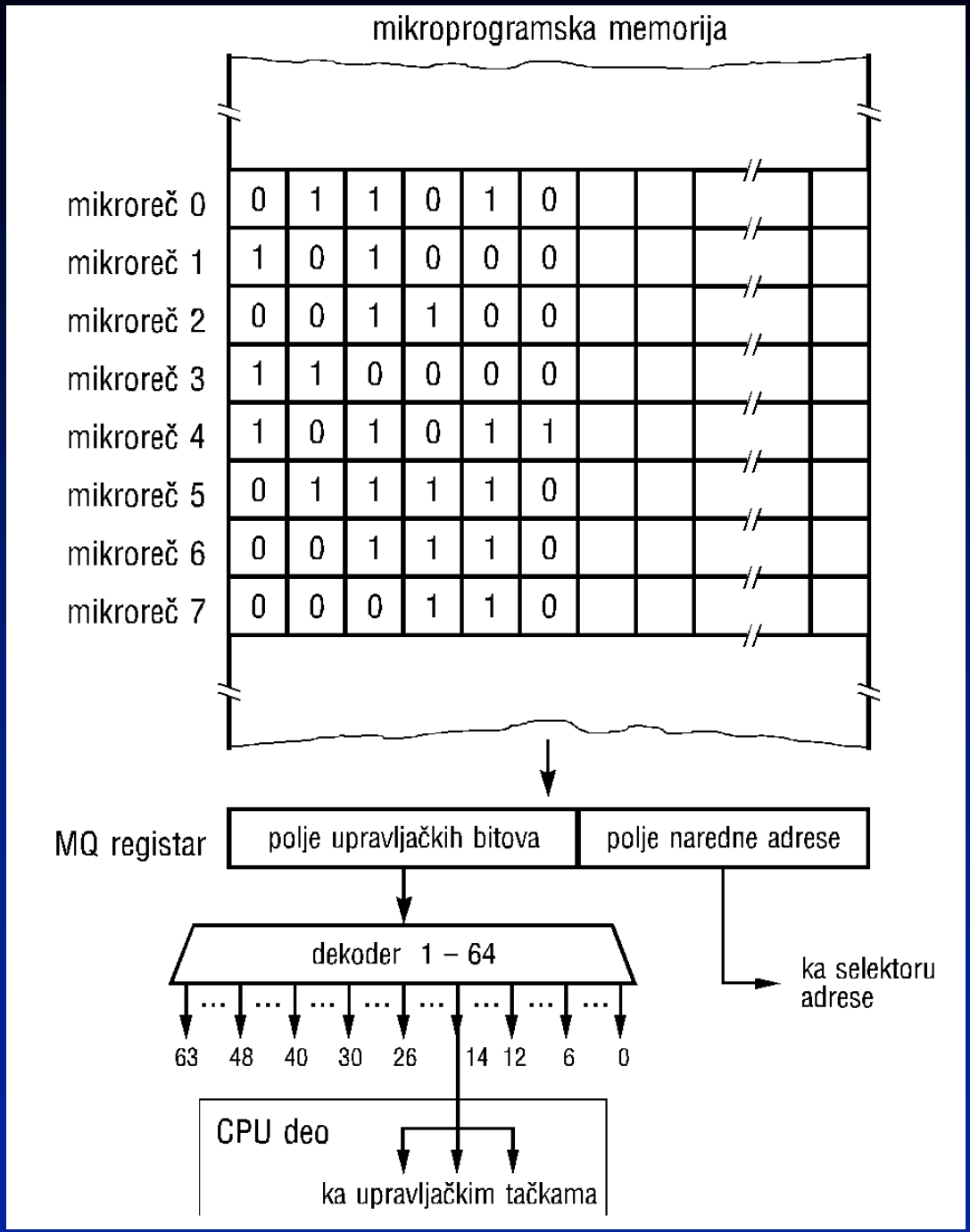
Tipična sekvenca izvršenja mikroinstrukcije



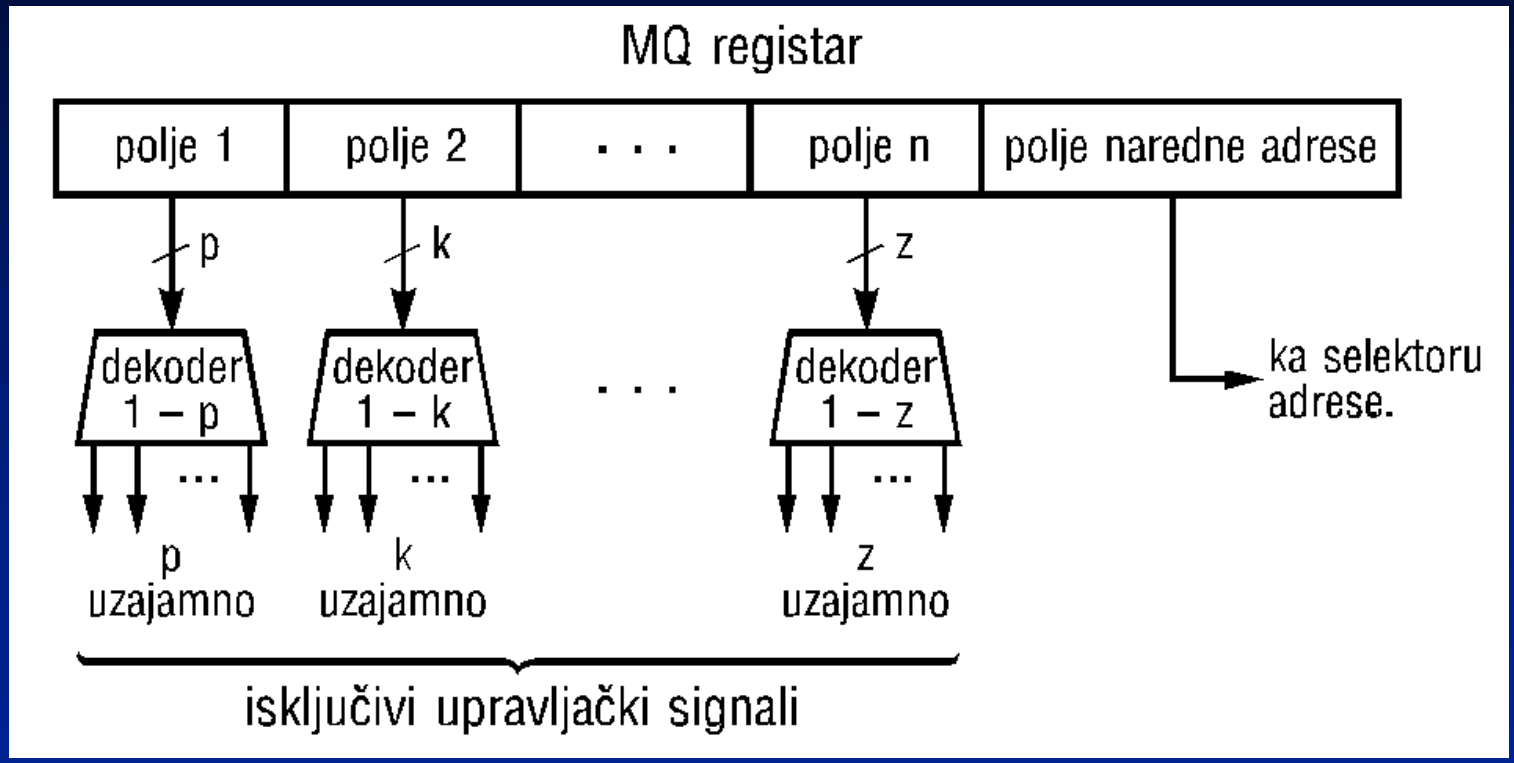
Format mikroinstrukcije



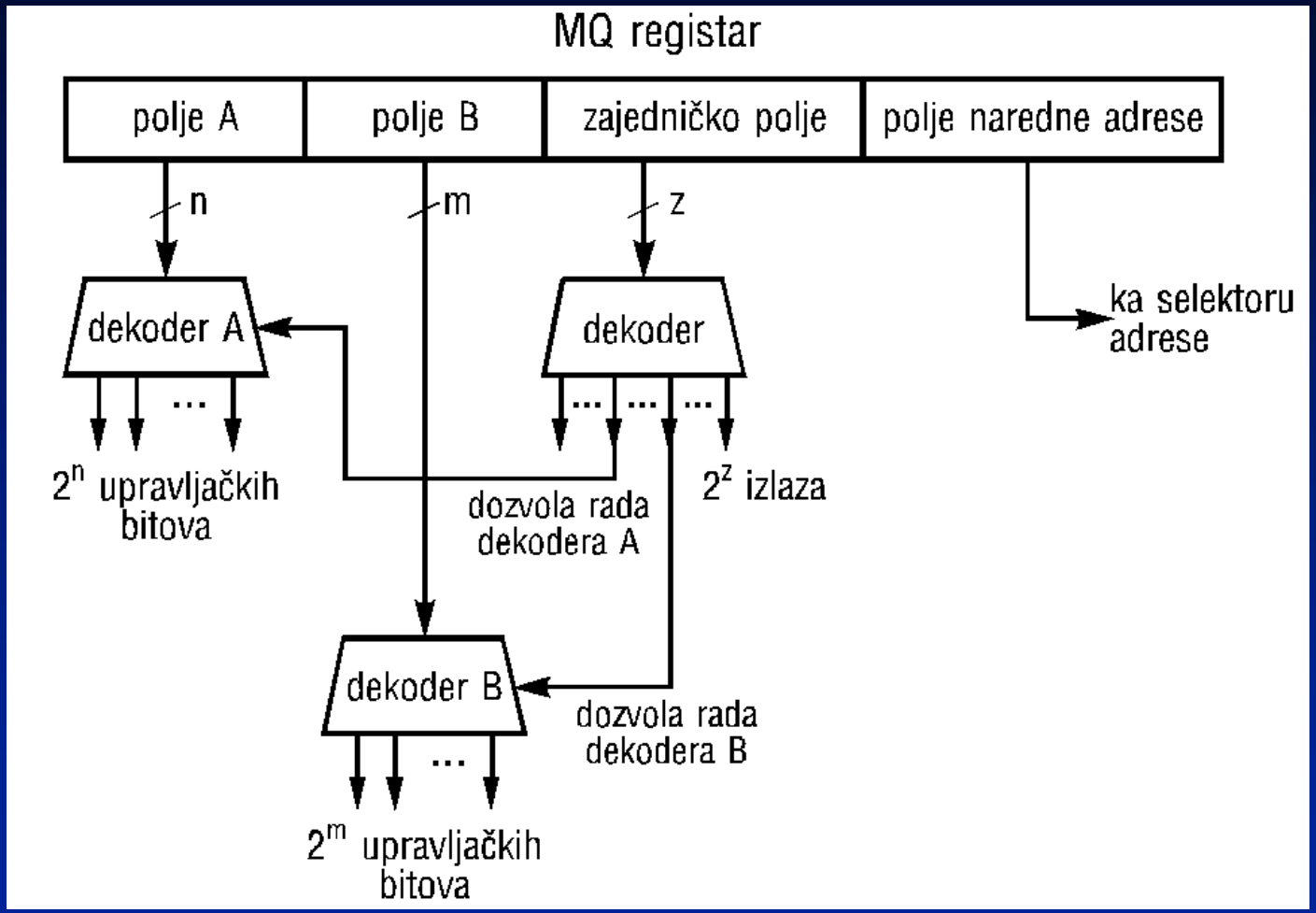
Organizacija mikro-programske reči kod metode maksimalnog dekodiranja



Organizacija kodiranja mikrooperacija kod metode minimalnog dekodiranja

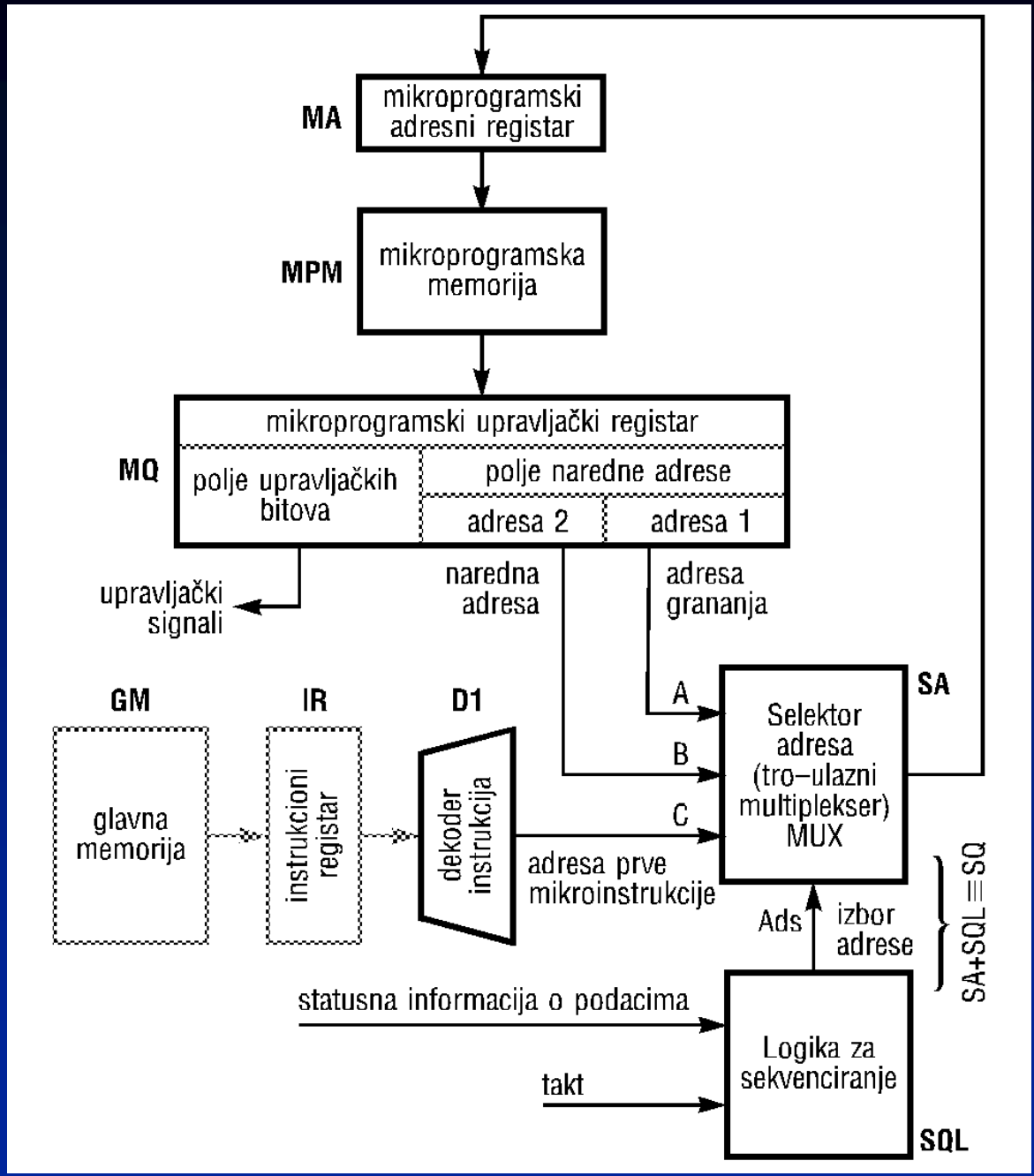


Organizacija indirektnog kodiranja mikroinstrukcije



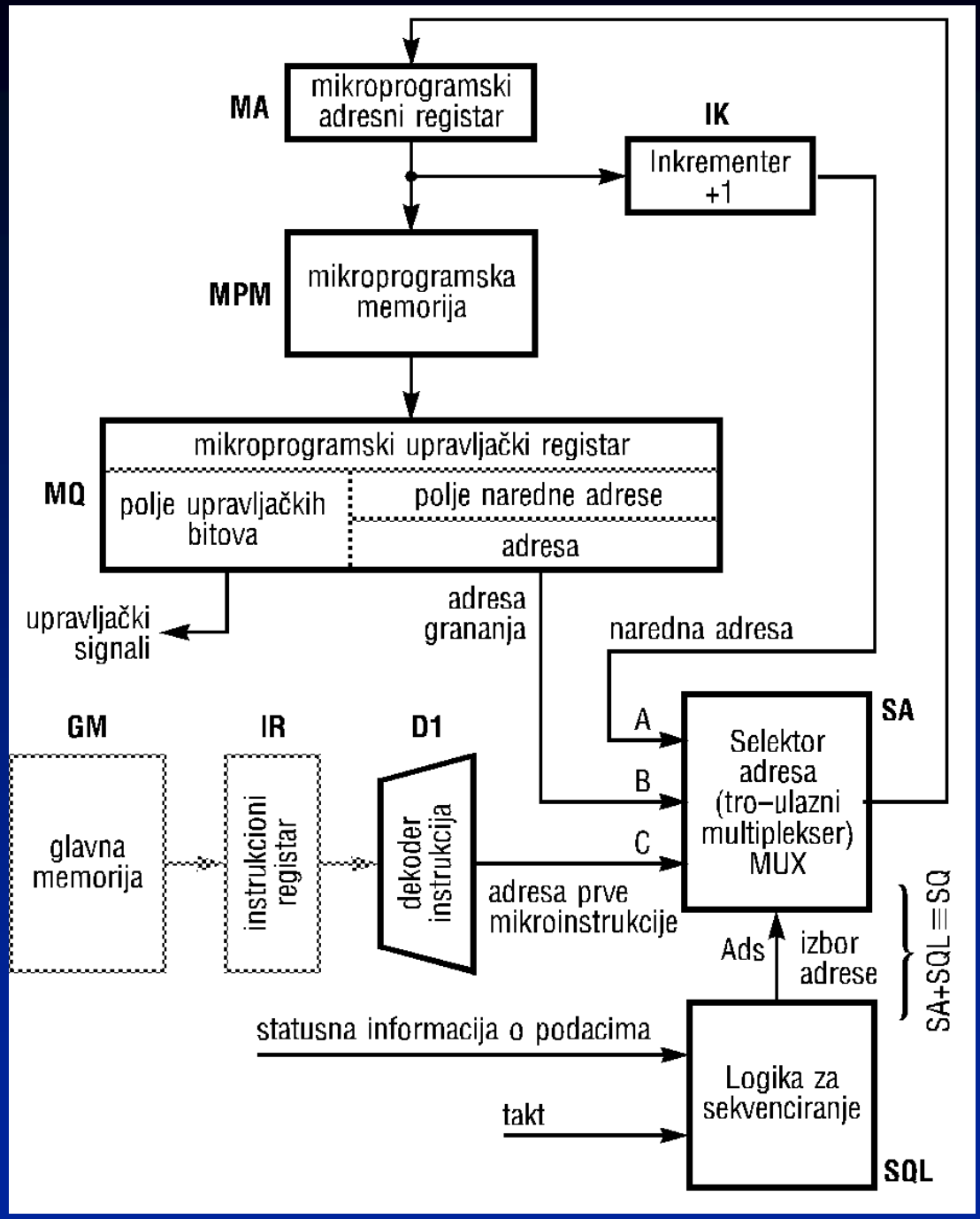
Sekvenciranje mikroinstrukcija

Upravljačka jedinica kod koje je polje naredna adresa dvo-adresno

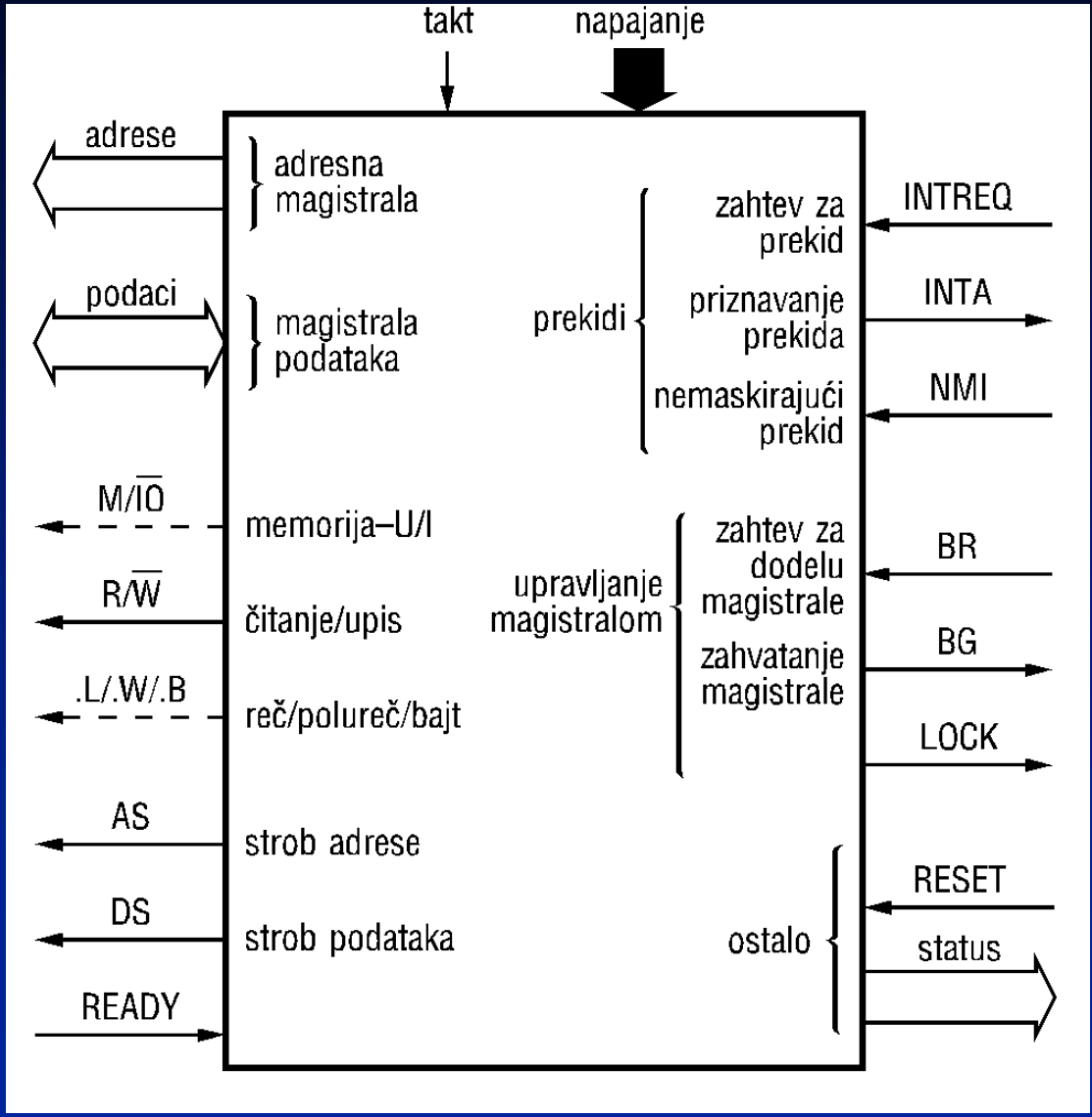


Sekvenciranje mikroinstrukcija

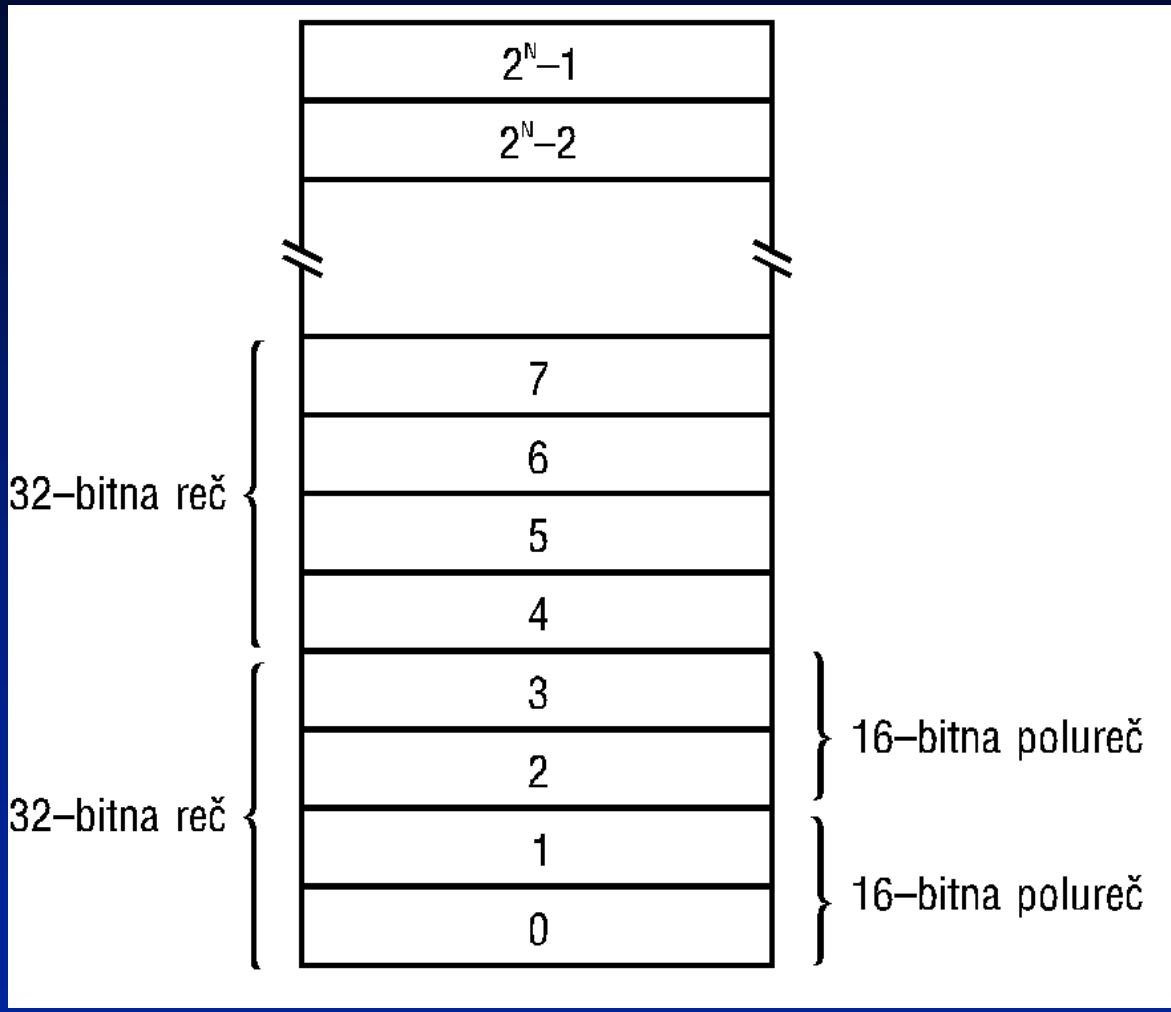
Upravljačka jedinica kod koje je polje naredna adresa jedno-adresno



Spoljni interfejs mikroprocesora

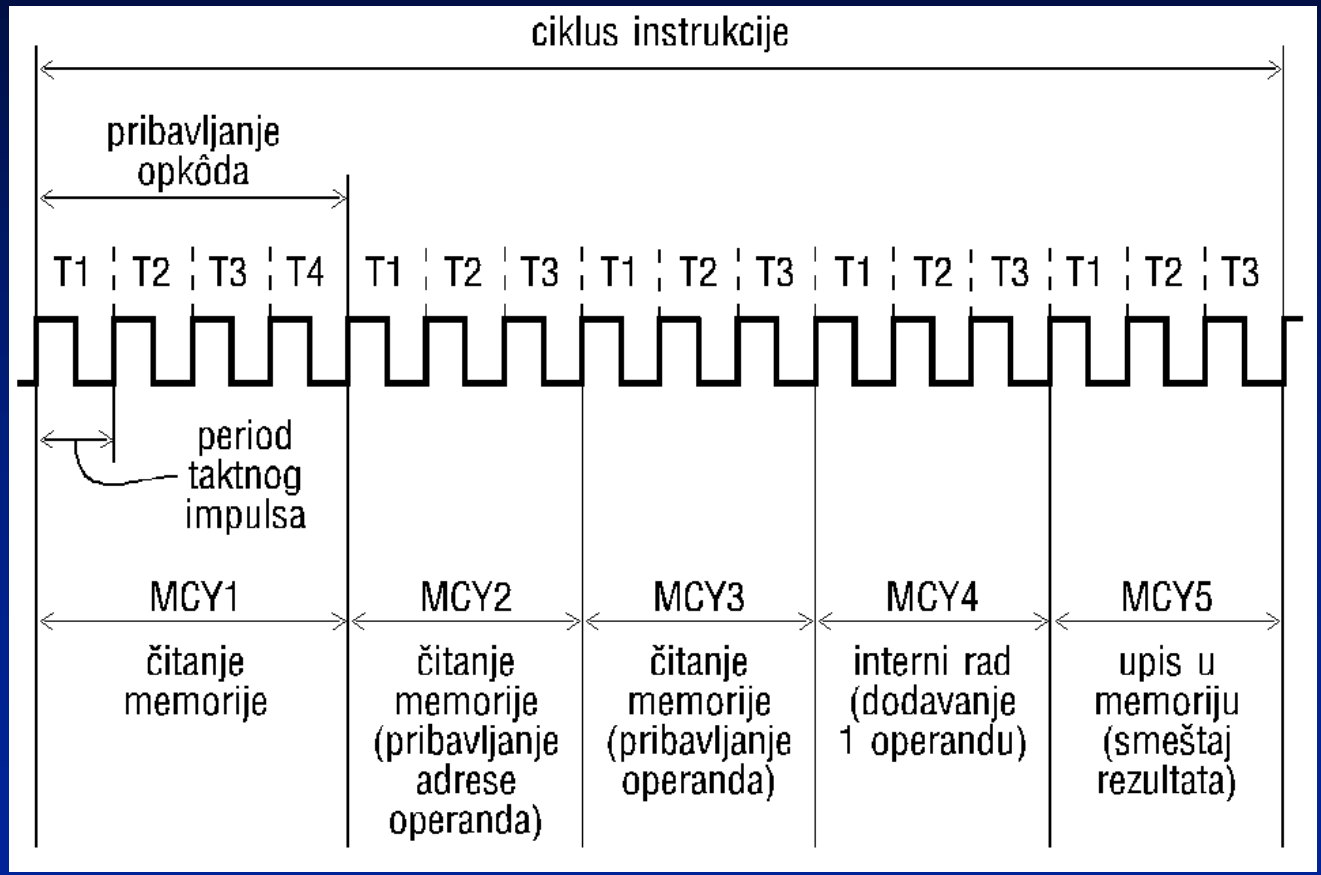


Organizacija adresnog prostora

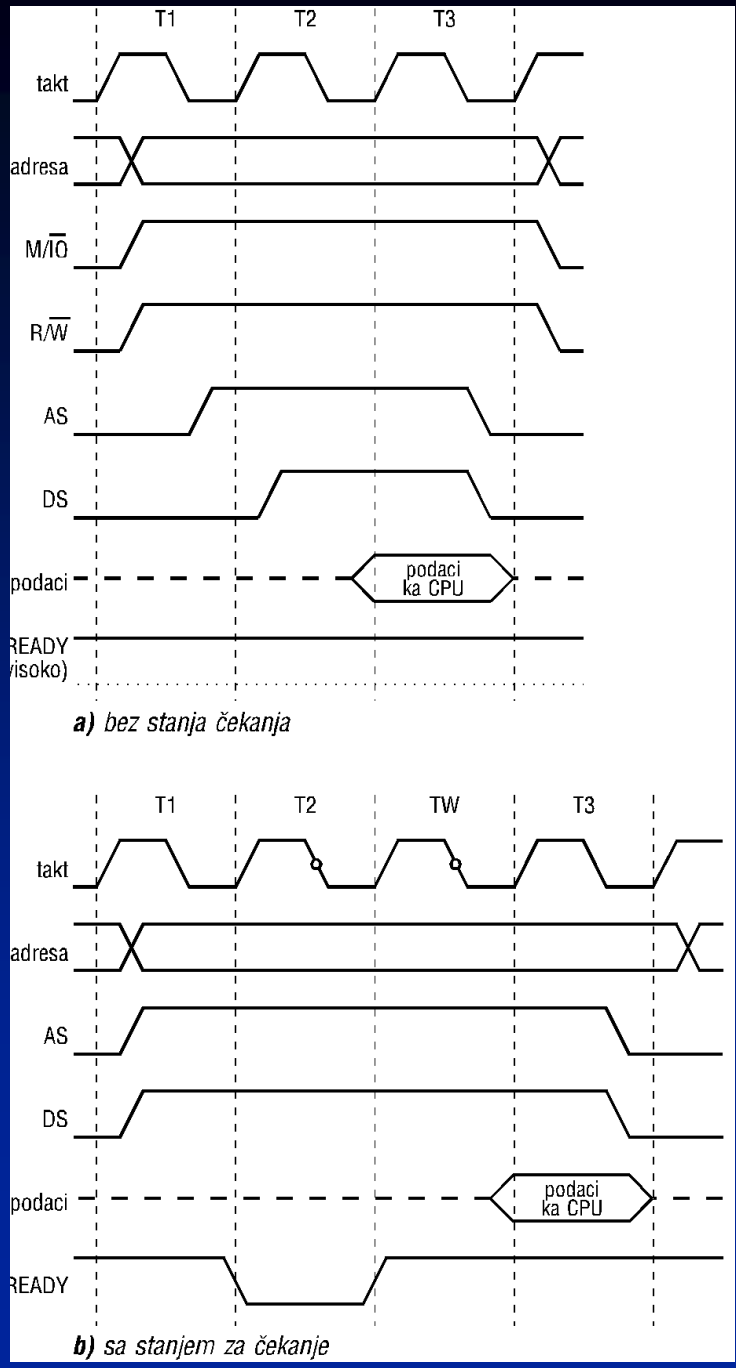


Vremenski redosled izvršenja instrukcije

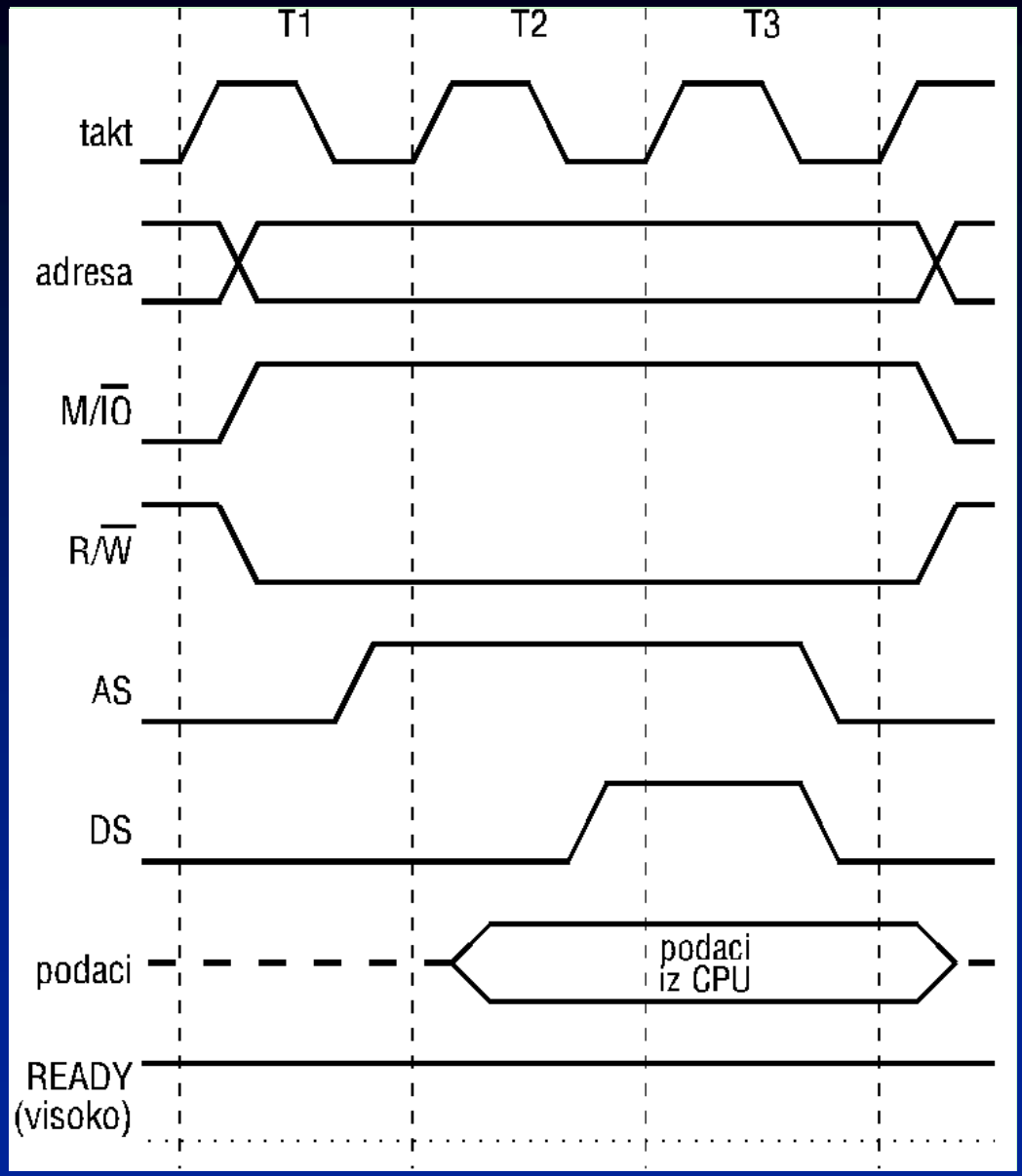
- **Inkrementiraj memoriju direktno**



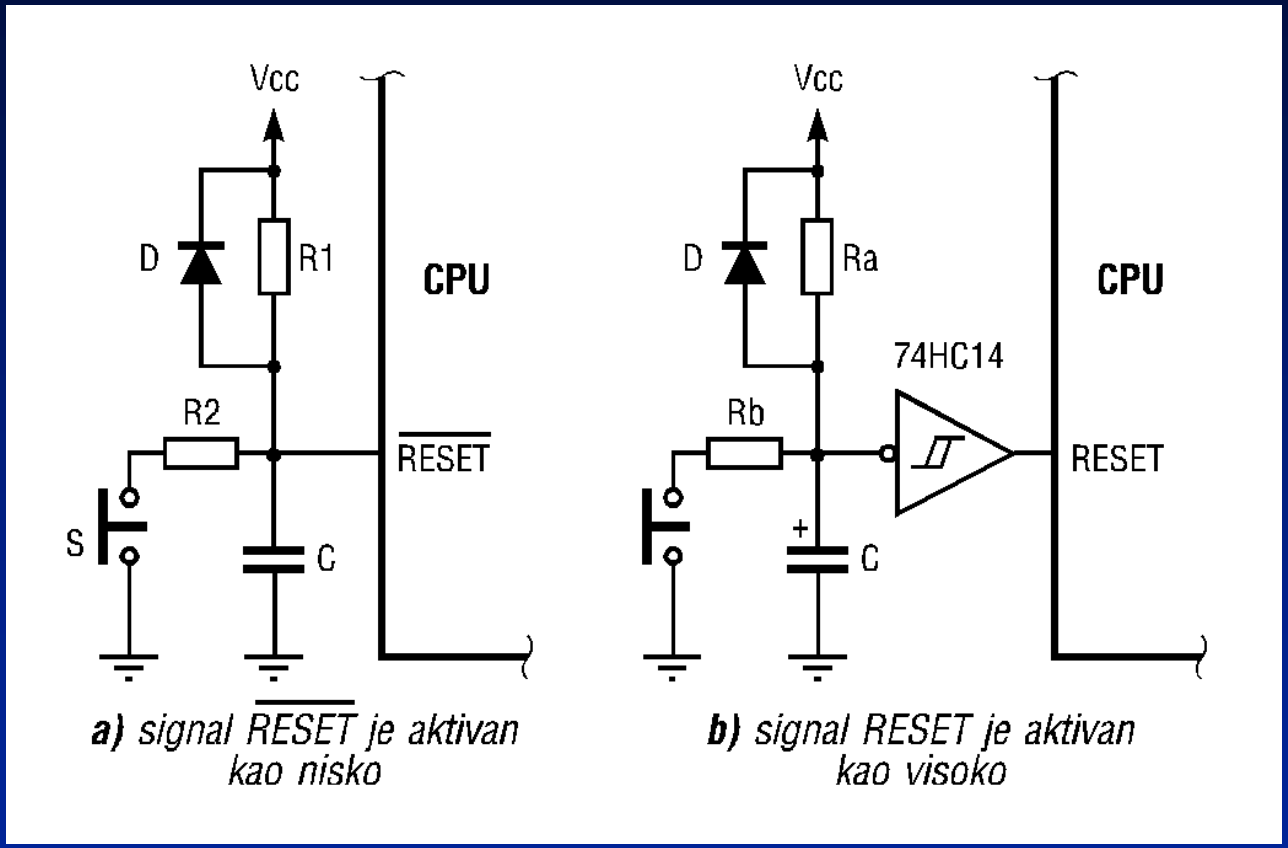
Talasni dijagrami karakterističnih signala kod mašinskog ciklusa čitanja memorije



Mašinski ciklus upis u memoriju



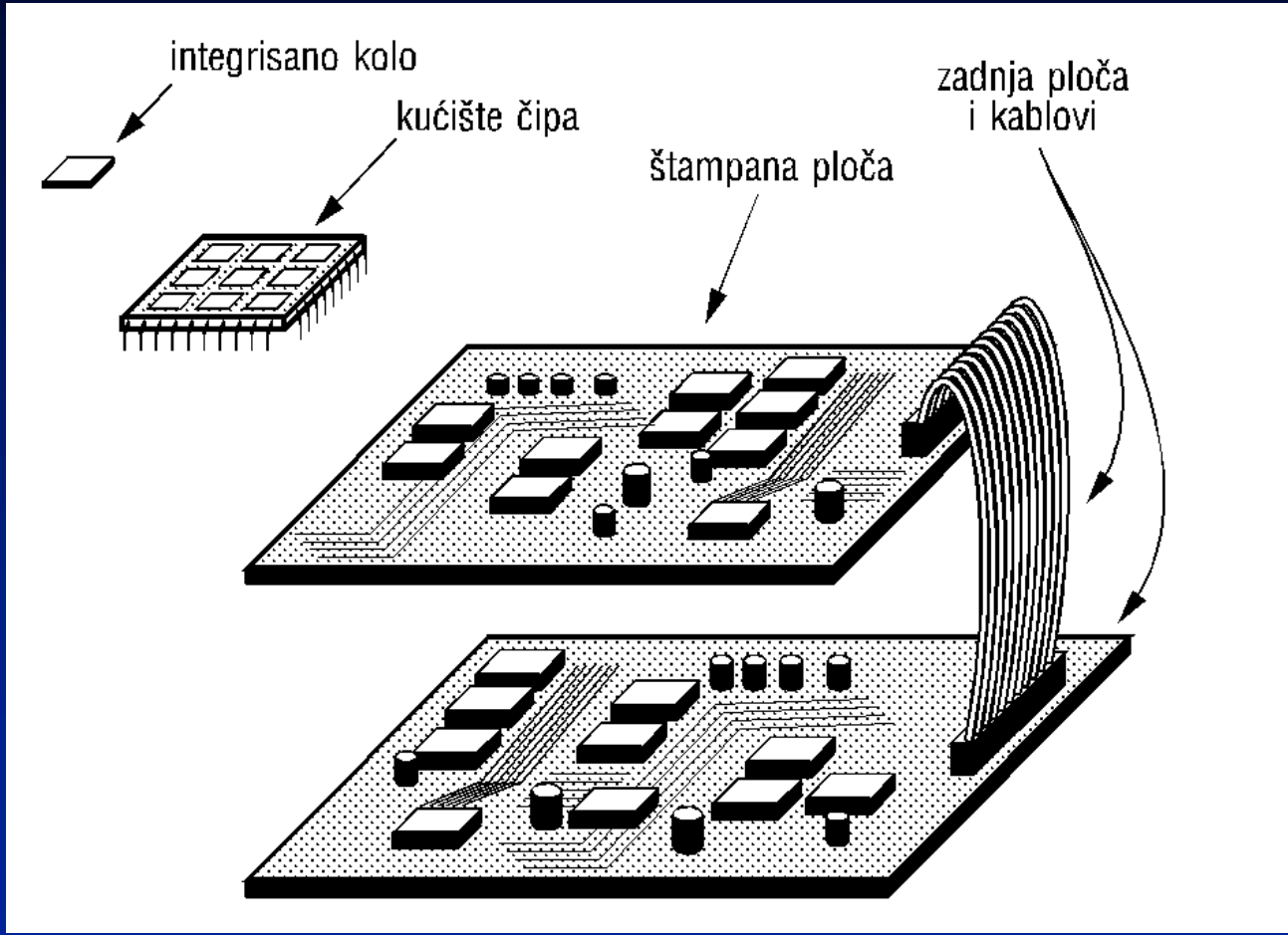
Jednostavno kolo za reset kod mikroprocesora



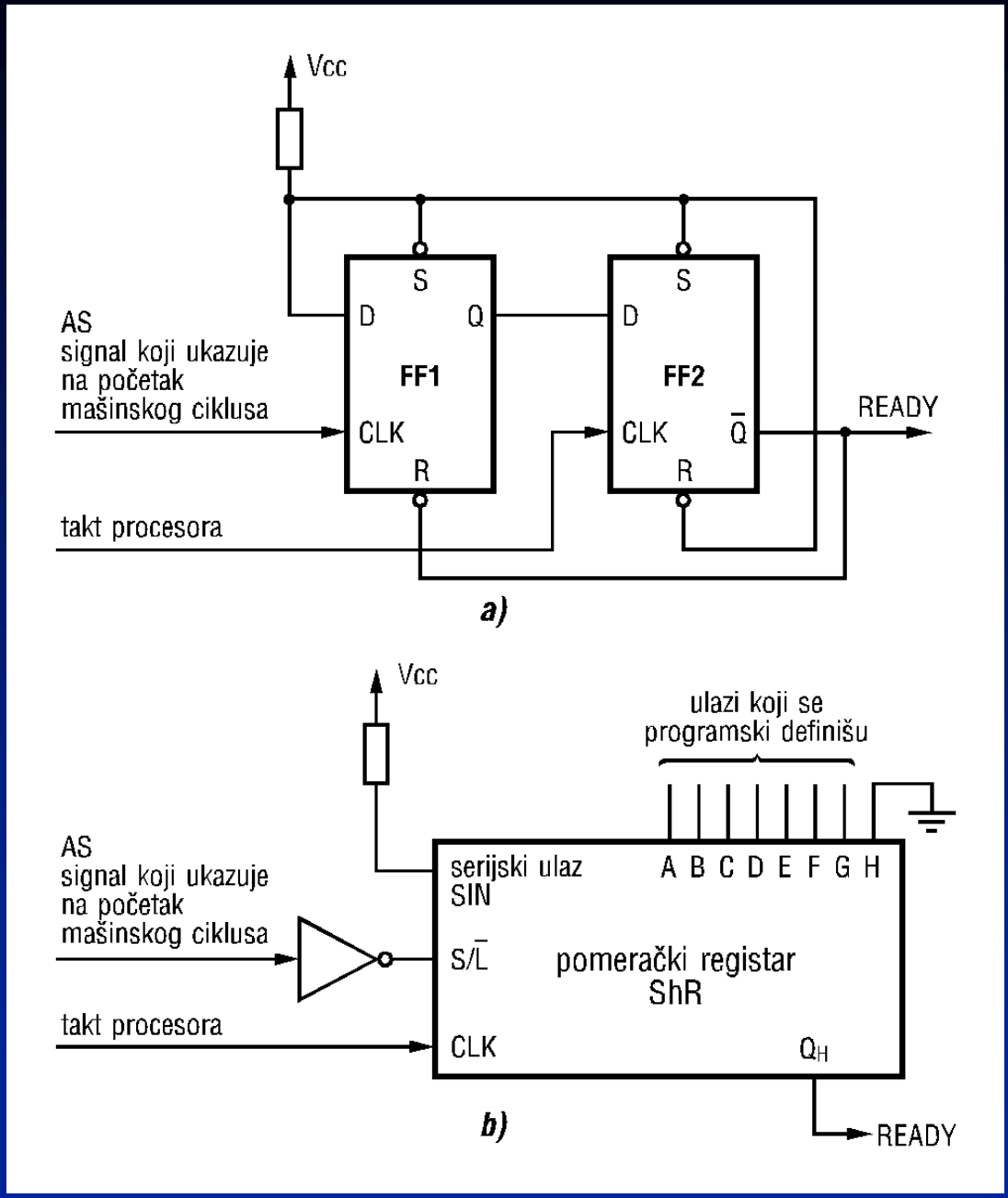
a) *signal RESET je aktivan kao nisko*

b) *signal RESET je aktivan kao visoko*

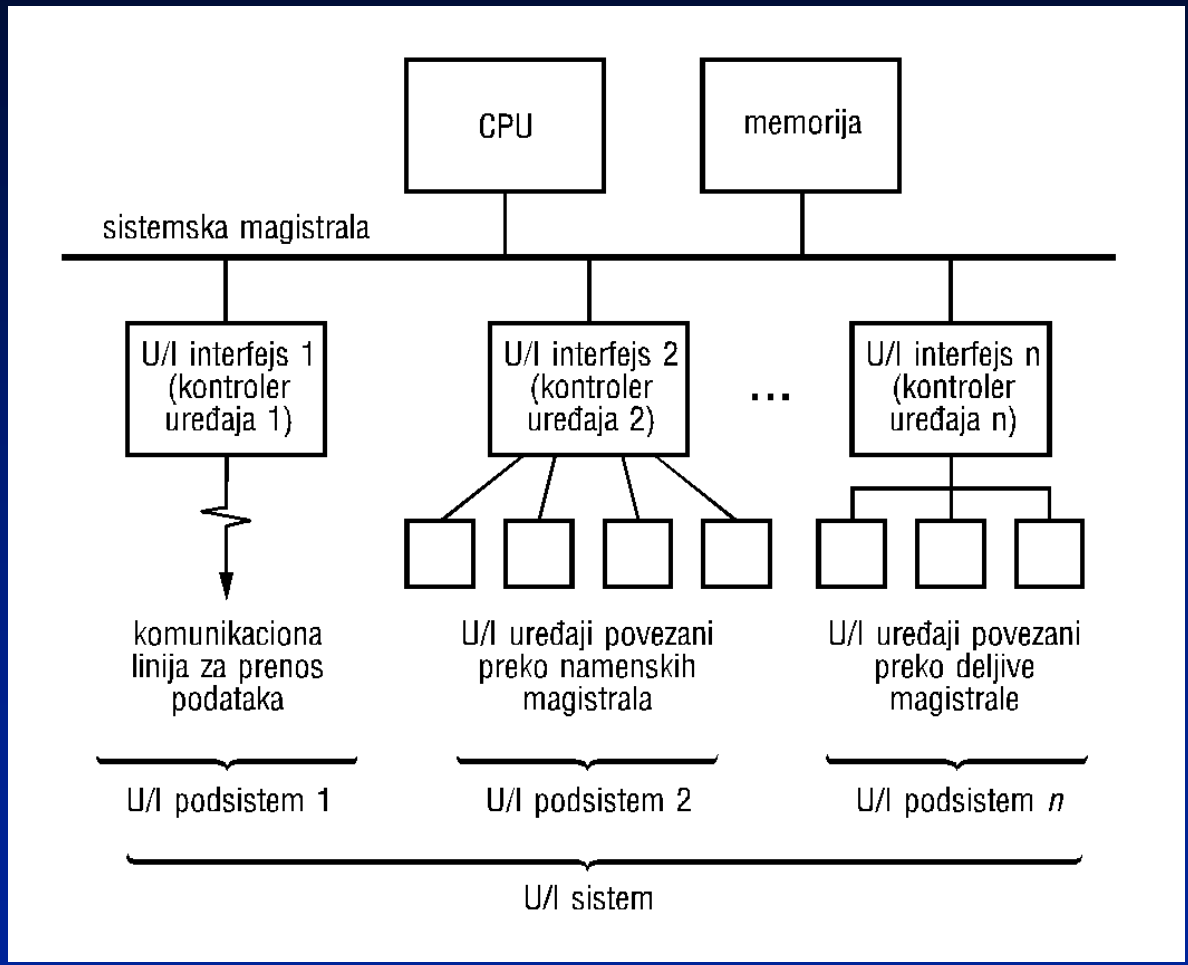
Hijerarhijsko pakovanje



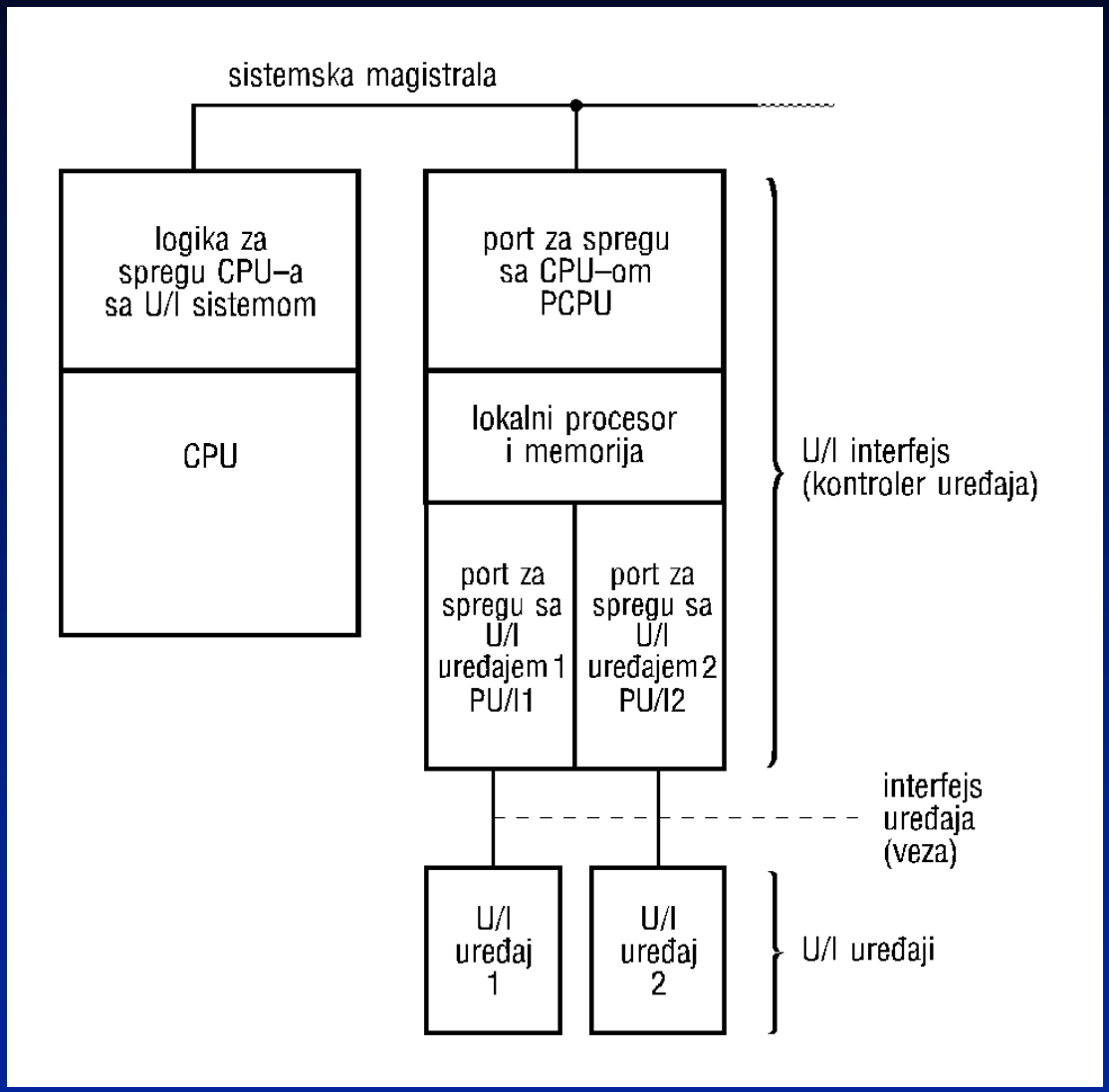
Kola za generisanje stanja čekanja



Organizacija sistema sa detaljnim prikazom U/I podsistema



Struktura U/I podsistema



Kategorije U/I uredjaja

- U/I uredjaje možemo svrstati u tri kategorije:

  - čitljivi od strane čoveka – pogodni su za komunikaciju sa korisnikom računara (video displej terminali, štampači, itd.)
  - čitljivi od strane mašine – pogodni su za komunikacijun sa opremom (magnetni diskovi, magnetne trake, itd.)
  - komunikacioni – pogodni za komunikaciju sa udaljenim uredjajima

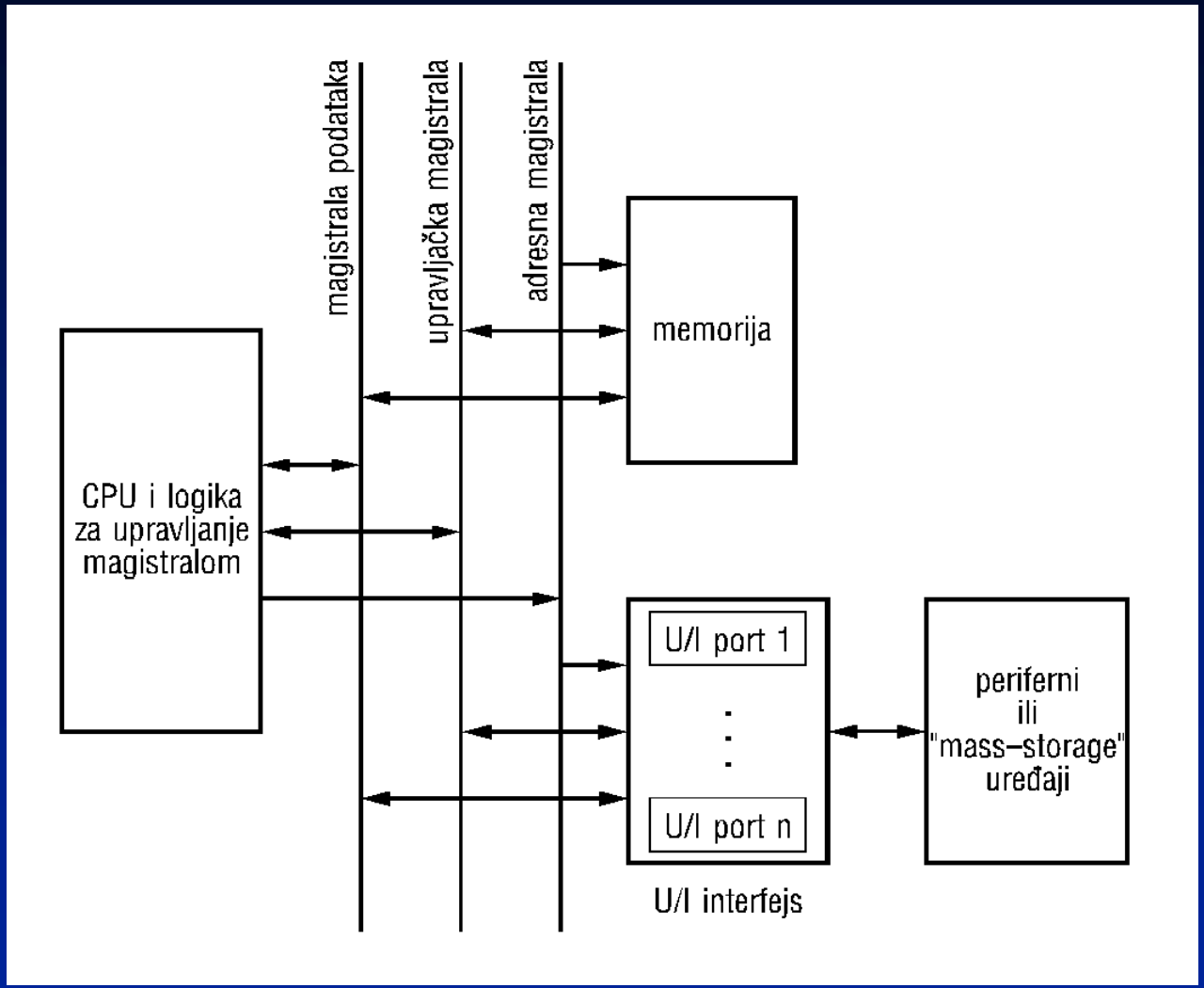
Problemi kod projektovanja U/I sistema

- **Postoje dva problema:**
  - **Na koji način se spreže periferal sa mikroračunarom i ostvaruje prenos podataka, statusa i upravljačkih signala**
  - **Kako se ostvaruje adresiranje U/I uredjaja**

Memorija vs. U/I sistem

- Memorija je specifična u odnosu na U/I podsistem iz sledećih razloga:
  - Radi skoro istom brzinom kao i CPU
  - Koristi iste tipove signala kao i CPU
  - Ne zahteva specijalne oblike ili bilo kakve upravljačke signale sa izuzetkom Read/Write i CS impusle
  - Automatski lečuje podatak koji se šalje ka njoj
  - Širina reči je ista kao i kod CPU-a

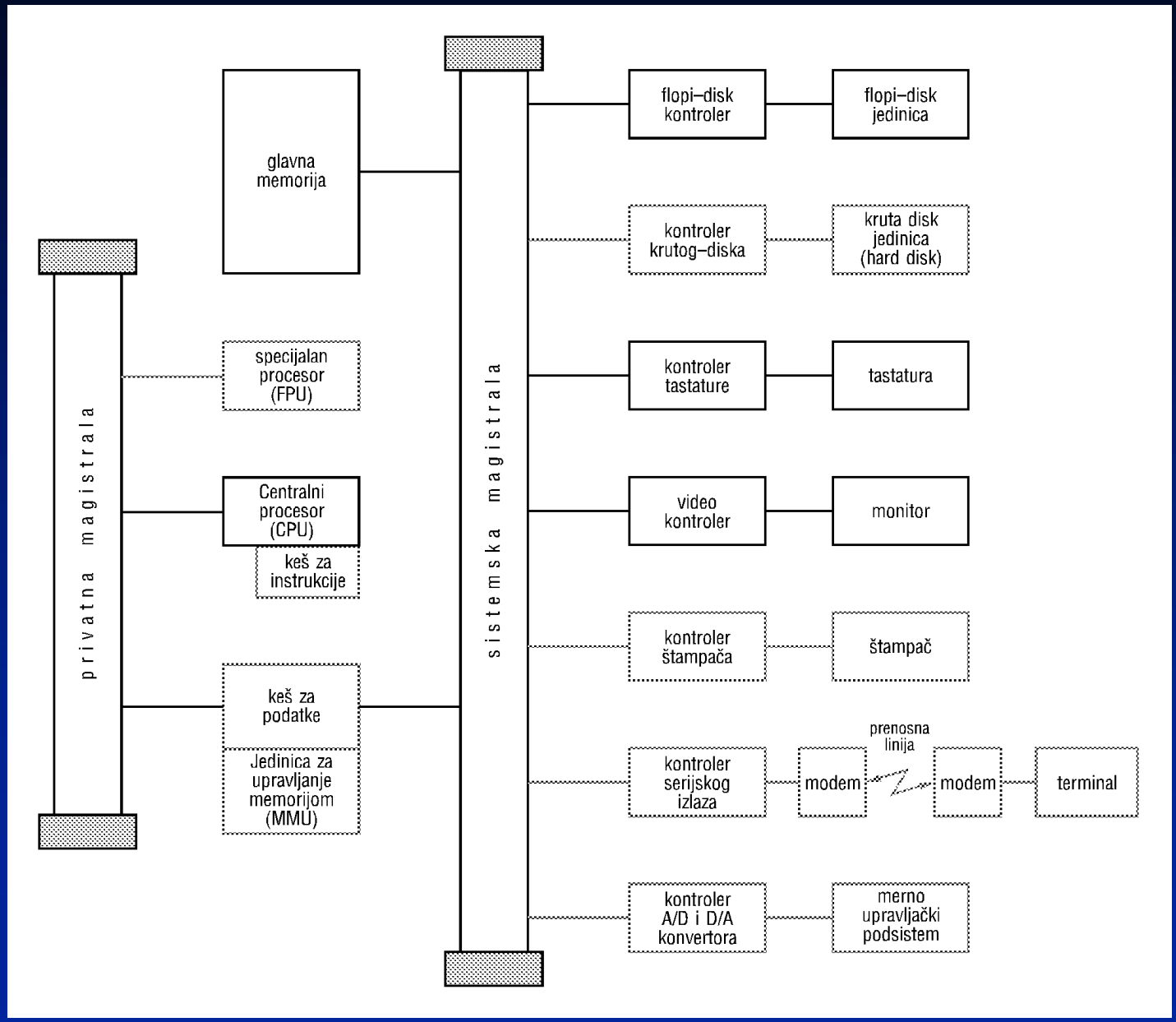
Povezivanje U/I interfejsa na magistralu



Specifičnosti U/I uredjaja

- Svaku periferiju karakteriše specifičan interfejs problem:
  - Postoji širok izbor perifernih uredjaja čiji je princip rada različit
  - Brzina prenosa kod perifernih uredjaja je često mnogo manja od one koja je svojstvena CPU ili memoriji
  - Periferali često koriste različite formate podataka kao i dužine blokova podataka u odnosu na CPU na koji su povezani

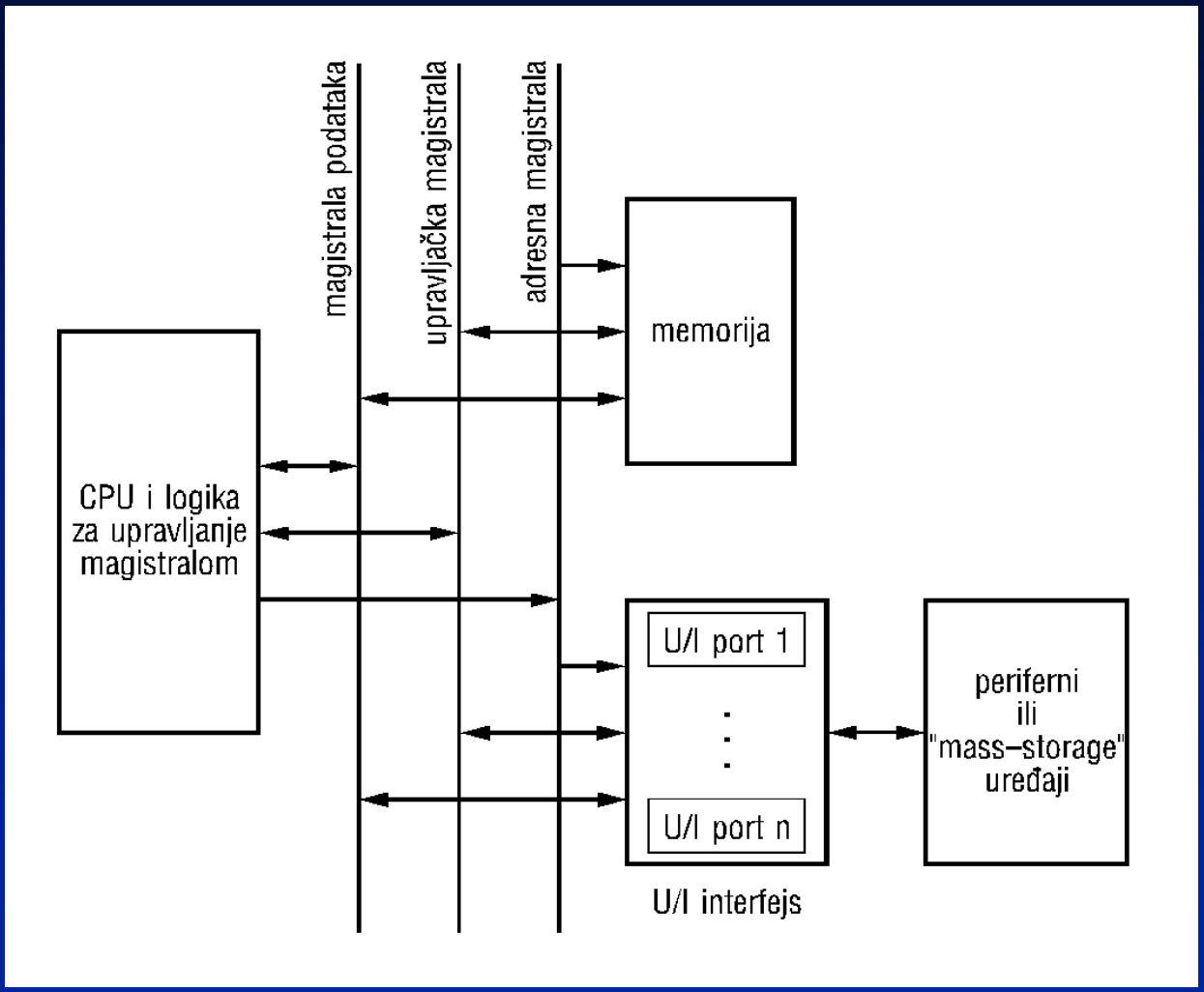
Povezivanje U/I periferija preko interfejsa



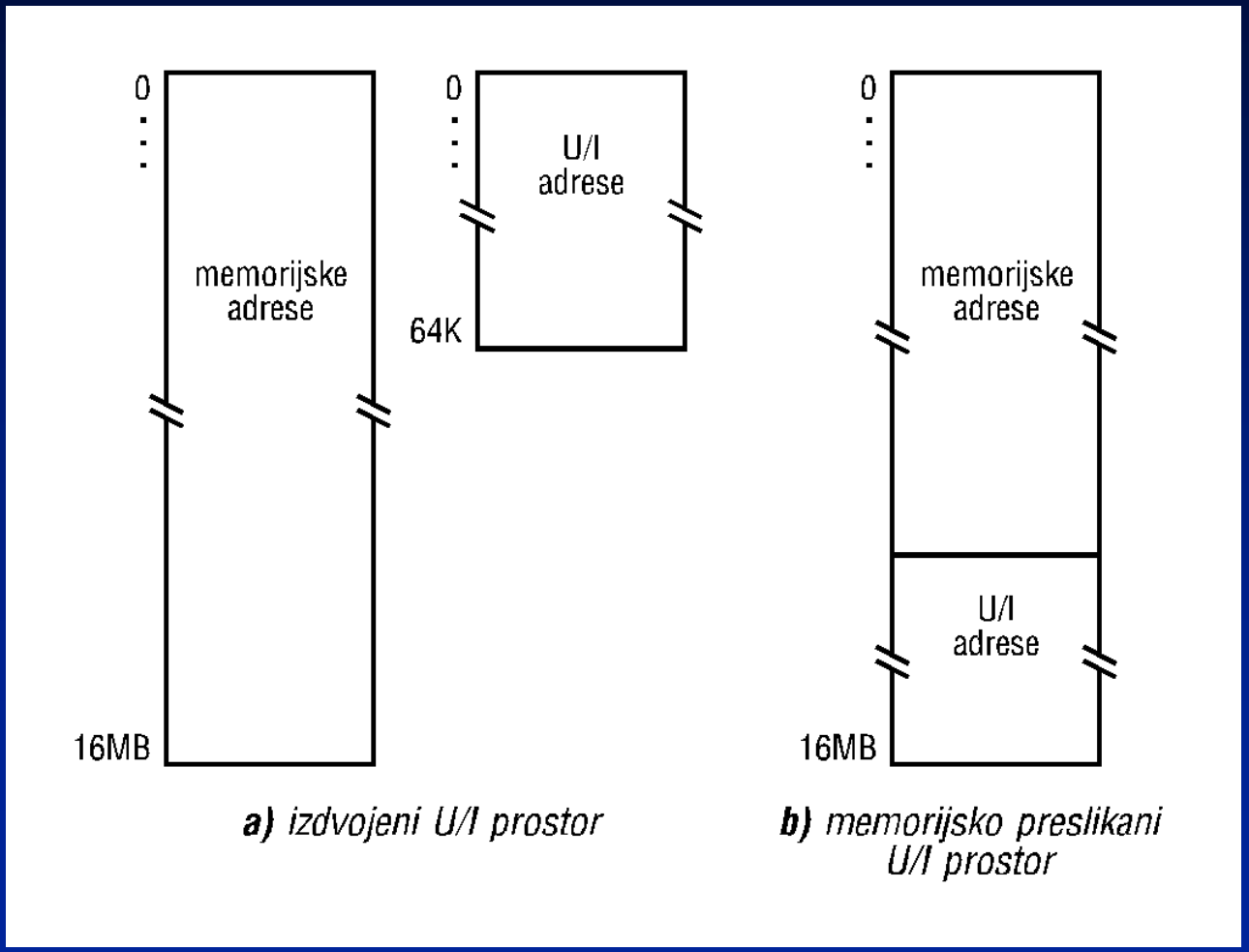
Tipovi U/I uredjaja u odnosu na brzinu

- **U zavisnosti od brzine sa kojom se vrši prenos podataka U/I uredjaje delimo na sledeće tri grupe:**
  - **Spori uredjaji – ne menjaju svoje stanje više od jedanput u sekundi (releji, mehanički senzori, aktuatori, itd.)**
  - **Uredjaji srednjih brzina – karakteriše ih brzina prenosa od 1 do 10 000 bitova u sekundi (tastature, štampači, itd.)**
  - **Uredjaji većih brzina – karakterišu se brzinom prenosa podataka koja je veća od 10 000 bitova u sekundi (brzi štampači, magnetni diskovi, brze komunikacione linije, CRT displeji, itd.)**

Način sprezanja U/I interfejsa na sistemsku magistralu



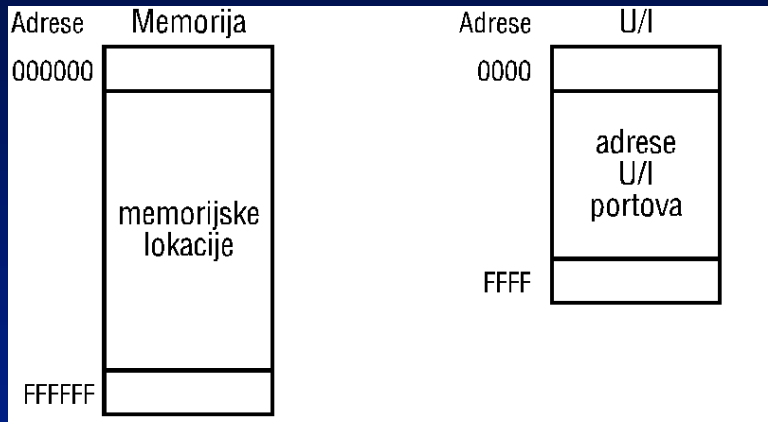
Adresiranje U/I uredjaja
- način podele adresnog prostora-



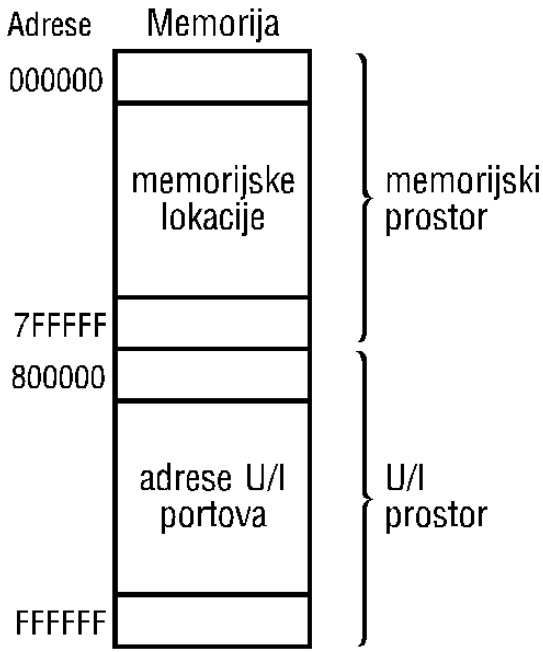
a) *izdvojeni U/I prostor*

b) *memorijsko preslikani U/I prostor*

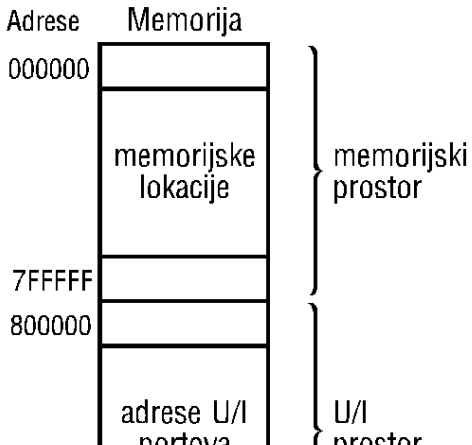
Realizacija memorijsko-preslikanog i izdvojenog U/I-a



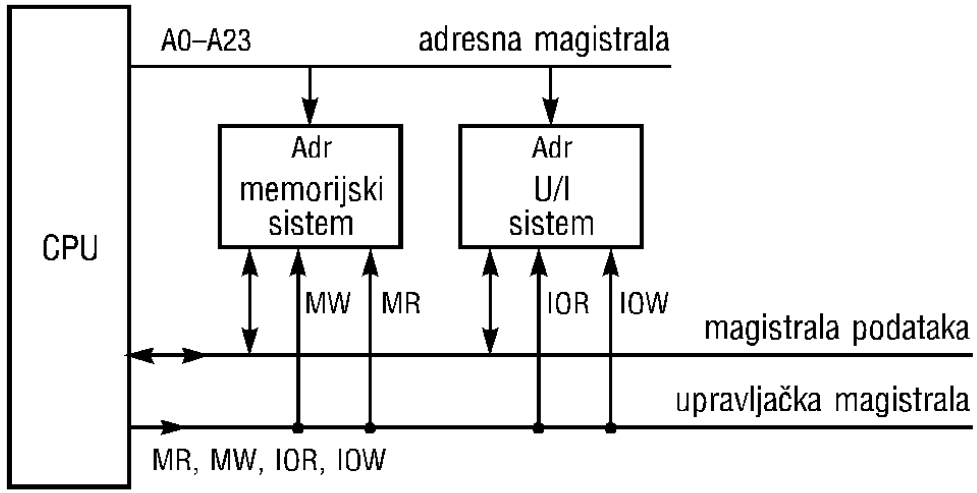
a) izdvojeni U/I – 16M memorijski i 64k U/I prostor



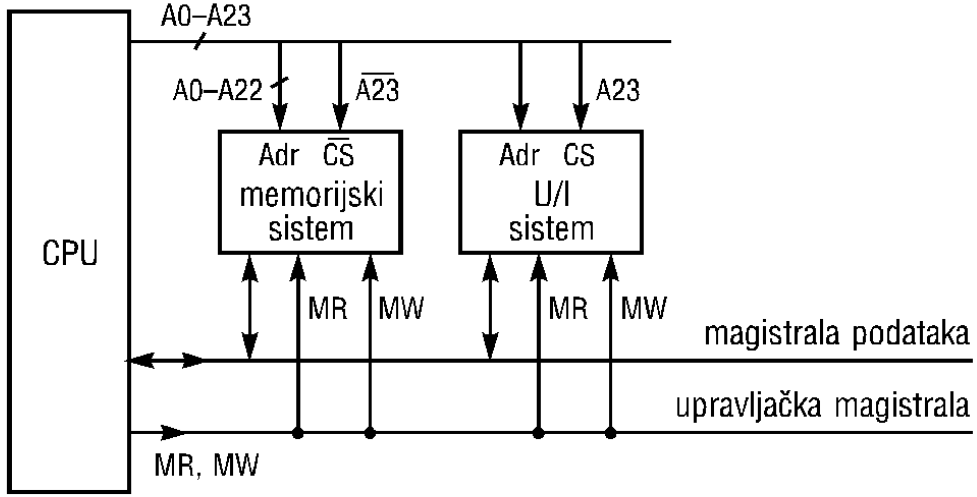
b) memorijsko preslikani U/I – 8M memorijski i 8M U/I prostor



Realizacija memorijsko-preslikanog i izdvojenog U/I-a - prod.



a) *izdvojeni U/I*
(MR, MW – čitanje i upis memorije; IOR, IOW – čitanje i upis U/I)



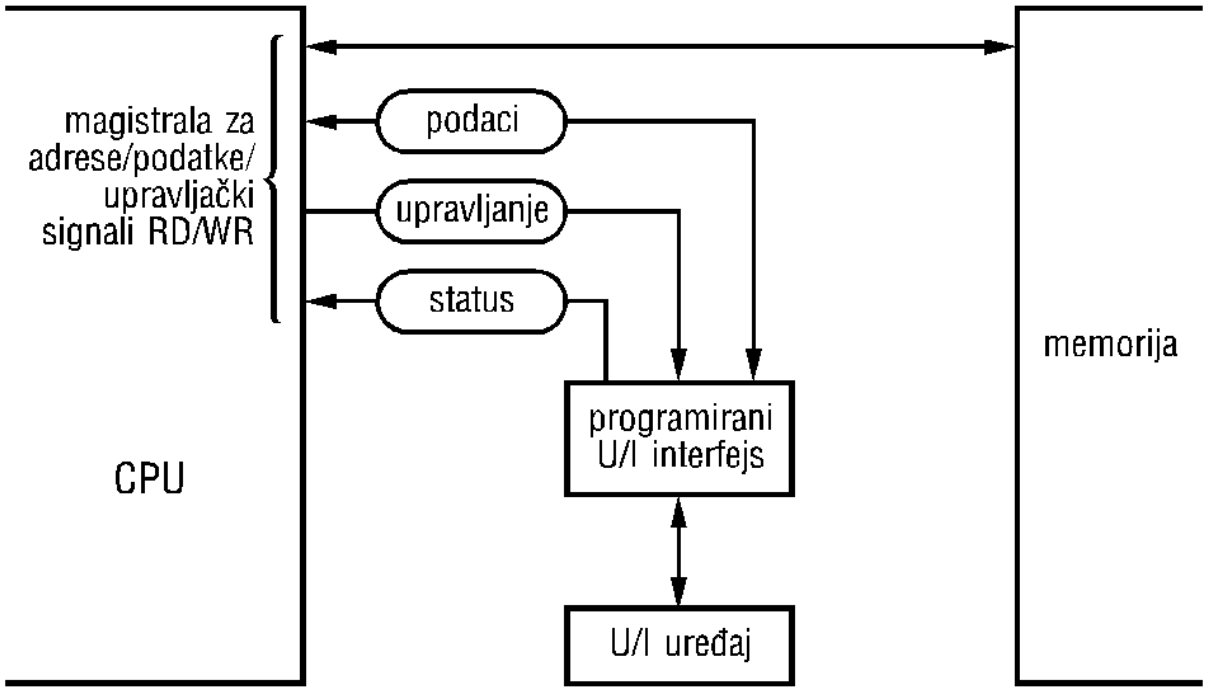
b) *memorijsko–preslikani U/I*

U/I tehnike za prenos podataka

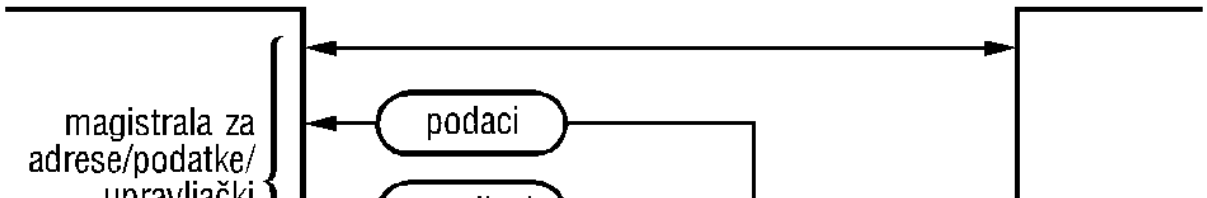
- **Tehnike za U/I prenos podataka se dele na sledeće tri grupe:**
  - **Programirana U/I tehnika – *Pragrammed I/O technique***
  - **Prekidna U/I tehnika – *Interrupt I/O technique***
  - **Direktni pristup memoriji - *DMA (Direct Memory Access)***

Programirana U/I i prekidna U/I tehnika se obavljaju pod programskom kontrolom, dok se DMA obavlja pod hardverskom kontrolom

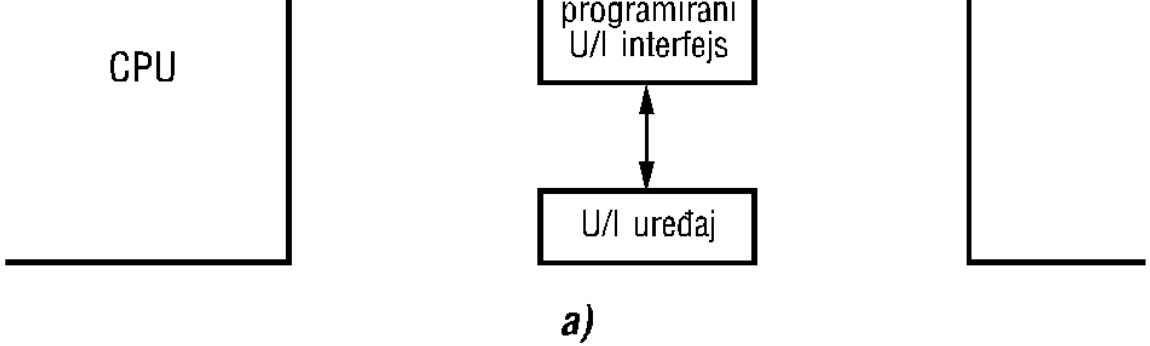
Tokovi podataka kod različitih tipova U/I prenosa - programirani U/I-



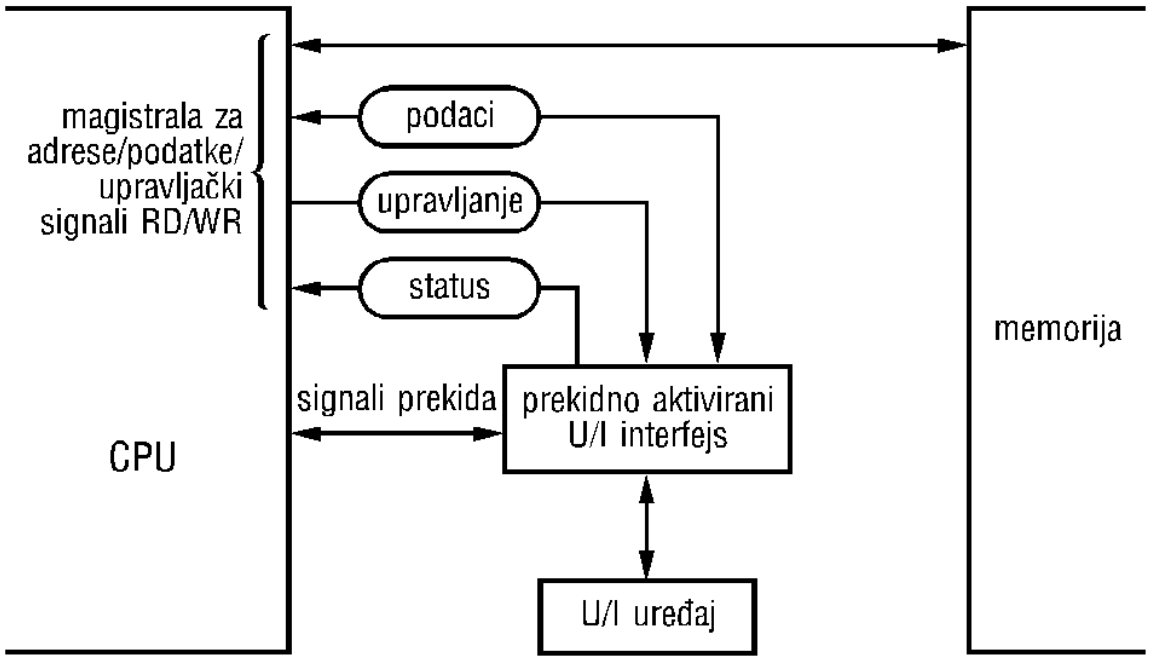
a)



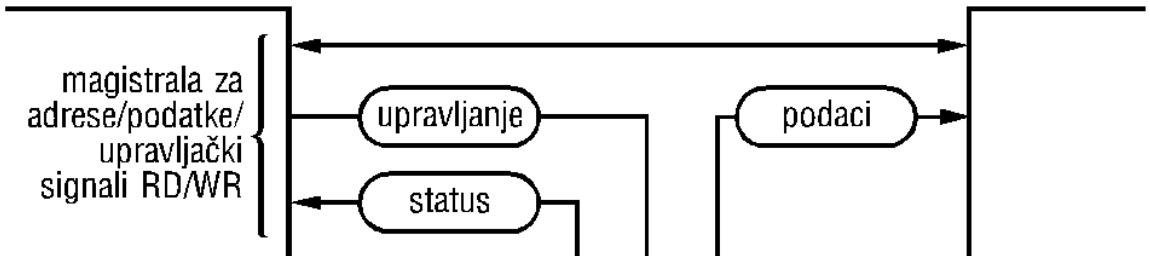
Tokov a U/I


CPU
programirani U/I interfejs
U/I uređaj

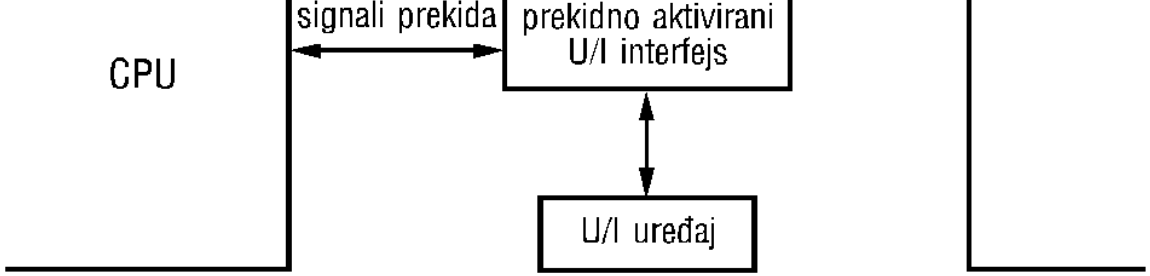
a)


magistrala za adrese/podatke/ upravljački signali RD/WR
podaci
upravljanje
status
signali prekida
prekidno aktivirani U/I interfejs
memorija
CPU
U/I uređaj

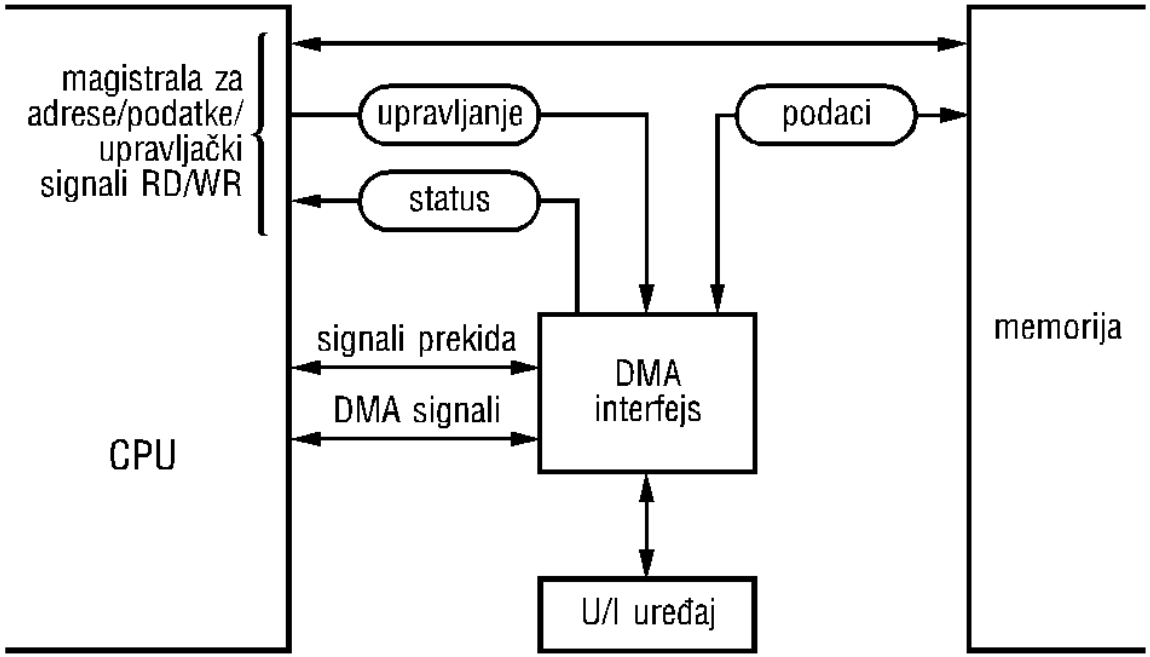
b)


magistrala za adrese/podatke/ upravljački signali RD/WR
upravljanje
status
podaci

Tokov a U/I


CPU
signali prekida
prekidno aktivirani
U/I interfejs
U/I uređaj

b)

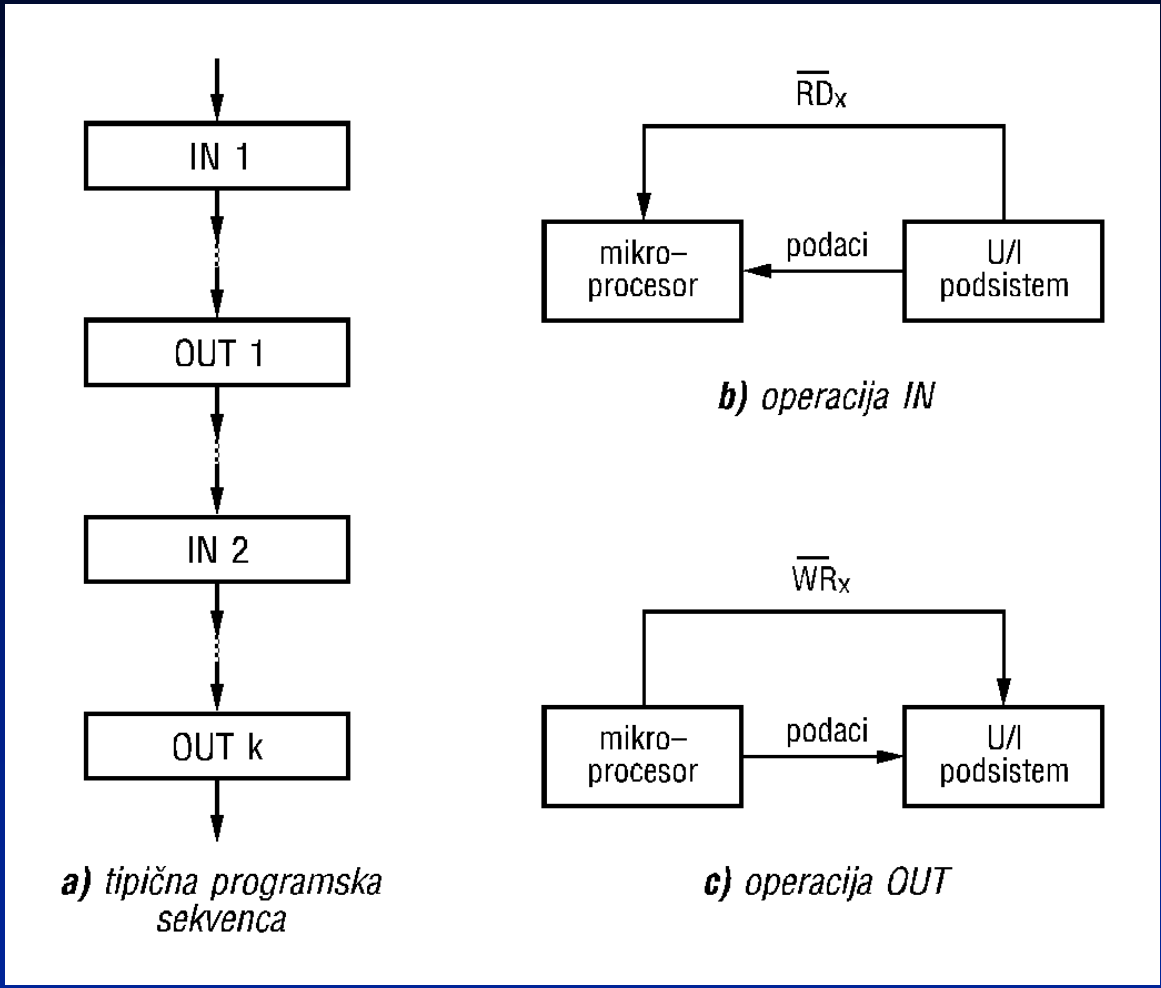

magistrala za
adrese/podatke/
upravljački
signali RD/WR
upravljanje
status
podaci
signali prekida
DMA signali
DMA
interfejs
memorija
CPU
U/I uređaj

c)

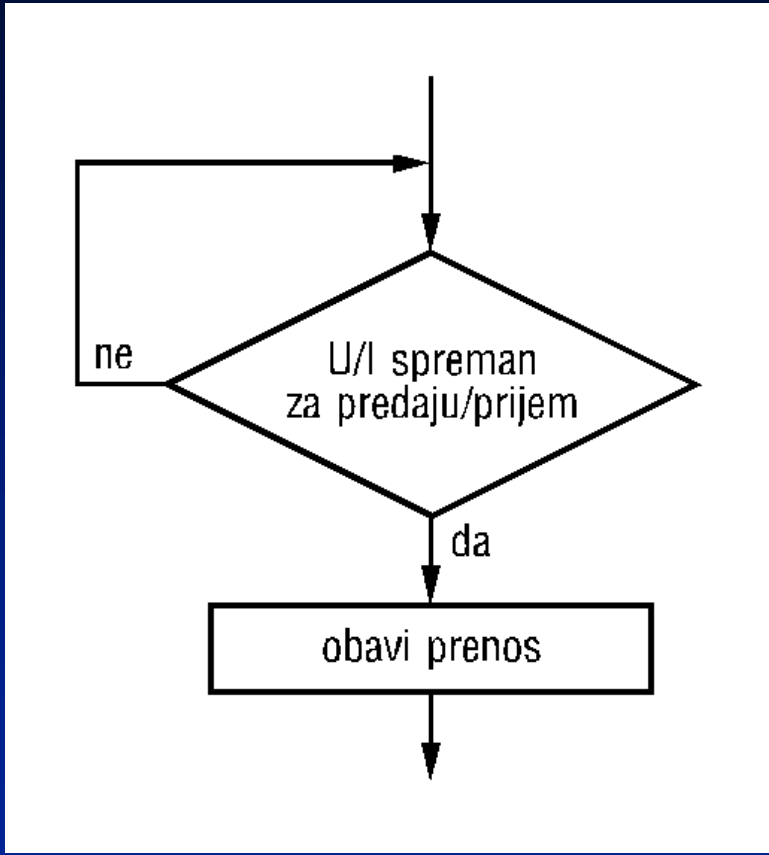
Tipovi programiranog U/I prenosa

- **Postoje dva tipa programiranog U/I:**
  - **Bezuslovni U/I prenos – karakteristike su sledeće:**
    - Koristi se tamo gde je vreme odziva spoljnih perifernih uredjaja poznato i stalno.
    - Uredjaji moraju biti spremni za prenos podataka u trenutku izvodjenja U/I naredbe.
    - Prenos se lako izvodi
  - **Uslovni U/I prenos – karakteristike su sledeće:**
    - U toku prenosa mikroračunar ispituje status U/I
    - Ako U/I uredjaj nije spreman programski se ispituje njegov status sve dok se ne ispuni uslov za predaju ili prijem
    - Ovakav tip prenosa koristi se na onim mestima i u onim slučajevima gde se ne sme preuzeti akcija ako prethodno nisu ispunjeni neki uslovi

Bezuslovni U/I prenos



Uslovni U/I prenos

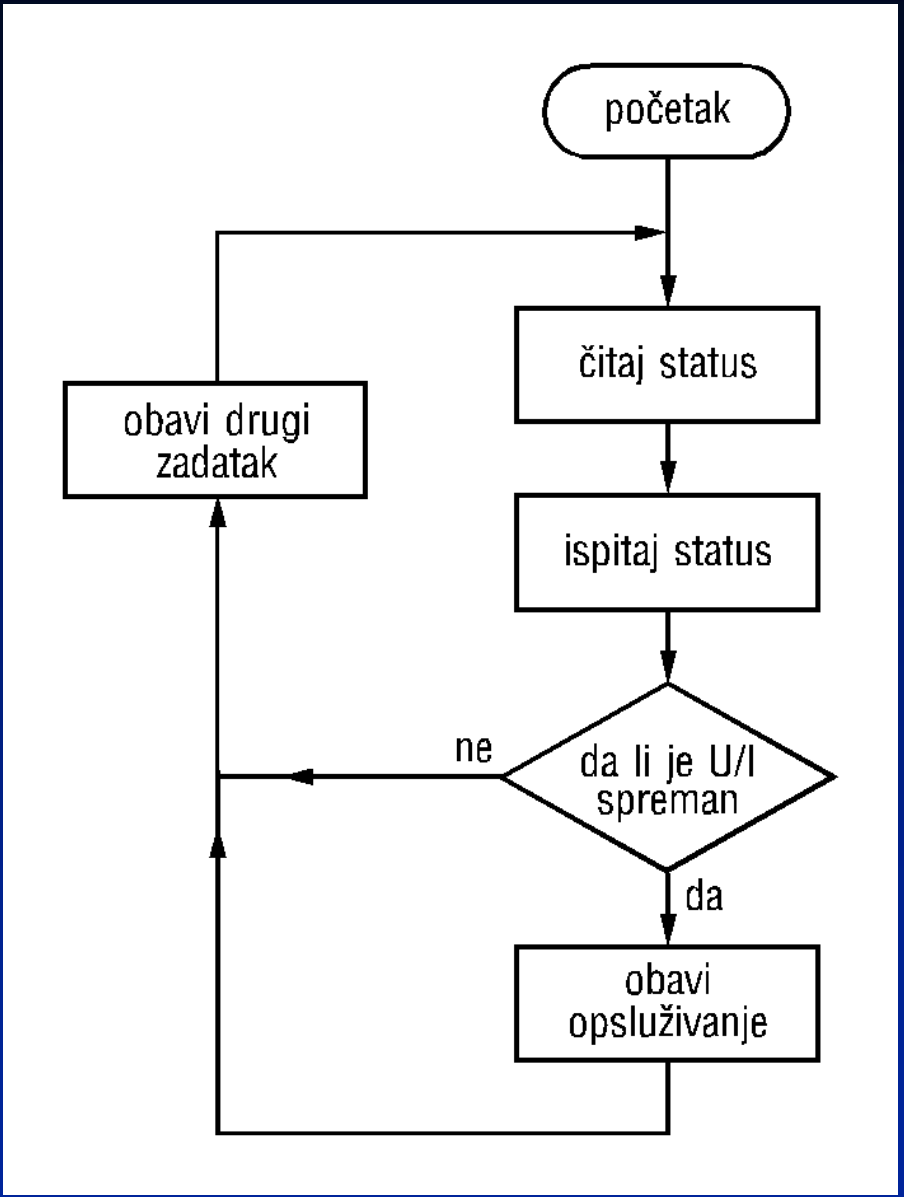


Nedostaci programiranog U/I prenosa

- **Programirani U/I prenos ima sledeća dva glavna nedostatka:**
  - **Brzina prenosa je ograničena brzinom sa kojom CPU može da testira i opslužuje U/I uredjaje**
  - **Vreme koje CPU potroši na testiranje statusa U/I uredjaja kao i vreme koje je potrebno da se obavi prenos podataka često je moguće efikasnije iskoristiti**

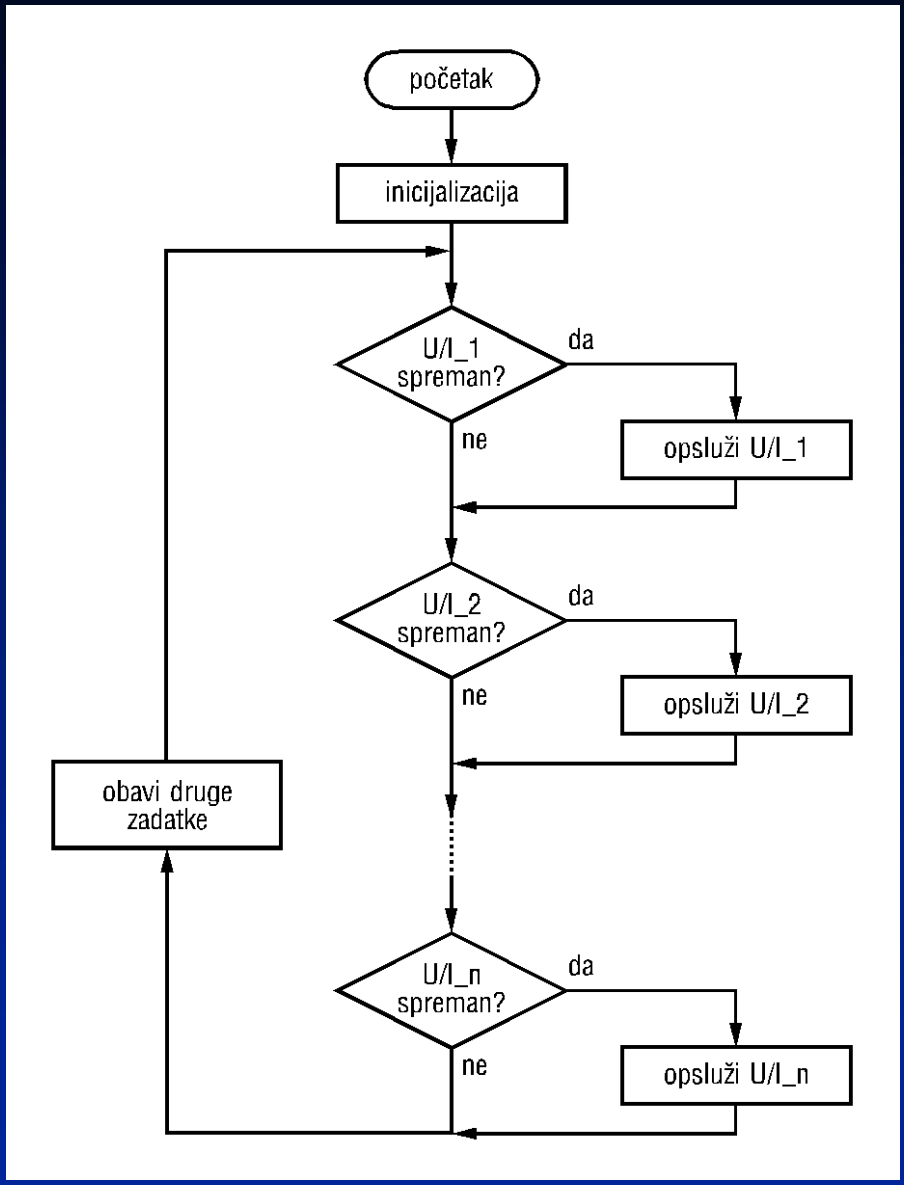
Tehnika za opsluživanje U/I uredjaja

-kružno ispitivanje (*pooled I/O*)-



Tehnika za kružno opsluživanje većeg U/I uredjaja

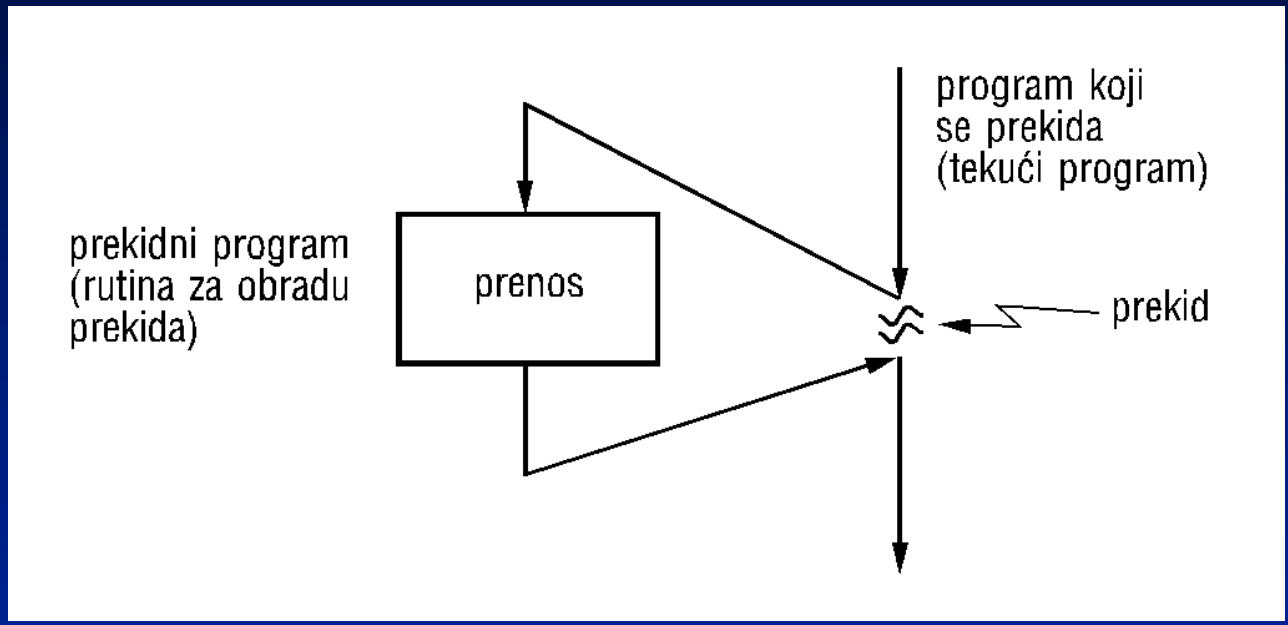
-Round-Robin Arangement-



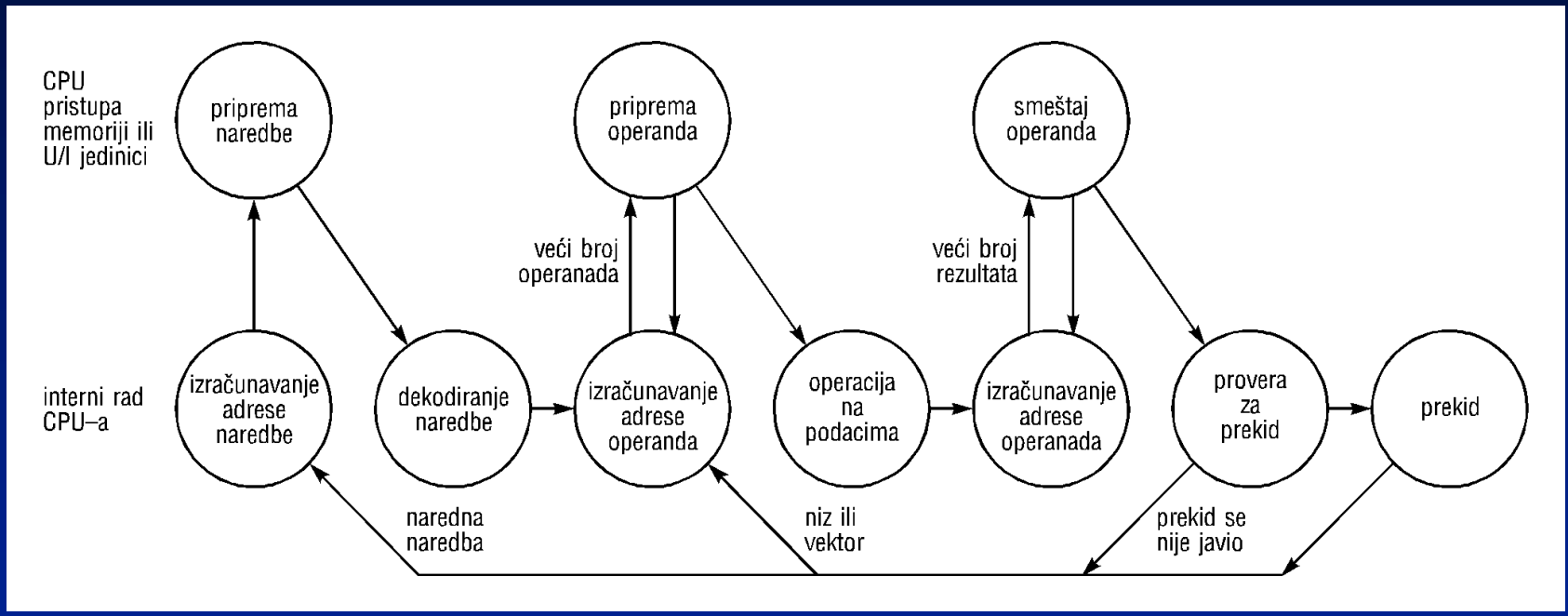
Prekidna U/I tehnika

- Kod prekidne U/I tehnike prenos podataka se inicira od strane U/I uredjaja koji signalaizira CPU-u kada je spreman za prenos podataka
- Ne postoji više potreba za permanentnim testiranjem statusa U/I uredjaja
- Dva osnovna problema koja se javljaju kod projektovanja prekidne U/I tehnike su sledeća:
  - Kako CPU odredjuje koji je uredjaj zahtevao prekid ?
  - Ako se veći broj tehnika javi istovremeno, kom će zahtevu CPU najpre posvetiti pažnju i obraditi ga ?

Prihvatanje zahteva za prekid i obrada

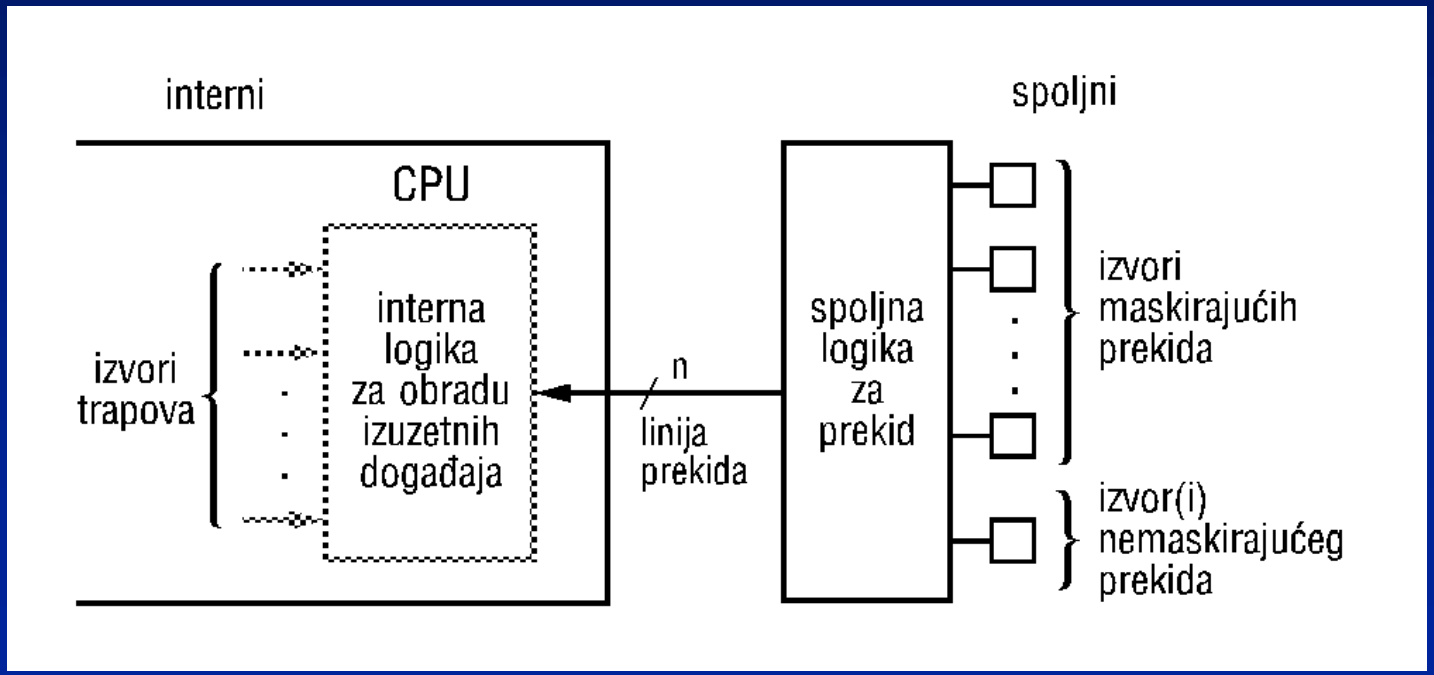


Prošireni dijagram stanja procesora u toku izvršenja ihnstrukcija

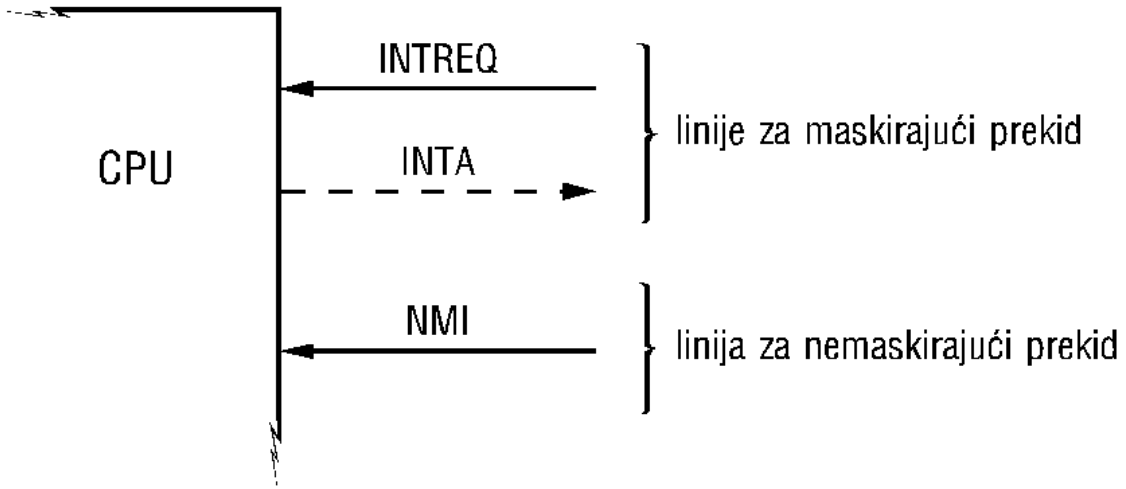


Izvori prekida

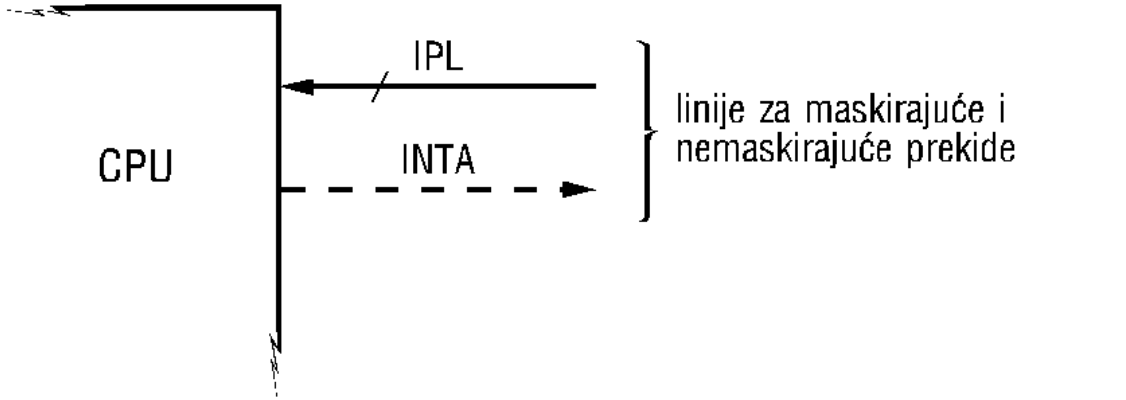
- **Prekidi se mogu inicirati:**
  - **Spoljašnje u odnosu na CPU – mogu biti maskirajući i nemaskirajući**
  - **Interno u okviru CPU-a kao rezultat izvršenja instrukcija, ove prekide nazivamo trap-ovi**



Spoljni interfejs signali za prekide

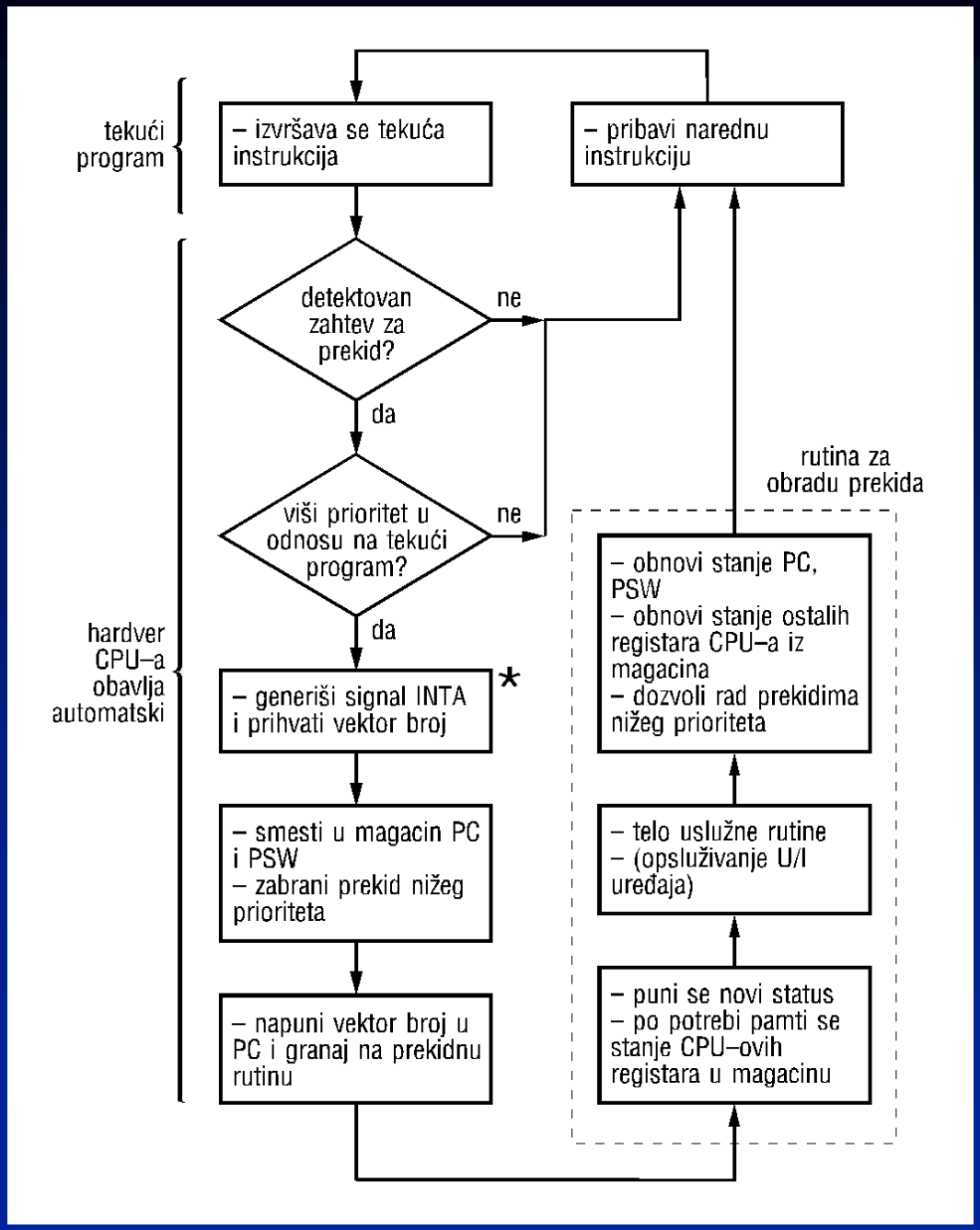


a) *izdvojene linije za maskirajuće i nemaskirajuće prekide*



b) *zajedničke linije za maskirajuće i nemaskirajuće prekide*

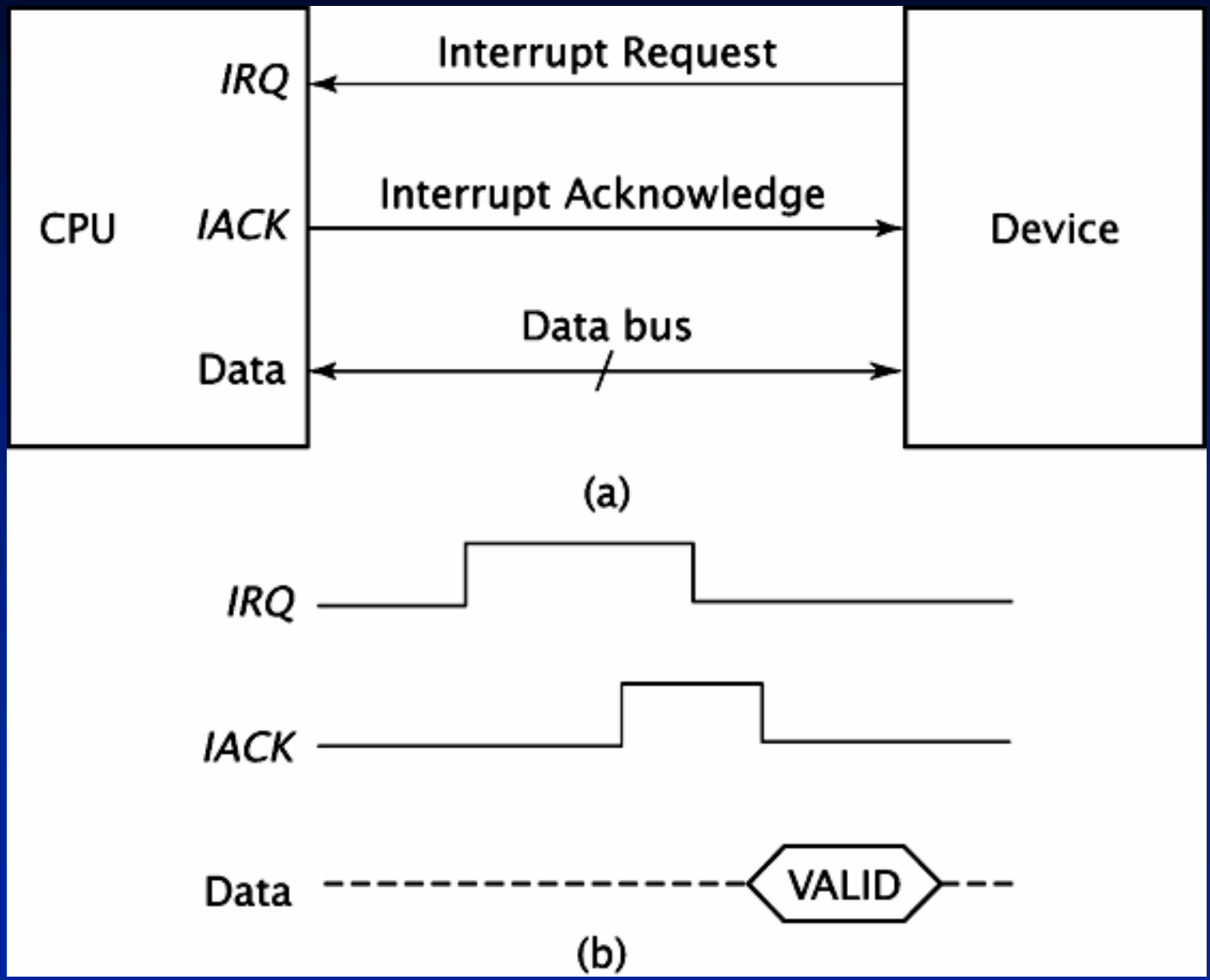
Obrada prekida



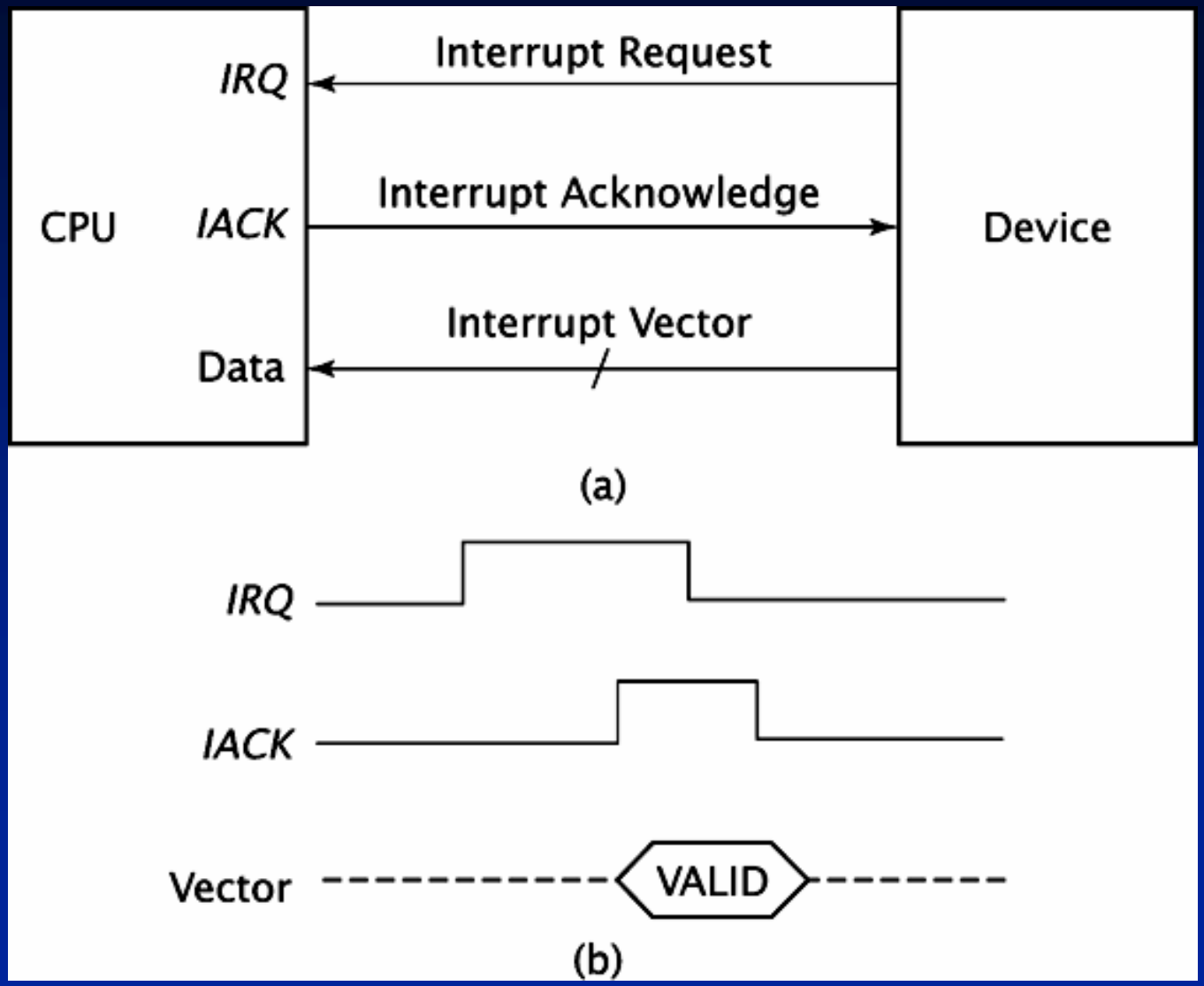
Prihvatanje adrese za obradu prekida

- **U odnosu na način kako CPU odredjuje adresu uredjaja koji je izazvao prekid razlikujemo dva tipa prekida:**
  - **Vektorski prekidi**
    - **U/I uredjaj predaje prekidni vektor koji se koristi da generiše početnu adresu rutine za obradu tog prekida.**
    - **Vektor može da predstavlja potpunu ili parcijalnu adresu.**
    - **Korisni su kod CPU-ova koji primaju zahteve za prekid od nekoliko uredjaja preko iste uprvaljačke linije**
  - **Nevektorski prekidi**
    - **U/I uredjaj predaje zahtev za prekid CPU-u aktiviranjem linije IRQ**
    - **Kada je CPU spreman da procesira zahtev on aktivira signal IACK kojim ga obaveštava da je prihvatio zahtev za prekid**

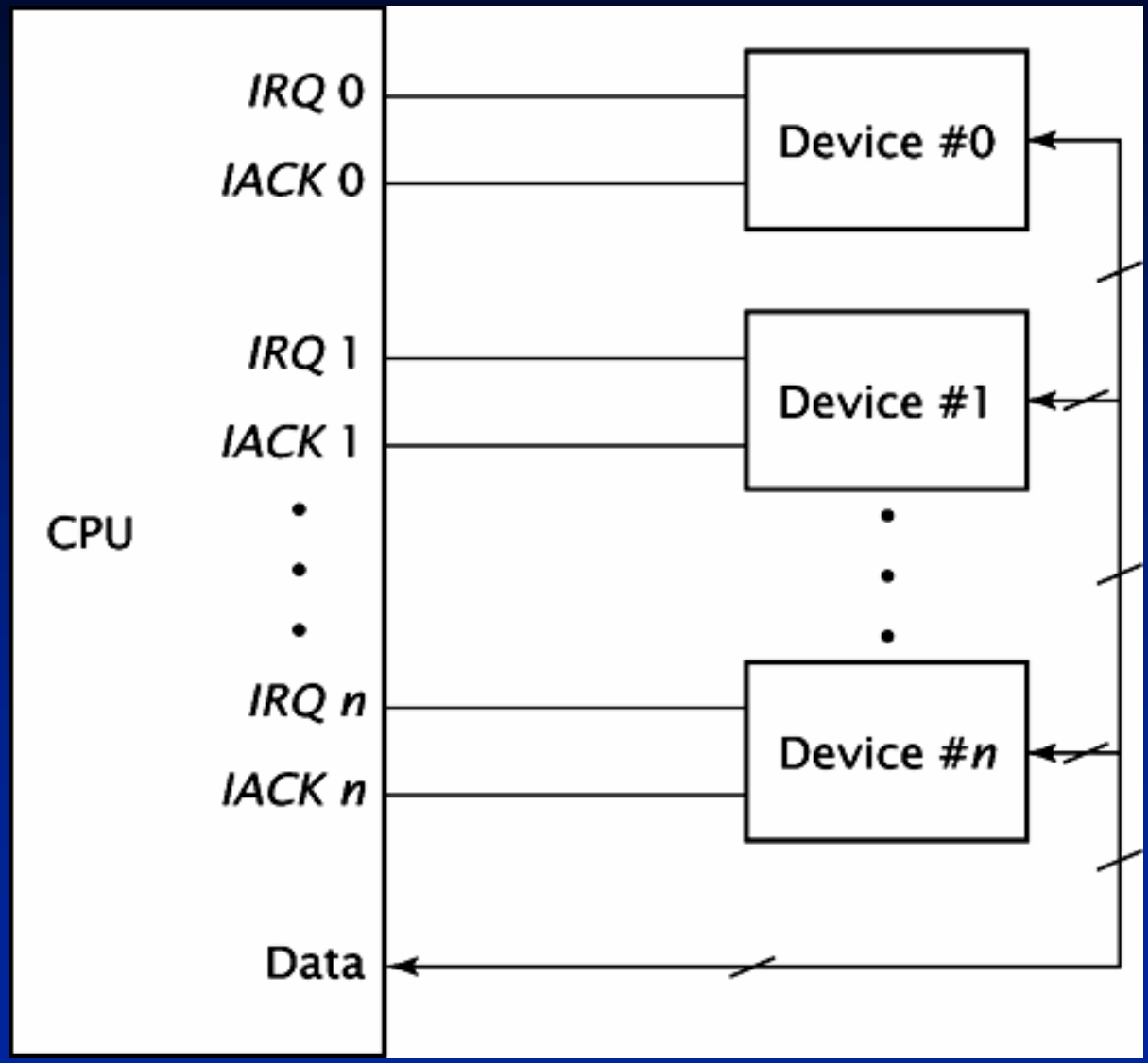
Hardver i tajming za nevektorski prekid iniciran od strane jedinstvenog uredjaja



Hardver i tajming za vektorski prekid iniciran od strane jedinstvenog uredjaja



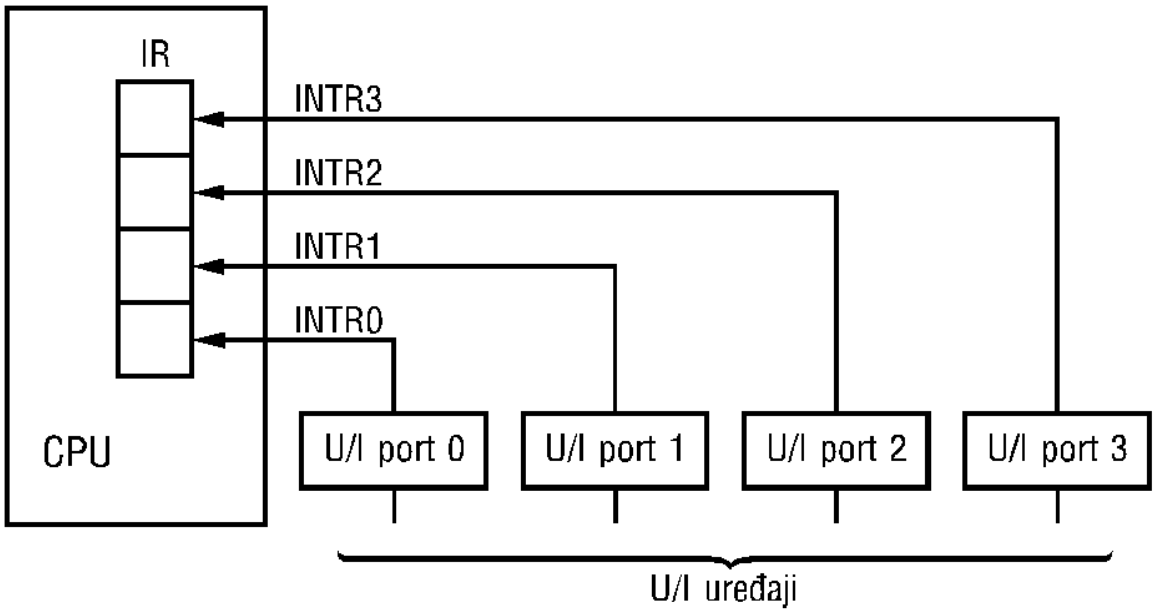
Hardver za nevektorski prekid iniciran od strane većeg broja uredjaja



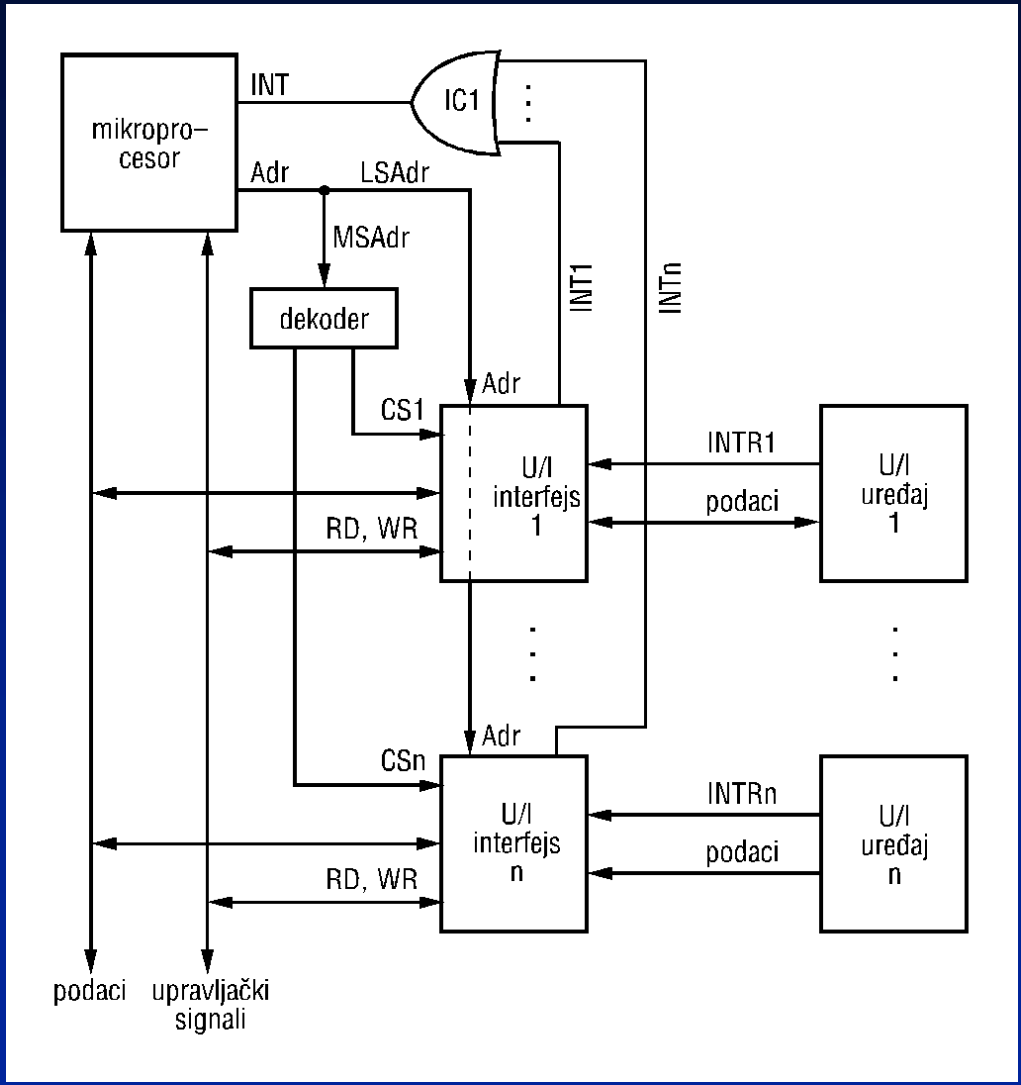
Tehnike za prihvatanje prekida

- **Četiri poznate kategorije prihvatanja zahteva za prekid kada istovremeno naidje veći broj zahteva za prekid su:**
  - **Veći broj prekidnih linija (*Multiple Interrupt Lines*)**
  - **Softversko kružno ispitivanje (*Software Pool*)**
  - **Lančanje (*Daisy Chain - hardware pool, vectored*)**
  - **Magistralna arbitraža (*Bus Arbitration, vectored*)**

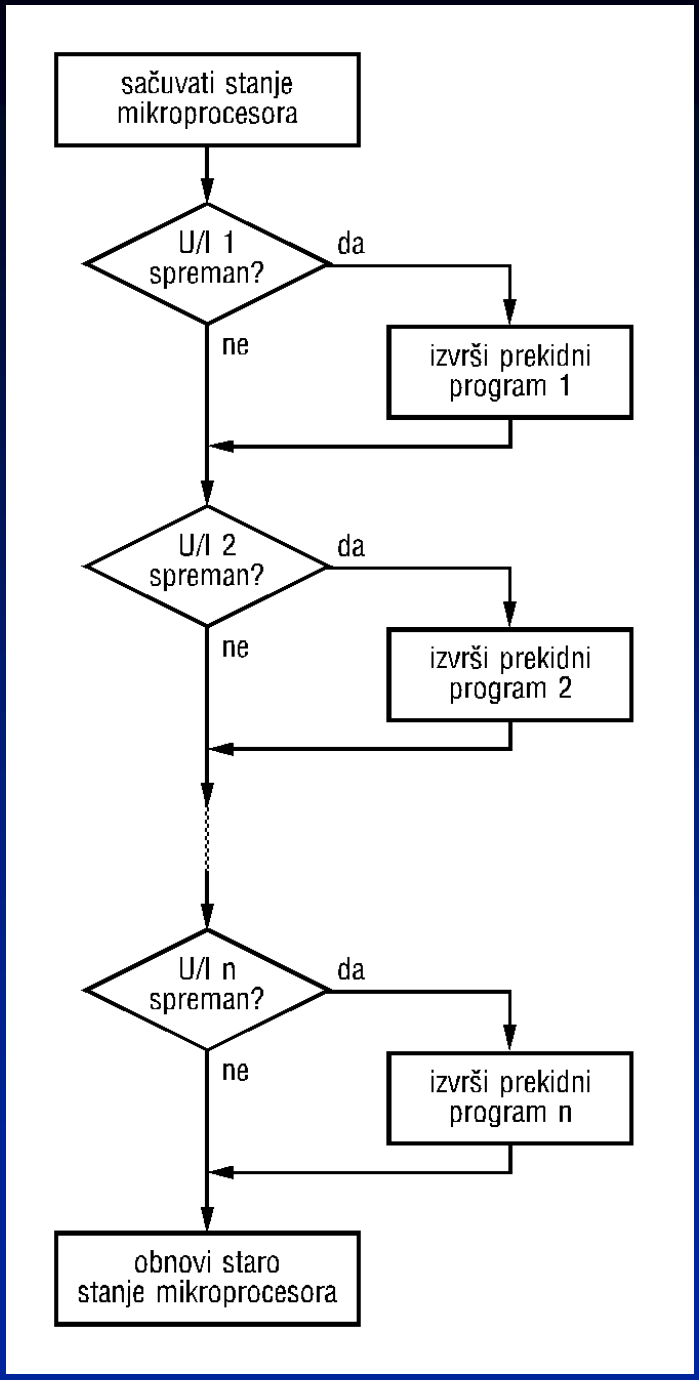
Sistem sa većim brojem prekidnih linija



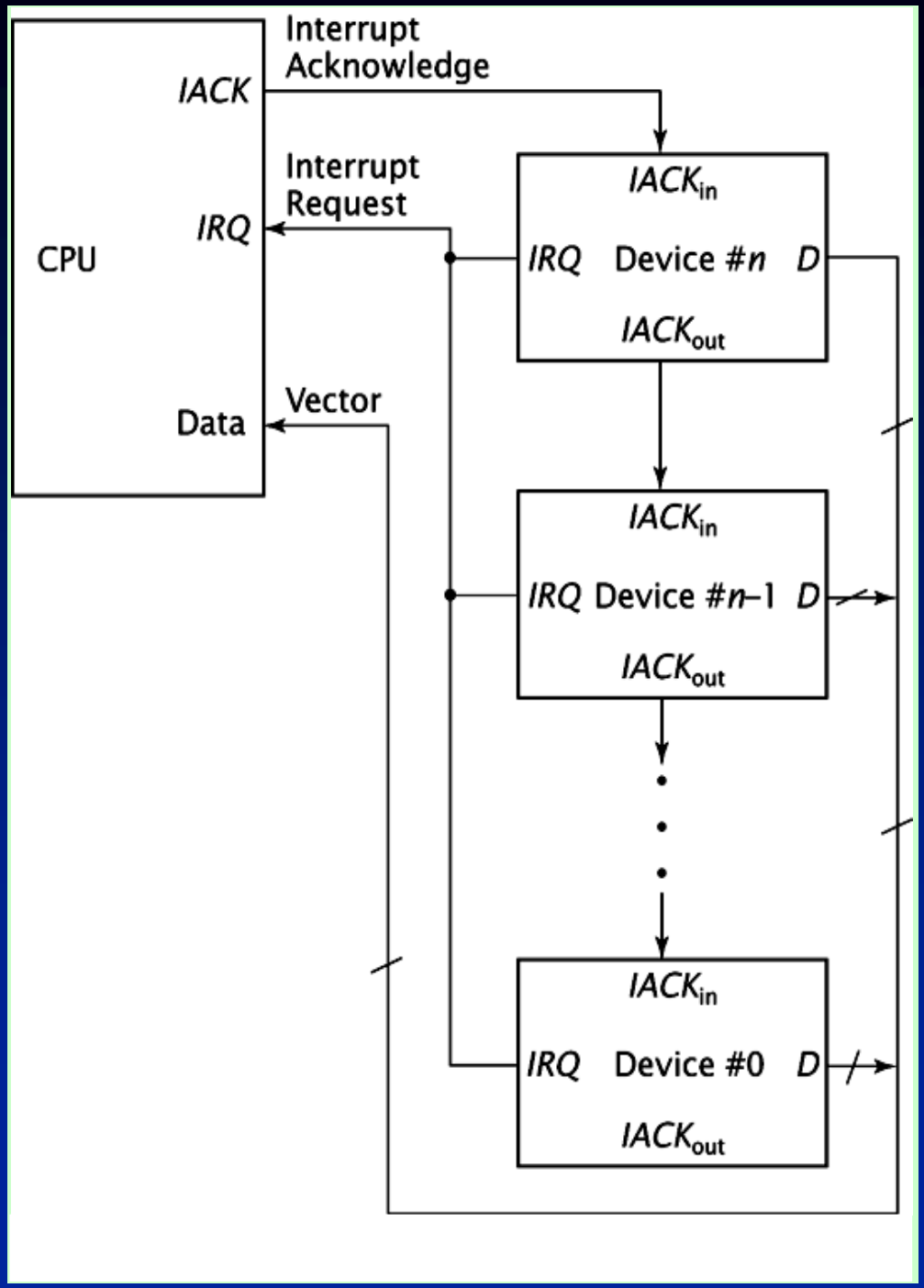
Softversko kružno ispitivanje zahteva za prekid - realizacija hardverskog dela -



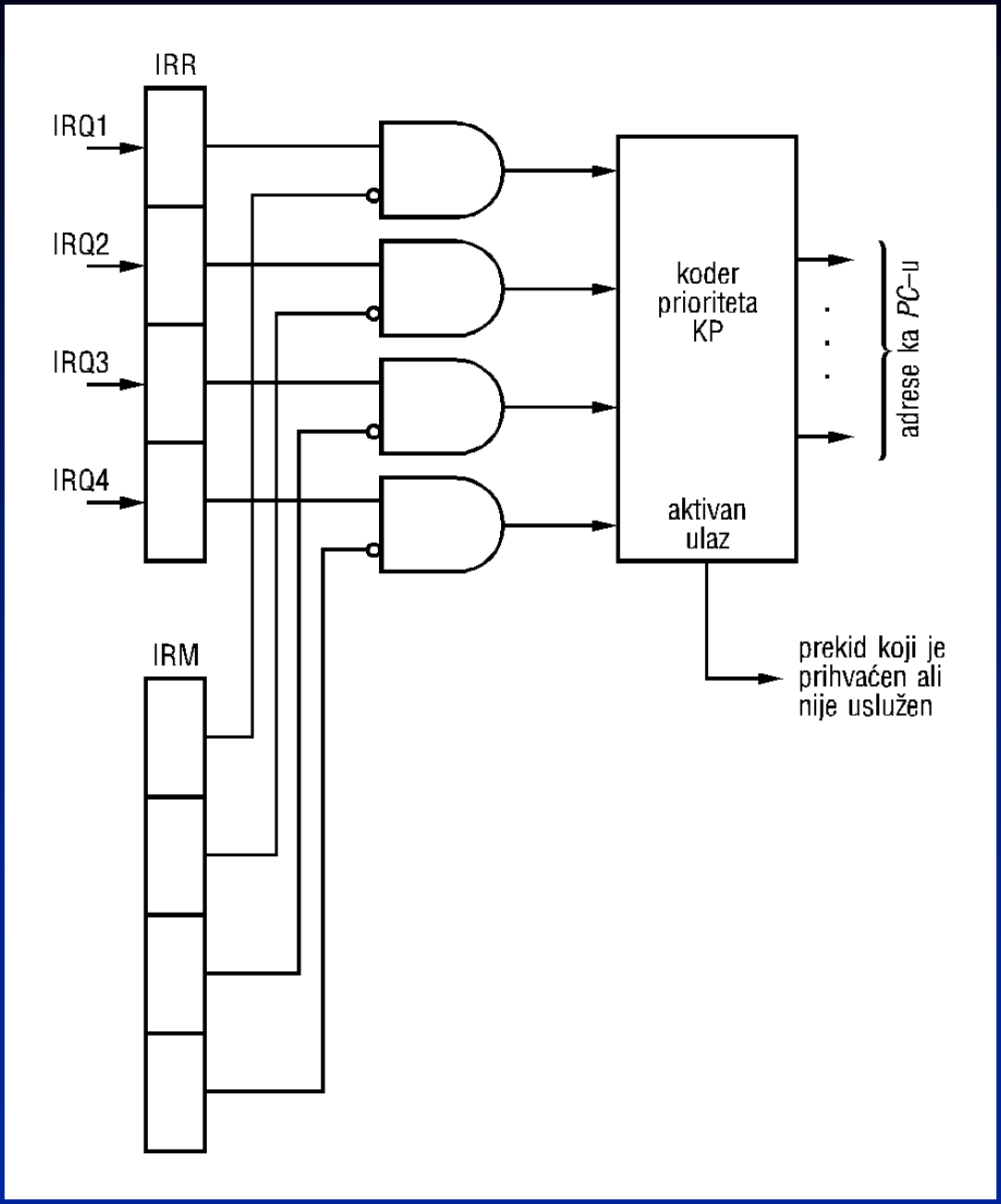
Softversko kružno ispitivanje zahteva za prekid -realizacija softverskog dela-



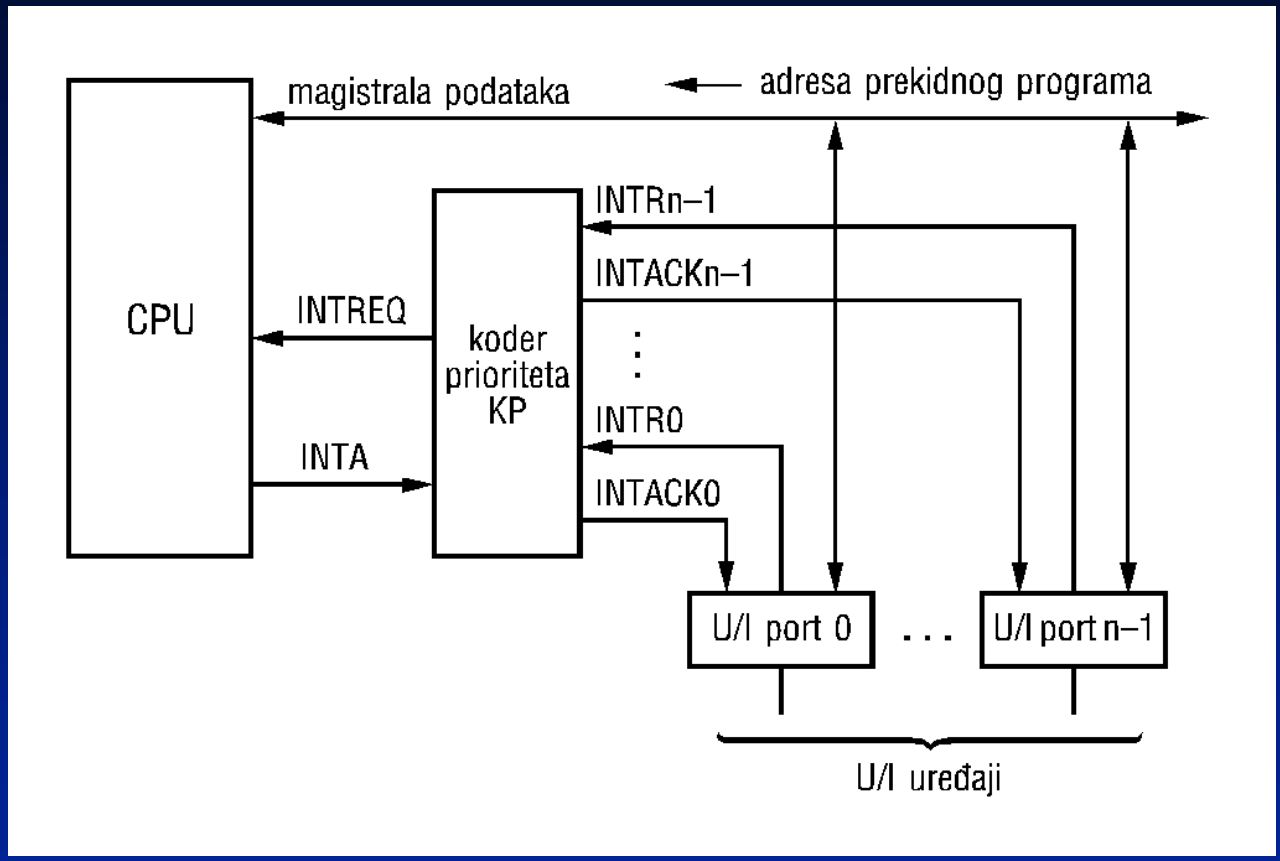
Lančanje


Interrupt
Acknowledge
IACK
Interrupt
Request
IRQ
CPU
Data
Vector
IACKin
IRQ Device #n D
IACKout
IACKin
IRQ Device #n–1 D
IACKout
IACKin
IRQ Device #0 D
IACKout

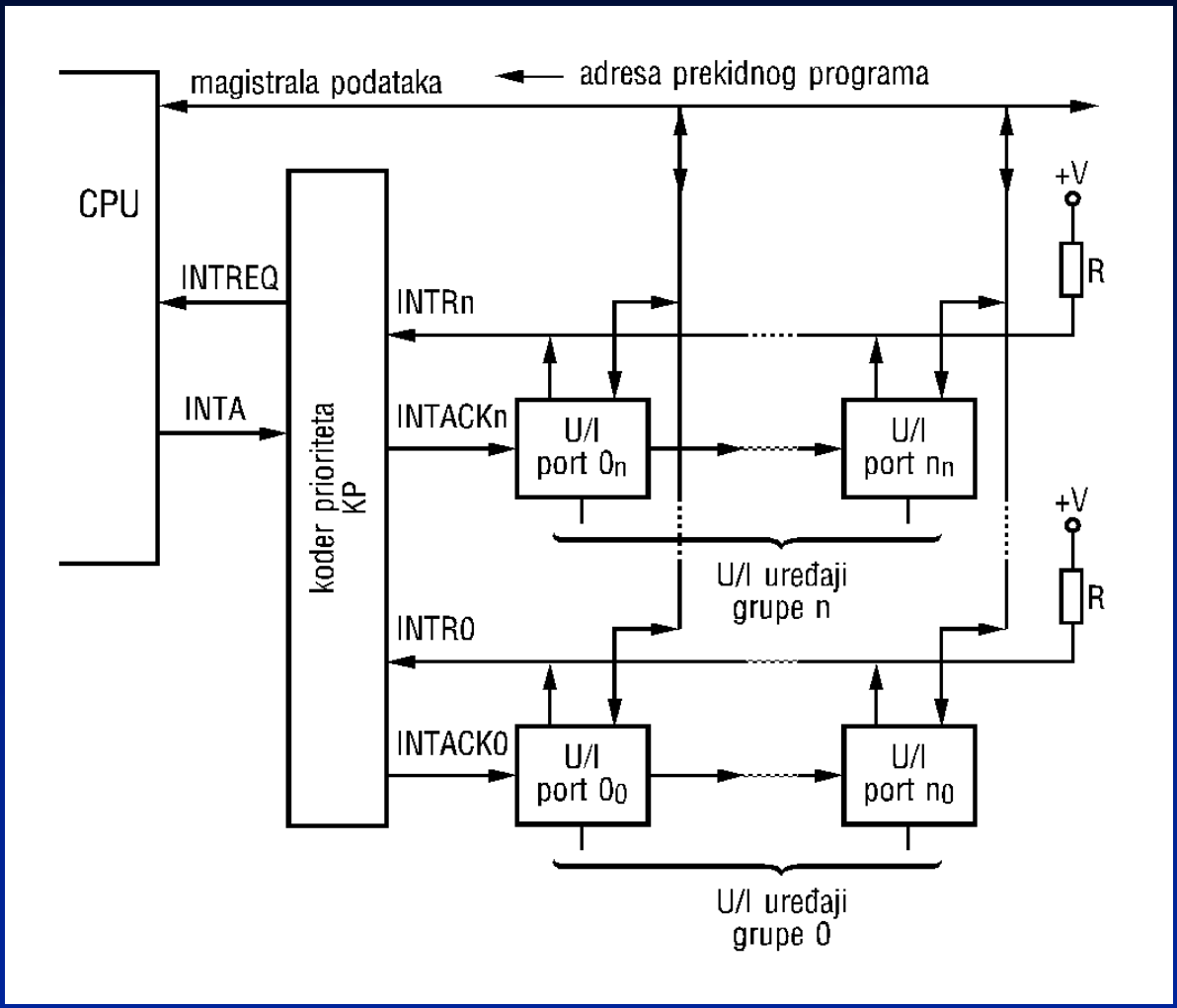
Prihvatanje vektorskog prekida zasnovanog na maskiranju



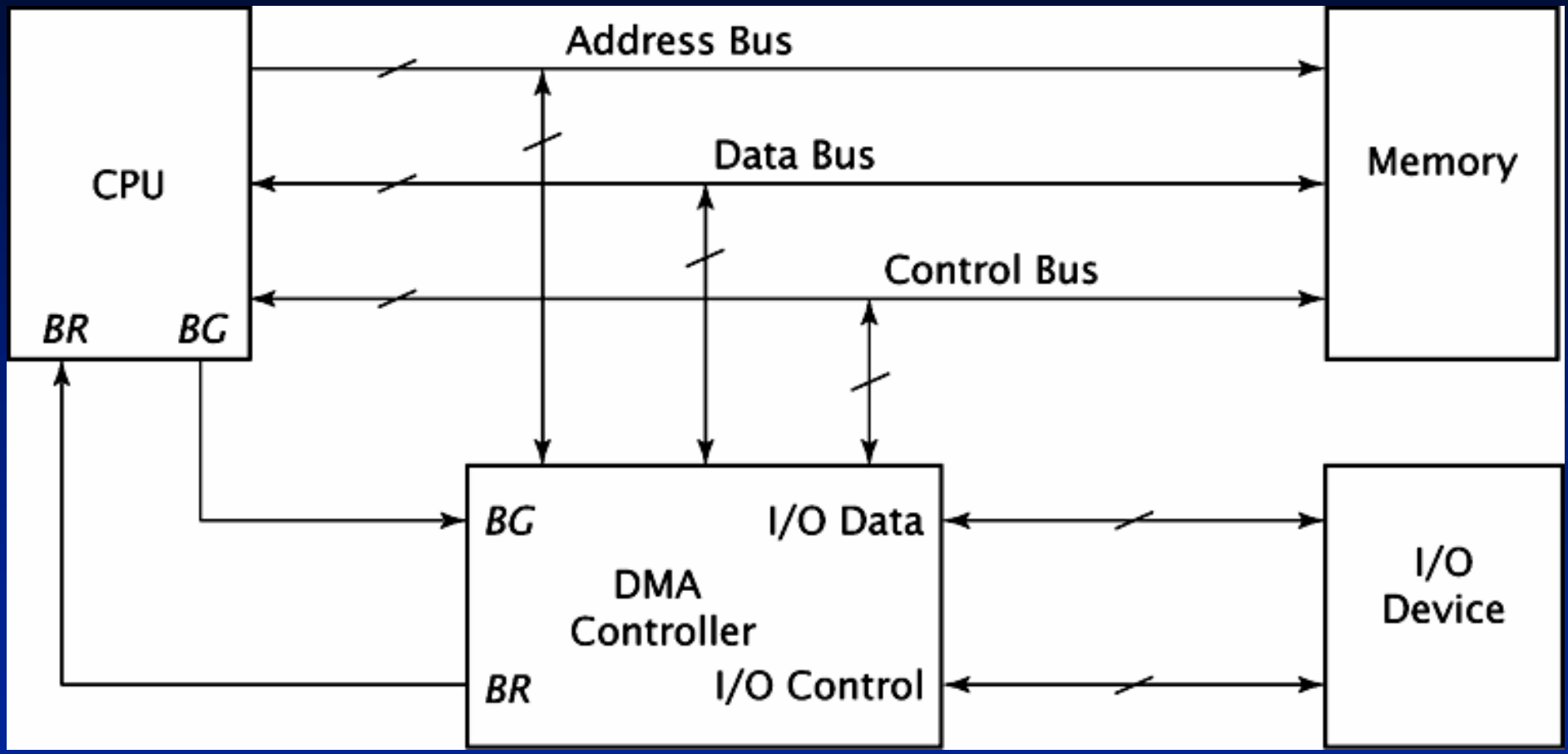
Tipična aplikacija vektorskog prekidanja



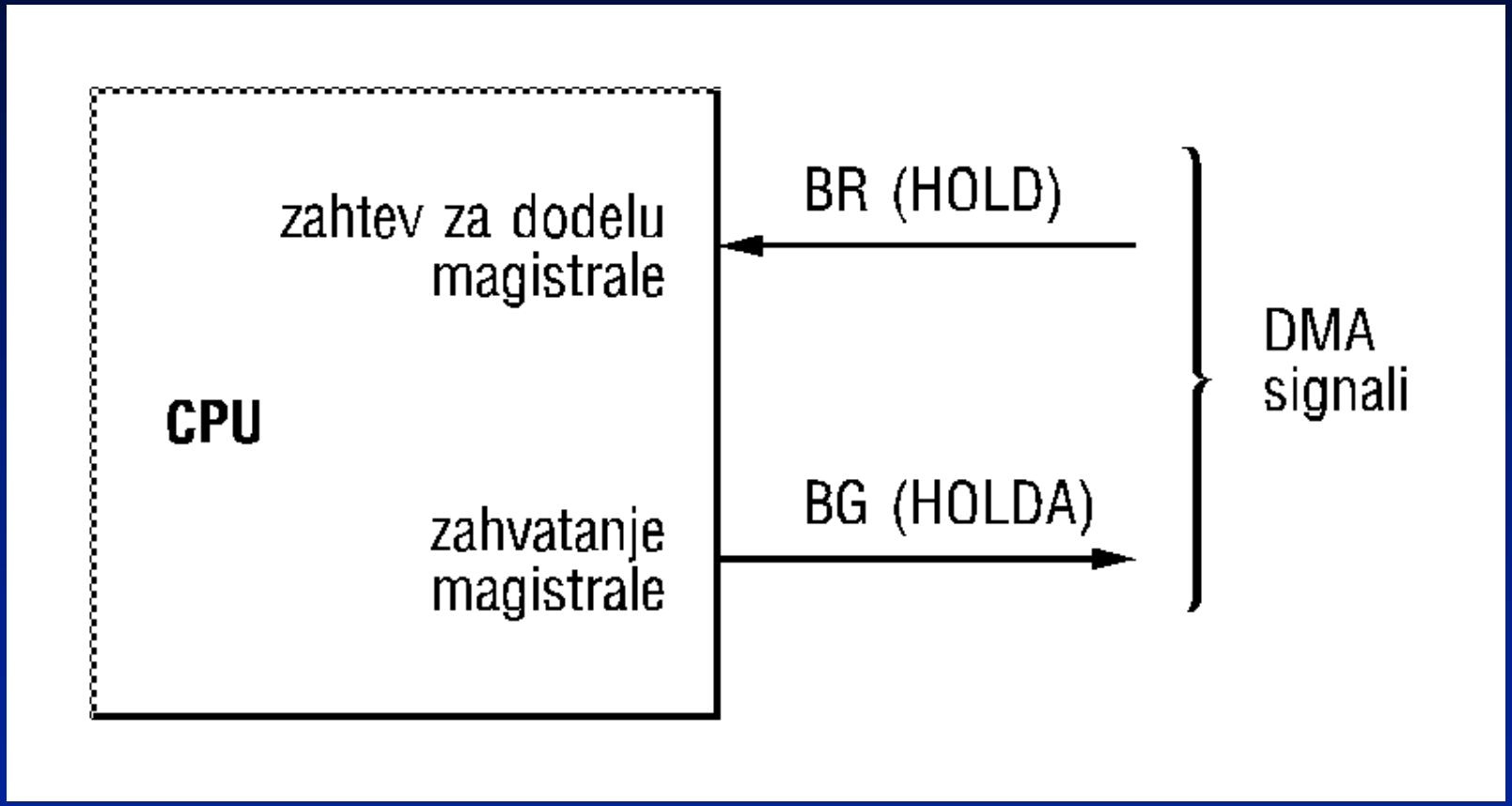
Šema za prihvatanje vektorskih prekida sa prioritetnim grupama



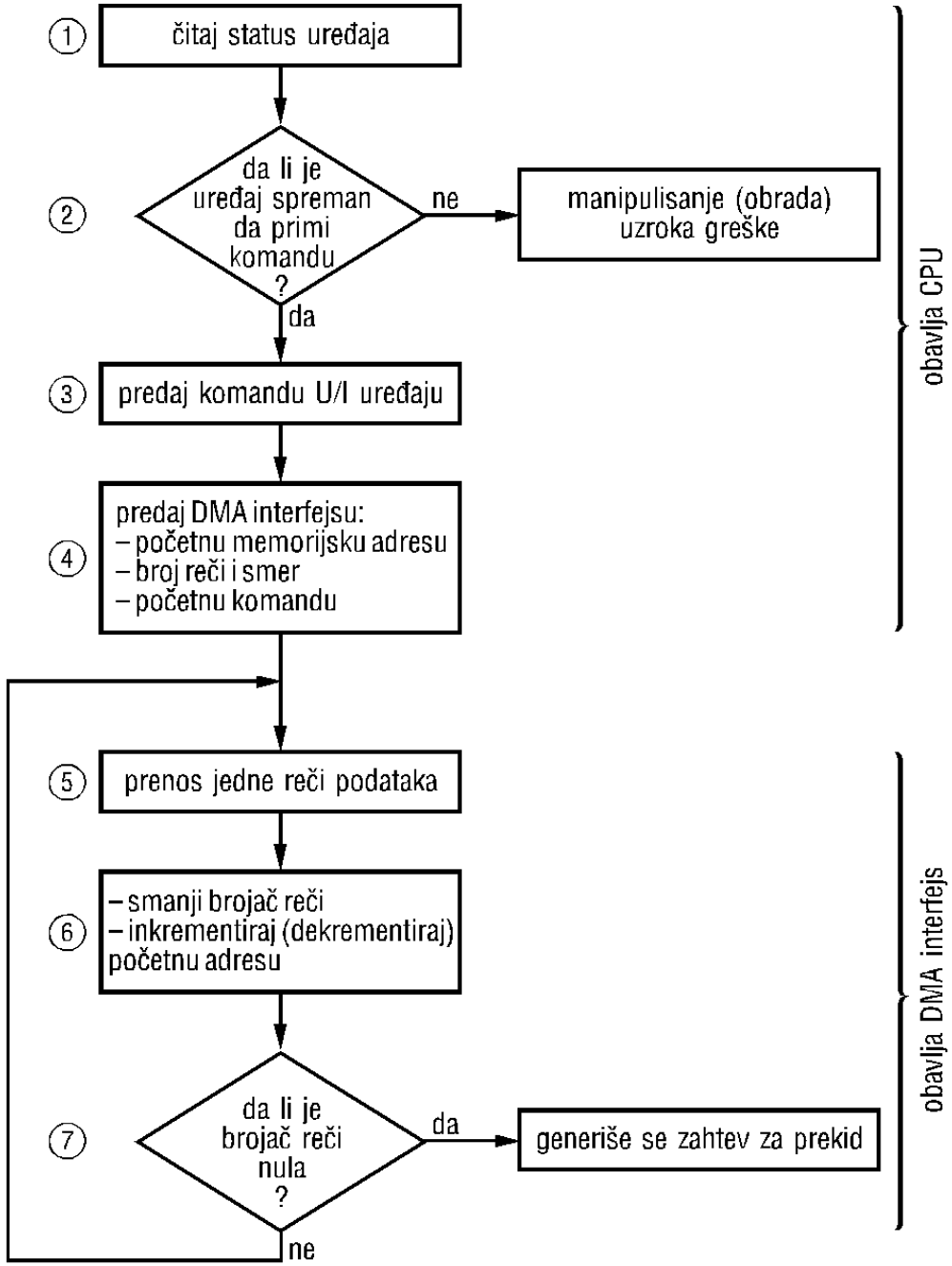
Računarski sistem sa DMA



DMA signalne linije tipičnog mikroprocesora



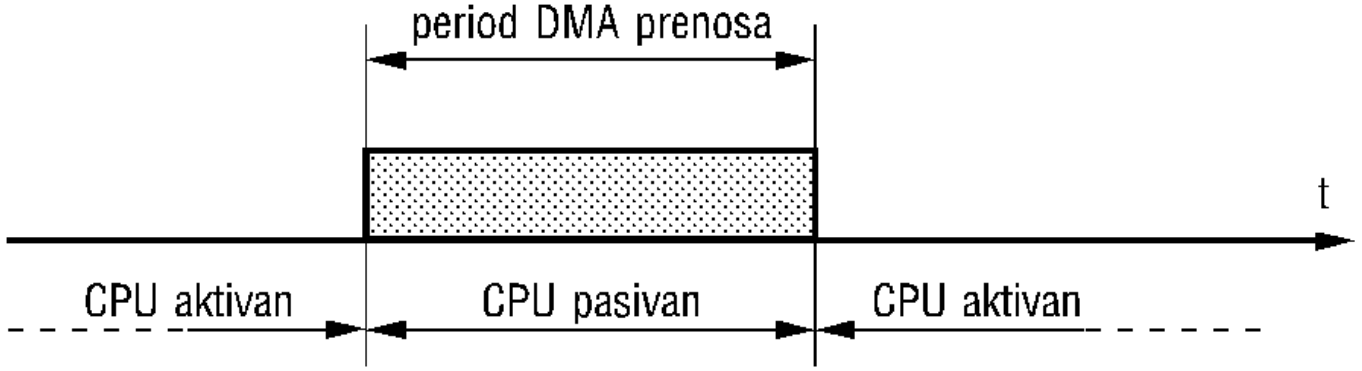
Tipična sekvenca u toku DMA transakcije



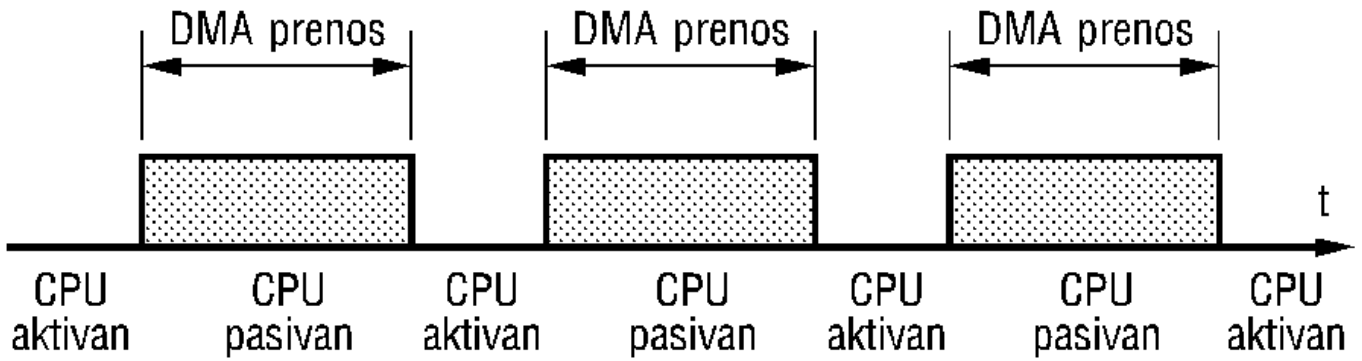
Tipovi DMA prenosa

- **Od tipa realizacije DMA prenosa:**
  - **kradja ciklusa**
  - **normalni**
  - **transparentni**
- **Od hardverske realizacije DMA prenosa:**
  - **direktni**
  - **indirektni**
- **Od smera prenosa podataka:**
  - **DMA IN**
  - **DMA OUT**
  - **DMA MEMORY-TO-MEMORY**

Normalni i *burst* DMA način rada

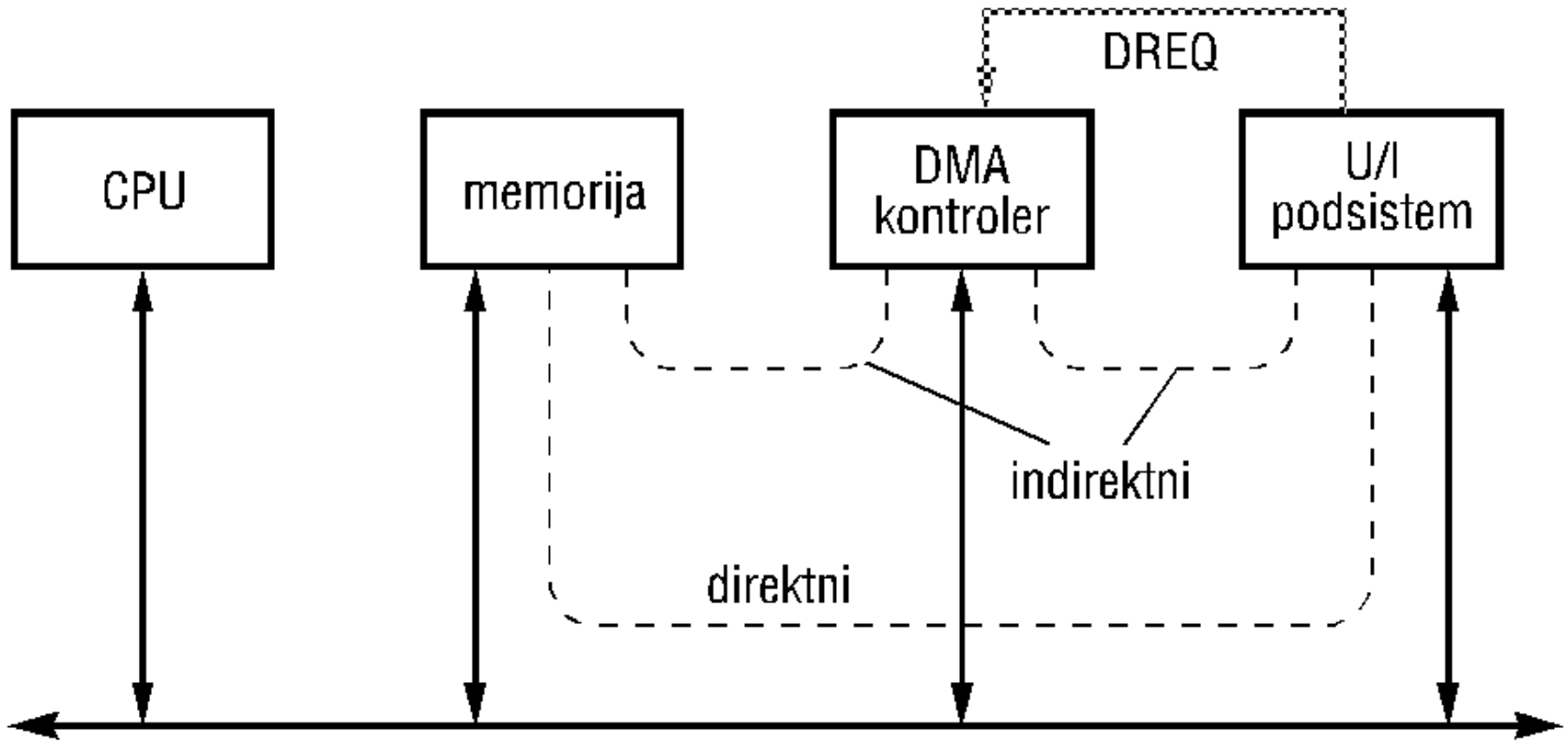


a) *normalni DMA*

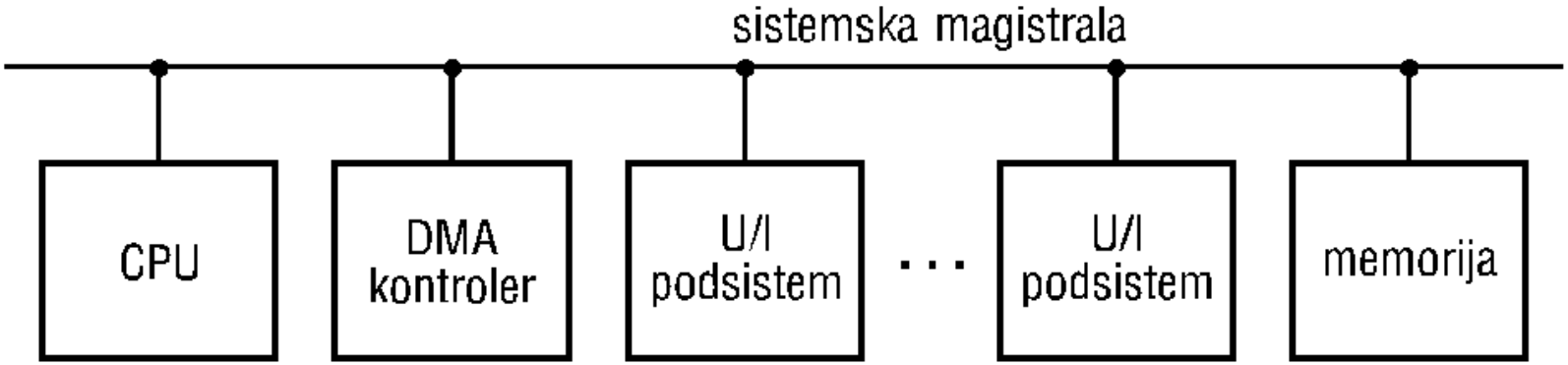


b) *"burst" DMA*

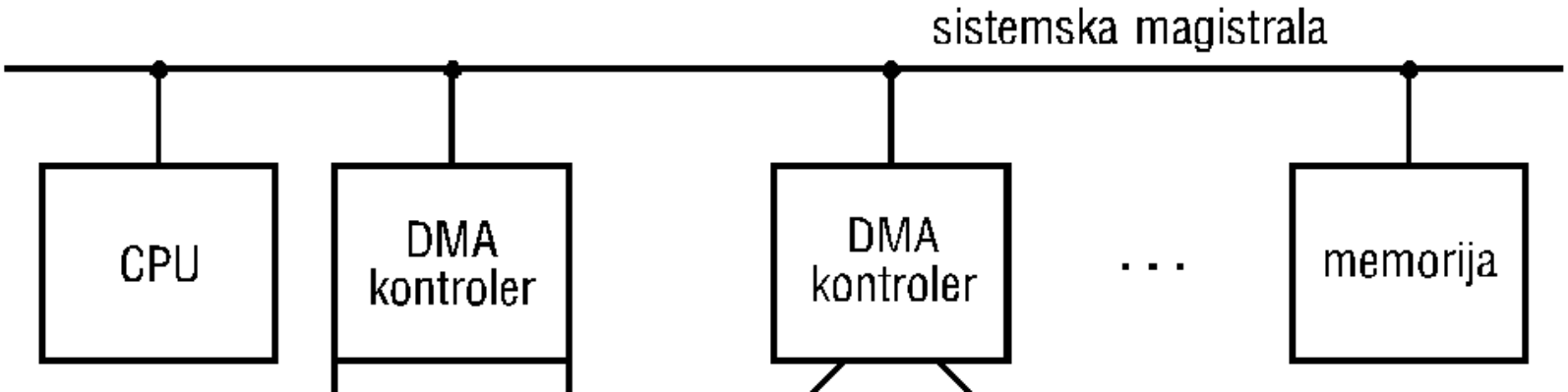
Direktni i indirektni DMA prenos

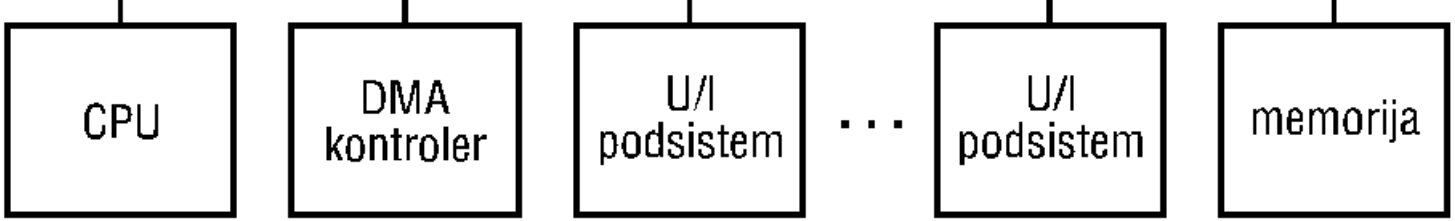


Moguće DMA hardverske konfiguracije

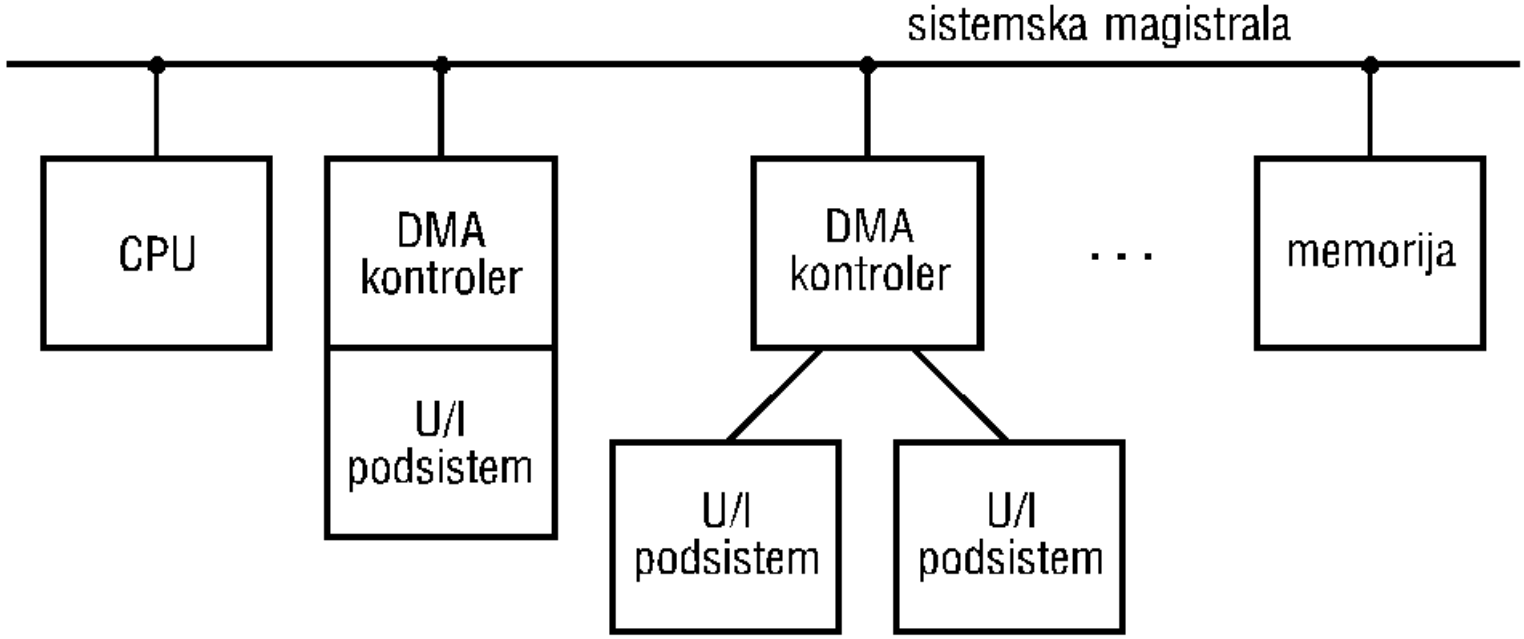


a) *jedinstvena magistrala, izdvojen DMA kontroler*

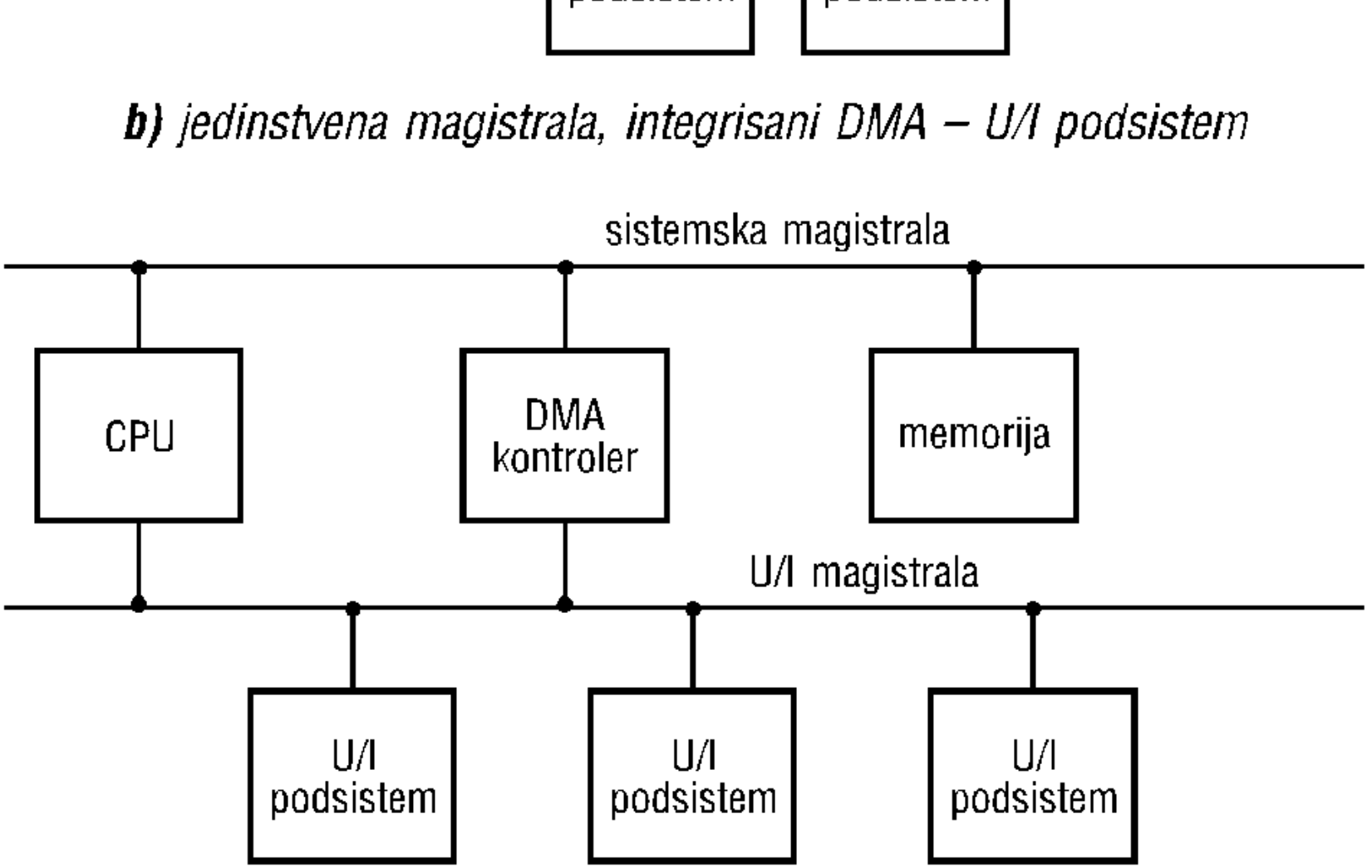




a) *jedinstvena magistrala, izdvojen DMA kontroler*



b) *jedinstvena magistrala, integrisani DMA – U/I podsistem*

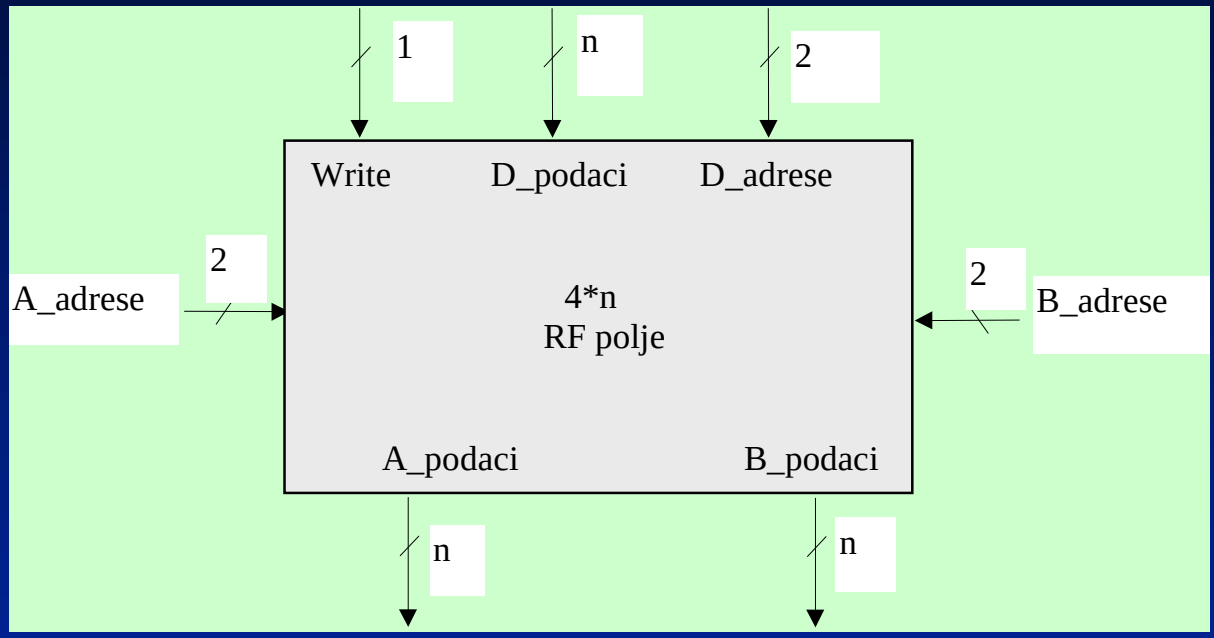


b) *jedinstvena magistrala, integrisani DMA – U/I podsistem*

c) *U/I magistrala*

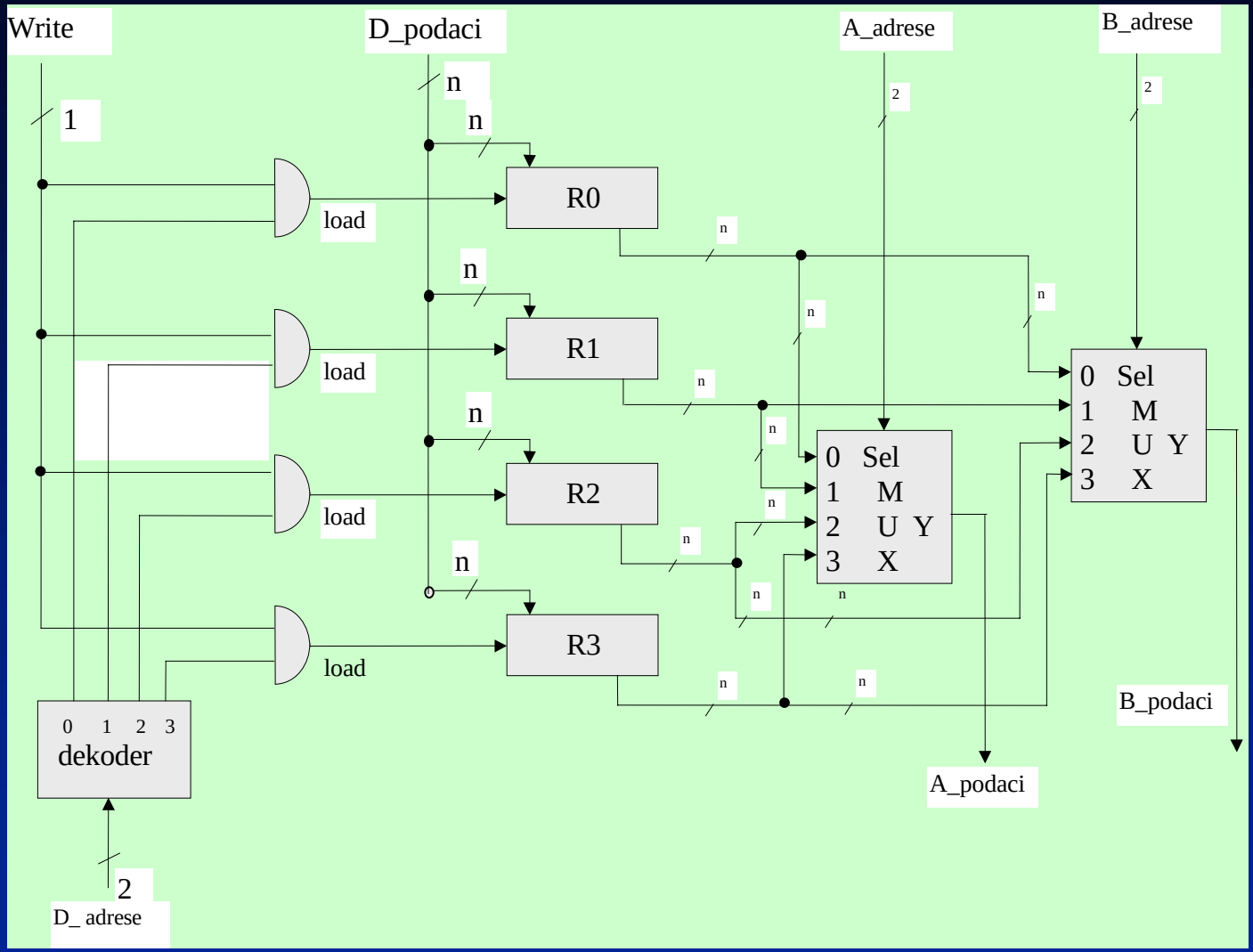
Zadatak 1.

Blok {ema jednog RF polja je data na slici:



Pokazati kako izgleda njegova interna struktura.

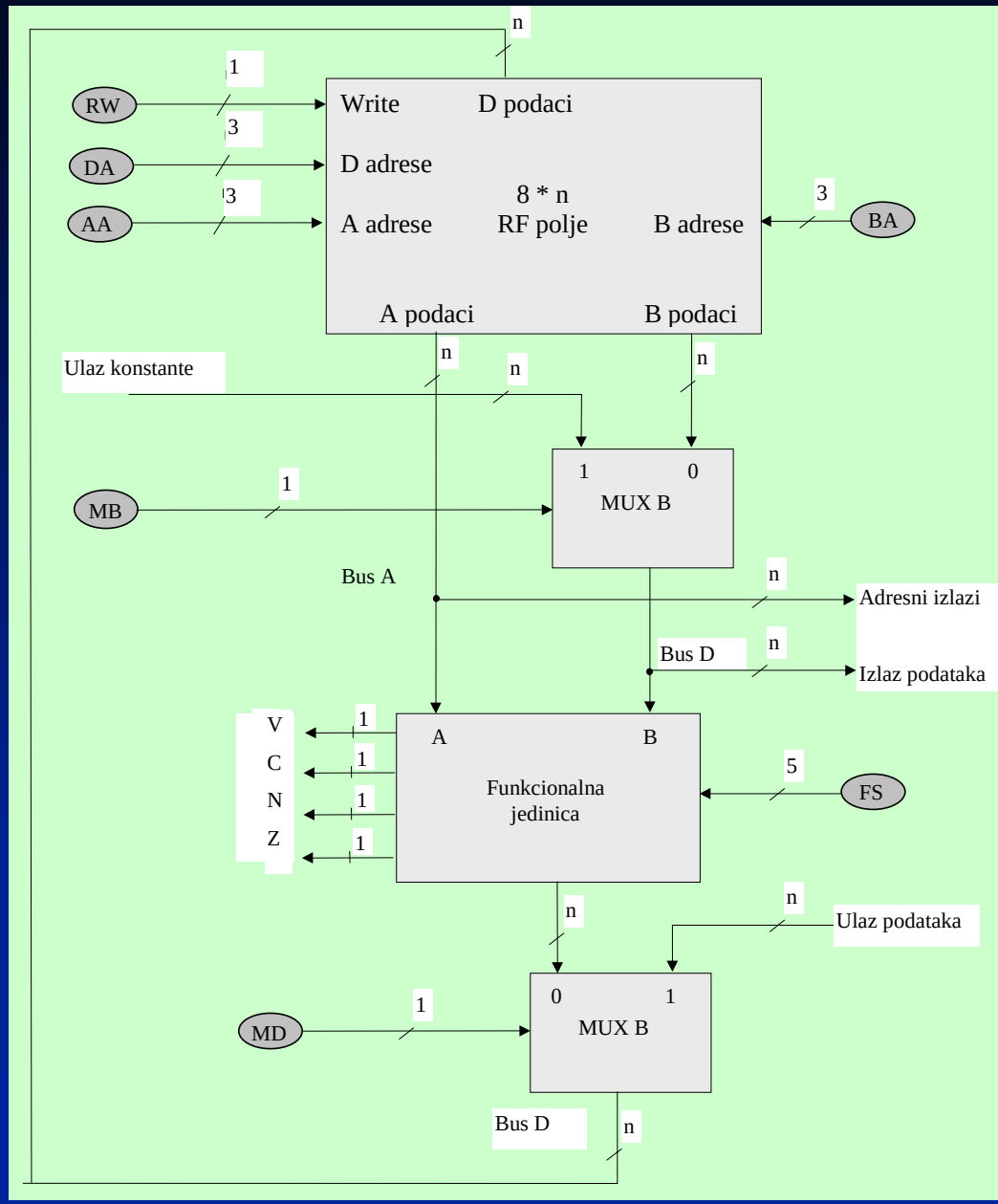
Odgovor



Logička struktura *4*n* RF polja

Zadatak 2.

Blok dijagram staze podataka koja sve operacije obavlja za jedan taktni interval, je oblika



17-bitna upravlja~ka re~ je oblika

16 15 14	13 12 11	10 9 8	7	6 5 4 3 2	1	0
DA	AA	BA	MB	FS	MD	RW

a kodiranje upravlja~ke re~i je oblika

		MB		*FS*		*MD*		*RW*	
funkcija	**kô**	**funkcija**	**kôd**	**funkcija**	**kôd**	**funkcija**	**kô**	**funkcija**	**kôd**
R0	**000**	**Registar**	**0**	**F=1**	**00000**	**Funkcija**	**0**	**nema**	**0**
R1	**001**	**Konstan**	**1**	**F =A+1**	**00001**	**ulaz**	**1**	**Write**	**1**
R2	**010**			**F=A+B**	**00010**				
R3	**011**			**F=A+B+1**	**00011**				
R4	**100**			**F=A+/B**	**00100**				
R5	**101**			**F=A+/B+1**	**00101**				
R6	**110**			**F=A-1**	**00110**				
R7	**111**			**F=A**	**00111**				
				F=A and B	**01000**				
				F=A or B	**01010**				
				F=A xor B	**01100**				
				F=not A	**01110**				
				F=sr A	**10000**				
				F=sl A	**10001**				

Ilustracije radi, operacije oduzimanja sadr`aja dva registra R2 i R3 i sme{tanje rezultata u R1 definisana kao:

R1←R2+ not R3 + 1,

kodira se na slede}i na~in:

Polje:	*DA*	*AA*	*BA*	*MB*	*FS*	*MD*	*RW*
Simbol:	R1	R2	R3	Registar	F=A+/B+1	Funkcija	Write
Binarna vrednost:	001	010	011	0	00101	0	1

Odrediti sadr`aj svih polja upravlja~ke re~i za slede}e tipove operacija:

- **R1←R2-R3;**
- **R4←slR6;**
- **R7←R7+1;**
- **R1←R0+2;**
- **izlaz podatka ← R3;**
- **R4 ← ulaz podatka;**
- **R5 ← 0**

Odgovor

Simbolički upravljačka reč je kodirana na sledeći način:

mikro operacija	*D A*	*AA*	*B A*	*MB*	*F3*	*MD*	*RW*
R1←R2+/R3+1	R1	R2	R3	Registar	F=A+/B+1	Funkcija	Write
R4←sl R6	R4	R6	-	Registar	F=slA	Funkcija	Write
R7←R7+1	R7	R7	-	-	F=A+1	Funkcija	Write
R1←R0+2	R1	R0	-	Konstanta	F=A+B	Funkcija	Write
izlaz podatka←R3	-	-	R3	Registar	-	-	nema Write
R4← ulaz podatka	R4	-	-	-	-	ulaz podatka	Write
R5← 0	R5	R0	R0	Registar	F=A xor B	Funkcija	Write

Binarno upravljačka reč, za odgovarajuću operaciju, kodirana je na sledeći način:

mikrooperacija	*DA*	*AA*	*BA*	*MB*	*FS*	*MD*	*RW*
R1←R2-R3	001	010	011	0	00101	0	1
R4←sl R6	100	110	000	0	10001	0	1
R7←R7+1	111	111	000	0	00001	0	1
R1←R0+2	001	000	000	1	00010	0	1
izlaz podatka←R3	000	000	011	0	00000	0	0
R4←ulaz podatka	100	000	000	0	00000	1	1
R5←0	101	000	000	0	01100	0	1

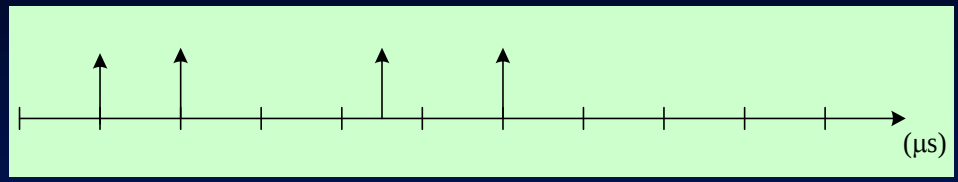
Zadatak 3.

Mikrora~unarski sistem prihvata zahteve za prekid iniciranih od strane U/I uredjaja. IRQ6 ima najvi{i prioritet opslu`ivanja a IRQ1 najni`i. Svaka rutina za obradu prekida (uklju~uju}i i opslu`ivanje zahteva za prekid) se obradjuje za 10 μs. Sekvenca dogadjaja je prikazana na slici.
Grafički pokazati koju }e rutinu CPU ivr{avati u vremenu od 0 do 100 μs u slu~ajevima:

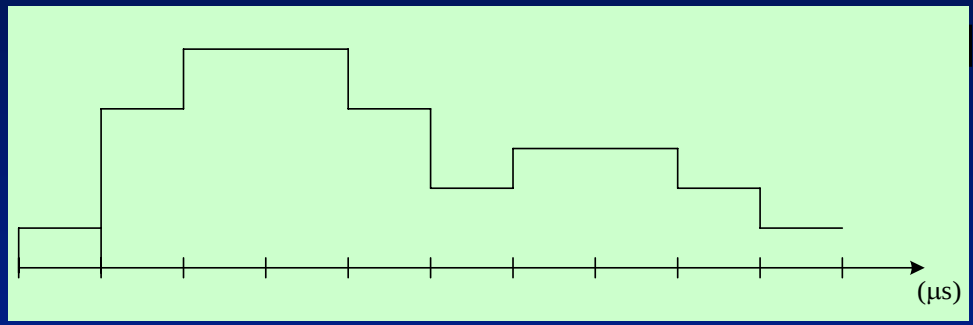
- **kada postoji istiskivanje (rutina vi{eg prioriteta prekine rad rutine ni`eg prioriteta)**
- **kada ne postoji istiskivanje**

vreme	akcija
0μs	po~etak glavnog programa
10μs	IRQ4
20μs	IRQ6
45μs	IRQ1
60μs	IRQ3

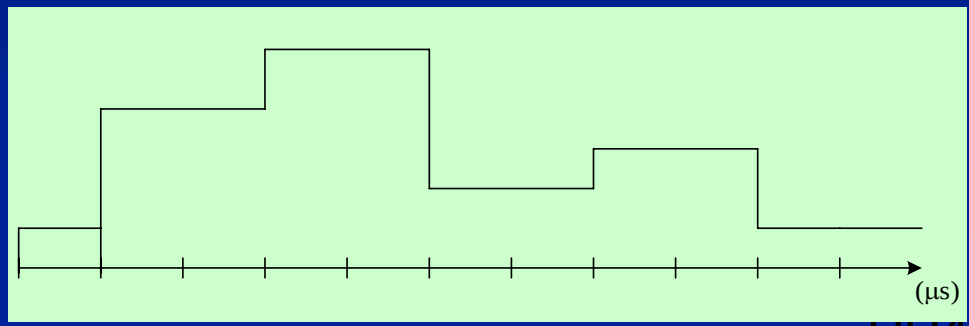
Odgovor



a)



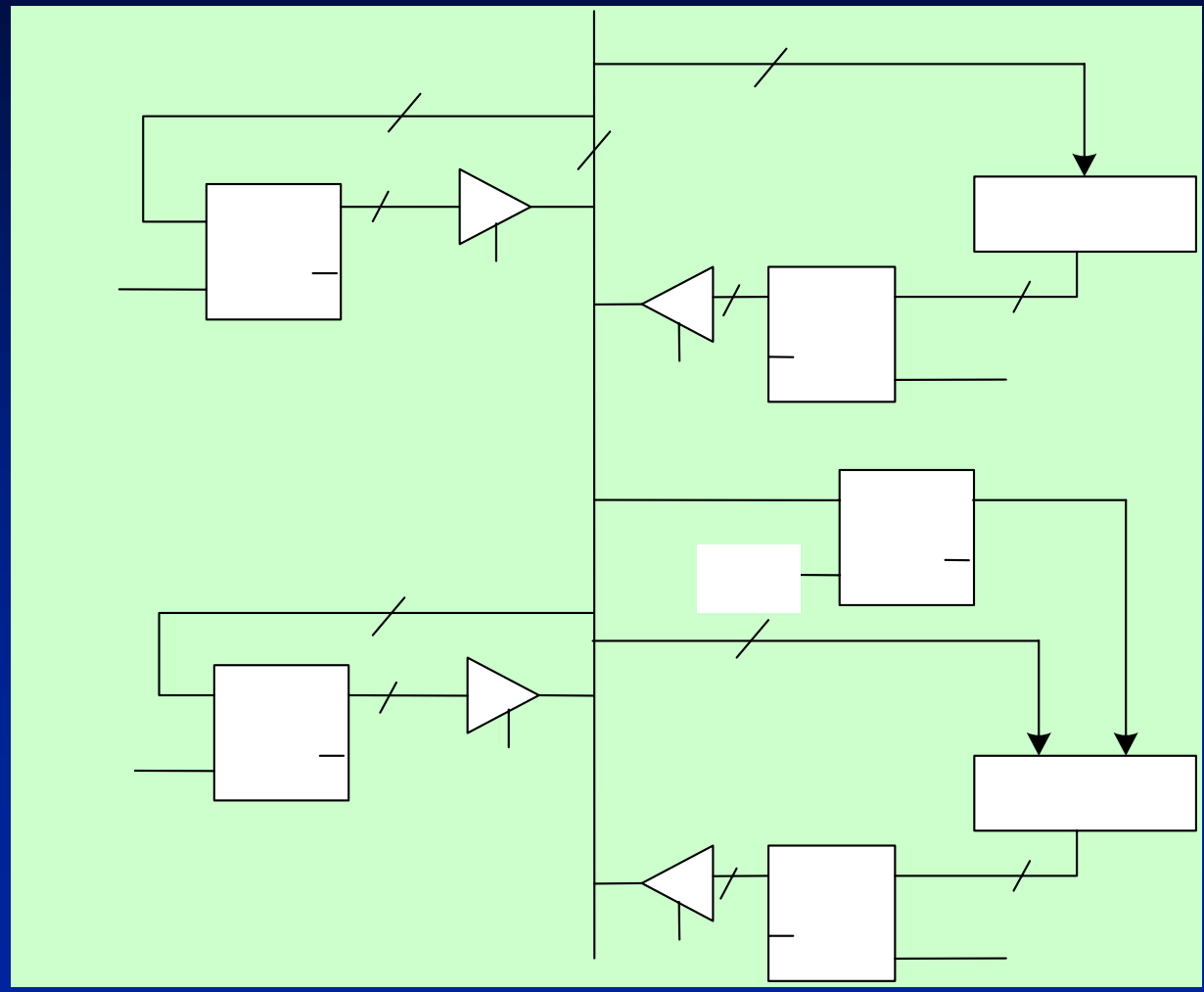
b)



Zadatak 4.

Koriste}i hardver prikazan na slici, napisati opis na nivou registarskog prenosa, a takodje i definisati upravlja~ku sekvencu koja implementira slede}e aktivnosti (operacije):

- **R3←R4+R6+1**
- **R2←R3+R4+R5**



Odgovor

R3←R4+R6+1

prenos	*upravlja~ki signali*
W←R6+1	R6out,Win
Y←W	Wout,Yin
Z←R4+Y	R4out,Zin
R3←Z	Zout,R3in

R2←R3+R4+R5

prenos	*upravlja~ki signali*
Y←R5	R5out,Yin
Z←R4+Y	R4out,Zin
Y←Z	Zout,Yin
Z←R3+Y	R3out,Zin
R2←Z	Zout,R2in

Mikroprocesorski sistemi

II deo

Napredne tehnike

Definicija ISA

- **Procesori koji su u stanju da izvršavaju unapred definisani skup instrukcija (*Instruction Set Processors - ISPs*) nazivaju se mikroprocesori.**

- **Funkcionalnost mikroprocesora karakteriše se skupom instrukcija koje je on u stanju da izvrši.**

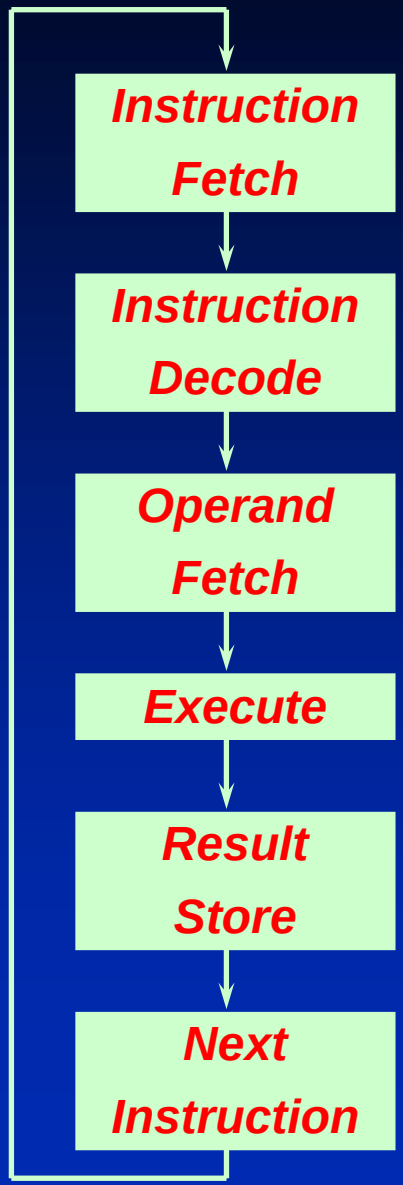
- **Svi programi koji se izvršavaju na tom mikroprocesoru kodirani su pomoću tog skupa instrukcija.**

- **Unapred definisani skup instrukcija naziva se skup instrukcija te arhitekture (*Instruction Set Architecture - ISA*).**

- ***ISA* se koristi kao interfejs izmedju softvera i hardvera, ili izmedju programa i procesora.**

- **Sa aspekta metodologije projektovanja *ISA* se odnosi na *specifikaciju* dizajna, dok mikroprocesor ili ISP predstavlja *implementaciju* dizajna.**

Instruction Set Architecture: What Must be Specified?



- Instruction Format or Encoding
  - how is it decoded?
- Location of operands and result
  - where other than memory?
  - how many explicit operands?
  - how are memory operands located?
  - which can or cannot be in memory?
- Data type and Size
- Operations
  - what are supported
- Successor instruction
  - jumps, conditions, branches
  - *fetch-decode-execute is implicit!*

Arhitektura računara – tri nivoa apstrakcije

- **Kada se govori o arhitekturi računara definišu se tri osnovna i različita nivoa apstrakcije:**
  - **arhitekturni – specificira funkcionalno ponašanje procesora,**
  - **implementacioni – odnosi se na logičku strukturu ili organizaciju koja važi za tu arhitekturu, i**
  - **realizacioni – predstavlja fizičku strukturu u koju se ugradjuje ta implementacija**

Arhitektura

- Arhitektura se često naziva *ISA*.
- ISA specificira skup instrukcija koji karakterišu funkcionalno ponašanje *ISP*-a.
- Sav softver se mora preslikati ili kodirati u ovaj skup instrukacija kako bi mogao da se izvršava od strane procesora.
- Svaki program se kompajlira u sekvencu instrukcija koje pripadaju tom skupu.
- Tipični primeri arhitektura su IBM 360, DEC VAX, Motorola 68k, Power PC, IA 32, i dr.
- Atributi koji prate arhitekturu su asemblerski jezik, format instrukcija, adresni način rada, i programski model.

Implementacija

- **Implementacija predstavlja specifični dizajn arhitekture koja se naziva mikroarhitektura.**
- **U toku života, jedna *ISA* arhitektura može da ima veći broj implementacija, pri čemu sve implementacije izvršavaju programe za tu *ISA*.**
- **Primeri implementacija poznatih arhitektura su IBM 360/91, VAX 11/780, Motorola 68040, PowerPC 604, Intel P6.**
- **Atributi koji su tipični za implementaciju su protočni dizajn, keš memorija, i prediktori granjanja.**
- **Implementacija se ostvaruje na nivou hardvera i nevidljiva je softveru.**

Realizacija

- Realizacija implementacije se odnosi na fizičko ugradjivanje dizajna.
- Kada se govori o mikroprocesorima tu se pre svega misli na to da li je to fizičko ugradjivanje sprovedeno na čipu ili multičipu.
- Za datu implementaciju postoji veći broj relizacija.
- Realizacije se mogu razlikovati u odnosu na taktnu frekvenciju, kapacitet memorije, interfejs magistrale, pakovanje i dr.
- Atributi koji prate realizaciju su površina čipa, disipacija, hladjenje, pouzdanost, i dr.

Projektovanje mikroprocesora- od čega se počinje ?

- *ISA* se koristi kao specifikacija za projektante procesora.
- Projektovanje mikroprocesora počinje sa *ISA*, a kao rezultat se dobija mikroarhitektura koja zadovoljava specifikacije.
- Svaka nova mikroarhitektura mora da se validira u odnodu na *ISA* kako bi se osigurali da li ona ispunjava funkcionalne zahteve specificirane od strane *ISA*.
- Validacija je od izuzetne važnosti ako želimo dokazati da će se postojeći softver korektno izvršavati na novoj mikroarhitekturi.
- Do danas je razvijen, i koristi se, veliki broj *ISA*, koje se razlikuju u odnosu na to kako se specificiraju operacije i operandi.

Projektovanje mikroprocesora- šta definiše ISA ?

- *ISA* definiše skup instrukcija nazvan asemblerske instrukcije.
- Svaka instrukcija specificira operaciju i jedan ili veći broj operanada.
- Svaka *ISA* na jedinstven način definiše asemblerski jezik.
- Program na asemblerskom jeziku čini sekvenca asemblerskih instrukcija.
- Uglavnom *ISA* se medjusobno razlikuju po tome koliki se broj operanada specificira instrukcijom.
- Postoje 3-, 2-, 1-, i 0-adresne mašine.

Projektovanje mikroprocesora- životni vek ISA ?

- Za razvoj efikasnih kompajlera i operativnih sistema potreban je period od desetak godina.

- Ovo znači da je inercija *ISA* u odnosu na razvoj softvera dosta velika.

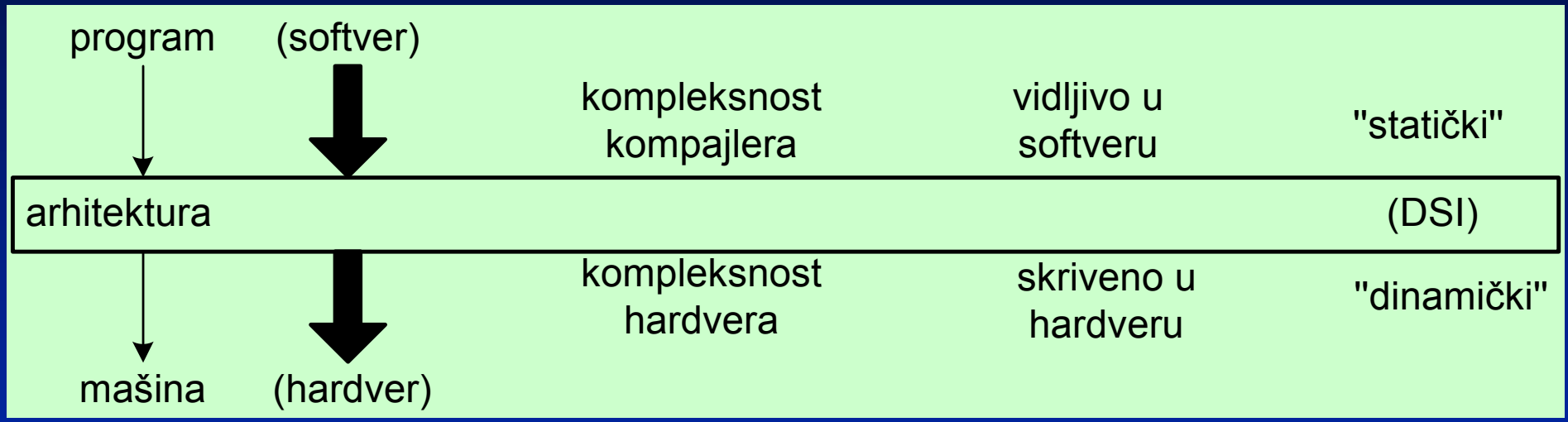
- U principu što je životni vek *ISA* veći to je softverska baza za tu *ISA* veća, pa je takvu *ISA* teže zameniti.

- Nasuprot sporom napretku na polju *ISA* inovacija, nove mikroarhitekture se razvijaju na svakih 3 do 5 godina.

Dinamičko-statički interfejs

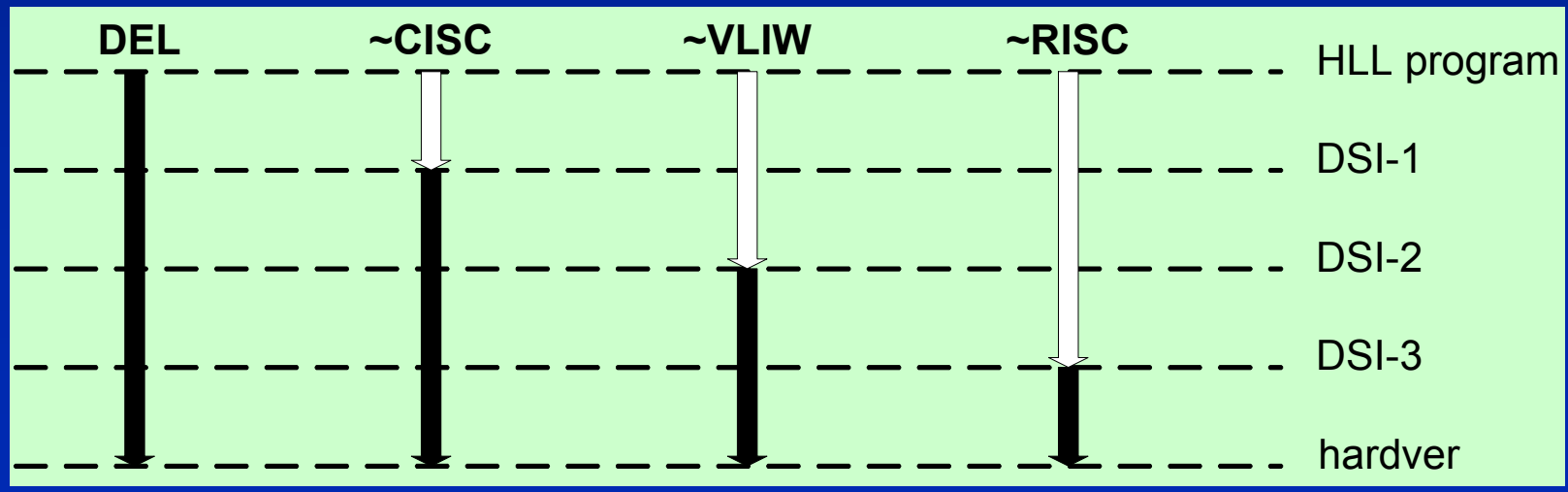
- **Dve ključne uloge koje ima *ISA* su:**
  - **ostvarivanje sprege izmedju hardvera i softvera što obezbedjuje nezavisni razvoj programa i mašina, i**
  - **specifikaciju dizajna mikroprocesora, sve implementacije moraju da ispune zahteve i podrže funkcionalnost specificiranu od strane *ISA***
- **Postoji i treća uloga kojom se definiše dinamičko-statički interfejs (*DSI*),**
- ***DSI* odredjuje šta se od strane *ISA* obavlja statički u toku kompajliranja programa, a šta dinamički u toku izvršenja programa.**

Dinamičko-statički interfejs – vizuelna prezentacija



Prikaz mogućih razmeštaja *DSI*-a kod *ISA* dizajna

- Predložen je veći broj različitih razmeštaja *DSI*-a.
- *Mike Flynn* je predložio postavljanje *DSI*-a veoma visoko, pri čemu se sve aktivnosti obavljaju ispod *DSI*-a, tj. programi su napisani na HLL-u i mogu se izvršavati od strane mašine direktno (*directly executable language machine*).
- CISC *ISA* smeštaju *DSI* na tradicionalnom asemblerskom jeziku, ili makrokôd, nivou.
- RISC *ISA* spuštaju *DSI* i očekuju da se najveći broj optimizacija obavi od strane kompajlera.
- Svrha trenda spuštanja *DSI*-a ka nižim nivoima je smanjenje kompleksnosti hardvera i realizacija mašina koje brže izvršavaju programe.



Kako bolje, brže i jeftinije projektovati efikasan računar ?

- Još od pojave prvih računara koji su sekvencijalno izvršavali naredbe projektanti su težili da pronadju nove tehnike, metode i algoritme za brže i efikasnije izvršenje programa

$$1000 = 10 * 10 * 10$$

Pretpostavimo da smo dobili zadatak da konstruišemo mašinu koja će imati 1000 puta bolje performanse u odnosu na neku sekvencijalnu koju tekuće koristimo.

Pitanje se postavlja: Kako ćemo ostvariti taj cilj ?

Kao prvo, bolje performanse za 10 puta možemo ostvariti zahvaljujući prednostima koje se nude od strane nove tehnologije (brža taktna frekvencija, nova tehnilogija izrade tranzistora, ...)

još 10 puta superiornije performanse mogu se ostvariti boljom arhitekturnom organizacijom (ugradnjom keš memorije, ugradnjom akceleratora, protočnom obradom, ...)

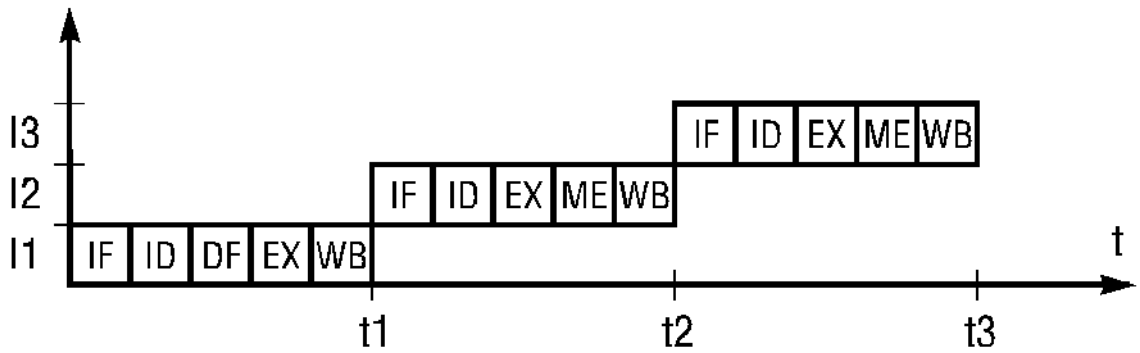
konačno, još 10 puta bolje performanse mogu se postići korišćenjem paralelizma

Serijsko, protočno i paralelno izvršenje

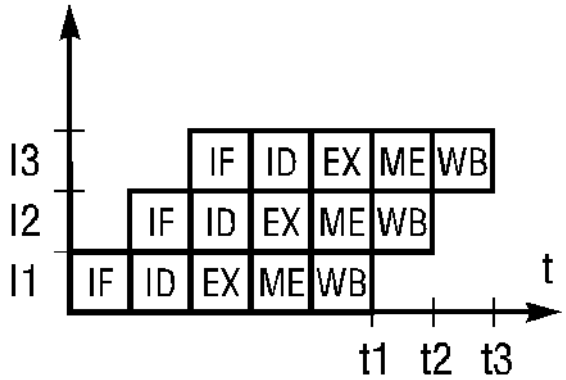
- Dve ključne tehnike za povećanje brzine serijskog načina izvršenja instrukcije zasnivaju se na korišćenju protočnosti i paralelizma

Kod protočnih sistema hardver koji obavlja odredjenu operaciju je podeljen na stepene

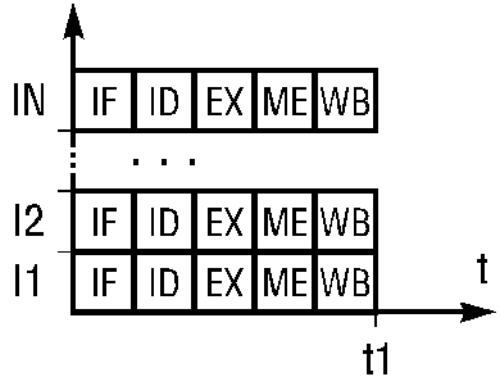
Kod paralelnih sistema hardver se replicira tako da se po nekoliko operacija mogu istovremeno izvršavati



a) serijski pristup – jedan rezultat na svakih pet taktnih intervala



b) protočnost – jedan rezultat po ciklusu



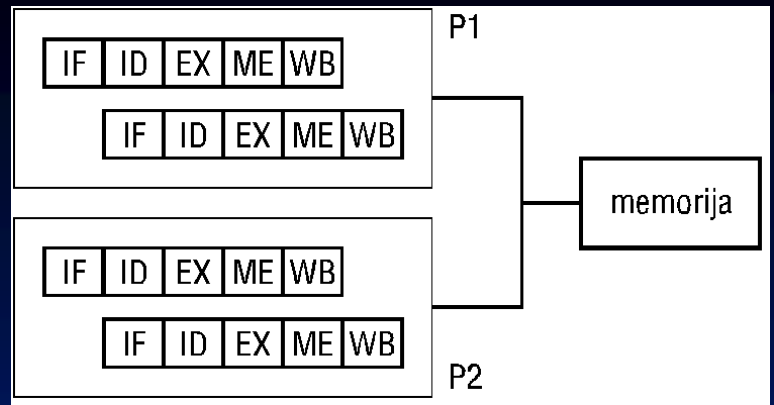
c) paralelizam – N rezultata za pet taktna intervala

Koja obrada je bolja: protočna ili paralelna ?

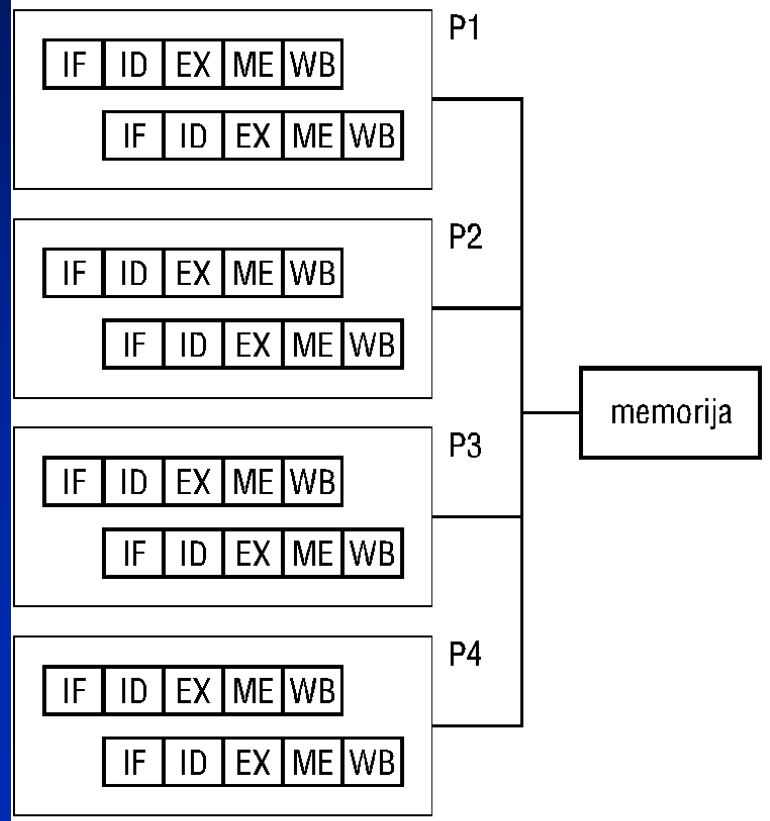
Nezavisno od toga što je protočnost superiornija tehnika u odnosu na paralelizam iz razloga što ne zahteva multipliciranje hardvera nego samo ubacivanje lečeva izmedju stepena, ipak savremene arhitekture da bi ostvarile bolje performanse koriste kombinaciju protočnosti i paralelizma kakav je slučaj sa super računarima iz serije Cray

Dobitna kombinacija je:

protočnost + paralelizam



a) dvo–procesorsko protočna mašina



b) četvoro–procesorsko protočna mašina

Protočna obrada instrukcija

- Protočnost (*pipelining*) je standardna hardverska tehnika koja se kod računara koristi za postizanje boljih performansi

- Kada se govori o izvršenju instrukcija ono se ostvaruje na taj način što se obrada instrukcije deli na veći broj fiksnih koraka koji se izvršavaju sekvencijalno

- Protočnu realizaciju funkcije čine nekoliko hardverskih stepena S1, ..., Sn, medjusobno razdvojenih lečevima

- Kako instrukcija prolazi kroz protočni sistem, hardver svakog stepena obavlja odredjeni tip obrade

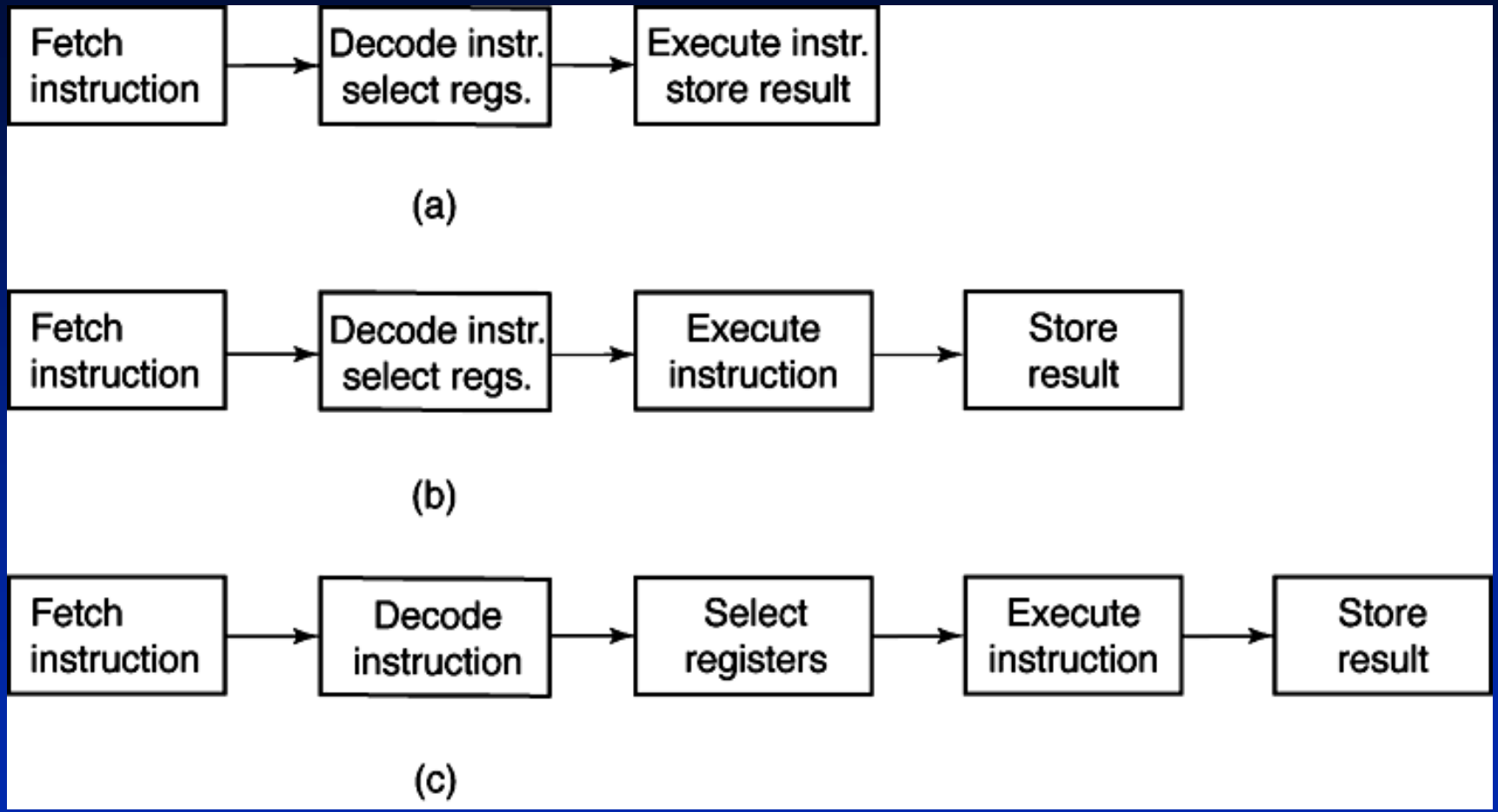
- Kada instrukcija napusti protočni sistem ona je u potpunosti izvršena

- Dobre performanse kod protočne obrade postižu se zahvaljujući paralelizmu. Paralelizam se odnosi na istovremenu obradu nekoliko insrtrukcija od kojih se svaka izvršava u različitom stepenu

Podela izvršenja instrukcije

- Izvršenje instrukcije čini sekvencu akcija
- Akcije se specificiraju semantikom instrukcije
- Neke od karakterističnih akcija, koje se obično u toku izvršenja instrukcije javljaju, su:
  - Pribavljanje instrukcije – **IF** (*Instruction Fetch*)
  - Dekodiranje instrukcije – **ID** (*Instruction Decode*)
  - Pribavljanje operanada – **OF** (*Operands Fetch*)
  - Izvršenje instrukcije – **EX** (*Instruction Execution*)
  - Pristup memoriji – **ME** (*Memory Access*)
  - Upis rezultata – **WB** (*Write-Back*)
  - . . .

Tro-, četvoro-, i peto-stepena protočna obrada



Tok podataka kod: tro-, četvoro-, i peto-stepenih protočnih sistema

Stage \ Clock cycle	1	2	3	4	5	6	7
1	I1	I2	I3	I4	I5	I6	I7
2	–	I1	I2	I3	I4	I5	I6
3	–	–	I1	I2	I3	I4	I5

(a)

Stage \ Clock cycle	1	2	3	4	5	6	7
1	I1	I2	I3	I4	I5	I6	I7
2	–	I1	I2	I3	I4	I5	I6
3	–	–	I1	I2	I3	I4	I5
4	–	–	–	I1	I2	I3	I4

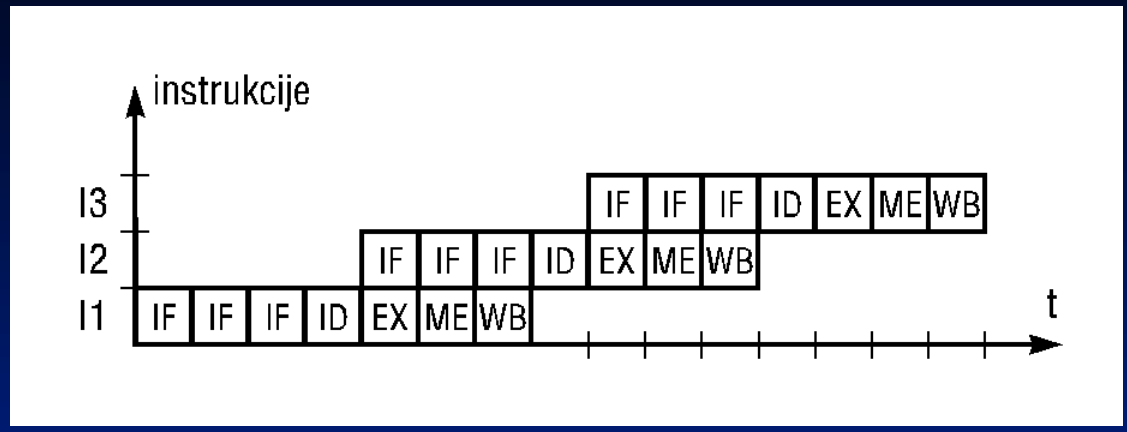
(b)

Stage \ Clock cycle	1	2	3	4	5	6	7
1	I1	I2	I3	I4	I5	I6	I7
2	–	I1	I2	I3	I4	I5	I6
3	–	–	I1	I2	I3	I4	I5
4	–	–	–	I1	I2	I3	I4
5	–	–	–	–	I1	I2	I3

(c)

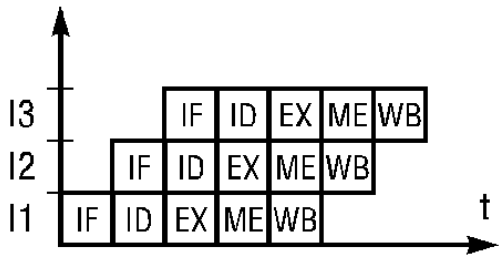
Preklapanje, plitka i duboka protočnost

Na tržištu su se prvo pojavile mašine koje su se zasnivale na preklapanju aktivnosti kod izvršenja instrukcija (overlaping)

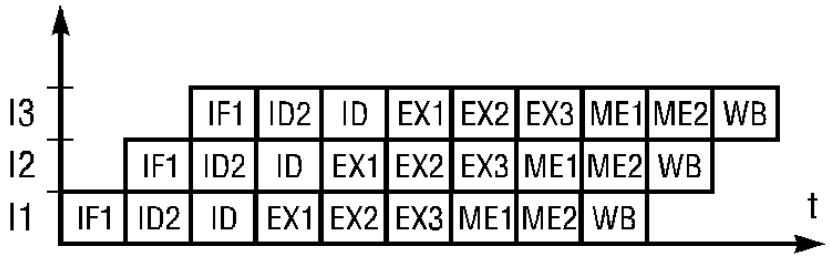


Protočnost se deli na:

1. **plitku – od 5 do 6 stepeni**
2. **duboku – više od 9 stepeni**



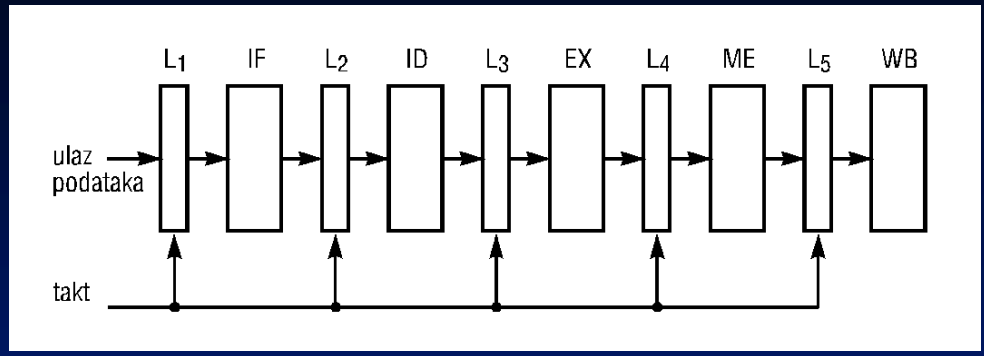
***a)** plitka protočnost*



***b)** duboka protočnost*

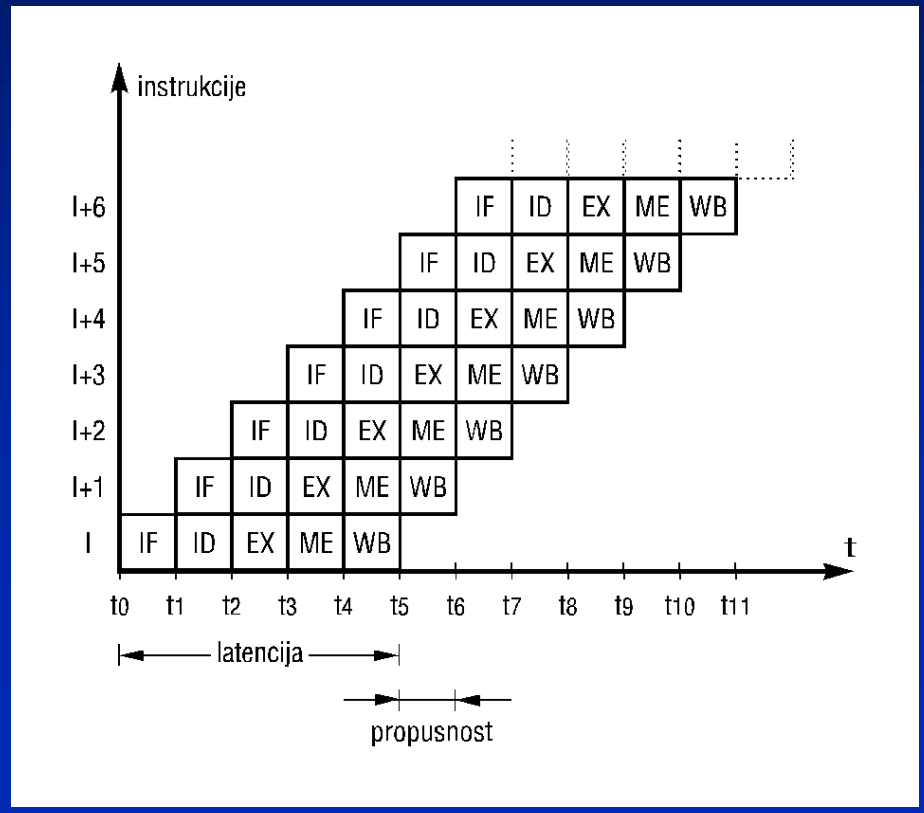
Peto-stepena protočna obrada

najtipičnija protočna obrada

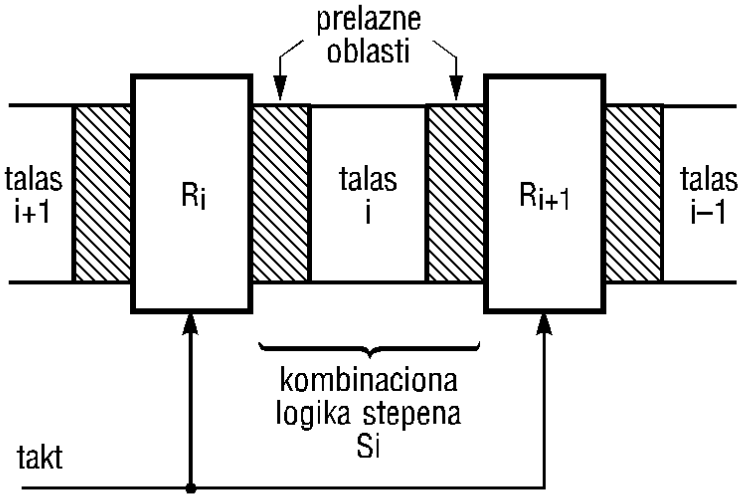


Taktni period odredjuje propusnost (throughput) i ukazuje na brzinu sa kojom se izvršavaju instrukcije

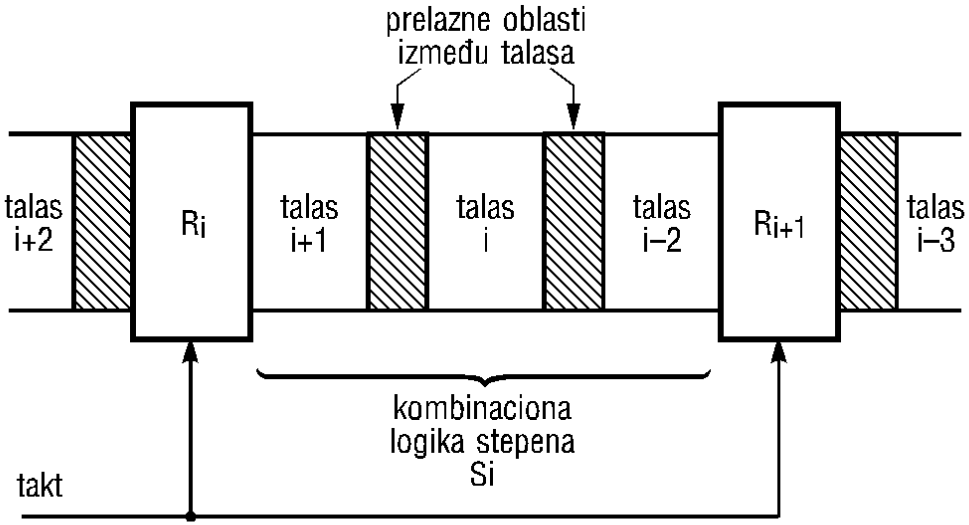
Vreme potrebno jednoj instrukciji da prodje kroz sve blokove protočnog sistema zove se latencija



Talasna protočnost kao jedna od superiornijih tehnika



a) *klasičan protočni sistem*



b) *talasno–protočni sistem*

Klase procesora

Postoje sledeće klase procesora:

- **Mikrorontroleri**
- **RISC procesori**
- **CISC procesori**
- **VLIW procesori**
- **Digitalni Signal Procesori (DSP)**
- **Multimedia procesori**
- **Aplikaciono Specifični *Instruction Set Processors* (ASIP)**
- **Druge klase**

U daljem izlaganju ograničavamo se na izučavanje RISC i CISC procesora

Šta znače skraćenice RISC i CISC ?

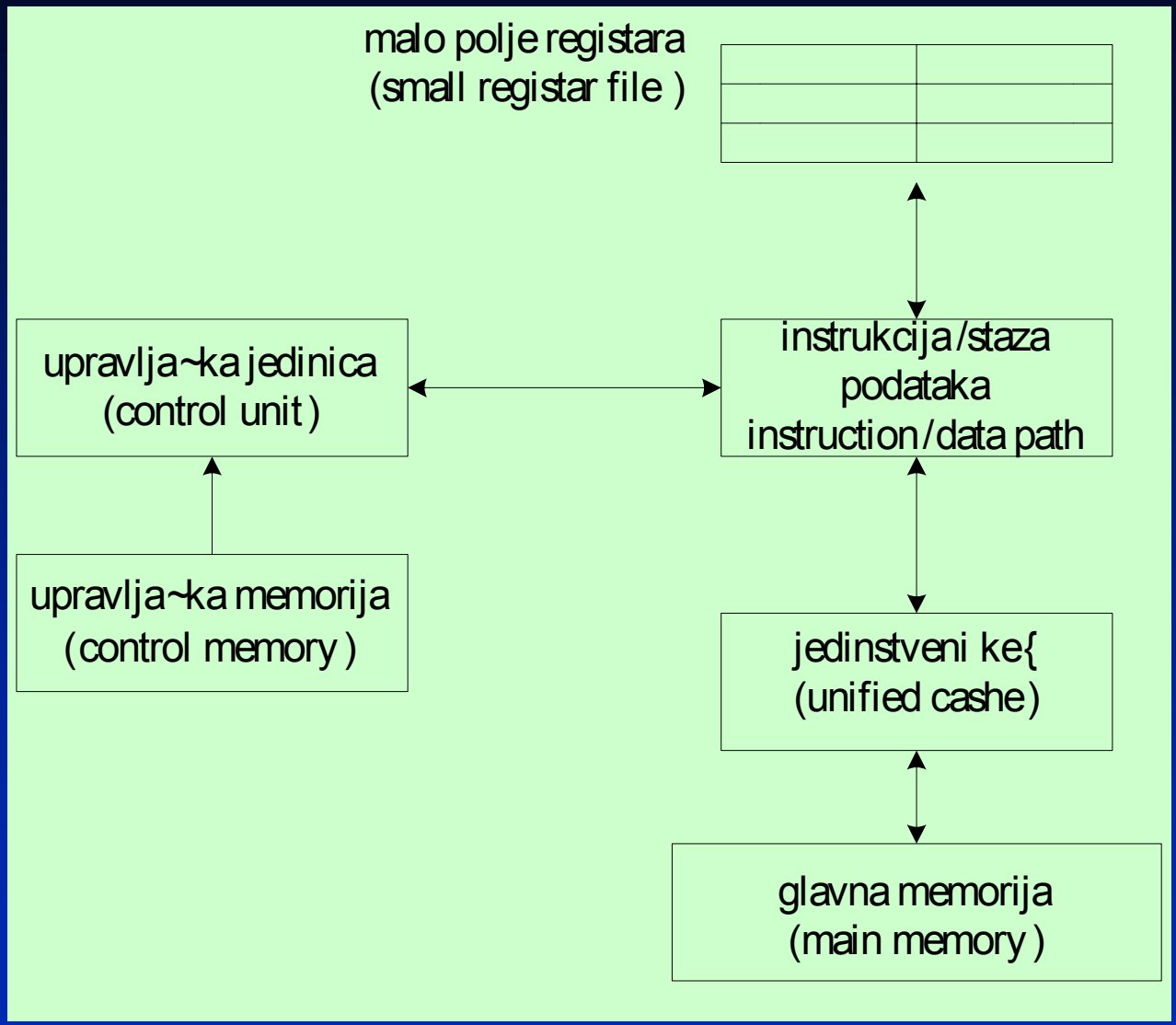
RISC je skraćenica od *Reduced Instruction Set Computer* što znači procesor sa redukovanim skupom naredbi – broj naredbi reda 100

CISC je skraćenica od *Complex Instruction Set Computer* što znači procesor sa složenim skupom naredbi – broj naredbi reda nekoliko stotina (600 – 800)

Prelazak sa CISC na RISC – CISC koncepti

- RISC i CISC koncepti predstavljaju dve škole koje se odnose na ISA
- Osnovne arhitekturne karakteristike klasičnih CISC procesora su:
  - Ranije CISC arhitekture imale su:
    - Jedinstveni keš u kome se čuvaju instrukcije i podaci.
    - Isti put za podatke/instrukcije i koristile malo registarsko polje (RF-*register file*).
    - Mikroprogramski upravljanu memoriju za implementiranje velikog skupa naredbi
  - Noviji CISC procesori kakvi su MC68030 i MC68040 koriste:
    - Posebne keševe za podatke i instrukcije.
    - Rešenja koja se okreću ka direktnom upravljanju.

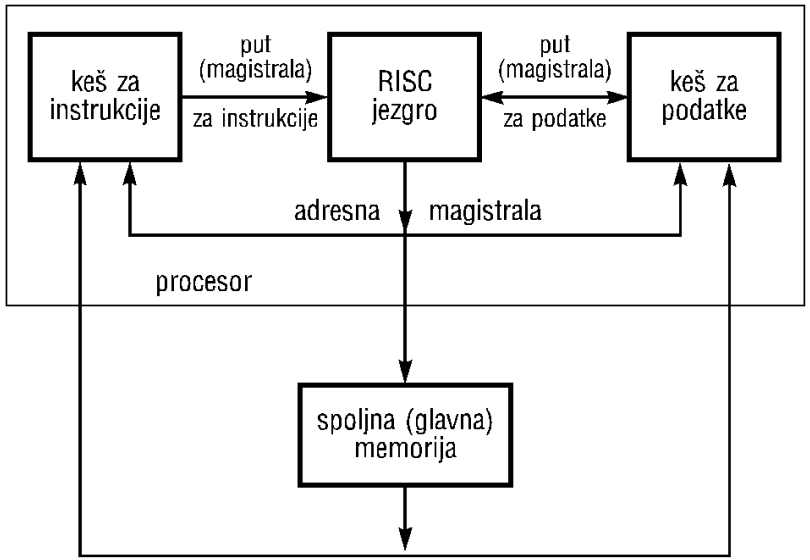
Klasični CISC procesor



Klasični CISC procesori koriste jedinstveni put za pristup memoriji podataka i memoriji instrukcija. Ovakav tip arhitekture se naziva ***Princeton arhitektura***

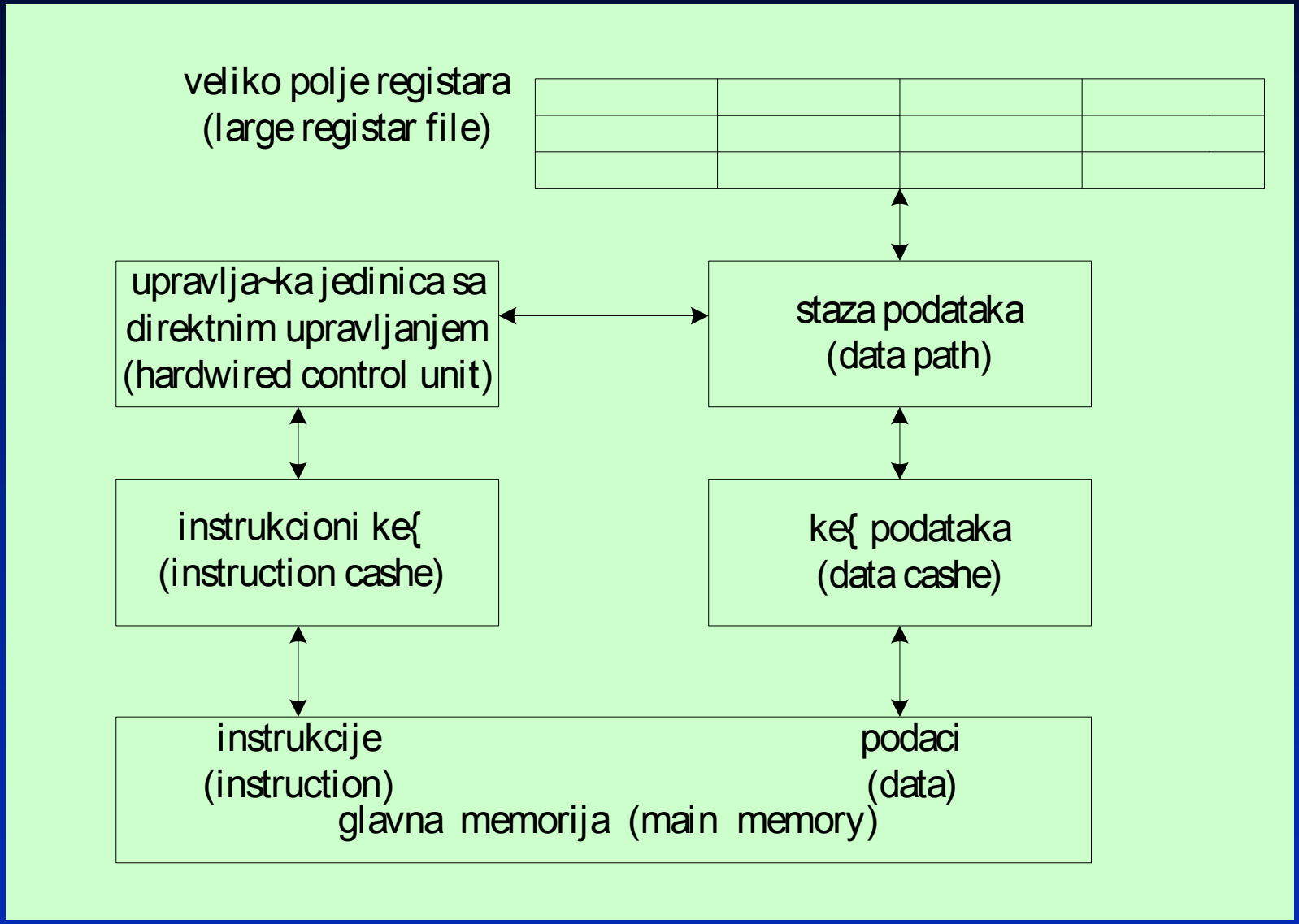
Prelazak sa CISC na RISC – RISC koncepti

- Keševi za podatke i instrukcije su izdvojeni, a takodje su različiti i putevi preko kojih se pristupa ovim memorijama (Harvard arhitektura).
- Kod RISC-ova se koristi registarsko polje nešto većeg obima
- Upravljačka jedinica je bazirana na direktnom upravljanju.
- Moguće je postići CPI reda jedan ciklus.

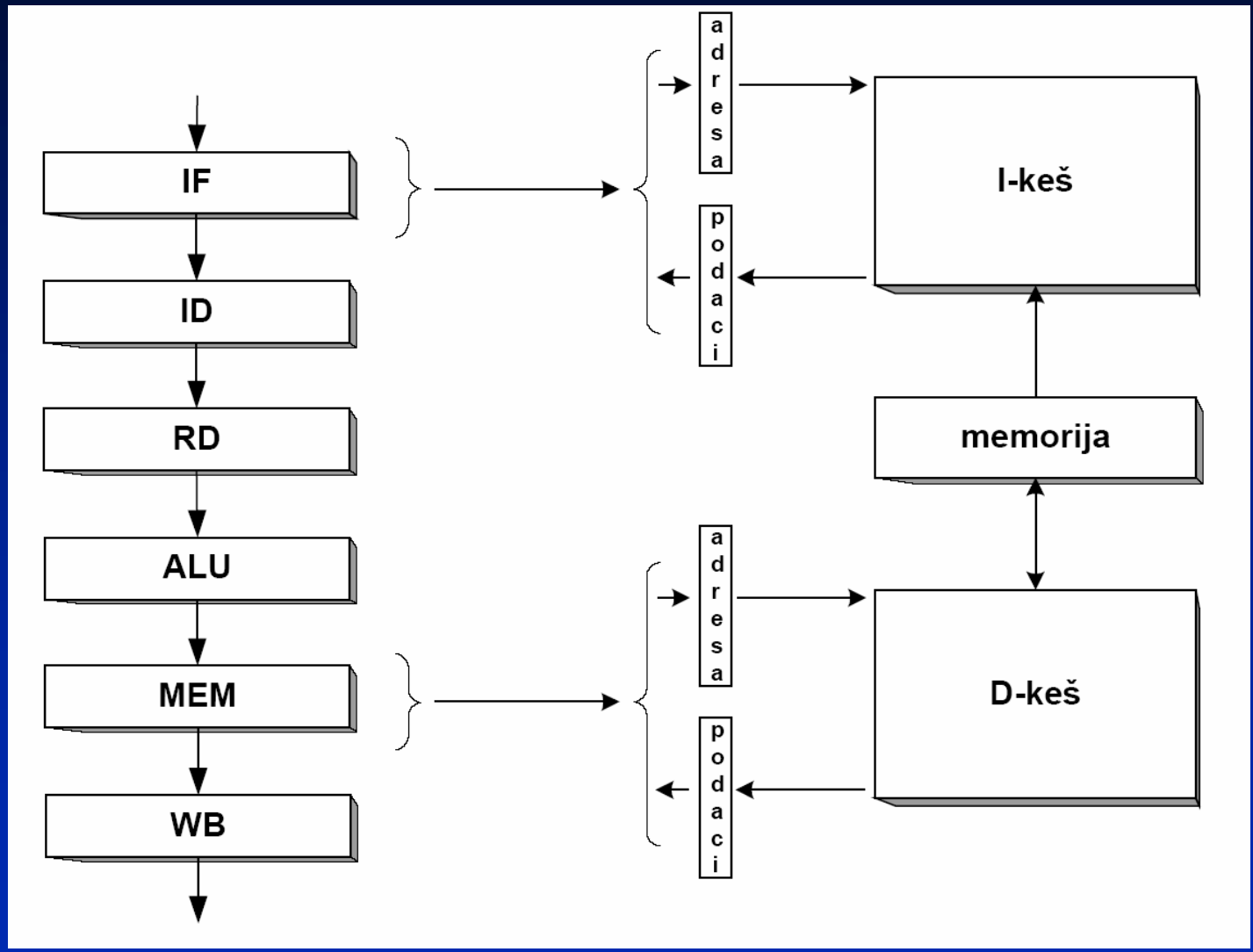


Arhitektura kod koje postoje posebni putevi za pristup podacima i instrukcijama naziva se Harvard arhitektura

Klasični RISC procesor



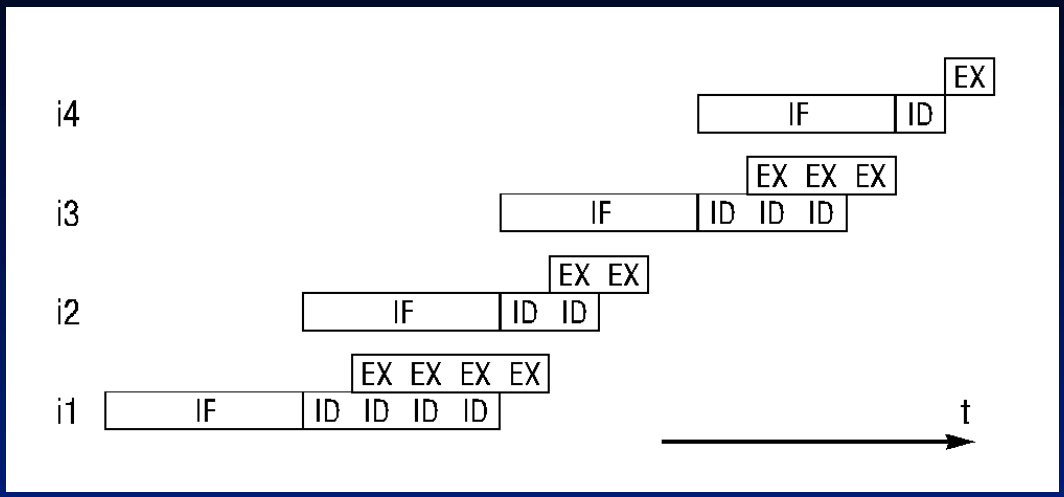
Pristup I-kešu i D-kešu



Smernice razvoja kod CISC-ova

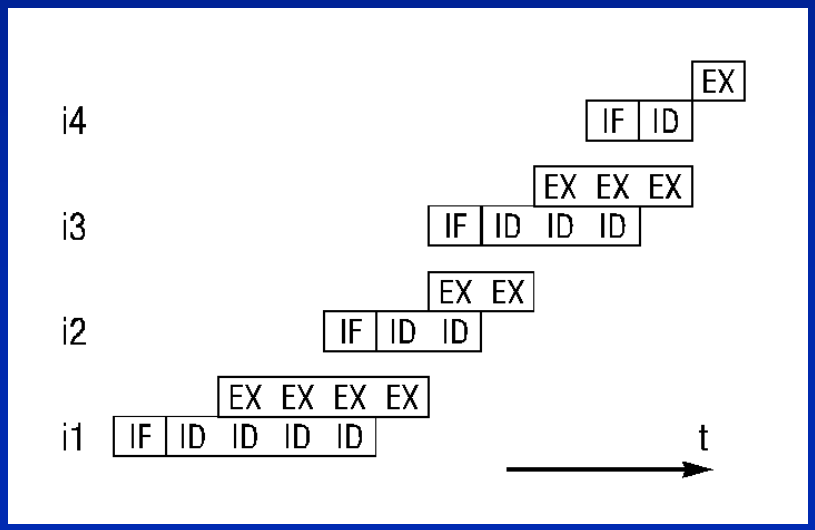
- Reprezenti su procesori iz serije Intel x86, Motorola MC 68xxx, Digital VAX serija, i neke IBM mašine.
- Porast skupa instrukcija je bio stimulisan popularnošću mikroprogramskog upravljanja 60-tih i 70-tih godina prošlog veka.
- Jedan tipičan CISC procesor ima više od 300 instrukcija.
- Koriste promenljive formate instrukcija/podataka, pri čemu obim tipičnih reči mo`e biti 8-, 16-, 32-, i 64-bita.
- Koristi se relativno mali skup registara opšte namene, od 8 do 24.
- Postoji veliki broj operacija obraćanja memoriji, koje koriste na desetinu adresnih načina rada (oko 20), uključujući indeksno i indirektno adresiranje.
- Skoro svi veliki proizvodjači mikroprocesorskih čipova imali su značajne investicije u razvoju CISC arhitektura od kasnih 60-tih do pojave prvih RISC čipova

Razlika izmedju CISC i RISC pristupa u pogledu memorije

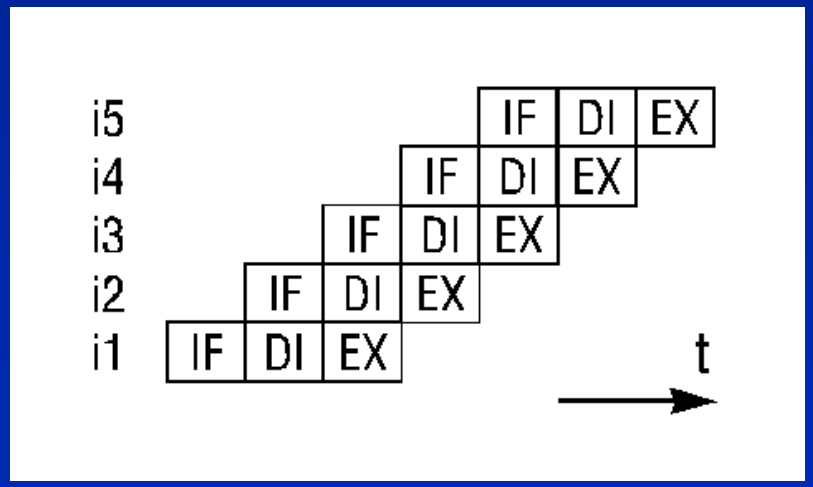


Protočna obrada kod CISC mašina kada se vreme pristupa memoriji u toku faze IF obavljalo za veći broj ciklusa

CISC procesori kod kojih se vreme pristupa memoriji u toku faze IF obavlja za jedan ciklus



Tajming RISC procesora pod uslovom da ne postoje zavisnosti



RISC izazovi

- Nakon tri decenije razvoja CISC arhitektura, korisnici računara su počeli da procenjuju odnos izmedju ISA i dostupnih hardversko/softverskih tehnologija.

- Analizama programa (uglavnom sprovedenih trasiranjem) ustanovljeno je da:

  - 25% od svih raspoloživih kompleksnih instrukcija troši 95% od ukupnog vremena potrebnog za izvršenje programa

  - ostalih 75% hardversko-podr`avanih instrukcija retko koristi u toku izvršenja programa (manje od 5%).

- Tako se i rodila ideja o RISC-ovima koja se bazira na sledećoj činjenici:

  Učini ono što se najviše koristi najbr`im.

- Kao posledica ovakvog pristupa ostvarilo se dramatično povećanje performansi u odnosu na CISC dizajn.

RISC izazovi – prod.

Klasično pitanje

Zbog čega tro{iti veliki deo površine čipa na instrukcije koje se retko koriste ?

Odgovor

- **Kompleksne instrukcije, nakon procene njihovog pojavljivanja u programima, treba kada je njihov procenat mali implementirati softverski (u vidu potprograma), a ne hardverski kakva je bila dotada{nja praksa kod CISC procesora.**
- **Implementacija retko korišćenih instrukcija u softveru omogućava da se skoro svi gradivni blokovi procesora smeste na jedinstveni VLSI čip. Šta više, u okviru jednog RISC procesorskog čipa, na dana{njem nivou tehnologije, moguće je ugraditi sada *on-chip* keš kao i ve}i broj FP jedinica.**
- **Skup instrukcija RISC procesora je manji od 100 instrukcija, pri čemu su instrukcije obima 32- ili 64- bita, ali ne oba formata istovremeno.**
- **Koriste se od tri do pet adresnih načina rada. Arhitektura RISC-a je tipa *Load/Store*. To znači da sve instrukcije pribavljaju operande iz registara i smeštaju rezultate i registre (*register-based*), a da su *Load* i *Store* jedine dve naredbe pomoću kojih se procesor obraća memoriji.**

RISC izazovi – prod.

- **RISC procesor često koristi dva posebna registarska polja:**
  - **Jedno polje čine 32 registra za manipulisanje sa celobrojnim vrednostima i adresama (*integer register file*),**
  - **Drugo polje se sastoji od 32 registra koji se koriste kod manipulisanja brojeva u pokretnom zarezu (*floating point register file*).**
- **Neki od RISC procesora koriste i više od 100 registara.**
- **Pored korišćenja registarskih polja velikog obima, podeljeni keševi za instrukcije i podaci dodatno i značajno skraćuju vreme pristupa memoriji.**
- **Osnovna ideja RISC-a je da se najveći broj instrukcija izvrši za jedan ciklus što se postiže pribavljanjem operanda direktnim putem iz registara, bafera preuredjenja ili keša podataka, a ne kao kod CISC-ova iz memorije.**

CISC vs. RISC

karakteristike	klasična CISC arhitektura	prava RISC arhitektura
format instrukcije	promenljivi format 8,16,32i 64 bita	fiksna 32-bitne instrukcije
taktna frekvencija	100-500 MHz	200-400 MHz
polje registara	8-24 registara opšte nemene	32-192 registara opšte namene, izdvojeni *integer* i FP registri
broj instrukcija i tipovi	oko 300, sa preko četrdesetak varijanti instrukcionih tipova	oko 100, najveći broj su registarsko zasnovani sa izuzetkom *Load/Store*
adresni načini rada	na desetinu (reda 20) uključujući indeksno/ indirektno adresiranje	ograničeno na 3-5, samo Load/Store adresiraju meoriju
dizajn keša	raniji modeli su koristili jedinstveni keš, noviji koriste razdvojene keševe za instrukcije i podatke	najveći broj koristi podeljeni keš za instrukcije i podatke
CPI, i prosečni CPI	1-20 ciklusa, u proseku 4 ciklusa	1 ciklus za jednostavne operacije, 1,5 ciklus u proseku
upravljačka jedinica CPU-a	najveći broj koristi mikroprogramsko upravljanje, a kod nekih imamo i direktno	najveći broj koristi direktno upravljanje bez upravljačke memorije
tipični reprezentativni rocesori	Intel x86, VAX 8600, IBM 390, MC 68040, Intel Pentium, AMD 486, Cyrix 686	Sun UltraSparc, MIPS R10000, Power PC 604, HP PA-8000, Digital 21264

Šta je donela dvo-decenijska ISA debata ?

- Korišćenjem velikog registarskog polja, bafera podataka, i izdvojenih keševa za instrukcije i podatke, imalo je povoljan efekat na interno prosledjivanje podataka kroz CPU, kao i eliminisanju nepotrebnih memorisanja medjurezultata operacija.

- Smanjenom kompleksnošću skupa instrukcija, RISC procesor se može lakše projektovati i realizovati kao jedinstveni VLSI čip.

- Prednost ovog pristupa su:

  - rad na višim frekvencijama,
  - niži prosečni CPI koji se mo`e posti}i,
  - ni`i procenat keš promašaja koji se može ostvariti,
  - bolje mogućnosti koje stoje na raspolaganju za optimizaciju kompilatora

Šta je donela dvo-decenijska ISA debata ? - prod

- Prelazak sa CISC na RISC koncept predstavljao je radikalan preokret u arhitekturi.
- Glavna žrtva je predstavljala gubitak binarne kompatibilnosti sa tradicionalnim CISC aplikacionim programima.
- Postoje}i programi kreirani za neku od CISC ma{ina nisu mogli vi{e da se sada izvršavaju na RISC mašinama.
- Istra`ivanja koja su se odnosila na trasiranje osobina programa pokazala su da konverzija sa CISC programa na ekvivalnetni RISC program podrazumeva povećanje dužine programa (broj instrukcija) za oko 40%.

Negativni efekat povećanja dužine RISC programa, sa aspekta vremena izvršenja, je ipak ~ini se manji u odnosu na pozitivne efekte koji se postižu povećanjem taktne frekvencije i smanjenjem prosečnog broja CPI-a kod RISC procesora.

Ko je dobio, a ko izgubio ?

- Debata izmedju projektanata RISC i CISC procesora, oko toga koji je pristup bolji, po svemu izgleda je sada završena.
- Obe strane u proteklom periodu naučile su i prihvatile jedna od druge puno dobrih i korisnih sugestija i rešenja.
- Na današnjem nivou razvoja po svemu sude}i te granice su postale takoreći nevidljive.
- Veliki broj procesora danas se implementira u istoj tehnologiji ugradjivanjem hibridnih rešenja koje uzimaju u obzir dobre osobine kako RISC tako i CISC procesora.

Verovatno je kompromis, kao i uvek, najboje rešenje.

Zavisnosti kod protočne obrade

- **Zavisnost je okolnost koja potencijalno sprečava da CPU izvršava instrukciju svojom maksimalnom specificiranom brzinom**

- **Kada protočni sistem izvršava dve instrukcije izmedju kojih postoje zavisnosti tada može da dodje do zastoja ili nekorektnog rada sistema pa se takva situacija naziva hazard (opasna situacija)**

- **Hazardi se uglavnom javljaju zbog medjusobne zavisnosti dve instrukcije kada su one u redosledu izvršenja suviše bliske jedna drugoj**

Zavisnost može da dovede do hazarda, ali ne mora

Tri klase zavisnosti/hazarda

- **Razlikujemo tri klase zavisnosti/hazarda:**

Strukturni – nastaju zbog konflikta kod korišćenja resursa, tj kada dve ili više instrukcija koje se nalaze u različitim fazama obrade zahtevaju angažovanje istog hardverskog resursa

Po podacima – tipično ih srećemo u situacijama kada dalje izvršenje instrukcije zavisi od rezultata koji generiše prethodna instrukcija

Upravljački – uslovljeni su od strane instruikcija grananja i svih drugih instrukcija koje menjaju sadržaj programskog brojača (PC)

Zavisnosti po podacima

- **Ove zavisnosti se često nazivaju i programske zavisnosti**

Na ISA abstraktnom nivou, program se specificira kao sekvenca asemblersko jezičkih instrukcija. Jedna tipična instrukcija se specificira kao funkcija:

$$i : T \leftarrow S_1 \; op \; S_2$$

gde je:

$D(i) = \{S_1 , S_2\}$ - **domen instrukcije** i ; $R(i) = \{T\}$ predstavlja **opseg instrukcije** i ; a preslikavanje iz domena u opseg je definisano operacijom op.

Neka su date dve instrukcije i i j , pri čemu u leksičkom redosledu obeju instrukcija, j sledi i . Izmedju instrukcija i i j kažemo da postoji **zavisnost-po-podacima** (*data dependencies*), koju označavamo sa $i \; \delta \; j$, ako važi jedan od sledeća tri uslova:

$$R(i) \cap D(j) \neq 0 \qquad (1)$$

$$R(j) \cap D(i) \neq 0 \qquad (2)$$

$$R(i) \cap R(j) \neq 0 \qquad (3)$$

Definicija RAW, WAR i WAW

$$R(i) \cap D(j) \neq 0 \qquad (1)$$

$$R(j) \cap D(i) \neq 0 \qquad (2)$$

$$R(i) \cap R(j) \neq 0 \qquad (3)$$

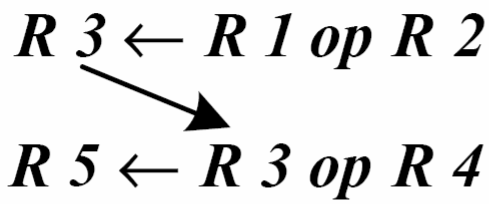
Prvi uslov ukazuje da instrukcija *j* zahteva operand koji se nalazi u opsegu instrukcija *i*. Ovo se naziva *read-after-write* (*RAW*) ili **prava** (*true*) **zavisnost po podacima**, i označava se sa $i\,\delta_d\,j$. Implikacija prave *zavisnosti po podacima* je takva da instrukcija *j* ne može da počne sa izvršenjem sve dok instrukcija *i* ne završi.

Drugi uslov ukazuje da se operand koji koristi instrukcija *i* nalazi u opsegu instrukcija *j*, ili da instrukcija *j* modifikuje promenljivu koja predstavlja operand instrukcije *i*. Ova zavisnost se naziva *write-after-read* (*WAR*) ili **anti zavisnost po podacima**, i označava se sa $i\,\delta_a\,j$. Egzistencija anti zavisnosti iziskuje da instrukcija *j* ne završi pre instrukcije *i*, inače će instrukcija *i* pribaviti pogrešan operand.

Treći uslov ukazuje da ode instrukcije *i* i *j* dele u svom opsegu zajedničku promenljivu, što znači da ode modifikuju tu promenljivu. Ova se naziva *write-after-write* (*WAW*) ili **izlazna zavisnost po podacima**, a označava se kao $i\,\delta_o\,j$. Postojanje izlazne zavisnosti iziskuje da prvo završi instrukcija *i*, a tek nakon toga instrukcija *j*, inače će instrukcije koje slede nakon instrukcije *j* koristiti pogrešan operand.

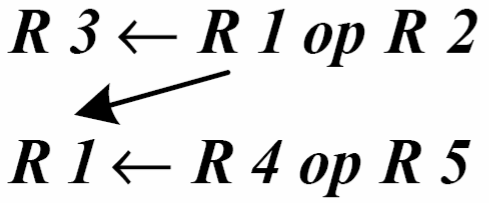
Konkretni slučajevi RAW, WAR i WAW

- prava zavisnost



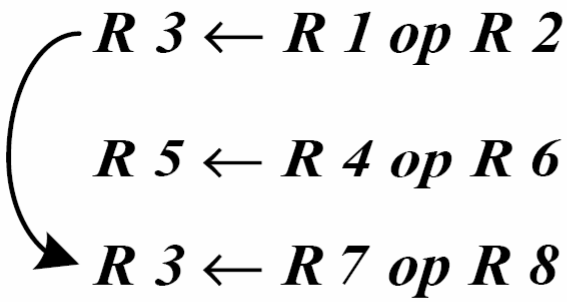
RAW zavisnost

- anti zavisnost



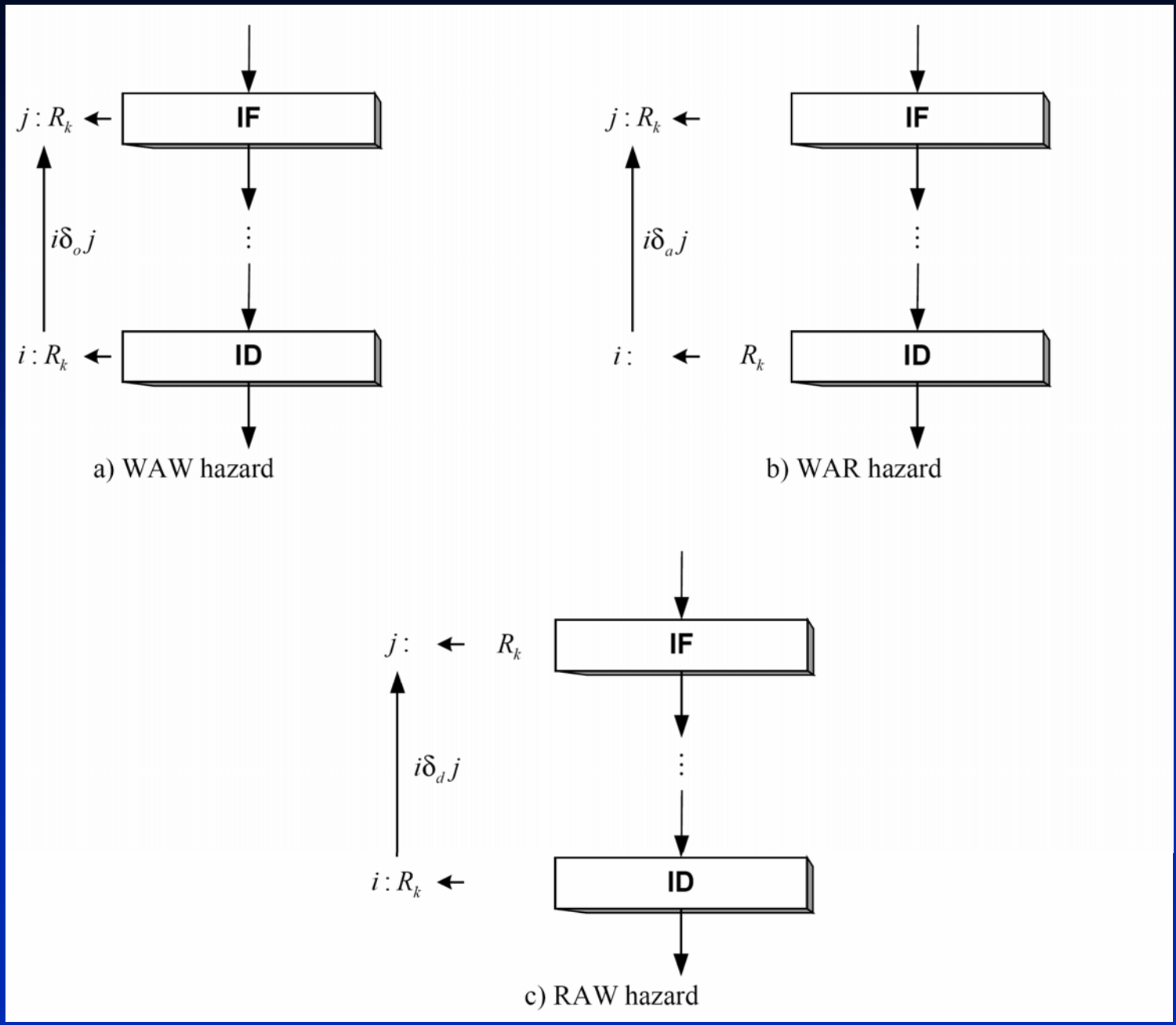
WAR zavisnost

- izlazna zavisnost



WAW zavisnost

Neophodni uslovi da se u protočnom sistemu jave WAW, WAR i RAW

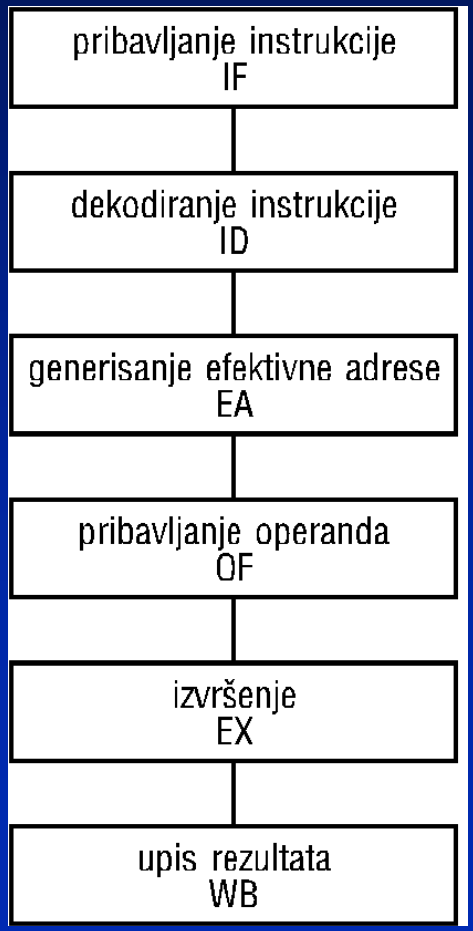


Upravljačke zavisnosti/hazardi

Pored *zavisnosti po podacima* izmedju dve instrukcije mogu da postoje i **kontrolne zavisnosti**, ili tzv. *upravljačke zavisnosti*. Neka su date dve instrukcije *i* i *j*, pri čemu instrukcija *j* sledi nakon instrukcije *i*. Tada za instrukciju *j* kažemo da je upravljački zavisna od instrukcije *i*, što se označava sa $i\ \delta_c\ j$ .Uslovne instrukcije grananja uzrokuju neodredjenost u sekvenciranju instrukcija. Naime instrukcije koje slede nakon uslovnog grananja mogu biti zavisne po upravljanju od instrukcije *Branch*.

Primer

Procesor sa šestostepenom protočnom obradom



I_i

I_{i+1}

I_{i+2} ;granaj se na I_j

I_{i+3}

.

.

.

I_j

I_{j+1}

.

.

.

niz instrukcija

Upravljačke zavisnosti/ hazardi

Primer - rešenje

Taktni interval	Protočni stepen						opis
	IF	ID	EA	OF	EX	WB	
1	I_i	–	–	–	–	–	početna latentnost
2	I_{i+1}	I_i	–	–	–	–	
3	I_{i+2}	I_{i+1}	I_i	–	–	–	
4	I_{i+3}	I_{i+2}	I_{i+1}	I_i	–	–	
5	I_{i+4}	I_{i+3}	I_{i+2}	I_{i+1}	I_i	–	
6	I_{i+5}	I_{i+4}	I_{i+3}	I_{i+2}	I_{i+1}	I_i	završetak I_i
7	I_{i+6}	I_{i+5}	I_{i+4}	I_{i+3}	I_{i+2}	I_{i+1}	završetak I_{i+1}
8	I_j	I_{i+6}	I_{i+5}	I_{i+4}	I_{i+3}	I_{i+2}	završetak I_{i+2}
9	I_{j+1}	I_j	I_{i+6}	I_{i+5}	I_{i+4}	I_{i+3}	cena koja se plaća zbog grananja
10		I_{j+1}	I_j	I_{i+6}	I_{i+5}	I_{i+4}	
11			I_{j+1}	I_j	I_{i+6}	I_{i+5}	
12				I_{j+1}	I_j	I_{i+6}	
13					I_{j+1}	I_j	završetak I_j
14						I_{j+1}	završetak I_{j+1}

Performansne mere – protočni sistem

- Osnovna ideja protočne obrade je da se ostvari preklapanje u izvršenju izmedju sukcesivnih instrukcija čime se štedi na vremenu i povećava propusnost.

- Skoro svi današnji savremeni procesori koriste tehniku protočne obrade čime na jedan indirektan način postižu bolje performanse u odnosu na strogo sekvencijalno izvršenje.

- Definišimo dva pojma koja se odnose na protočni dizajn:

  - **Protočni ciklus** (alternativni pojmovi su procesorski ciklus ili taktni period): perioda taktnog signala, *Cp* kojim se pobudjuje protočni sistem. Perioda je odredjena od najdužeg vremenskog kašnjenja potrebnog da se izvrši jedna operacija kroz neki od protočnih stepena. Tekući protočni procesori operišu sa procesorskim ciklusom koji je reda od 200 ps - 20 ns (tj. 5 GHz – 50 MHz)

  - **Latencija iniciranja instrukcija**: broj procesorskih ciklusa izmedju iniciranja dve susedne instrukcije u programskoj sekvenci.

Performansne mere – broj procesorskih ciklusa

- **Broj procesorskih ciklusa potreban za procesiranje sekvence od *k* instrukcija iznosi**

broj_procesorskih_ciklusa =

broj_procesorskih_ciklusa_da_se_generiše_prvi_rezultat
+ broj_instrukcija_u_sekvenci
- 1

= n + k -1

gde je: *n*-broj protočnih stepeni, a *k* - dužina sekvence instrukcija.

Performansne mere – broj procesorskih ciklusa po instrukciji

- **Broj procesorskih ciklusa po instrukciji (*CPI*) iznosi:**

$$CPI = (n + k - 1) / k$$
$$= 1 + (n - 1) / k$$

kada

$k \rightarrow 1$, **tada CPI** $\rightarrow n$,

a u slu~aju kada

$k \rightarrow \infty$, **tada CPI** $\rightarrow 1$,

to znači da se CPI nalazi u granicama

$$1 < CPI < n.$$

Performansne mere – ubrzanje

- **Ubrzanje (*speedup*) protočnog sistema u odnosu na neprotočni sistem se definiše kao**

-

$$S_{pro} = T_{neprotocno} / T_{protocno}$$

$$= \quad n * k / (n + k - 1)$$

to znači da $S_{pro} \rightarrow n$, **kada** $k \rightarrow \infty$

- **Važan faktor koga treba uzeti u obzir kod procene performansi predstavlja dodatno kašnjenje koje unosi svaki leč u lancu.**

- **Vreme potrebno protočnom stepenu da procesira podatak jednako je zbiru**

$$T_{ci} + T_{ri} \ ,$$

gde je: T_{ci} **- vreme procesiranja signala od strane stepena** *Si,* **a** T_{ri} **- predstavlja kašnjenje koje unosi leč** *Li* **.**

Performansne mere – ubrzanje (prod.)

- **Zbog uticaja kašnjenja koje unosi leč ubrzanje realnog protočnog sistema biće dato sledećom relacijom**

$$S_{pro} = ((n - \alpha) * k)/(k + n - 1)$$

to znači da

$$S_{pro} \rightarrow n - \alpha, \text{ kada } k \rightarrow \infty$$

gde je: α prekoračenje usled T_{ri}, tj. prekoračenje usled dodatnog vremena, svakog protočnog stepena, potrebnog da se ostvari sinhronizovan rad.

Performansne mere – efikasnost

- **Efikasnost protočnog sistema koji obavlja izračunavanje (ili niz izračunavanja) se definiše kao**

$$E_{ideal} = S_{pro} / n$$
$$= (n * k) / (n^2 + n * k - n)$$

za $n = 5$ i $k = 10$, $E = 0{,}714$;

za $n = 5$ i $k = 100$, $E = 0{,}962$;

za $n = 5$ i $k = 1000$, $E = 0{,}996$;

Tehnika projektovanja protočnog procesora

- Skup instrukcija koje izvršava protočni procesor ima veoma veliki uticaj na njegovu strukturu i funkcionisanje.

- Još u fazi projektovanja protočnog procesora mora prvo da se sagleda na koji način se različite klase mašinskih instrukcija preslikavaju, tj. uklapaju, u protočnu strukturu.

Klasifikovaćemo instrukcije na osnovu sledećeg kriterijuma:
Kakav je i gde se ostvaruje tok podataka u toku izvršenja instrukcije ?

Shodno odabranom kriterijumu moguće je identifikovati sledeća tri tipa instrukcija:

ALU **– aritmetičko/logičke**

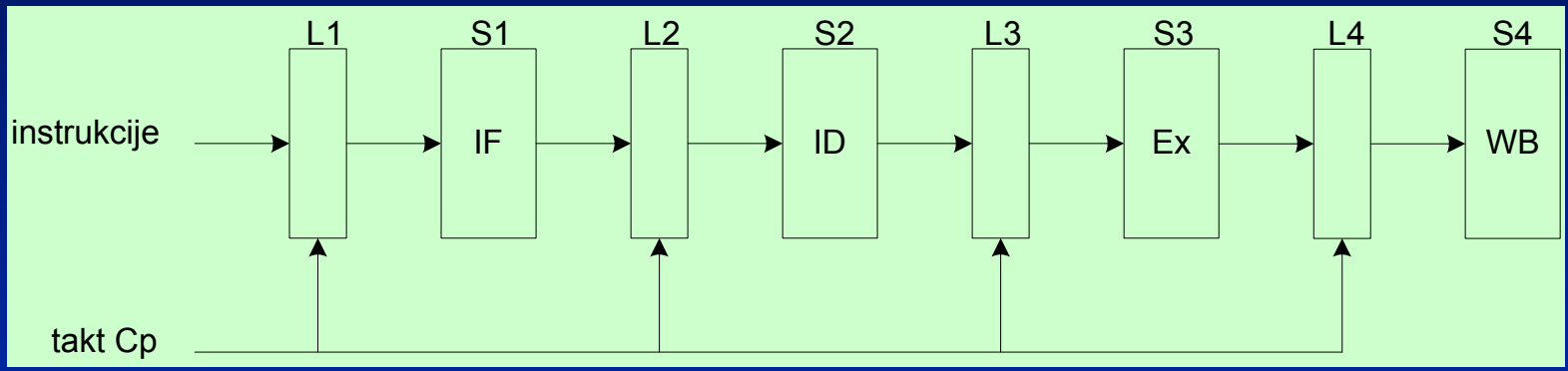
Load/Store **– punjenje registra iz memorije/upis stanja registra u memoriju**

Branch **– grananja, mogu biti uslovna i bezuslovna**

ALU instrukcije

- Analizira}emo prvo kakva treba da bude struktura 32-bitnog procesora kada on izvršava aritmetičke instrukcije tipa *registar-u-registar*.

- U konkretnom slučaju analiziraćemo četvoro-stepeni protočni sistem, *FX-4P*, koga čine stepeni IF, ID, EX i WB, ispunjava postavljene zahteve.



Operacija koja se izvršava je tipa

```
Add  R1,R2,R3
```

; (R1) ← (R2) + (R3)

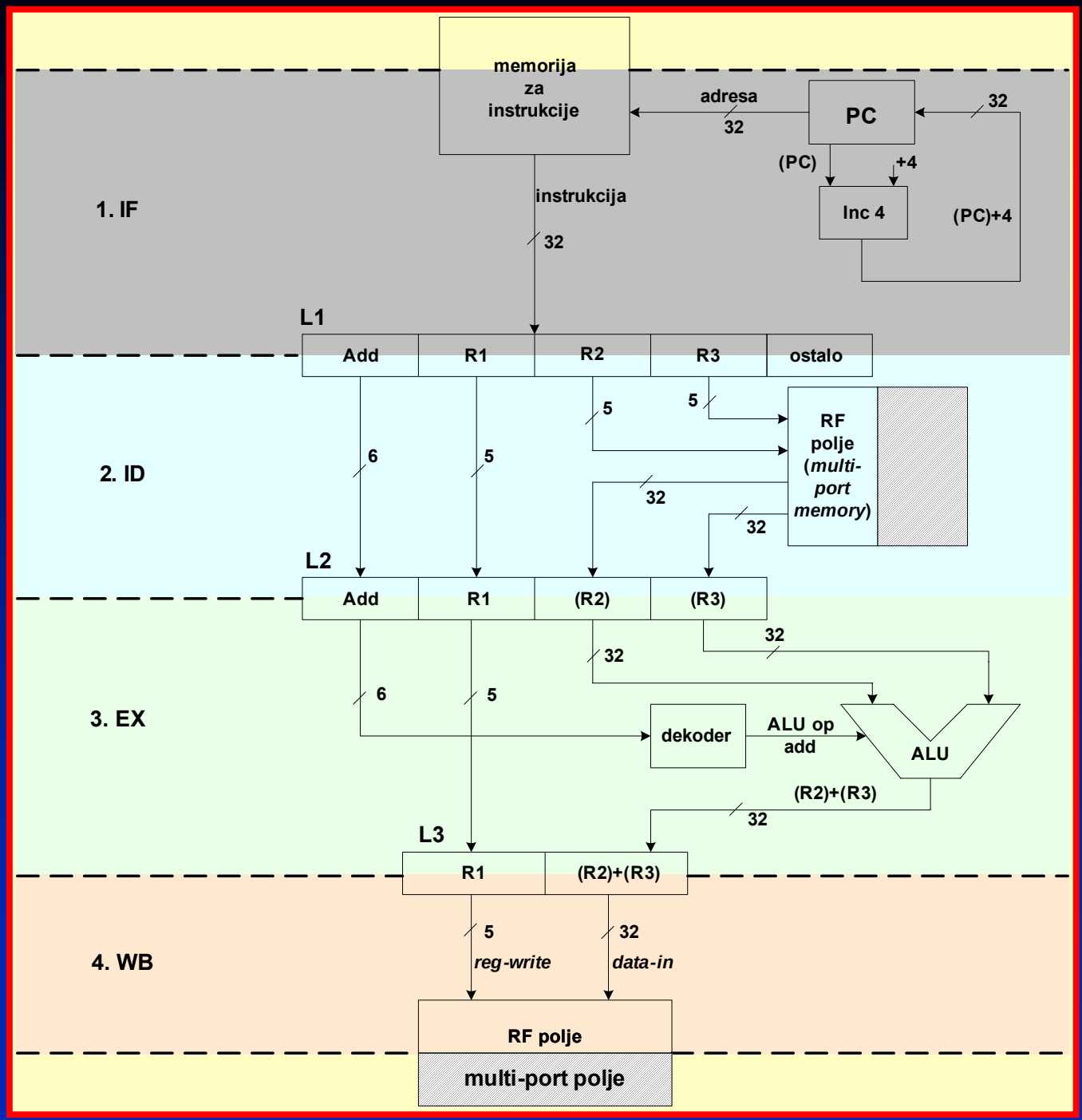
Format ALU instrukcije-ADD

6	5	5	5	11
opkôd	Rd	Rs_1	Rs_2	ostalo

- **Format instrukcije *Add* čine polja:**
  - ***Rs1* = *Rs2* = *Rd*** **svi obima po 5 bitova (*RF* polje čine 32 registra),**
  - **opkôd polje obima 6 bitova (moguće je kodirati do 64 različite instrukcije)**
  - **polje *ostalo* veličine 11 bitova**

Potrebna struktura FX-4P za ADD

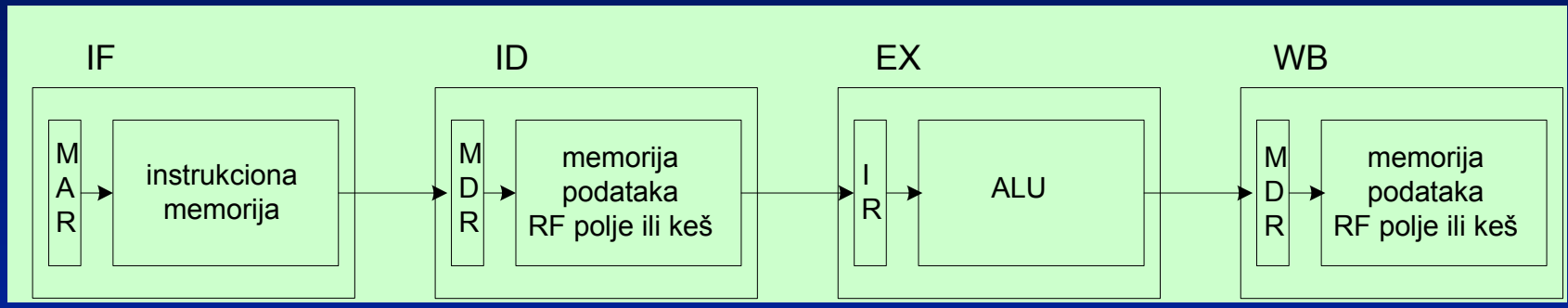
Add R1,R2,R3

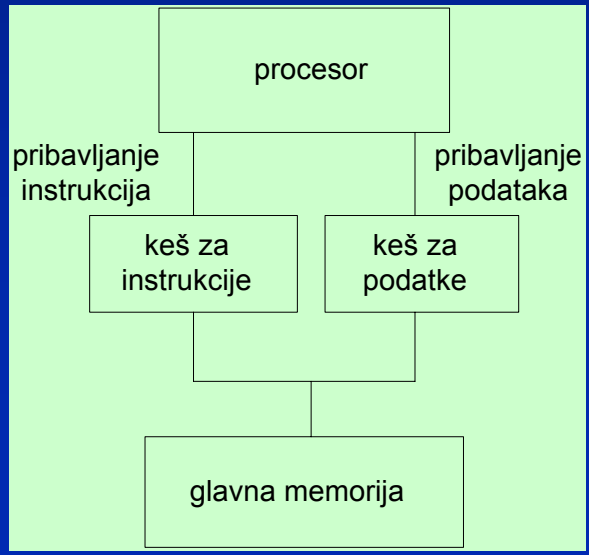


Instrukcije tipa *Load* i *Store*

- Kod instrukcija tipa *Load* i *Store* u okviru aktivnosti koje obavljaju protočni stepeni vrši se pristup memorijama za podatke.

- Saglasno ovome može se izvesti jedan alternativni način prikazivanja rada protočnog sistema koji za odgovarajući protočni stepen u prvi plan postavlja pristup memoriji (kešu) za podatke





Karakteristike *Load* i *Store* instrukcija

- Današnji RISC procesori se realizuju kao *Load*/*Store* mašine, pa je pogodno kod ovakvih mašina da postoji još jedan protočni stepen koji bi obavljao aktivnost tipa pristup-memoriji.

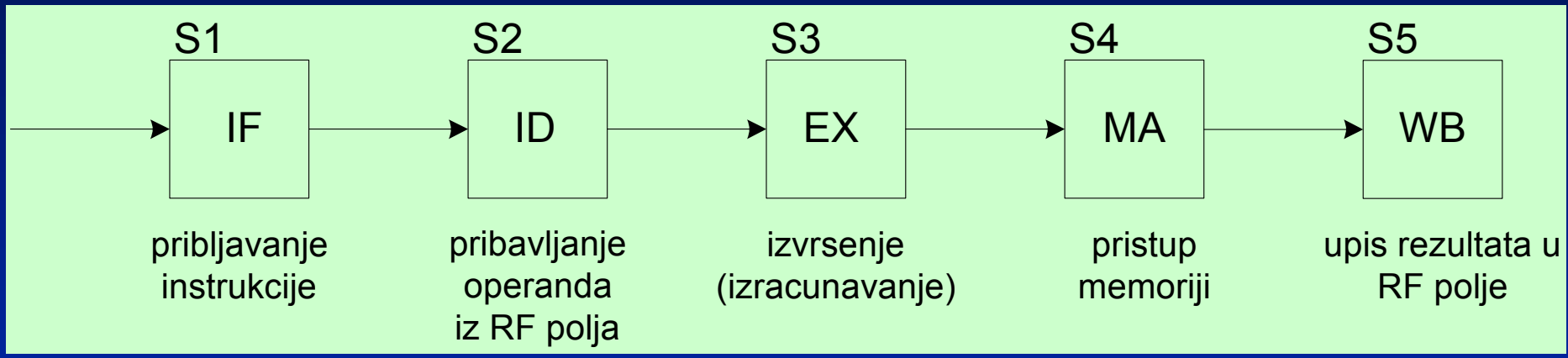
- Memorijske adrese za obe, *Load* i *Store,* operacije izračunavale bi se putem registarsko indirektnog načina adresiranja.

- Ovakav pristup nalaže da protočni sistem čine sledećih pet stepeni:

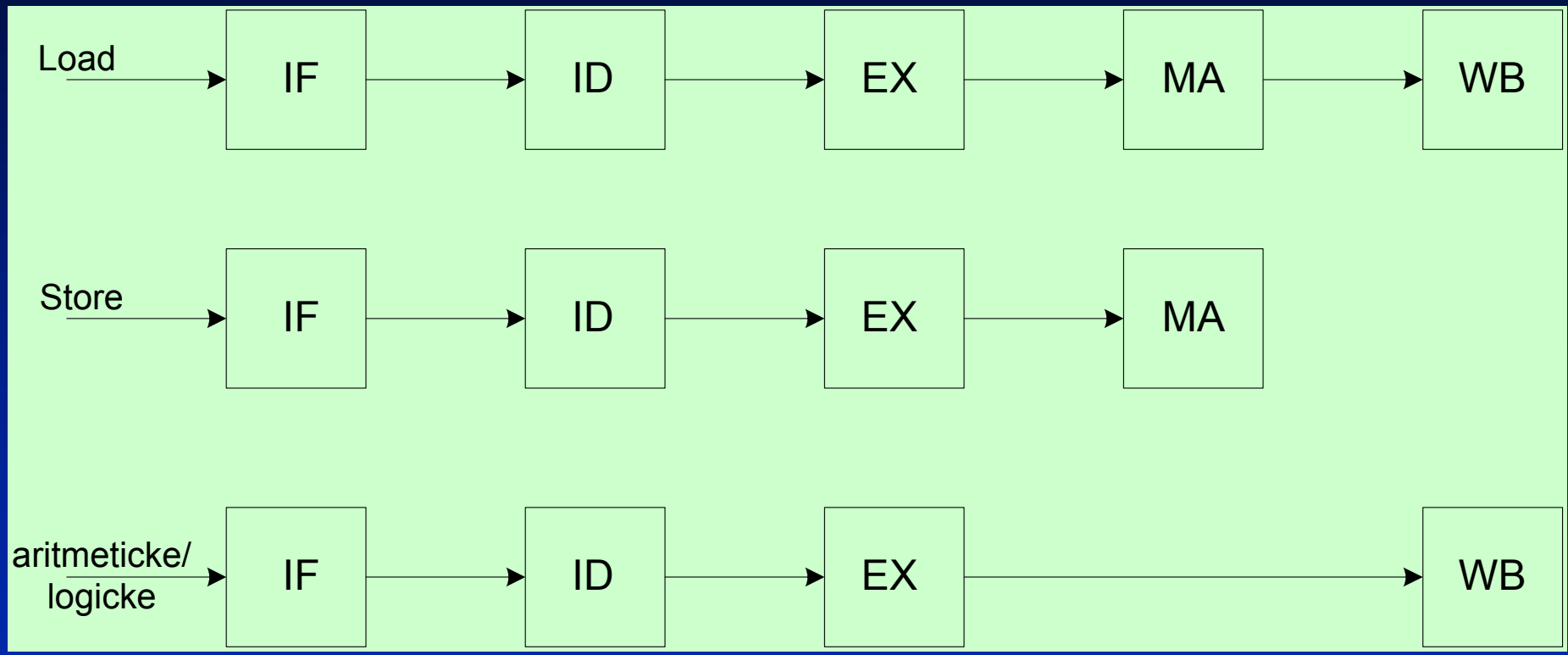
  - S1: FI – pribavljanje instrukcije
  - S2: DI – dekodiranje/pribavaljanje operanada iz *RF* polja
  - S3: EX – izvršenje *ALU* operacije ili izračunavanje efektivne adrese
  - S4: MA – pristup memoriji radi upisa podataka ili čitanje memorijskog operanda
  - S5: WB – upis rezultata operacije u *RF* polje

Peto-stepeni protočno organizovan sistem *FX-5P*.

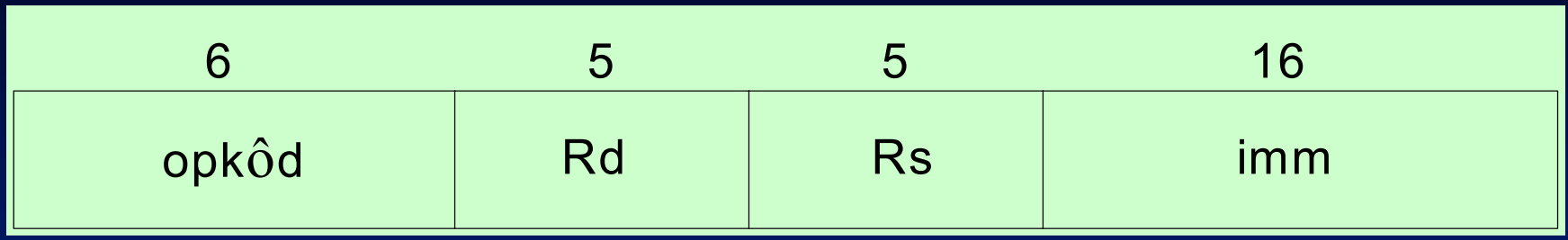
Aktivnosti stepena od S1 do S5



Iskorišćenost stepeni kod *FX-5P* za različite instruckije



Format instrukcije *Load*



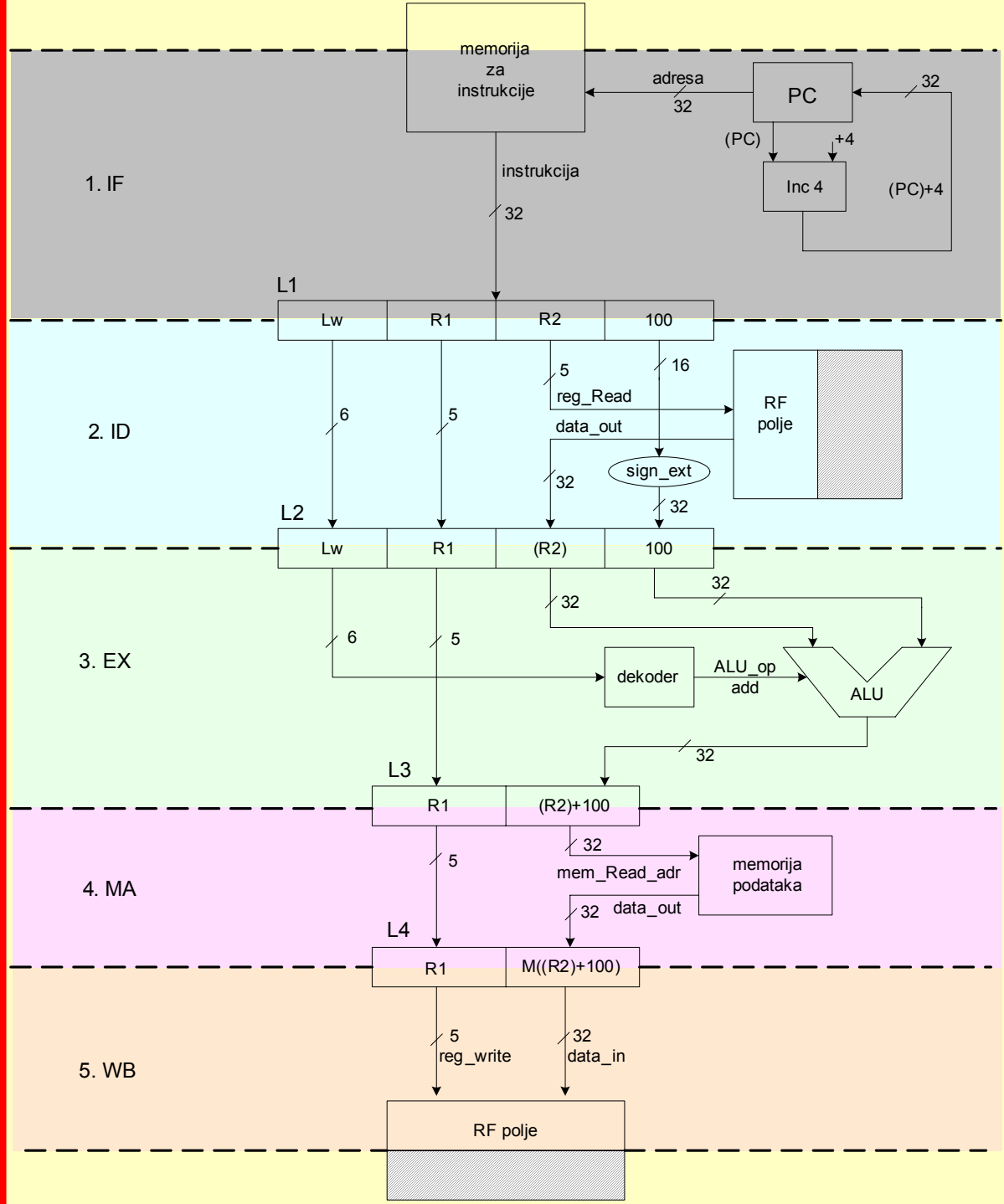
- **Format instrukcije *Load* čine polja:**
  - ***Rd* = *Rs*** **oba obima po 5 bitova (*RF* polje čine 32 registra),**
  - **opkôd polje obima 6 bitova (moguće je kodirati do 64 različite instrukcije)**
  - **polje *imm* veličine 16 bitova definiše konstantu**

Operacija koja se izvršava je tipa

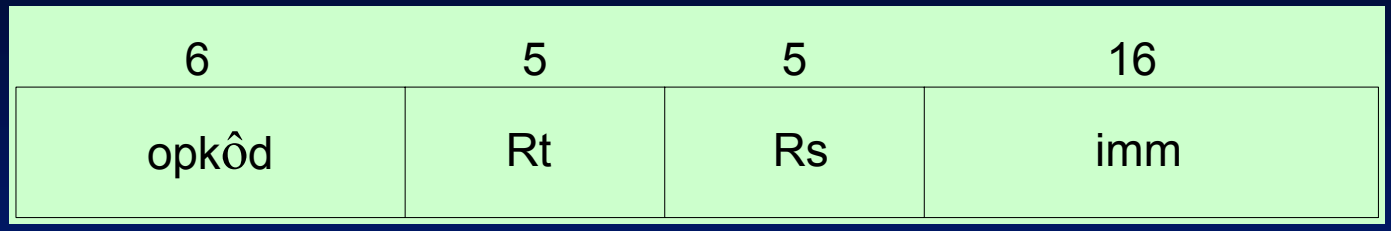
```
Lw   R1,100(R2)      ;  (R1)← M(100+R2)
```

Potrebna struktura FX-5P za *Load*

`Lw  R1,100(R2)`



Format instrukcije *Store*



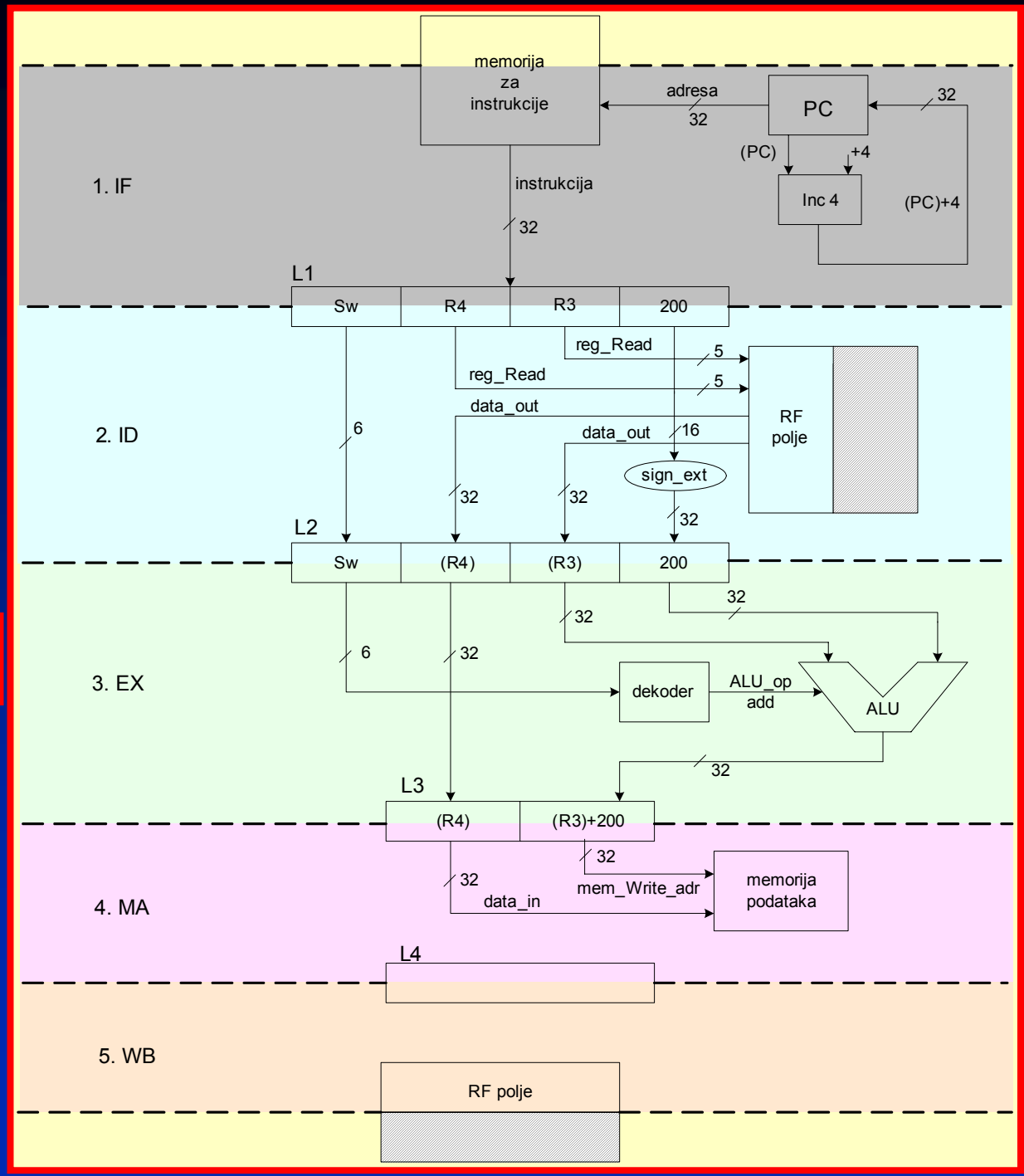
- **Format instrukcije *Store* čine polja:**
  - ***Rd* = *Rs*** **oba obima po 5 bitova (*RF* polje čine 32 registra),**
  - **opkôd polje obima 6 bitova (moguće je kodirati do 64 različite instrukcije)**
  - **polje *imm* veličine 16 bitova definiše konstantu**

Operacija koja se izvršava je tipa

```
Sw R4,200(R3)
```
; (R4)→ M(200+R3)

Potrebna struktura FX-5P za *Store*

Sw R4,200(R3)



Instrukcije tipa Branch

- **Instrukcije grananja (*Branch*) formiraju različitu klasu instrukcija u odnosu na aritmetičko/logičke i *Load*/*Store* instrukcije.**

- **Razlika se sastoji u tome što instrukcije tipa *Branch* upisuju novu vrednost u programski brojač, *PC*.**

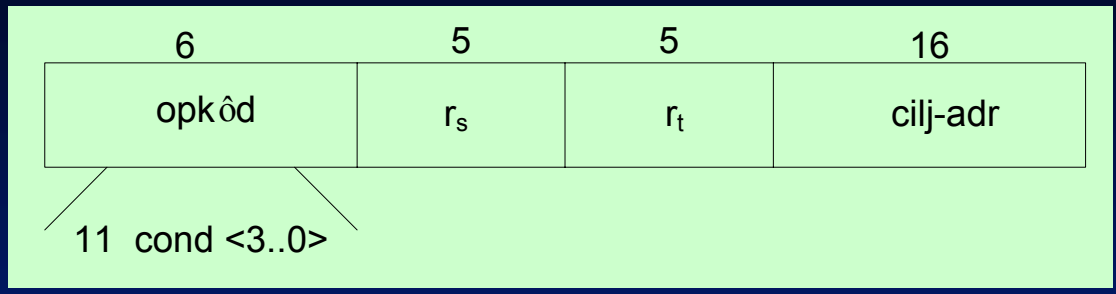
- **Instrukcije grananja mogu biti:**

  - **uslovne, pa ih tada nazivamo *Bcc* (*Branch Conditional*)**

  - **bezulsovne, poznate kao *Jump* (*Jump Unconditional*)**

Format Branch instrukcija

Uslovne i bezuslovne instrukcije *Branch* imaju jedinstveni format



Polja r_s i r_t su registri čije se vrednosti uporedjuju

Polje *cilj_adr* predstavlja ciljnu adresu grananja

Polje *cond* ima značenje

Cond:={ c<3..0>=0 →*0 ; ne granaj se nikad*
c<3..0>=1 →*1 ; branch* (granaj se) uvek na cilj_adr, instrukcija *Jump*
c<3..0>=2 → *; rs – rt = 0; if rs – rt = 0 branch* na cilj_adr
c<3..0>=3 → *; rs – rt≠0; if rs - rt ≠ 0 branch* na cilj_adr
c<3..0>=4 → *; rs – rt > 0; if rs > rt branch* na cil_adr
c<3..0>=5 → *; rs – rt < 0; if rs < rt branch* na cilj_asr

. . . . . .

vrednost uslova *c<3..0>* od 6 do 15 definisane su na slici 9

Spisak instrukcija Branch

Tip instrukcije	Mnemonik	Marker uslova koji se testira	
nikad se ne granaj	*Jnev*	---	*C<3..0>=0*
uvek se granaj (bezuslovni skok)	*Jump*	---	*C<3..0>=1*
granaj se ako je nula	*Bz*	*If {z}=1 go to cilj_adr else next_adr;*	*C<3..0>=2*
granaj se ako je nije nula	*Bnz*	*If {z}=0 go to cilj_adr else next_adr;*	*C<3..0>=3*
granaj se ako je parnost parna	*Bpe*	*If {p}=1 go to cilj_adr else next_adr;*	*C<3..0>=6*
granaj se ako je parnost neparna	*Bpo*	*If {p}=0 go to cilj_adr else next_adr;*	*C<3..0>=7*
granaj se ako postoji prenos	*Bc*	*If {c}=1 go to cilj_adr else next_adr;*	*C<3..0>=8*
granaj se ako ne postoji prenos	*Bnc*	*If {c}=0 go to cilj_adr else next_adr;*	*C<3..0>=9*
granaj se ako je negativan	*Bsn*	*If {s}=1 go to cilj_adr else next_adr;*	*C<3..0>=10*
granaj se ako je pozitivan	*Bsp*	*If {s}=0 go to cilj_adr else next_adr;*	*C<3..0>=11*
granaj se ako ima premasaj	*Bo*	*If {o}=1 go to cilj_adr else next_adr;*	*C<3..0>=12*
granaj se ako nema premašaj	*Bno*	*If {o}=0 go to cilj_adr else next_adr;*	*C<3..0>=13*

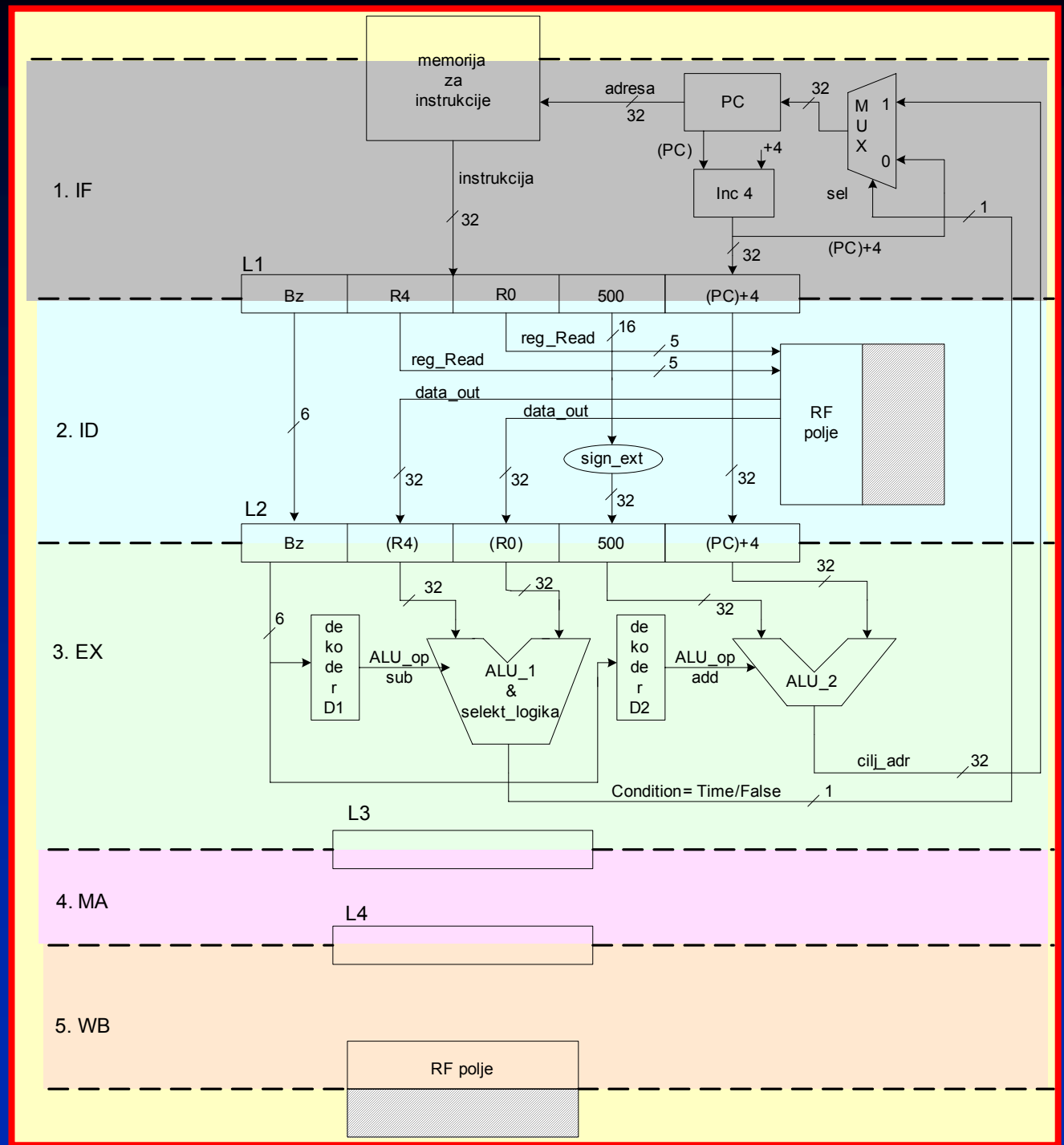
Kombinacije C<3..0>=4 i 5 su rezervisane za *Bgt (branch if greater then), i Bet (branch if less then)*, dok su ostale dve kombinacije ostavljene za buduća proširenja

Potrebna struktura FX-5P za *Branch*

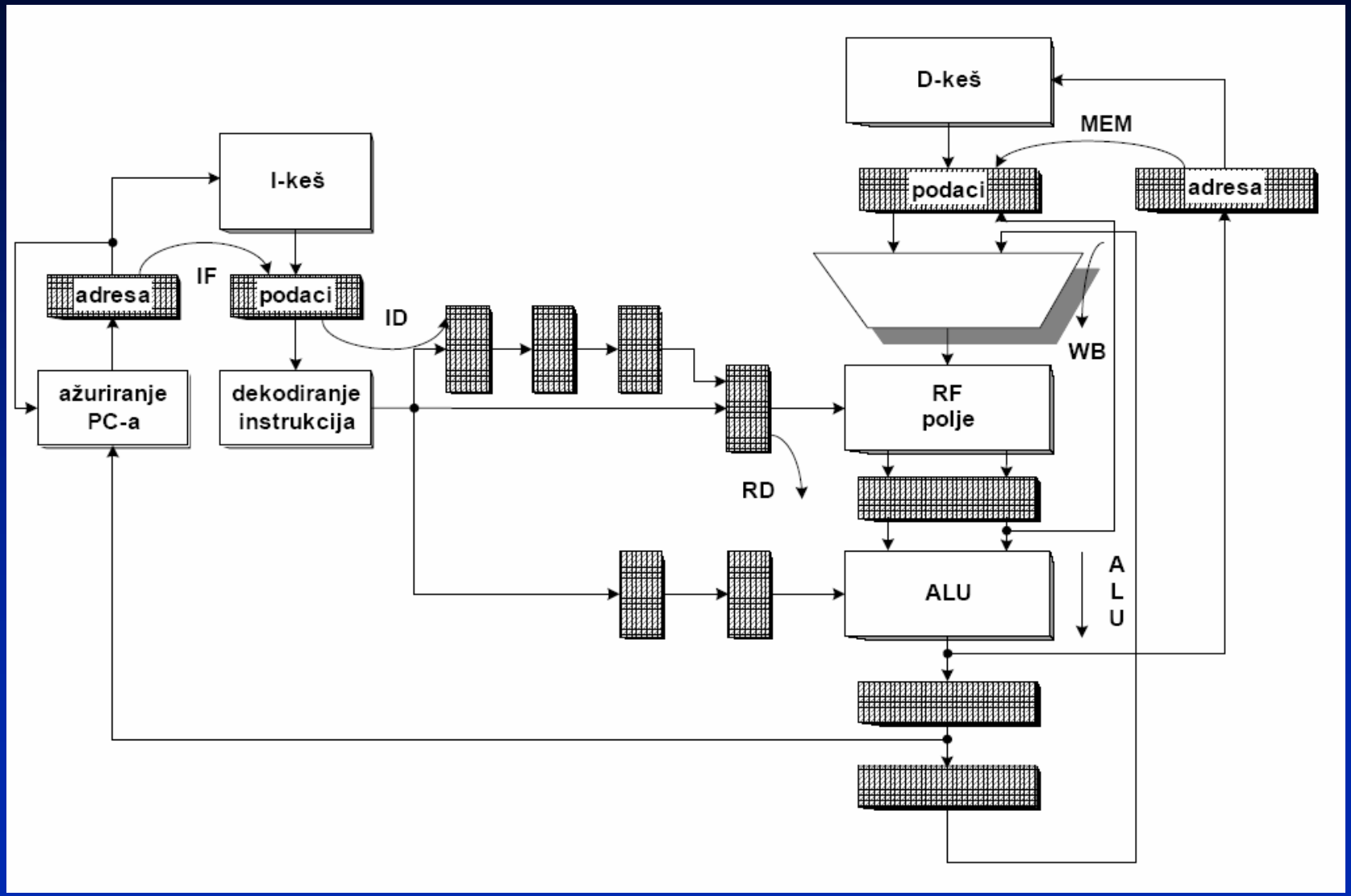
Operacija koja se izvršava je tipa

```
Bz R4,R0,500
```

; *if* R4 - R0 = 0 *go to* (PC) + 4 + 500, *else* *next_adr*



Jedna fizička organizacija protočnog sistema



Zastoji u radu protočnog sistema

- Analizirajmo sekvencu:

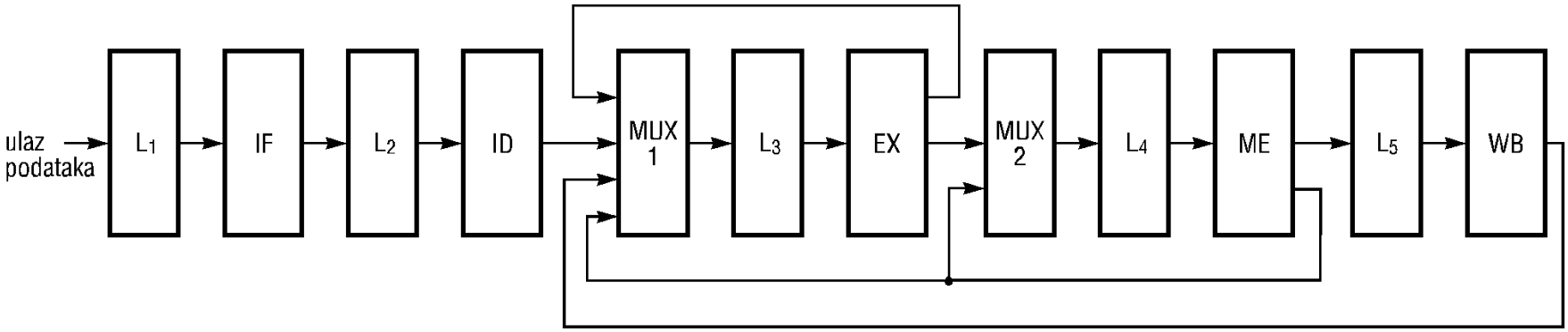
```
Loadf fp1,Mem(r1+r2)
Loadf fp2,Mem(r3+disp)
Mulf fp3,fp1,fp2
Storef Mem(r1+r2),fp3
```

- Vizuelna prezentacija zastoja u radu protočnog sistema

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Loadf fp1, Mem (r1+r2)	IF	ID	EX	ME	WB										
Loadf fp2, Mem (r3+disp)		IF	ID	EX	ME	WB									
Mulf fp3,fp1,fp2			IF	●	●	●	ID	EX	ME	WB					
Storef Mem (r1+r2),fp3							IF	●	●	●	ID	EX	ME	WB	

Rešavanje zastoja uz pomoć premošćavanja

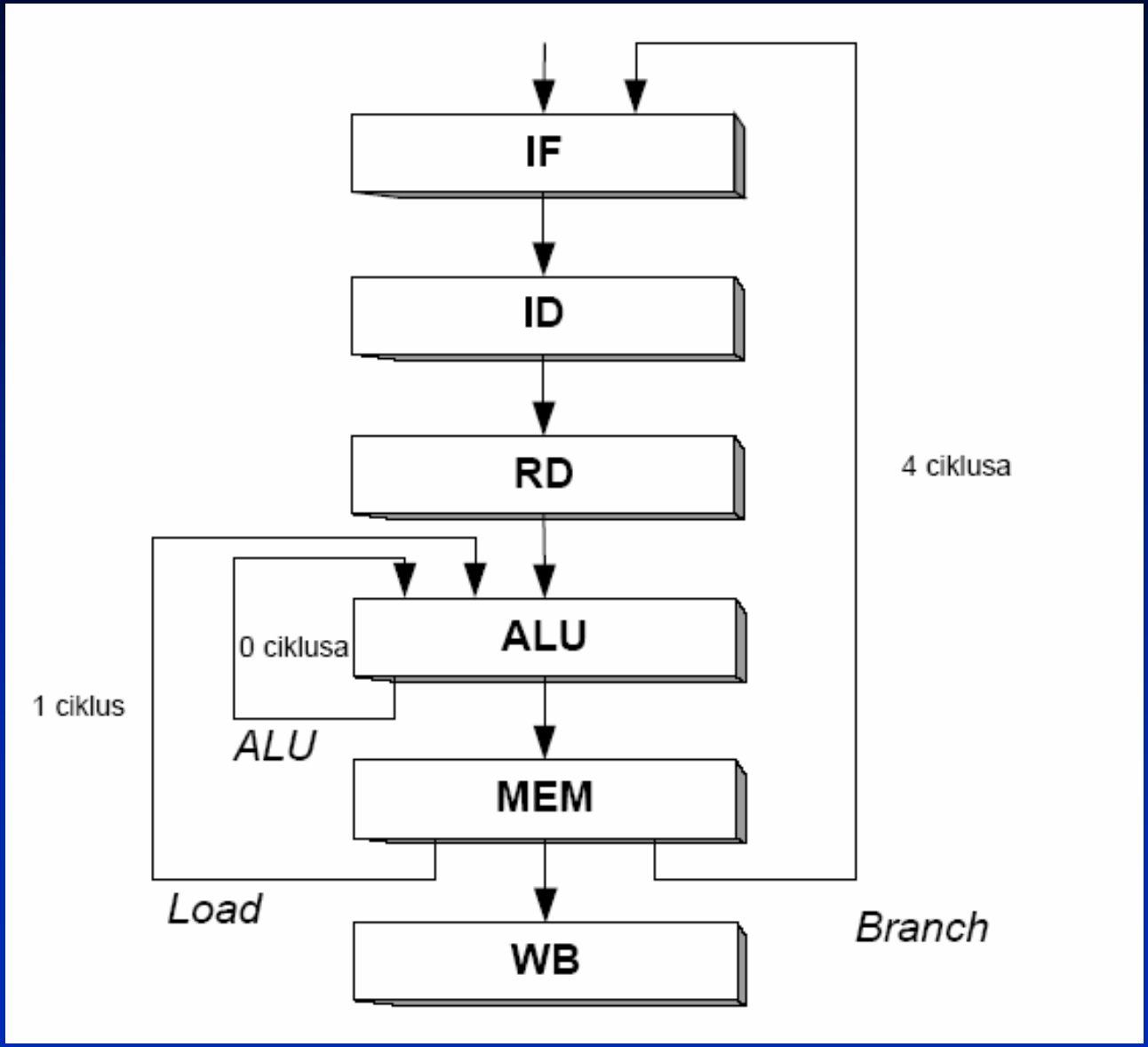
- Protočni sistem sa izvedemnim premošćavanjima:



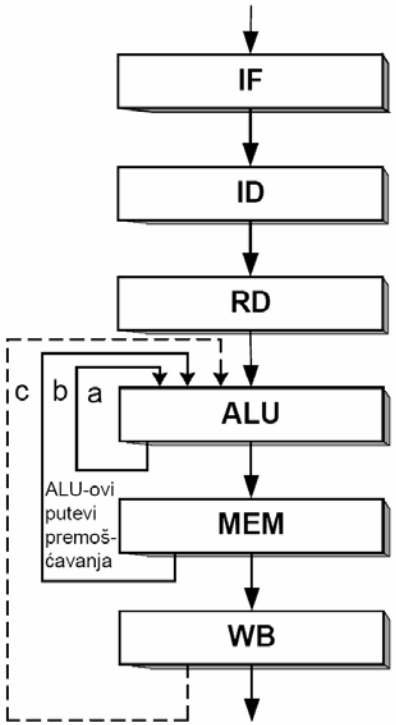
- Tok izvršenja programske sekvence sa izvedenim premošćavanjima

	0	1	2	3	4	5	6	7	8
Loadf fp1, Mem (r1+r2)	IF	ID	EX	ME	WB				
Loadf fp2, Mem (r3+disp)		IF	ID	EX	ME	WB			
Mulf fp3,fp1,fp2			IF	●	ID	EX	ME	WB	
Storef Mem (r1+r2),fp3					IF	ID	EX	ME	WB

Tehnike za ubrzanje rada-premošćavanje da bi se smanjile cene zastoja usled ALU i *Load*

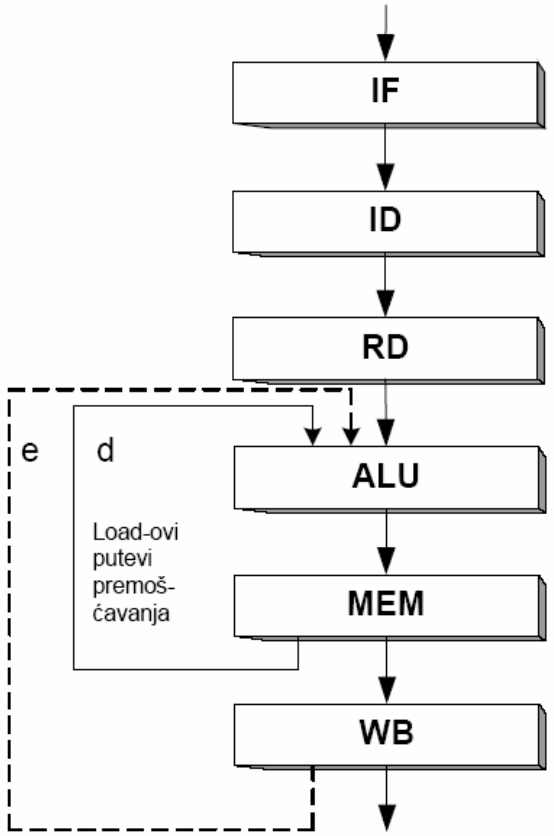


Putevi premošćavanja kada je vodeća instrukcija tipa ALU



$i+1:\ \leftarrow\ R_1$	$i+2:\ \leftarrow\ R_1$	$i+3:\ \leftarrow\ R_1$
$i:\ R_1\leftarrow$	$i+1:\ \leftarrow\ R_1$	$i+2:\ \leftarrow\ R_1$
	$i:\ R_1\leftarrow$	$i+1:\ \leftarrow$
		$i:\ R_1\leftarrow$
$(i \rightarrow i+1)$	$(i \rightarrow i+2)$	$(i \rightarrow i+3)$
premošćavanje preko puta "*a*"	premošćavanje preko puta "*b*"	i upisuje u R_1 pre nego što $i+3$ čita R_1

Putevi premošćavanja kada je vodeća instrukcija tipa *Load*



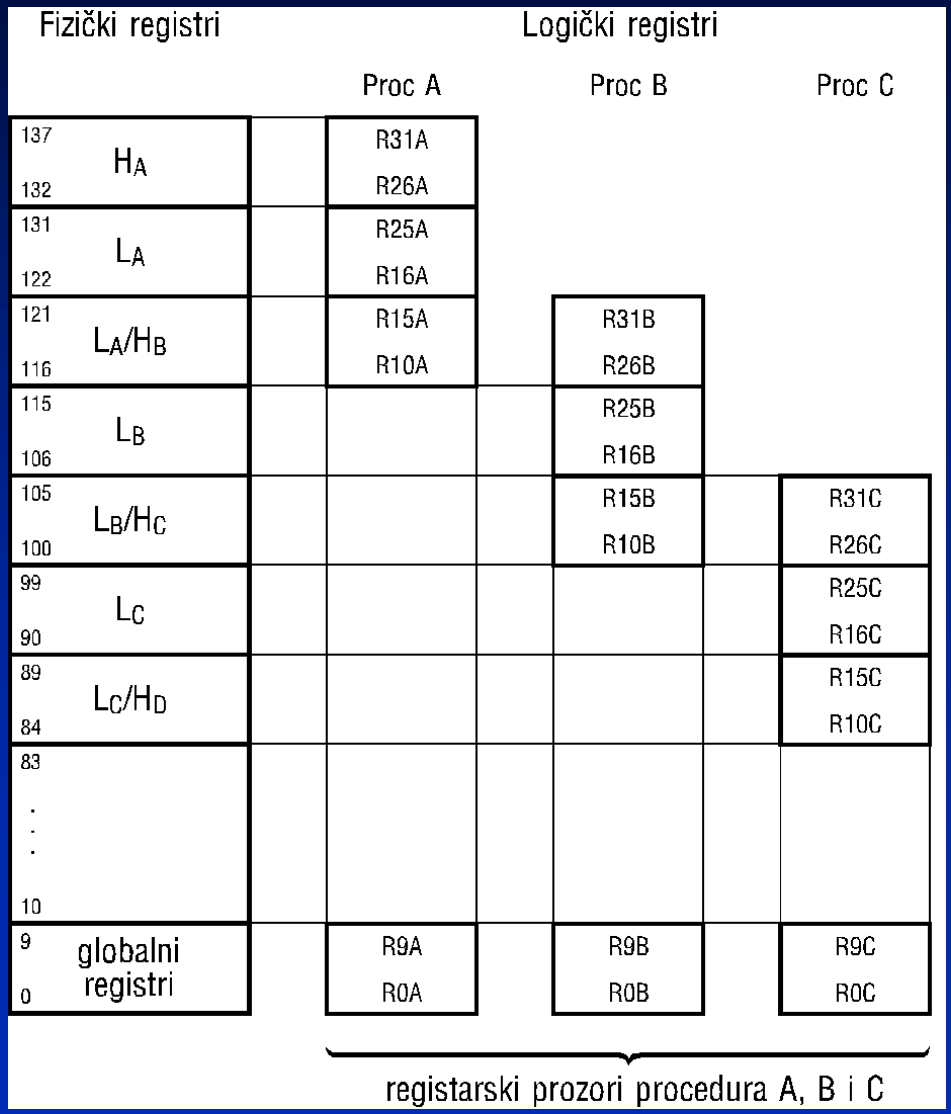
$i+1:\ \leftarrow\ R_1$	$i+1:\ \leftarrow\ R_1$	$i+2:\ \leftarrow\ R_1$
$i:\ R_1 \leftarrow\ MEM[\ ]$		$i+1:\ \leftarrow\ R_1$
	$i:\ R_1 \leftarrow\ MEM[\ ]$	
		$i:\ R_1 \leftarrow\ MEM[\ ]$
$(i \rightarrow i+1)$	$(i \rightarrow i+2)$	$(i \rightarrow i+3)$
zastoj $i+1$	premošćavanje preko puta "d"	i upisuje u R_1 pre nego što $i+2$ čita R_1

Registarski model i rad sa prozorima

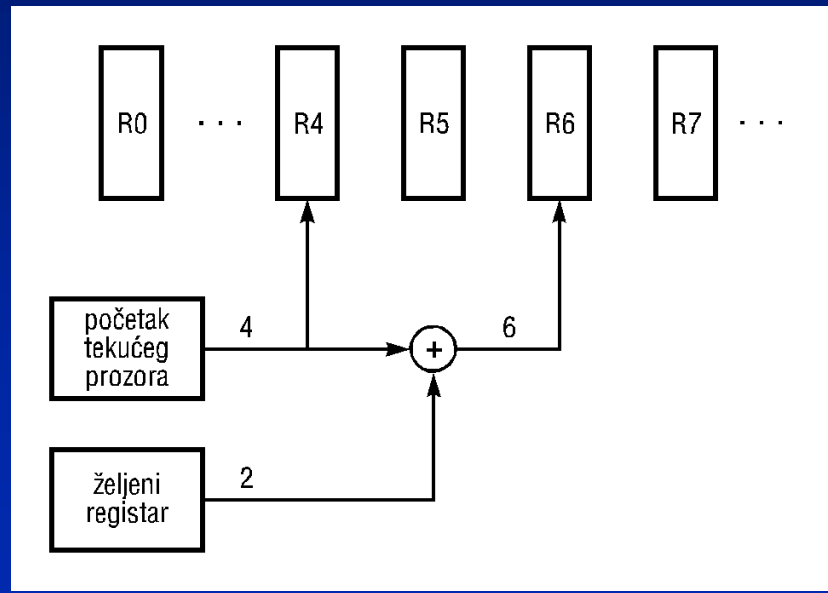
- **Razlikujemo dve hardverske tehnike za rad sa registarskim prozorima:**
  - **Prva se odnosi na rad sa prozorima fiksne dužine (projektant prozora je taj koji definiše broj prozora po banci)**
  - **Druga se odnosi na rad sa prozorima promenljive dužine – softverskim sredstvima se specifira obim banke i trenutak dodele**

Prozori fiksnog i promenljivog obima

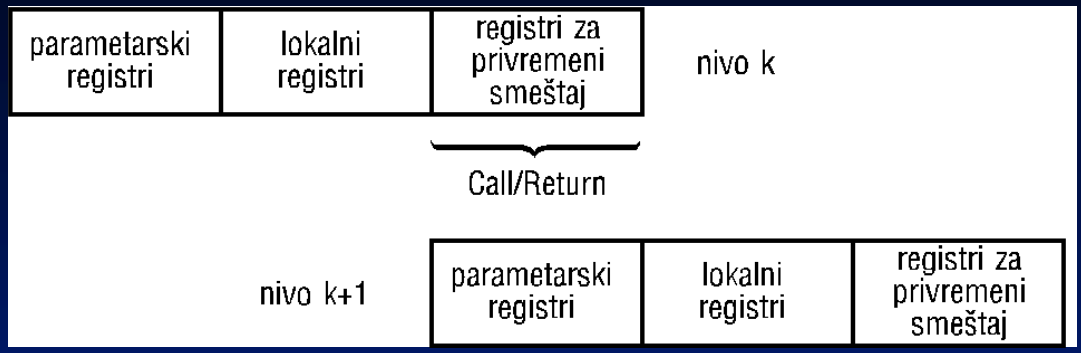
Prozori fiksnog obima kod mikroprocesora Berkeley RISC II



Organizacija registarskog prozora promenljive dužine

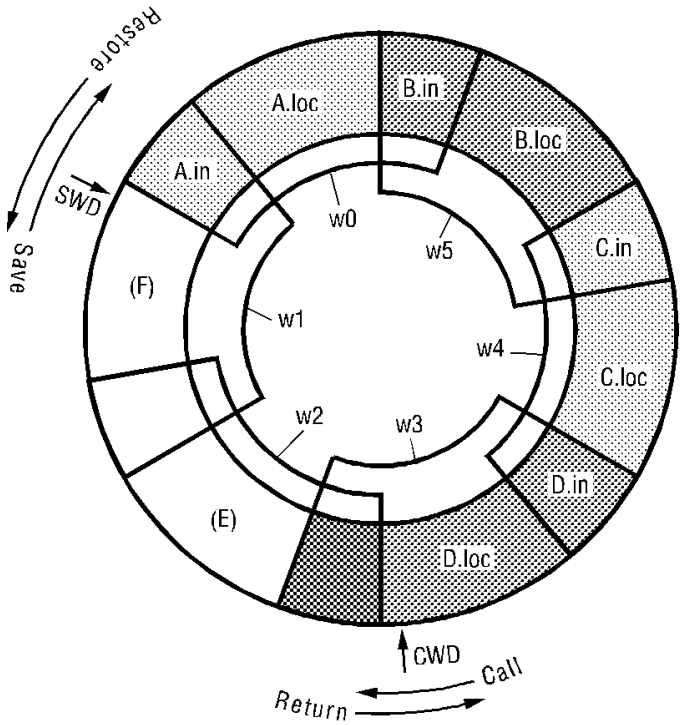


Rad sa prozorima fiksnog obima



Koncept rada sa prozorima fiksnog obima kod mikroprocesora Berkeley RISC II

Kružni bafer sa prozorima koji se preklapaju



Odmotavanje petlji

- Petlje čine značajni deo programskih sekvenci
- Od velike važnosti je eliminisati zastoje koji se javljaju u toku njihovog izvršenja
- U cilju postizanja veće brzine izvršenje programa vrši se odmotavanje petlje (*loop unrooling*)
- Ako se petlja odmota *n* puta dobija se *n* kopija tela petlje , a na kraju programske sekvence postoji instrukcija bezuslovnog grananja *Branch*
- Vrednosti koje su funkcije indeksa petlje, kao što su adresni indeksi, uskladjuju se kod svake od *n* kopija
- Na ovaj način smanjuje se broj grananja, a time se ubrzava izvršenje programa

Jednostavna petlja i odgovarajući mašinski kôd

```
do i=1 to n.
A(i)=B(i)+C(i)
```

a) *jednostavna petlja*

```
        Loadi   r1, #A              /*A, B, C, 0 i n su */
        Loadi   r2, #B              /*neposredni operandi */
        Loadi   r3, #C
        Loadi   r4, #0              /*ekvivalentne instrukcije */
        Loadi   r5, #n
LOOP:   Loadf   fp1, Mem(r1+r4)     /*Loadf fp1, i(r1) */
        Loadf   fp2, Mem(r2+r4)     /*Loadf fp2, i(r2) */
        Addf    fp3, fp1, fp2,
        Storef  Mem(r3+r4), fp3     /*Store 0(r3), fp3 */
        Addi    r4, r4, #1
        CmpLT   C1, r4, r5          /*Compare Less Than i */
                                    /*postavi marker uslova C1*/
        BC      LOOP.C1             /*uslovno grananje  */
                                    /*ako je C1=True */
```

b) *mašinski kôd jednostavne petlje za slučaj n ≥ 1*

Odmotavanje petlje – slučaj 1

Slučaj kada jednu grupu čine dve iteracije, *n* je parno i važi uslov $n \geq 2$

```
        Loadi   r1, #A
        Loadi   r2, #B
        Loadi   r3, #C
        Loadi   r4, #0
        Loadi   r5, #n
LOOP:   Loadf   fp1, Mem(r1+r4)
        Loadf   fp2, Mem(r2+r4)
        Addf    fp3, fp1, fp2
        Storef  Mem(r3+r4), fp3
        Addi    r4, r4, #1
        Loadf   fp1, Mem(r1+r4)
        Loadf   fp2, Mem(r2+r4)
        Addf    fp3, fp1, fp2
        Storef  Mem(r3+r4), fp3
        Addi    r4, r4, #1
        CmpLT   C1, r4, r5
        BC      LOOP,C1
```

Odmotavanje petlje – slučaj 2

Odmotavanje petlje korišćenjem različitih radnih registara u svakoj kopiji za slučaj, *n* parno i važi uslov $n \geq 2$

```
        Loadi   r1, #A
        Loadi   r2, #B
        Loadi   r3, #C
        Loadi   r4, #0
        Loadi   r5, #n
        Loadi   r6, #1          /*početni indeks druge kopije*/
LOOP:   Loadf   fp1, Mem(r1+r4)
        Loadf   fp2, Mem(r2+r4)
        Loadf   fp4, Mem(r1+r6)
        Loadf   fp5, Mem(r2+r6)
        Addf    fp3, fp1, fp2
        Addf    fp6, fp4, fp5
        Storef  Mem(r3+r4), fp3
        Store   Mem(r3+r6), fp6
        Addi    r4, r4, #2
        Addi    r6, r6, #2
        CmpLT C1, r4, r5
        BC      LOOP.C1
```

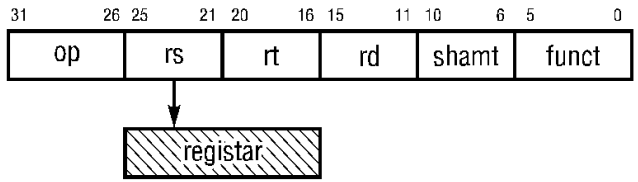
MIPS procesor

- **MIPS xx je jedan od najpoznatijih 32-bitnih RISC procesora koji se karateriše sledećim osobinama**

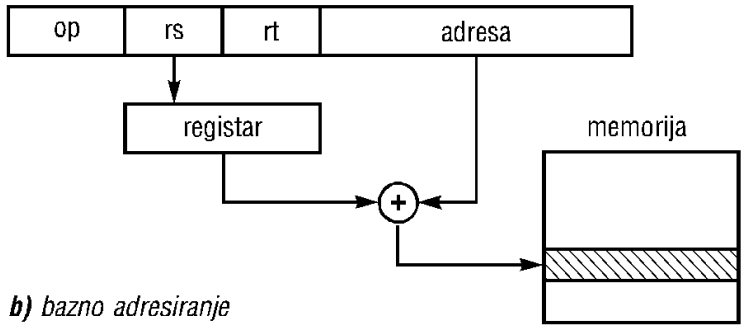
  - **CPU poseduje tridesetdva 32-bitna registra opšte namene R0 do R31 koji čine RF polje. U R0 se uvek čuva konstanta 0**
  - **Mmeorija je bajt adresibilna, koriste se BE način rada sa 32-bitnim adresama**
  - **Sve instrukcije su 32-bitne i poravnjane su**
  - **Novije generacije MIPS procesora imaju i tridesetdva 32-bitna FP registra**

Adresni načini rada kod MIPS-a

Samo četiri adresna načina rada

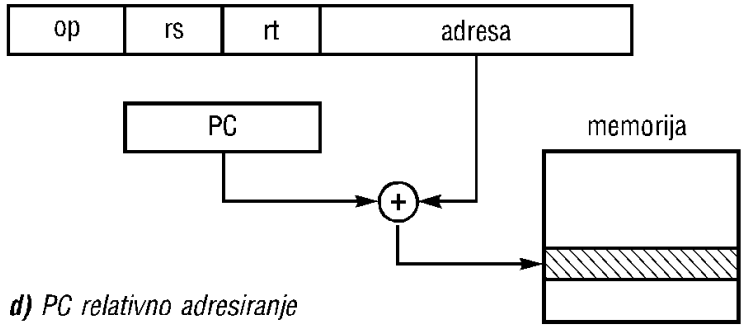


a) registarsko adresiranje



b) bazno adresiranje

c) neposredno adresiranje

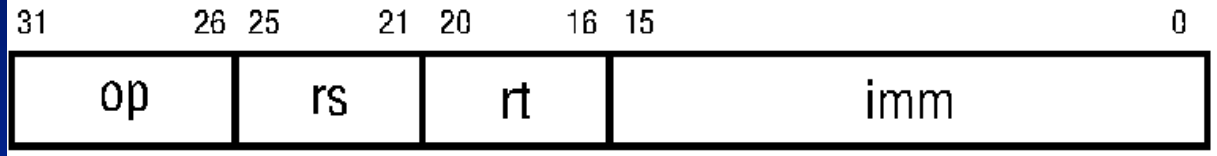


d) PC relativno adresiranje

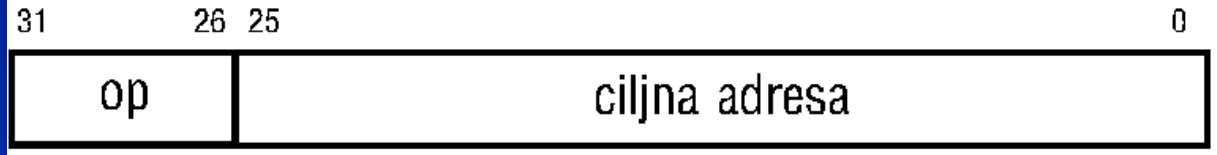
Formati instrukcija kod MIPS-a

Samo tri formata instrukcija

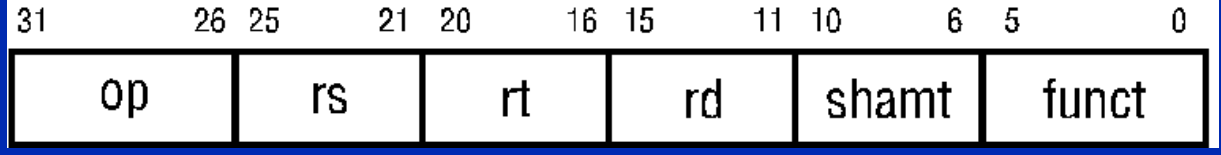
I – tip *(neposredni – immediate)*



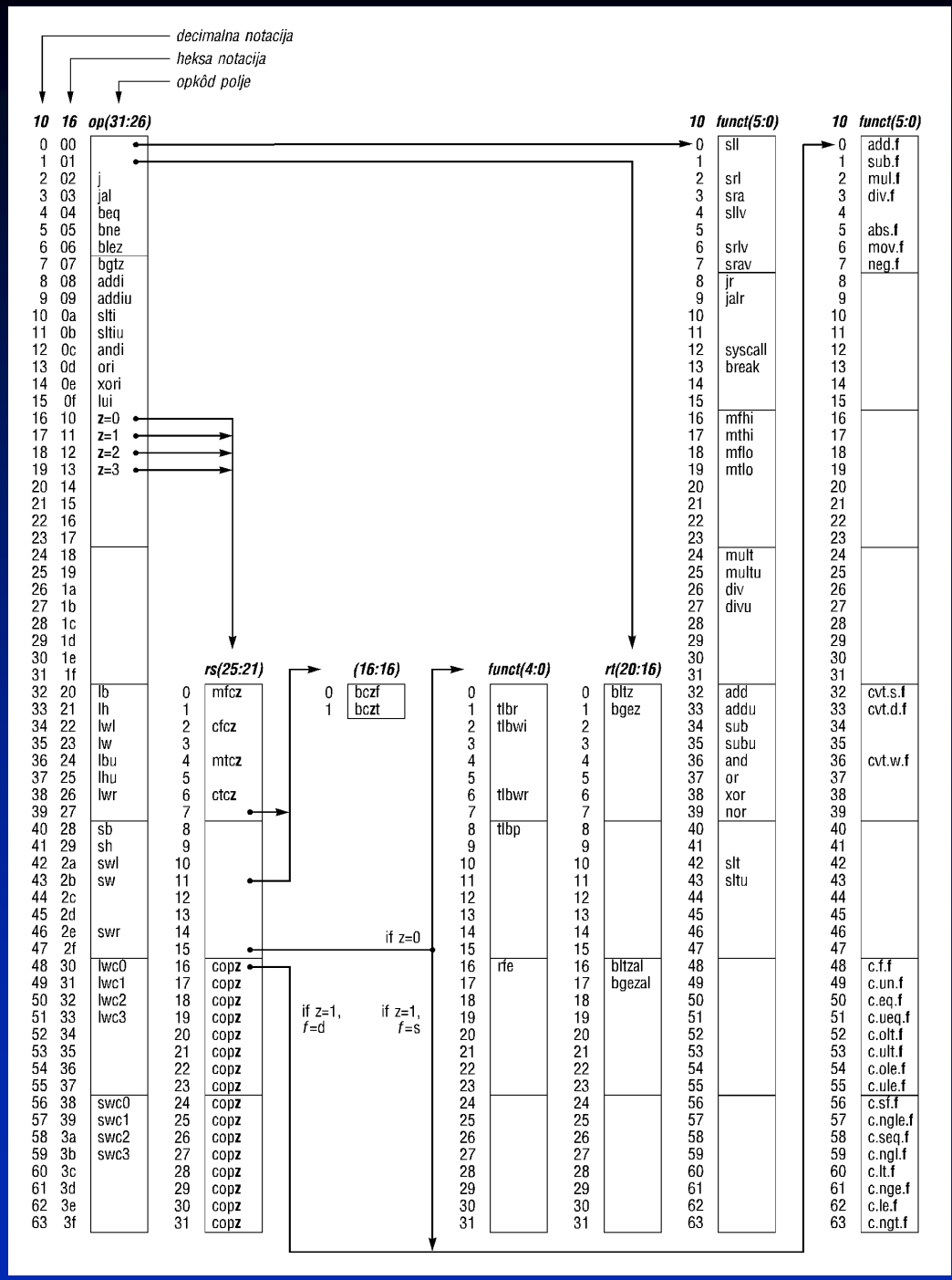
J – tip *(grananje – jump)*



R – tip *(registarski – register)*



Kodiranje naredbi kod MIPS-a



Zadatak 1.

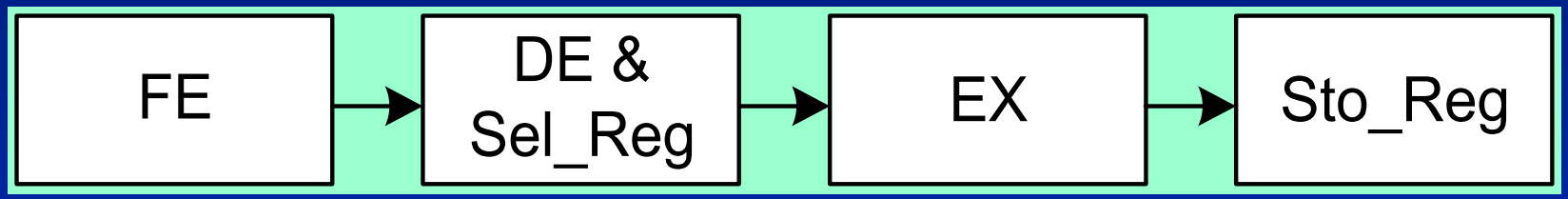
- Slede}i programski segment se izvr{ava na tro-stepenom proto~no organizovanom RISC procesoru sa slike.

```
1:      R1←3
2:      R2←R2+R3
3:      R3←R3+R4
4:      R4←R1+R2
5:      R1←R1-1
6:      if (R10≠0) then go to 2
7:      R5←R6+R7
8:      R6←R7+R8
```

Pokazati kona~nu verziju programa i trag izvr{enja ako se konflikti zbog grananja re{avaju:

- ubacivanjem *Nop* operacija
- preuredjenjem redosleda izvr{enja instrukcija
- ubacivanjem zastoja.

- Ponoviti prethodne aktivnosti ako RISC procesor koristi ~etvorostepenu proto~nu obradu.



Tro- i četvoro-stepeno organizovan RISC procesor

Odgovor:

a) i)

1	FE	DS	ES	R1←3								
2		FE	DS	ES	R2←R2+R3							
3			FE	DS	ES	R3←R3+R4						
4				FE	DS	ES	R4←R1+R2					
5					FE	DS	ES	R1←R1-1				
6						FE	DS	ES	if (R10≠0) then go to 2			
6'							FE	DS	ES	Nop		
6''								FE	DS	ES	Nop	
2/7									FE	DS	ES	R2←R2+R3/R5←R6+R7

ii)

1	FE	DS	ES	R1←3					
2		FE	DS	ES	R2←R2+R3				
3			FE	DS	ES	R3←R3+R4			
6				FE	DS	ES	if (R10≠0) then go to 2		
4					FE	DS	ES	R4←R1+R2	
5						FE	DS	ES	R1←R1-1
7							FE	DS	ES	R5←R6+R7
8								FE	DS	ES	R6←R7+R8

iii)

1	FE	DS	ES	R1←3							
2		FE	DS	ES	R2←R2+R3						
3			FE	DS	ES	R3←R3+R4					
4				FE	DS	ES	R4←R1+R2				
5					FE	DS	ES	R1←R1-1			
6						FE	DS	ES	if (R10≠0) then goto 2		
2/7							FE	*	FE	DS	ES
3/8										FE	DS	ES

Odgovor:

b) i)

1	FE	DS	EX	SR	R1←3							
2		FE	DS	EX	SR	R2←R2+R3						
3			FE	DS	EX	SR	R3←R3+R4					
4				FE	DS	EX	SR	R4←R1+R2				
5					FE	DS	EX	SR	R1←R1-1			
6						FE	DS	EX	SR	if (R10≠0) then go to 2		
6'							FE	DS	EX	SR	Nop	
6"								FE	DS	EX	SR	Nop
2/7									FE	DS	EX	SR

ii)

1	FE	DS	EX	SR	R1←3							
2		FE	DS	EX	SR	R2←R2+R3						
3			FE	DS	EX	SR	R3←R3+R4					
6				FE	DS	EX	SR	if (R10≠0) then go to 2				
4					FE	DS	EX	SR	R4←R1+R2			
5						FE	DS	EX	SR	R1←R1-1		
7							FE	DS	EX	SR	R5←R6+R7	
8								FE	DS	EX	SR	R6←R7+R8

iii)

1	FE	DS	EX	SR	R1←3							
2		FE	DS	EX	SR	R2←R2+R3						
3			FE	DS	EX	SR	R3←R3+R4					
4				FE	DS	EX	SR	R4←R1+R2				
5					FE	DS	EX	SR	R1←R1-1			
6						FE	DS	EX	SR	if (R10≠0) then goto 2		
2/7							FE	*	FE	DS	EX	SR
3/8										FE	DS	EX	SR

Zadatak 2.

Registarsko zasnovana arhitektura izvr{ava sekvencu koju ~ine devet instrukcija. Za svaku instrukciju defini{u se dva izvorna registarska operanda i jedan odredi{ni registarski operand. Funkcije koje obavljaju operacije (instrukcije) su F1, F2 i F3. Registar procesora je ozna~en sa *Ri*, *i*=1,...6.

instrukcija	operacija
I1	R1 ← F1(R2,R3)
I2	R5 ← F3(R1,R2)
I3	R4 ← F2(R2,R2)
I4	R3 ← F1(R1,R4)
I5	R6 ← F1(R5,R6)
I6	R1 ← F1(R2,R3)
I7	R4 ← F2(R2,R5)
I8	R3 ← F1(R1,R4)
I9	R5 ← F3(R5,R5)

Izlistati sve zavisnosti-po-podacima koje se javljaju izmedju instrukcija u toku izvr{enja ove programske sekvence.

Odgovor:

registar	*RAW*	*WAR*	*WAW*
R1	I1-I2, I1-I4, I1-I8, I6-I8	I2-I6, I4-I6	I1-I6
R2			
R3	I4-I6	I1-I4, I1-I8, I6-I8	I4-I8
R4	I3-I4, I3-I8, I7-I8	I4-I7	I3-I7
R5	I2-I5, I2-I7, I2-I9	I5-I9, I7-I9	I2-I9
R6			

Zadatak 3.

Na slici prikazana je kôdna sekvenca koja se izvr{ava od strane staze podataka koja ima samo jedan ALU.

- **Bez promene redosleda operacija u kôdnoj sekvenci odrediti minimalan broj registara koji je potreban da se ona izvr{i.**
- **Ako se dozvoli preuredjenje u redosledu izvr{enja odrediti minimalan broj registara.**

Kôdna sekvenca je oblika

$$V1 = V2 + V3$$
$$V4 = V2 - V3$$
$$V5 = V1 * V2$$
$$V6 = V4 \text{ and } V3$$
$$V7 = V5 \text{ or } V6$$

Odgovor:

Postoji sloboda u preuredjenju operacija kôdne sekvence sve dok su ograni~enja koja se odnose na prethodjenje operacija kao i ograni~enja u odnosu na broj procesnih jedinica zadovoljene.

- **Kôdna sekvenca koja ne koristi preuredjenje redosleda instrukcija prikazana je na slici.**

```
R1 = R2 + R3          ; ukupno je potrebno 4 registara
R4 = R2 - R3
R1 = R1 * R2
R4 = R4 and R3
R4 = R1 or R4
```

Dodela registara bez preuredjenja

Odgovor-prod.

- **Kada se dozvoli preuredjenje oblik kôdne sekvence je prikazan na slici.**

```
R1 = R2 + R3          ; ukupno je potrebno 3 registra
R1 = R1 * R2
R2 = R2 - R3
R2 = R2 and R3
R3 = R1 or R2
```

Dodela registara sa preuredjenjem

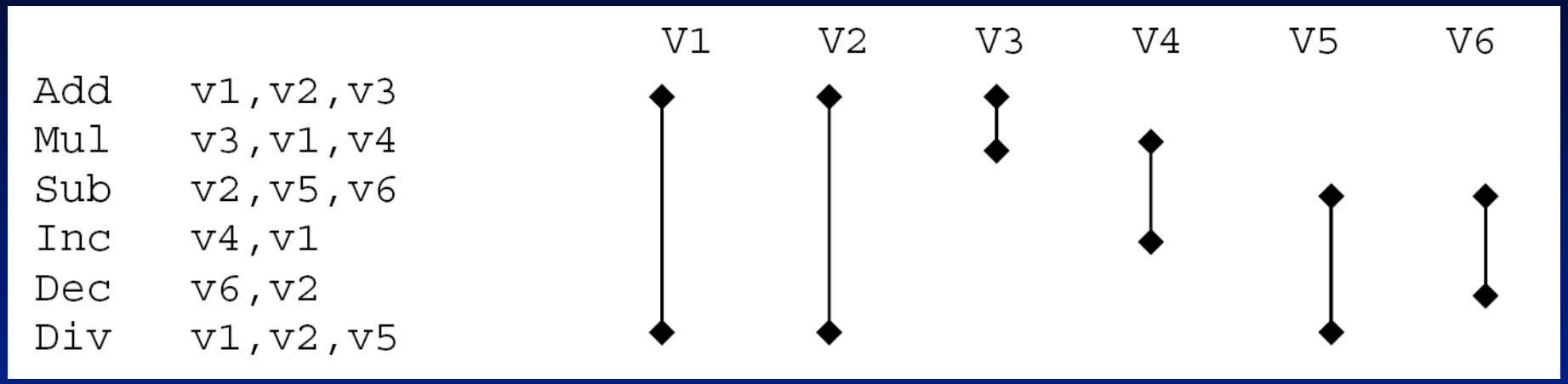
Zadatak 4.

Za kôdnu sekvencu sa slike odrediti vremena `ivota svih promenljivih.

***Napomena*: Vreme `ivota jedne promenljive se defini{e kao vremenski period izmedju njene dodele i zadnjeg kori{}enja. Promenljiva se dodeljuje registru.**

```
Add   v1,v2,v3
Mul   v3,v1,v4
Sub   v2,v5,v6
Inc   v4,v1          ; inkrementiraj v4 za v1
Dec   v6,v2          ; dekrementiraj v6 za v2
Div   v1,v2,v5
```

Odgovor:

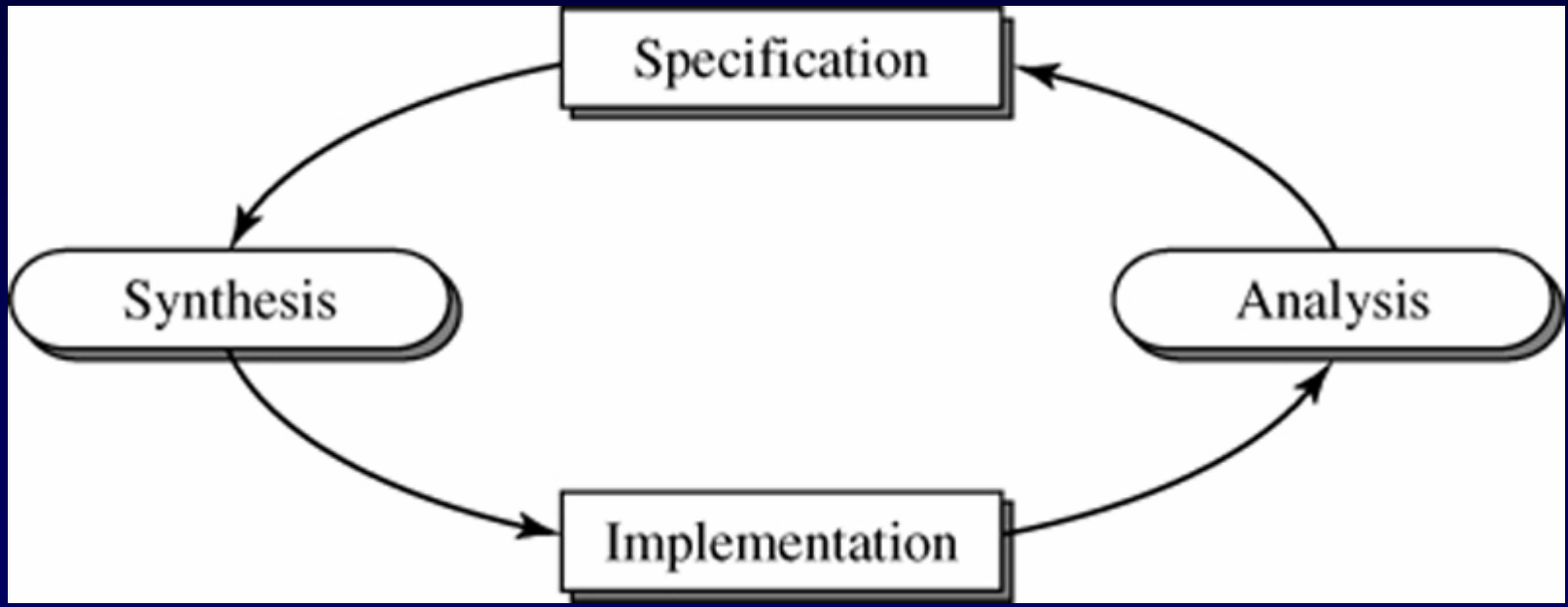


Vreme života promenljivih

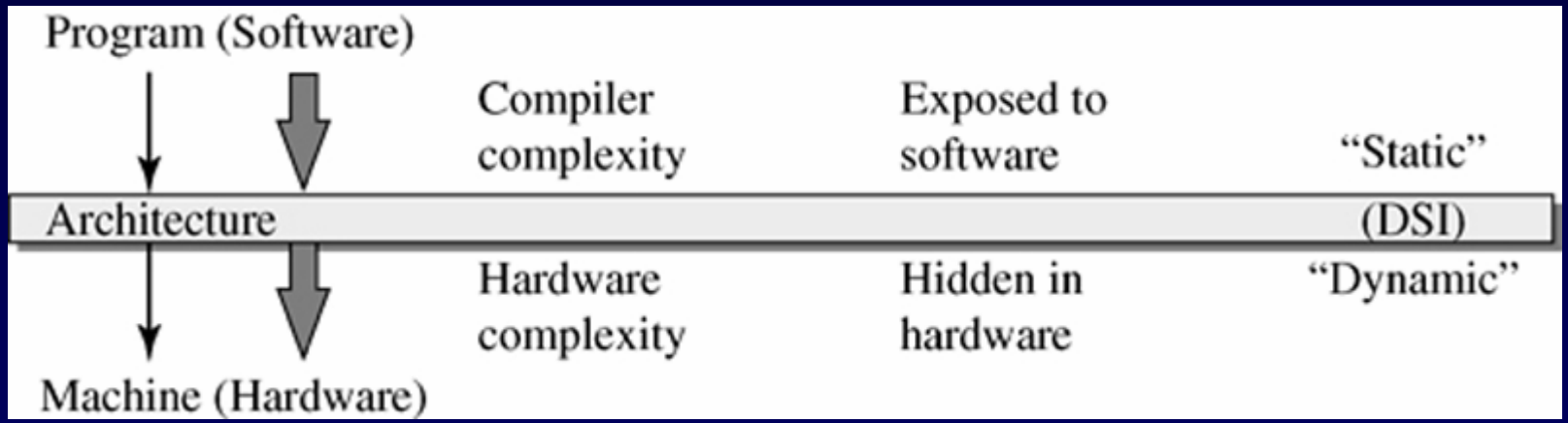
Disjoint **promenljive su one ~ija se vremena `ivota ne preklapaju. Dodela (alokacija) registara promenljivama se svodi na nala`enje najboljeg mogu}eg grupisanja *disjoint* promenljivih u skupovima tako da se minimiziraju ti skupovi.**

Modern processor design

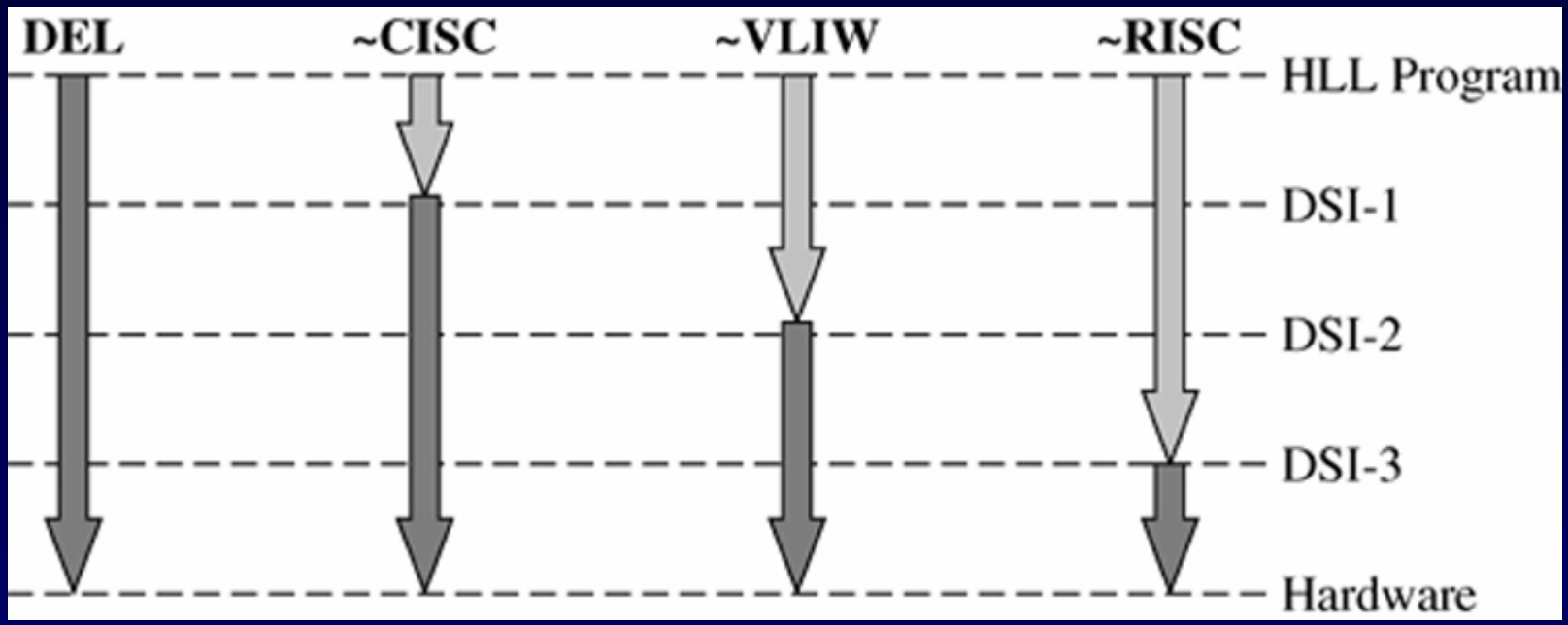
Engineering design



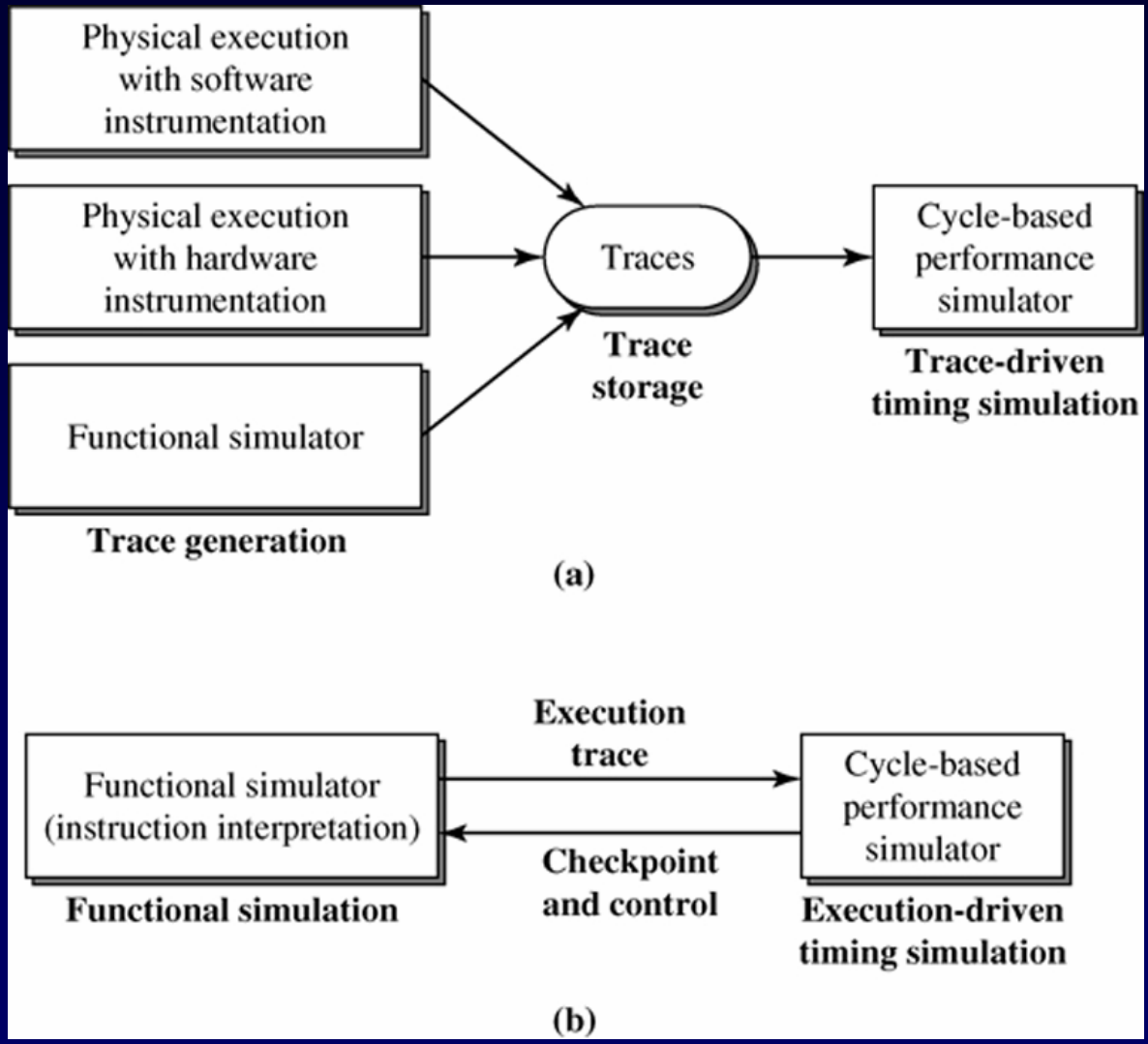
The dynamic-static interface (DSI)



Conceptuall illustration of possible placement of DSI in ISA design



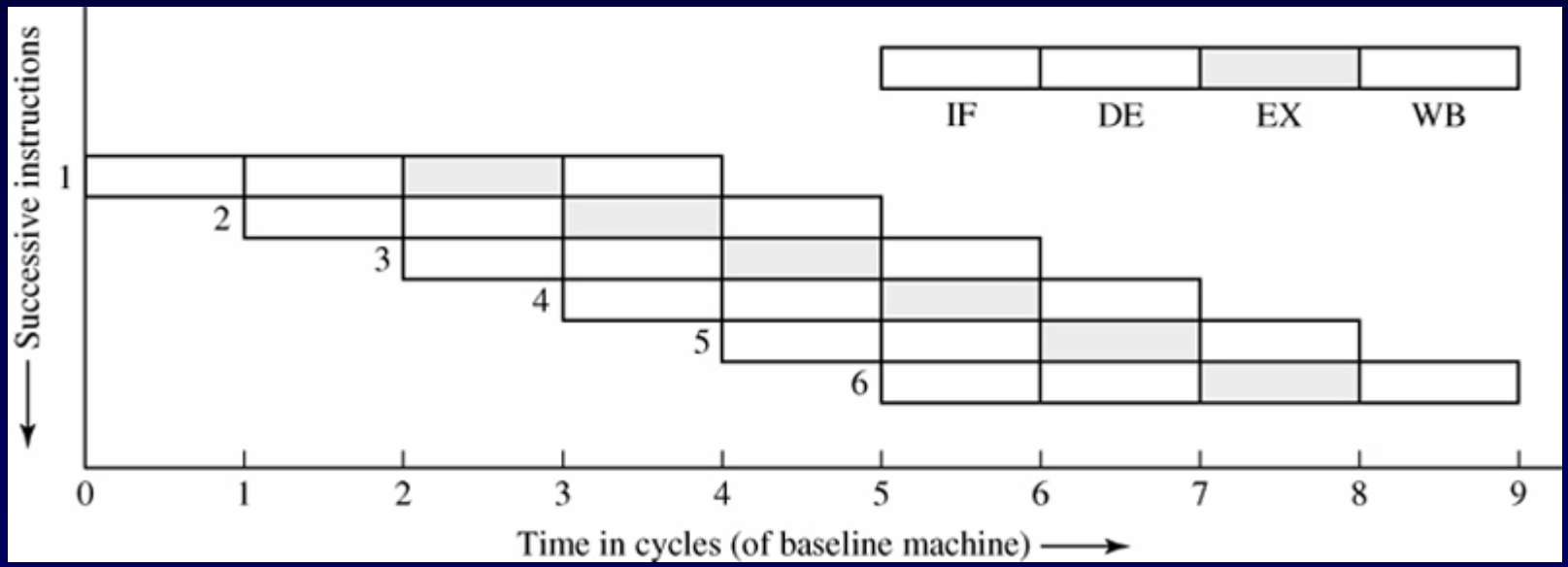
Performance simulation metods



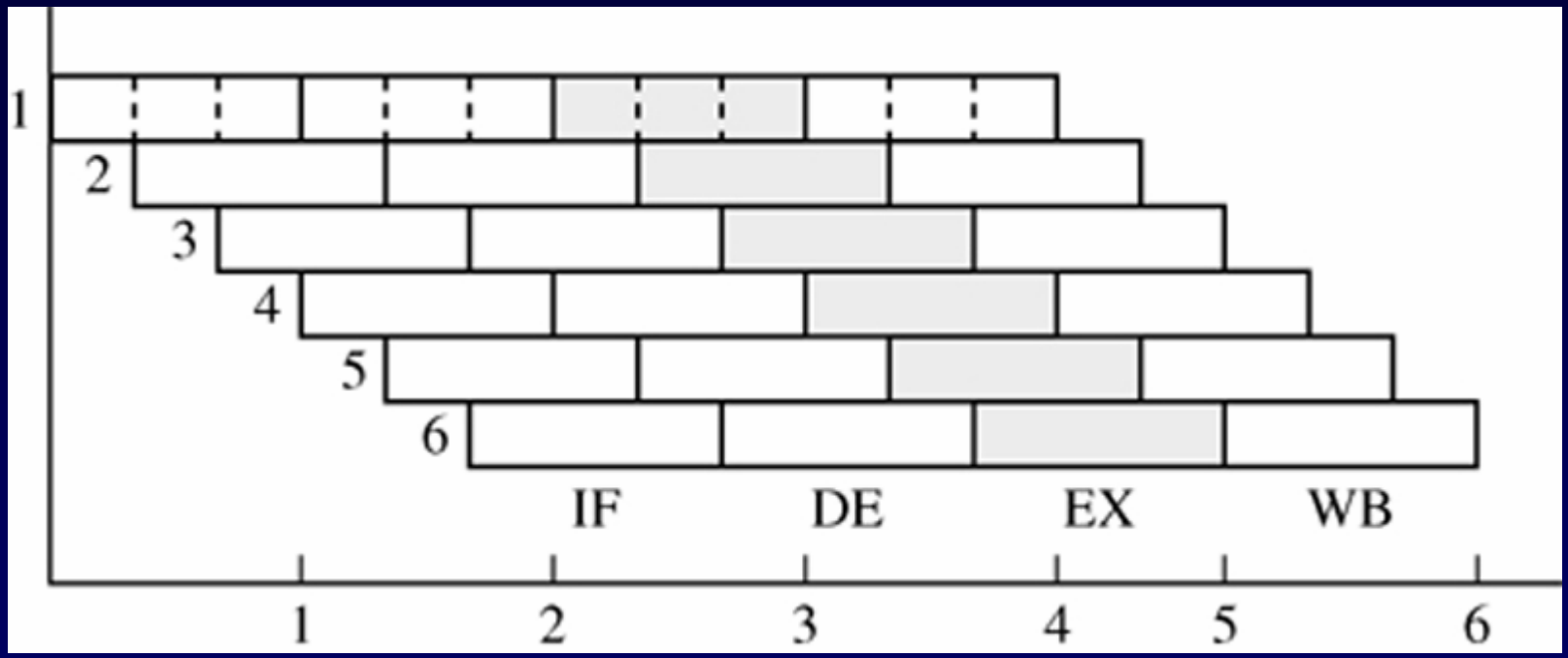
Trace-driven simulation

Execution-driven simulation

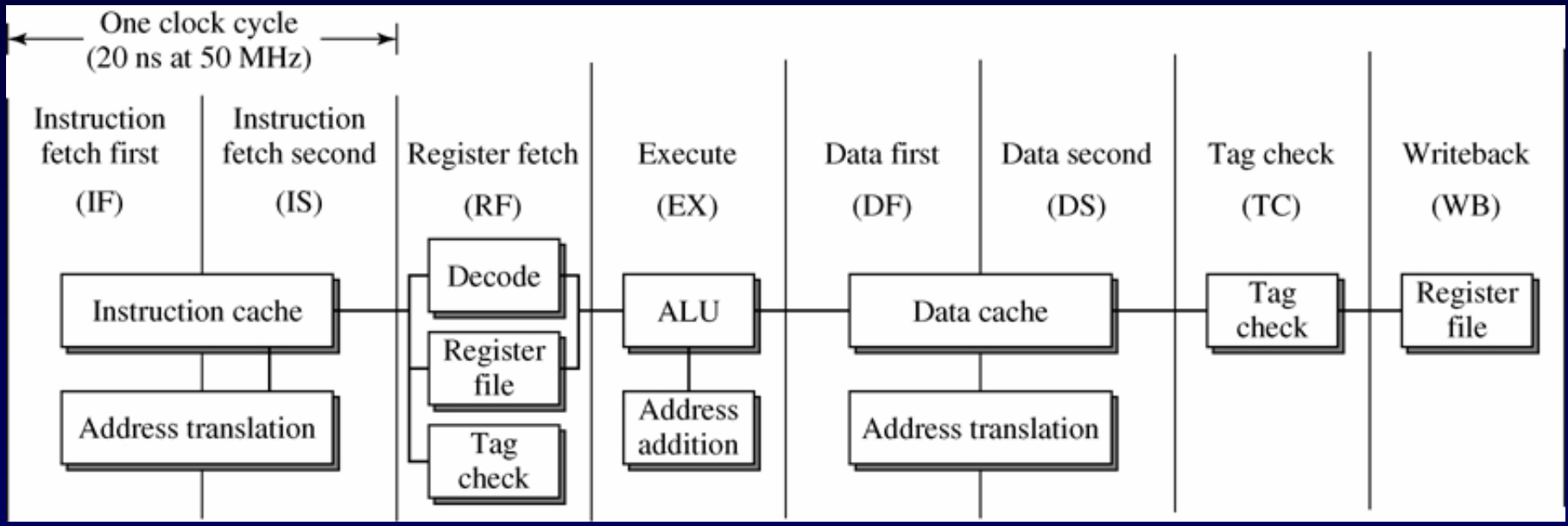
Scalar pipeline machine



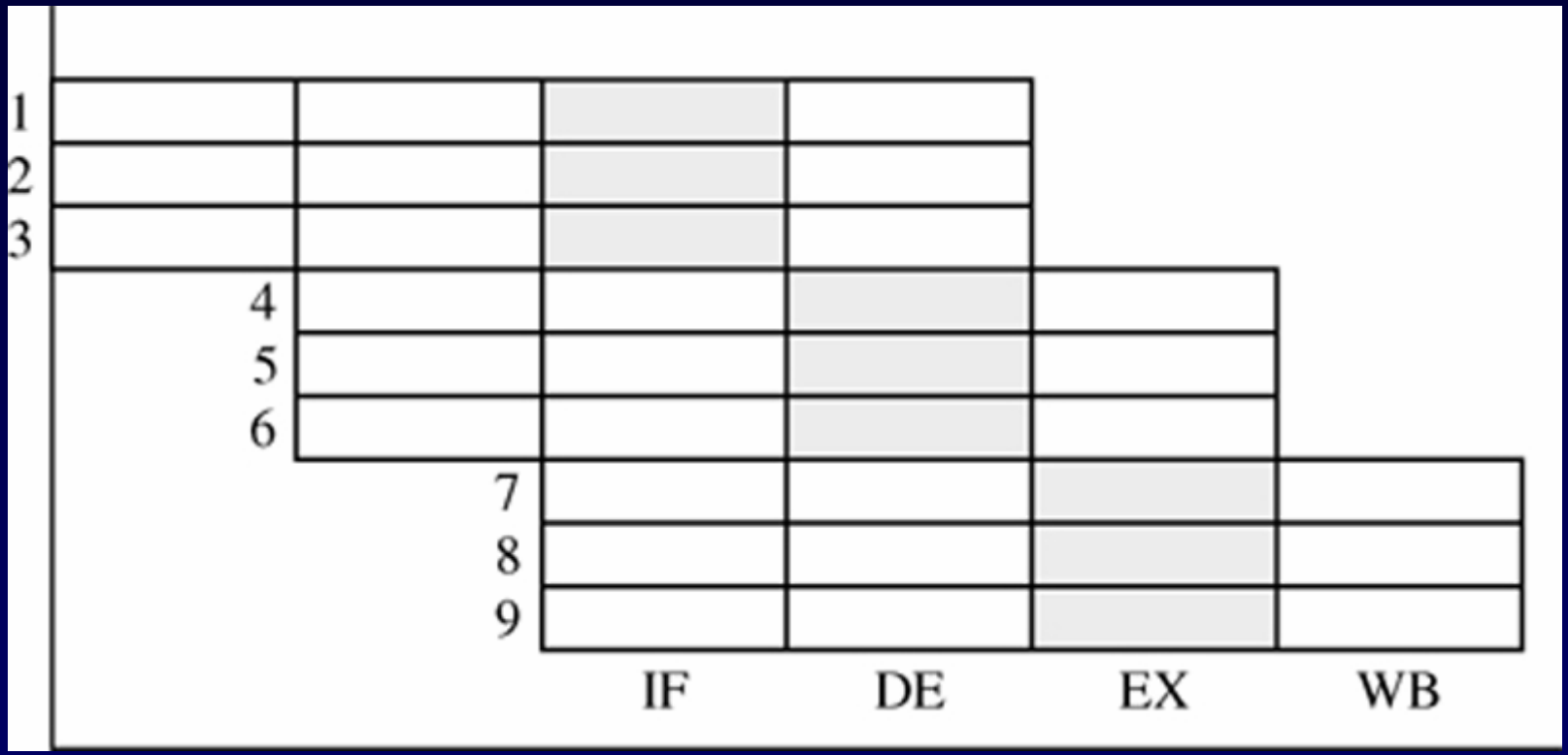
Superpipelined machine of degree *m*=3



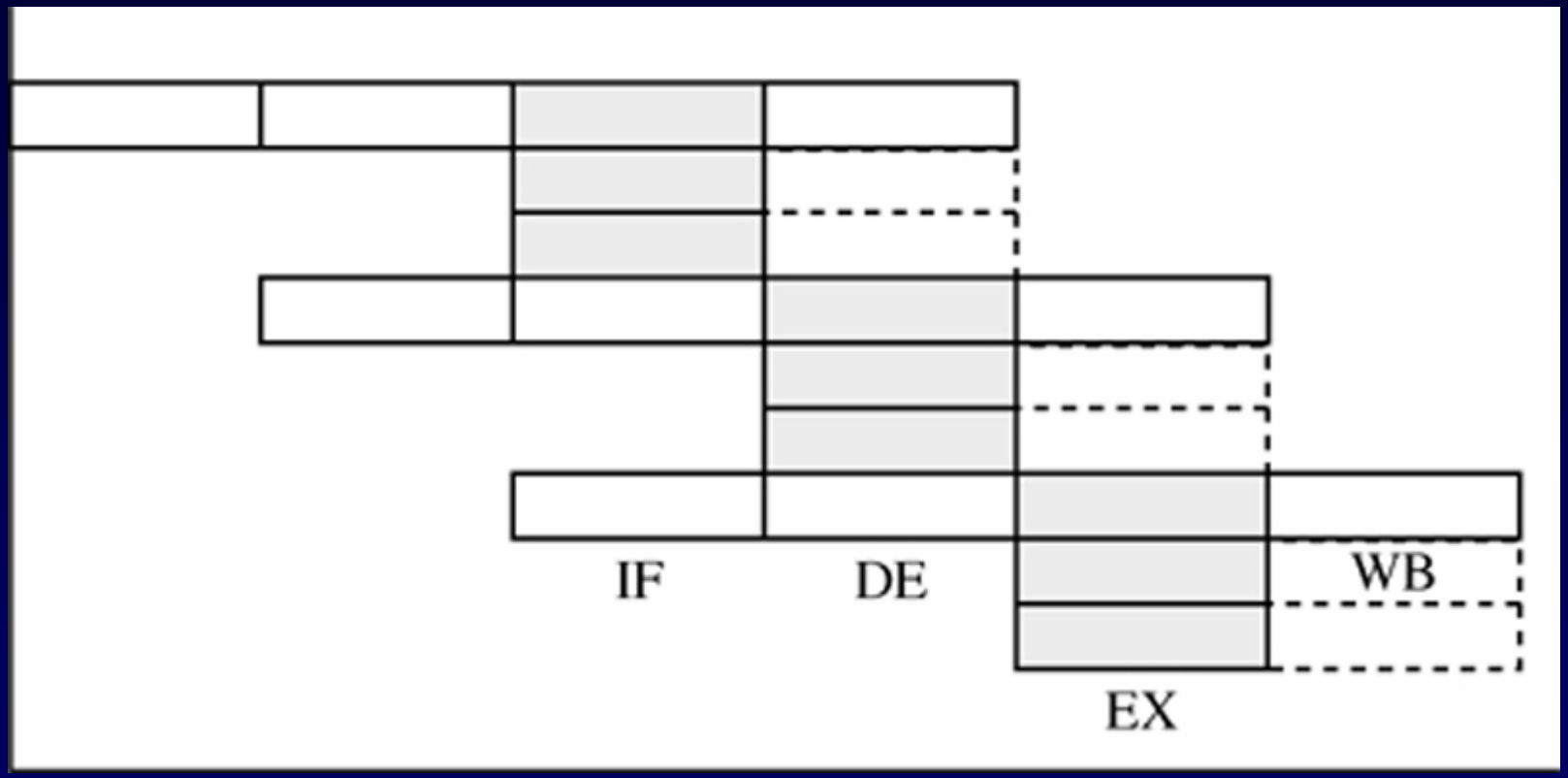
Superpipelined MIPS R4000 8-stage pipeline



Superscalar machine of degree *n*=3

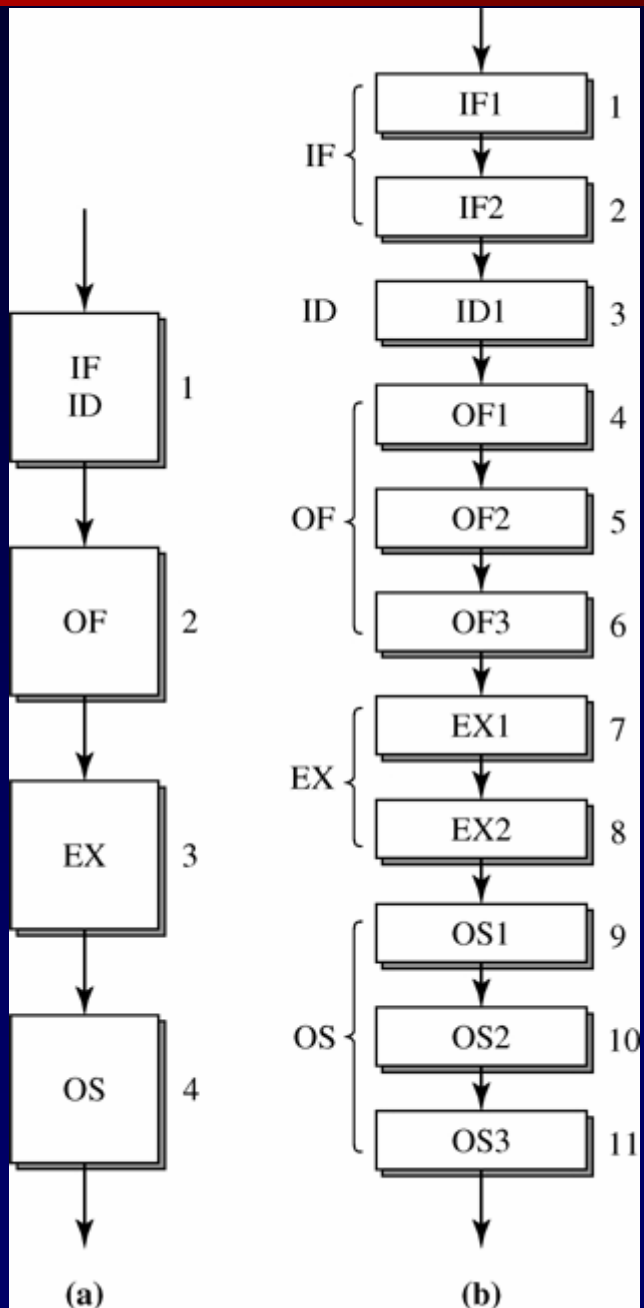


VLIW machine of degree *n*=3



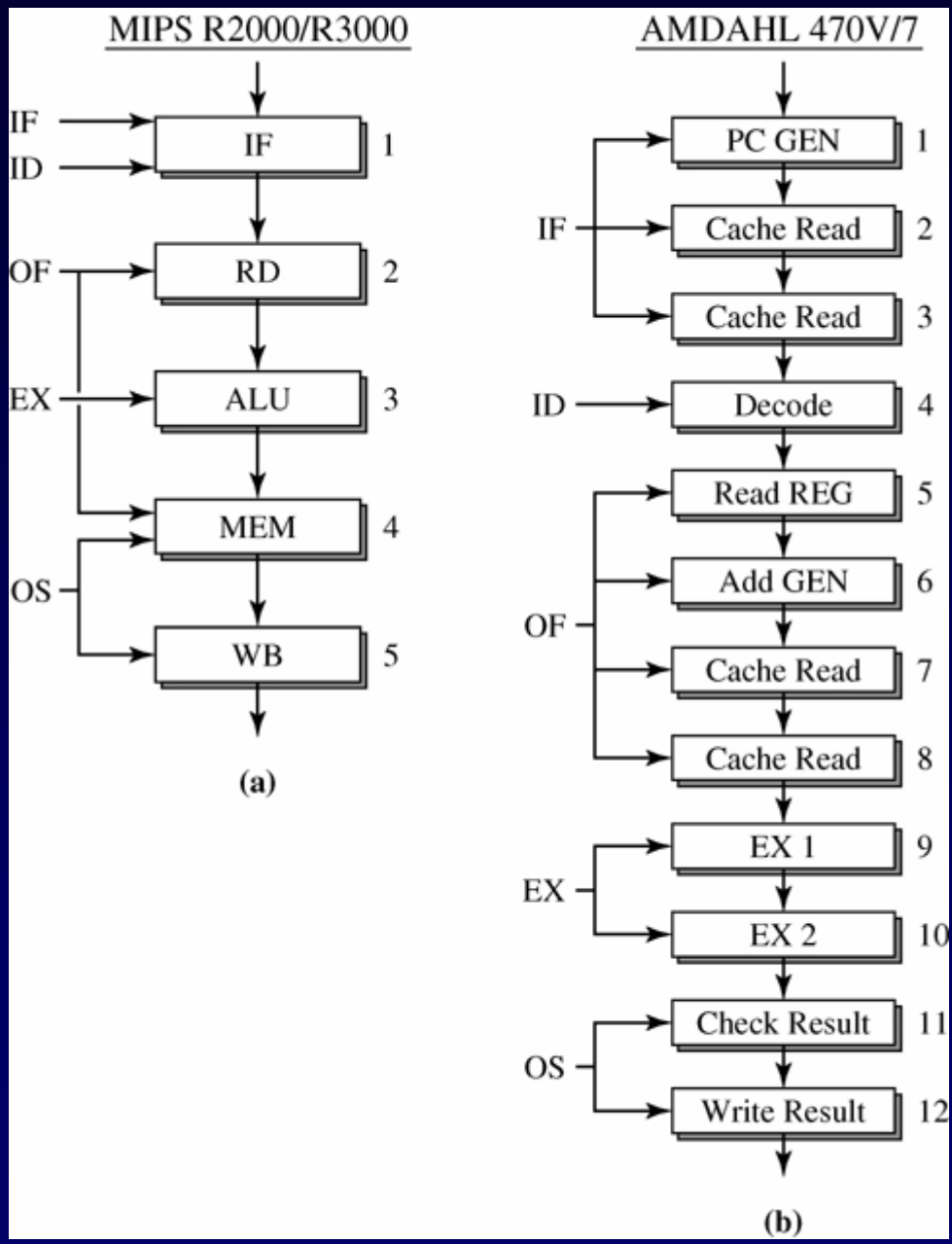
Pipeline example

4-stage

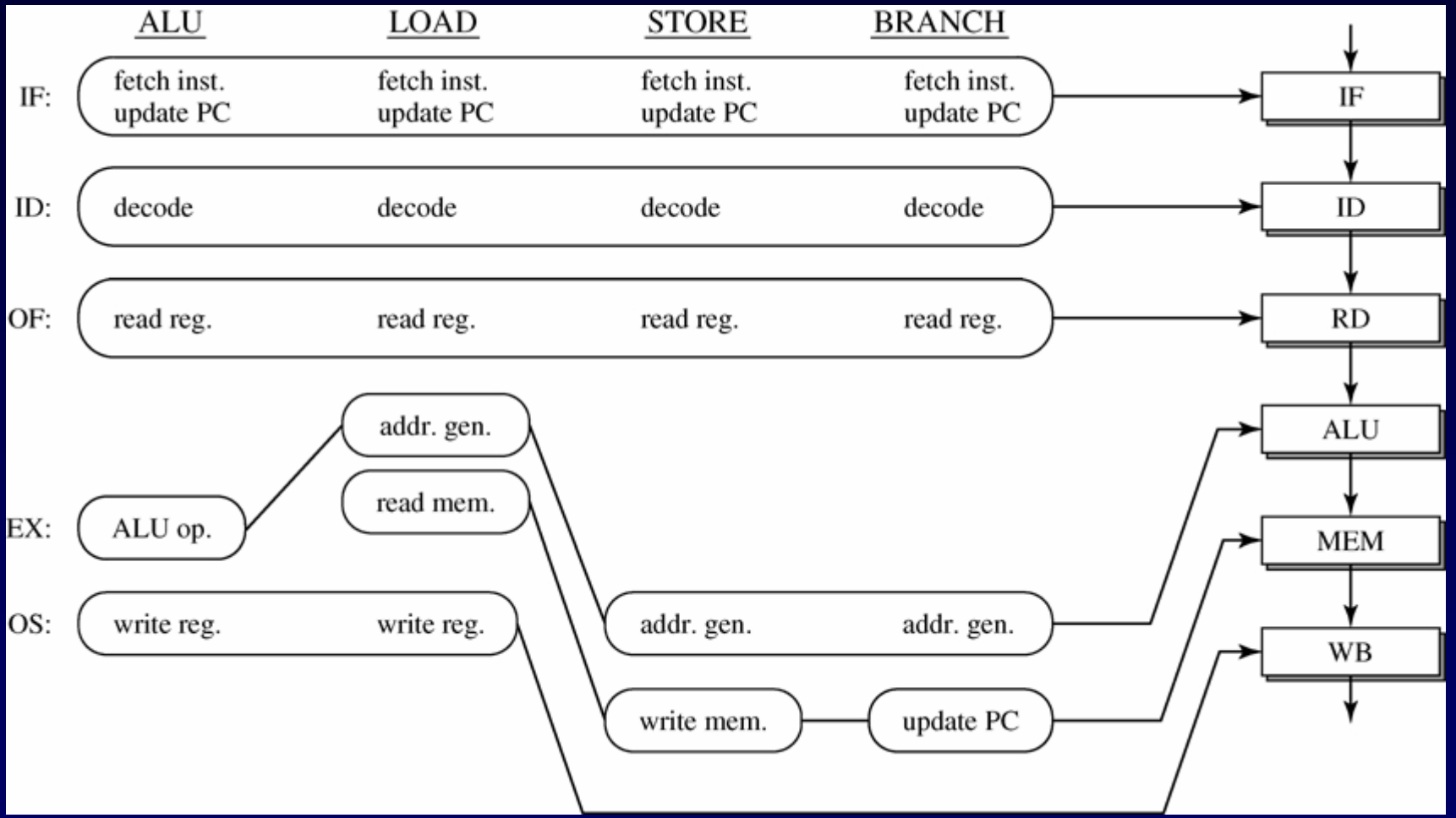


11-stage

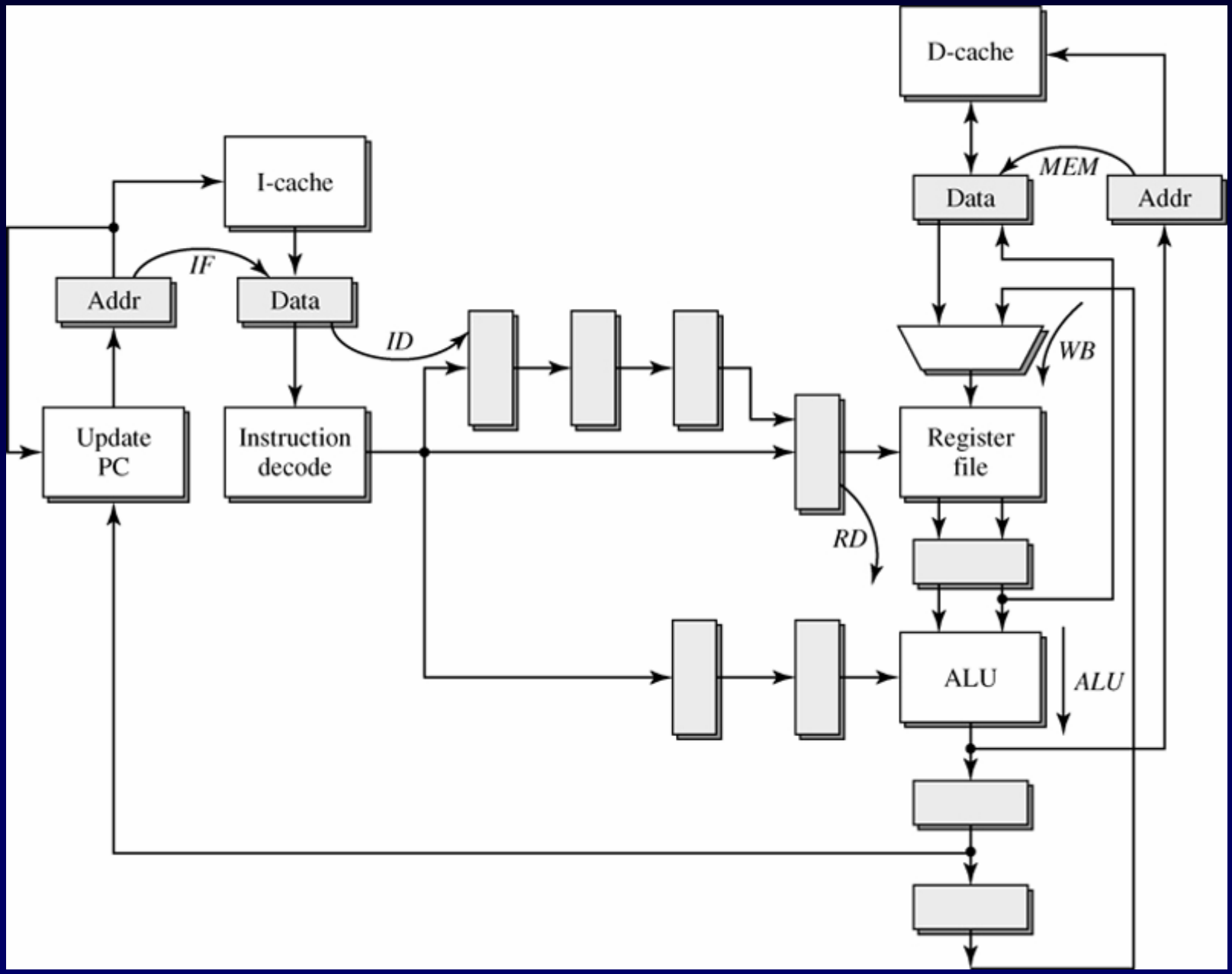
Two commercial instruction pipelines



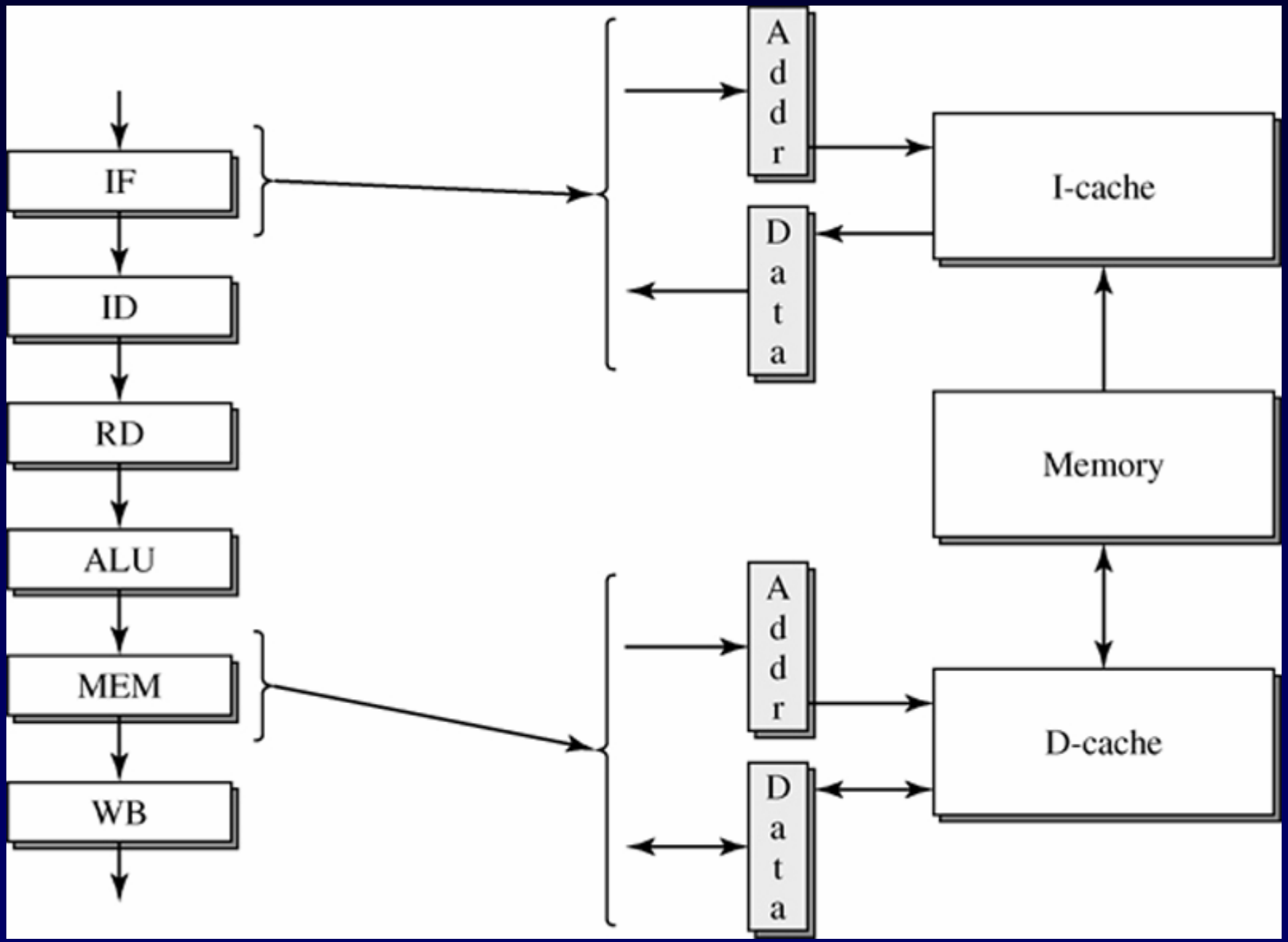
Activity of pipeline stages



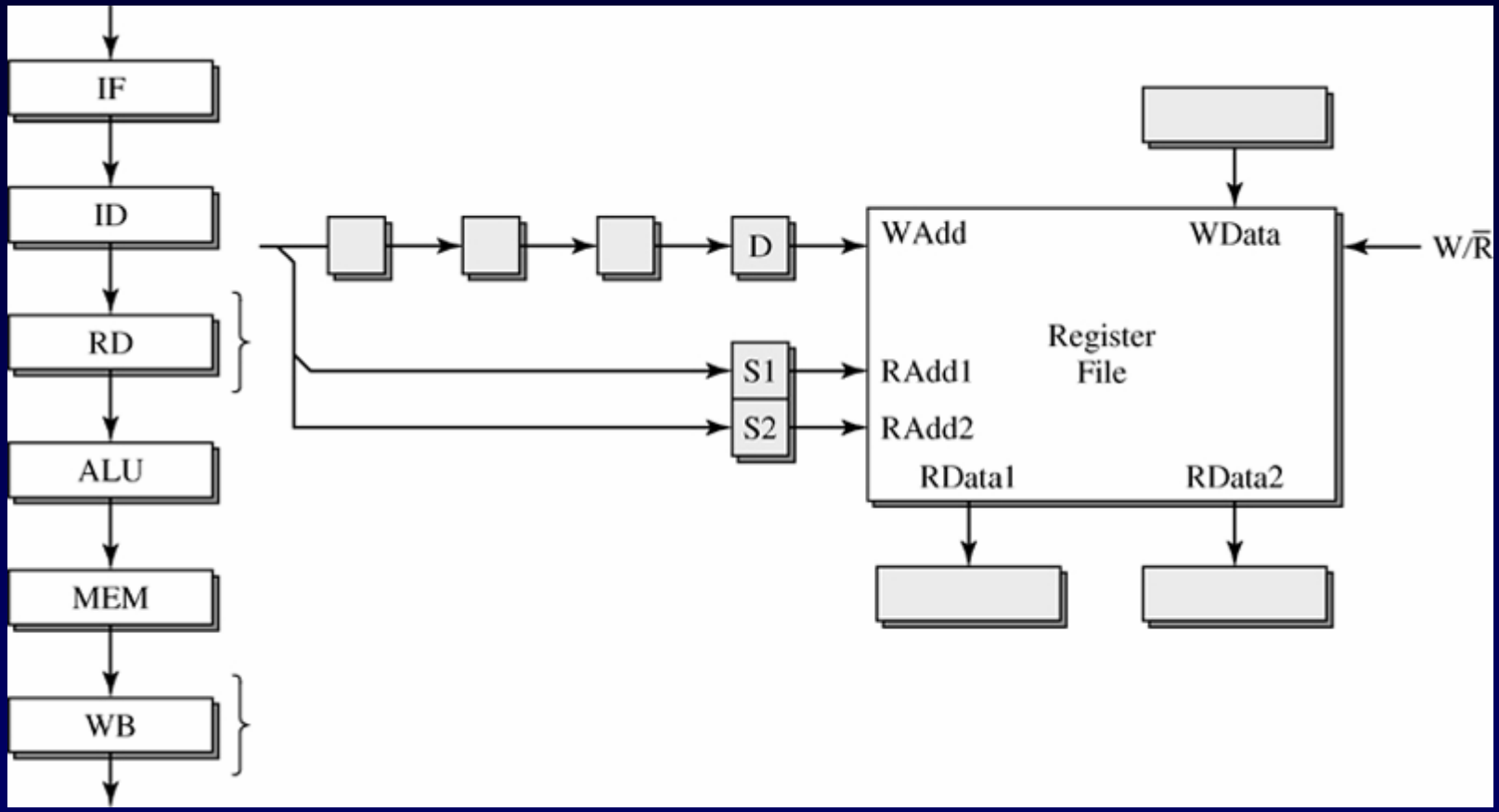
6-stage instruction pipeline



I-cache and D-cache



Access to RF


IF
ID
RD
ALU
MEM
WB
D
S1
S2
WAdd
WData
W/R
Register
File
RAdd1
RAdd2
RData1
RData2

RAW, WAR, and WAW data dependencies

- True dependence

$$R_3 \leftarrow R_1 \; op \; R_2$$
$$R_5 \leftarrow R_3 \; op \; R_4$$

Read-after-write (RAW)

- Anti-dependence

$$R_3 \leftarrow R_1 \; op \; R_2$$
$$R_1 \leftarrow R_4 \; op \; R_5$$

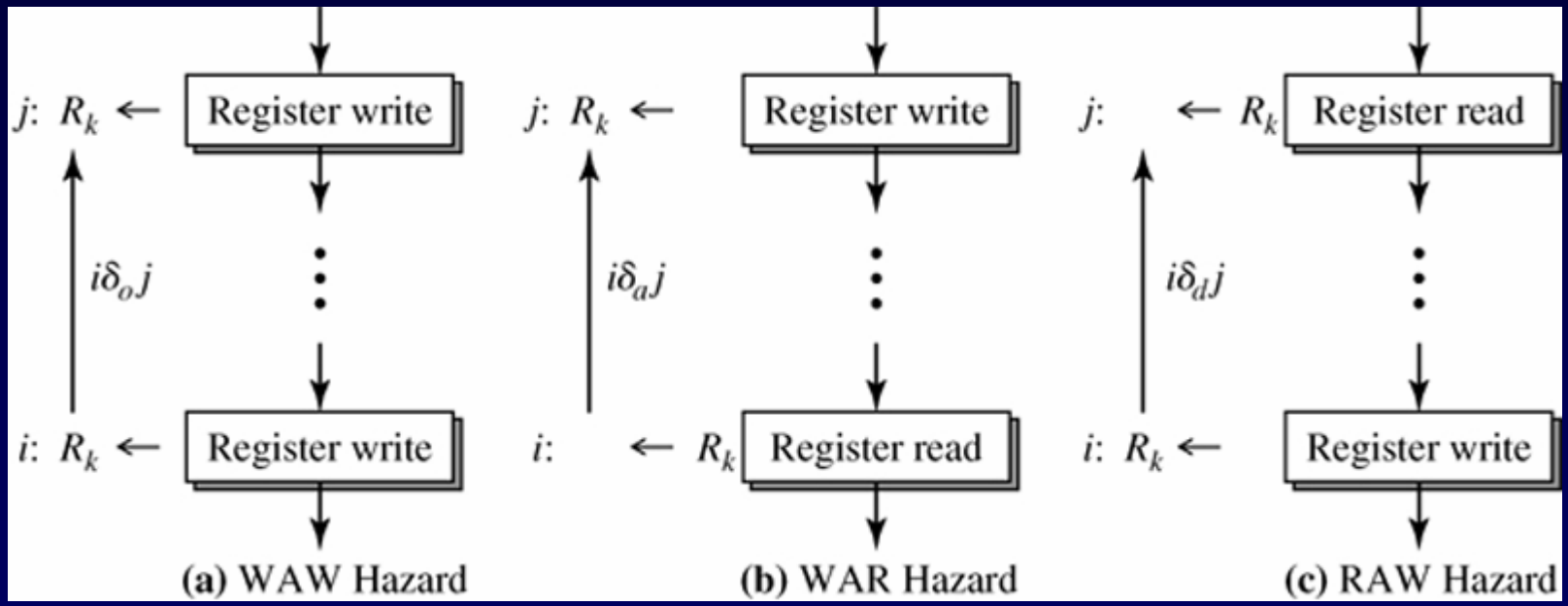
Write-after-read (WAR)

- Output dependence

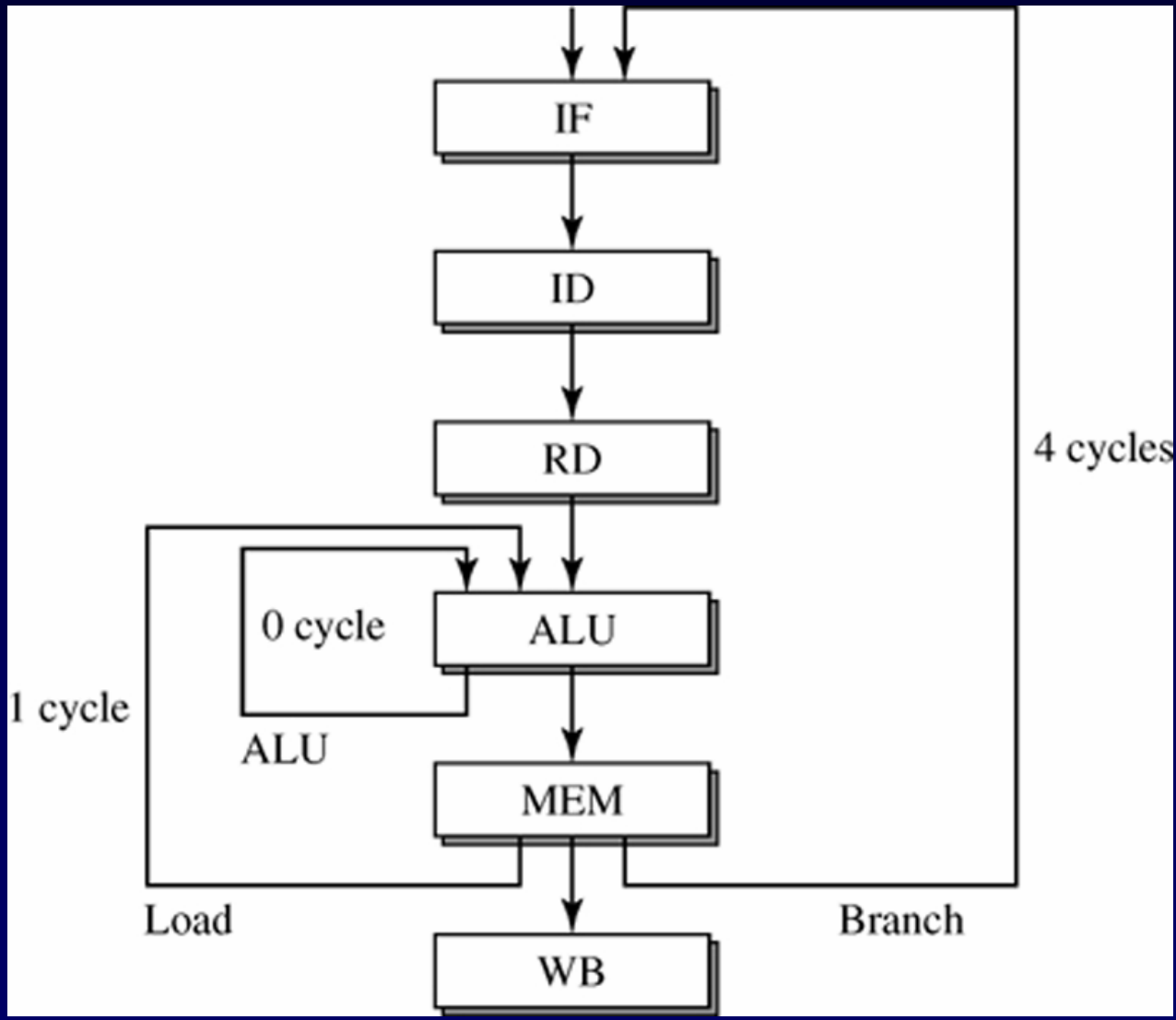
$$R_3 \leftarrow R_1 \; op \; R_2$$
$$R_5 \leftarrow R_3 \; op \; R_4$$
$$R_3 \leftarrow R_6 \; op \; R_7$$

Write-after-write (WAW)

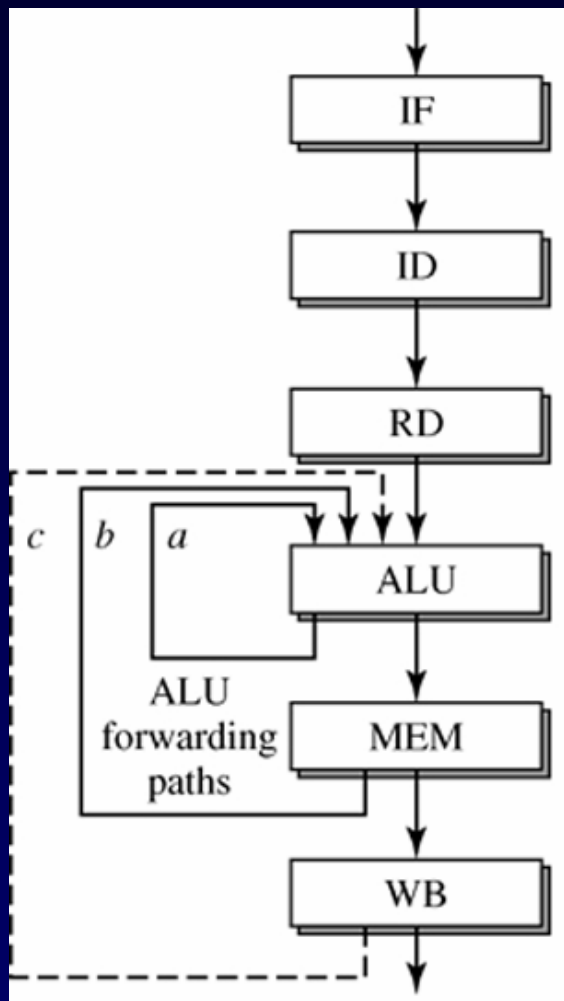
WAW, WAR, and RAW hazards



Forwarding paths

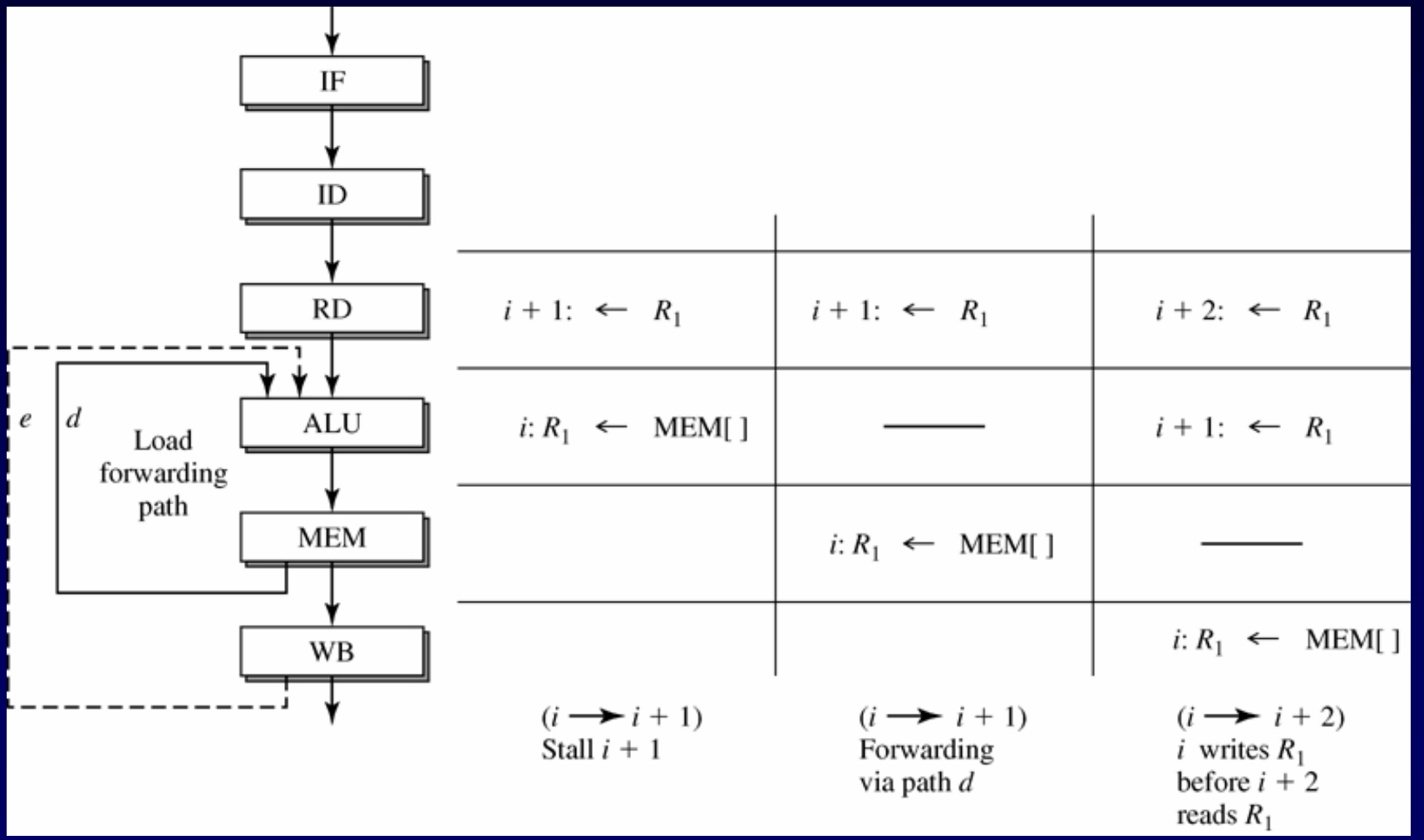


Forwarding paths for ALU leading instructions

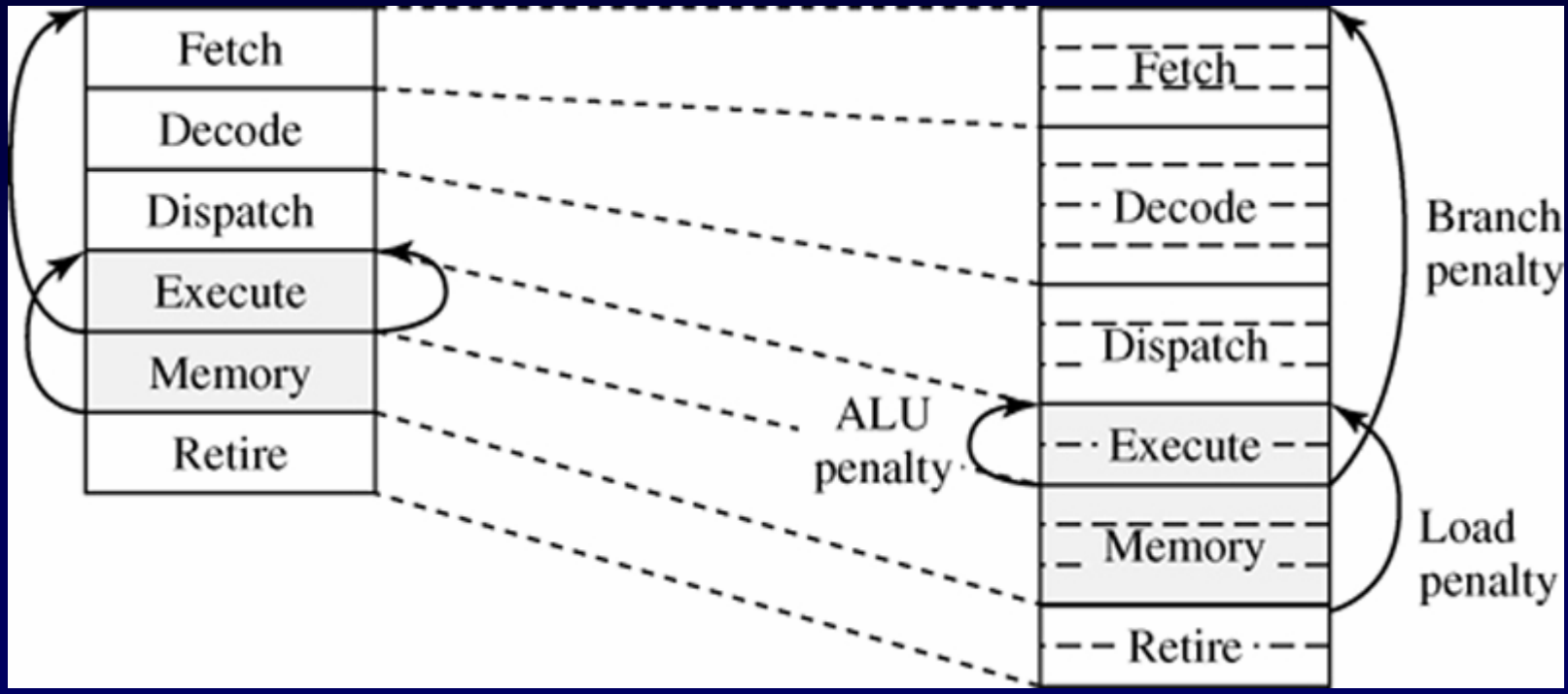


	Cycle t1:	Cycle t2:	Cycle t3:
	$i+1: \leftarrow R_1$	$i+2: \leftarrow R_1$	$i+3: \leftarrow R_1$
	$i: R_1 \leftarrow$	$i+1: \leftarrow R_1$	$i+2: \leftarrow R_1$
		$i: R_1 \leftarrow$	$i+1: \leftarrow$
			$i: R_1 \leftarrow$
	$(i \rightarrow i+1)$ Forwarding via path a	$(i \rightarrow i+2)$ Forwarding via path b	$(i \rightarrow i+3)$ i writes R_1 before $i+3$ reads R_1

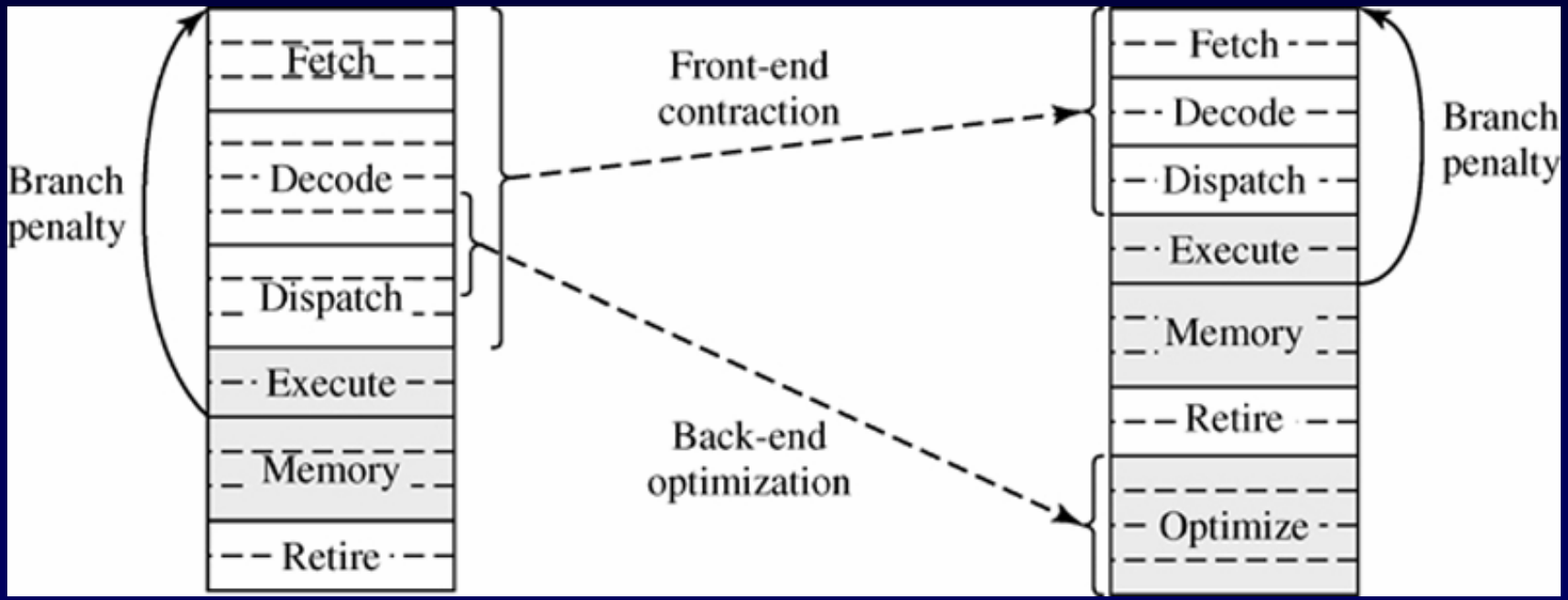
Forwarding paths for Load leading instructions



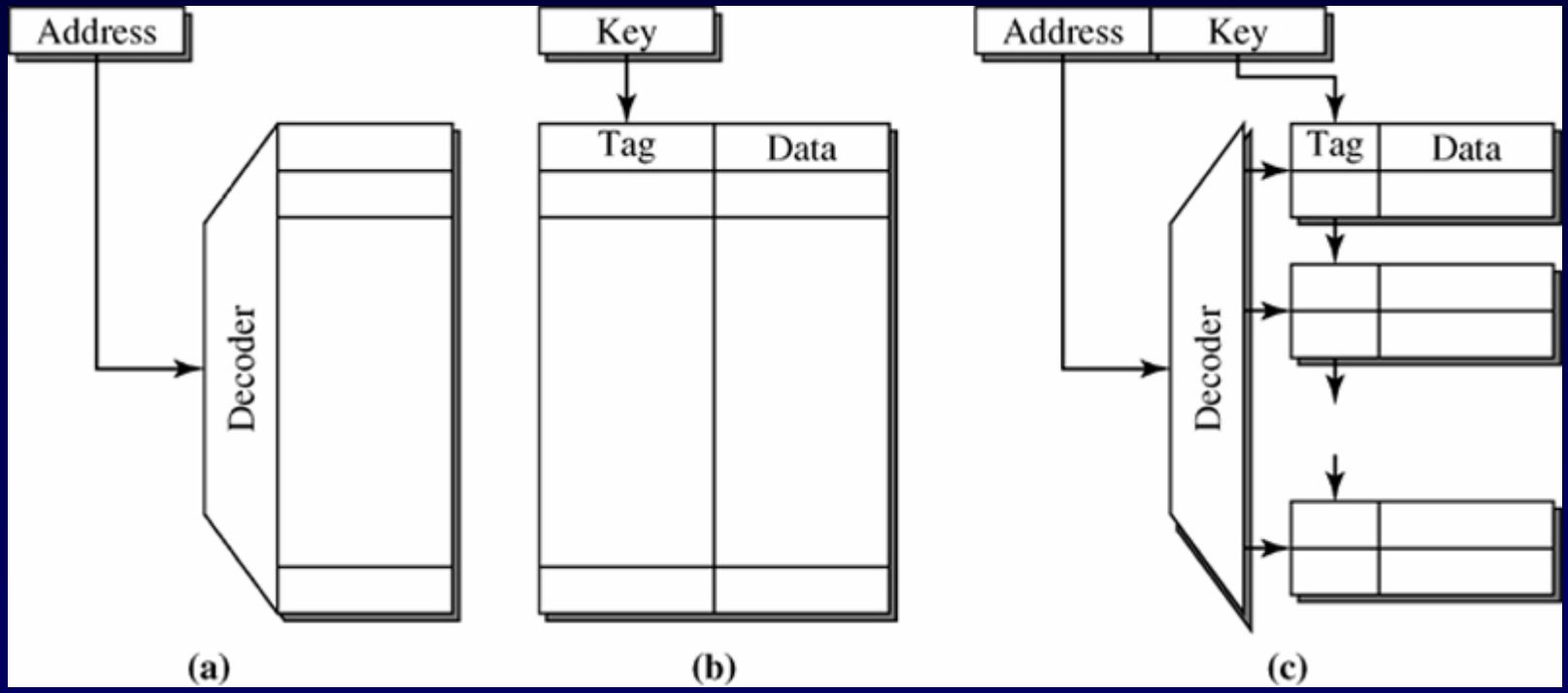
Impact on ALU, Load, and Branch penalties with increasing pipeline depth



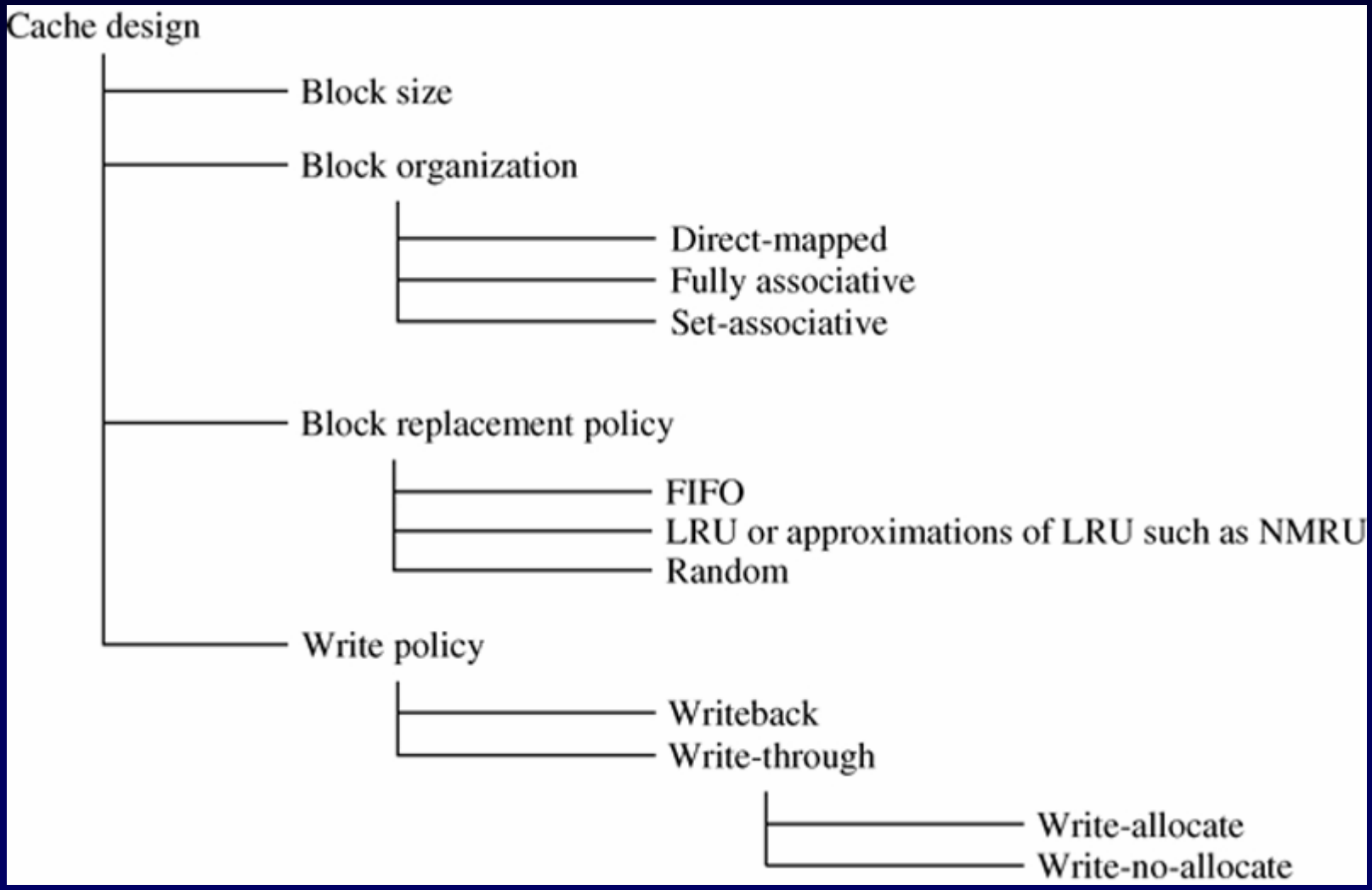
Mitigating the Branch penalty impact of deep pipelines



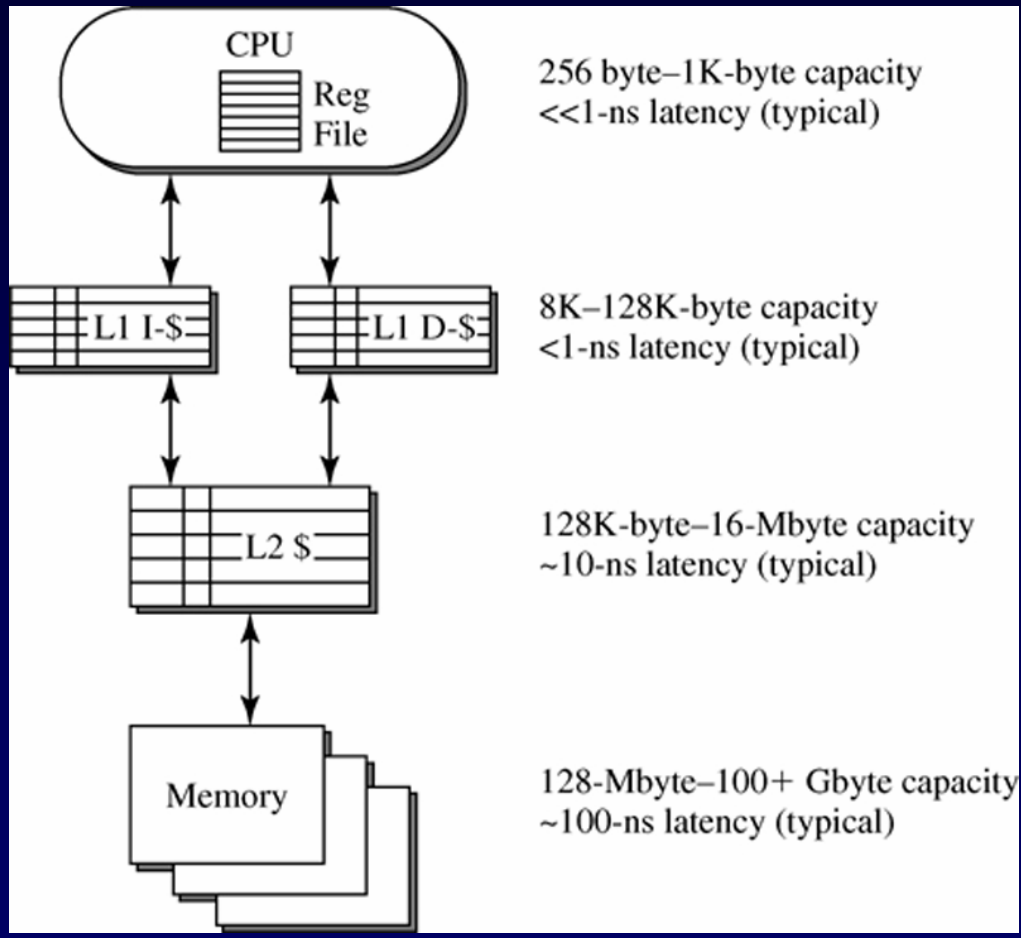
Direct, associative, and set-associative caches



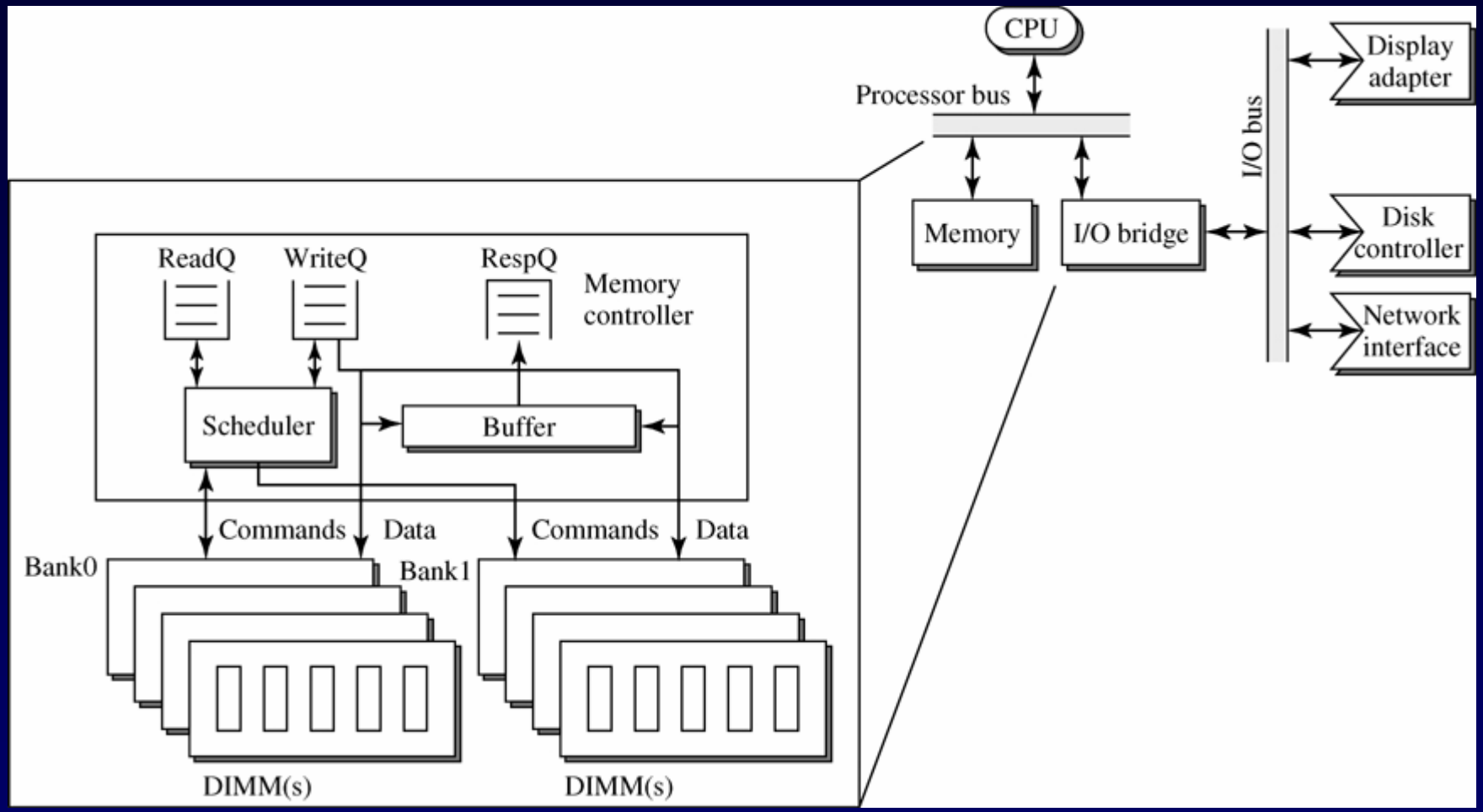
Strategy of cache design



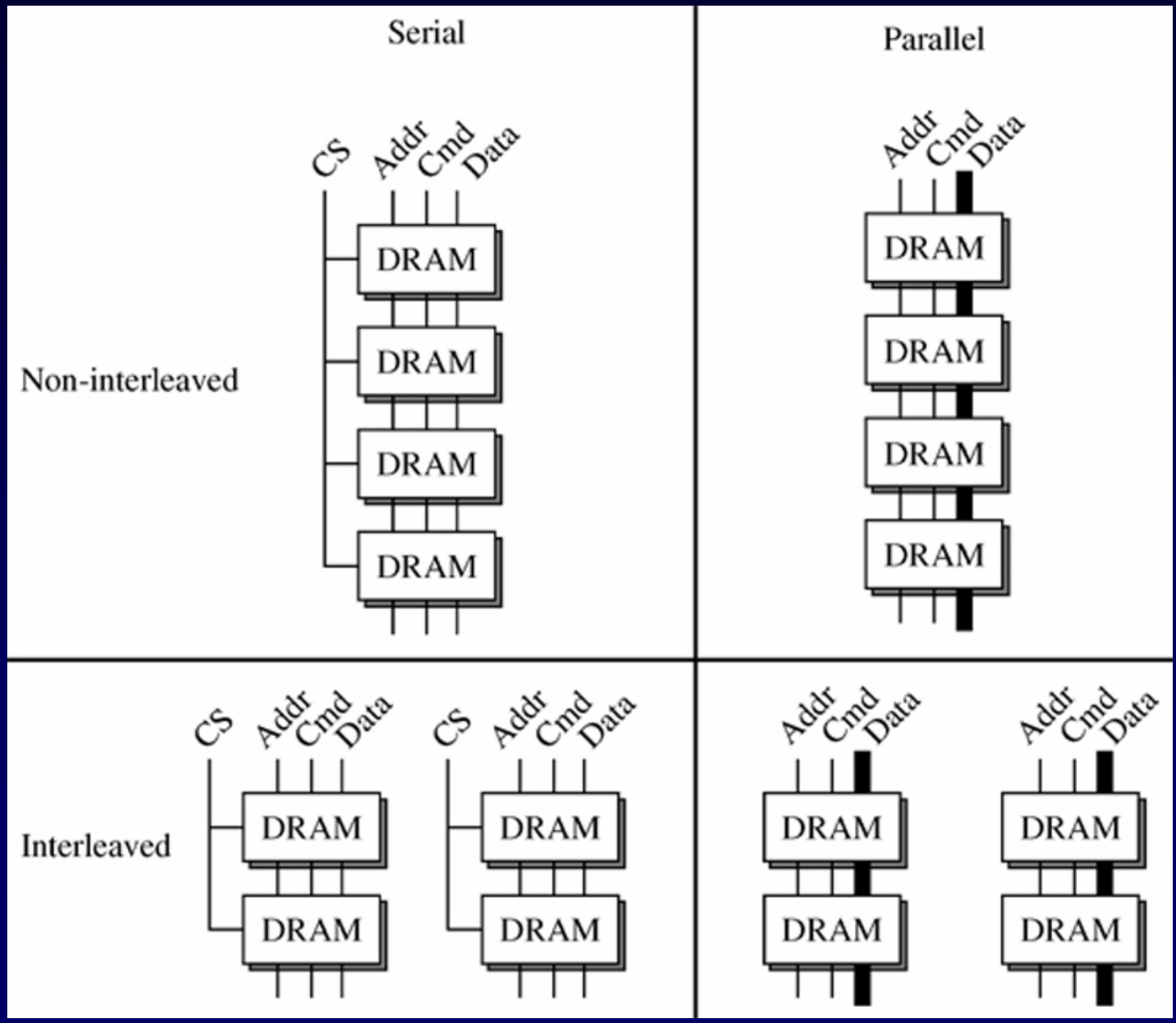
Memory hierarchy



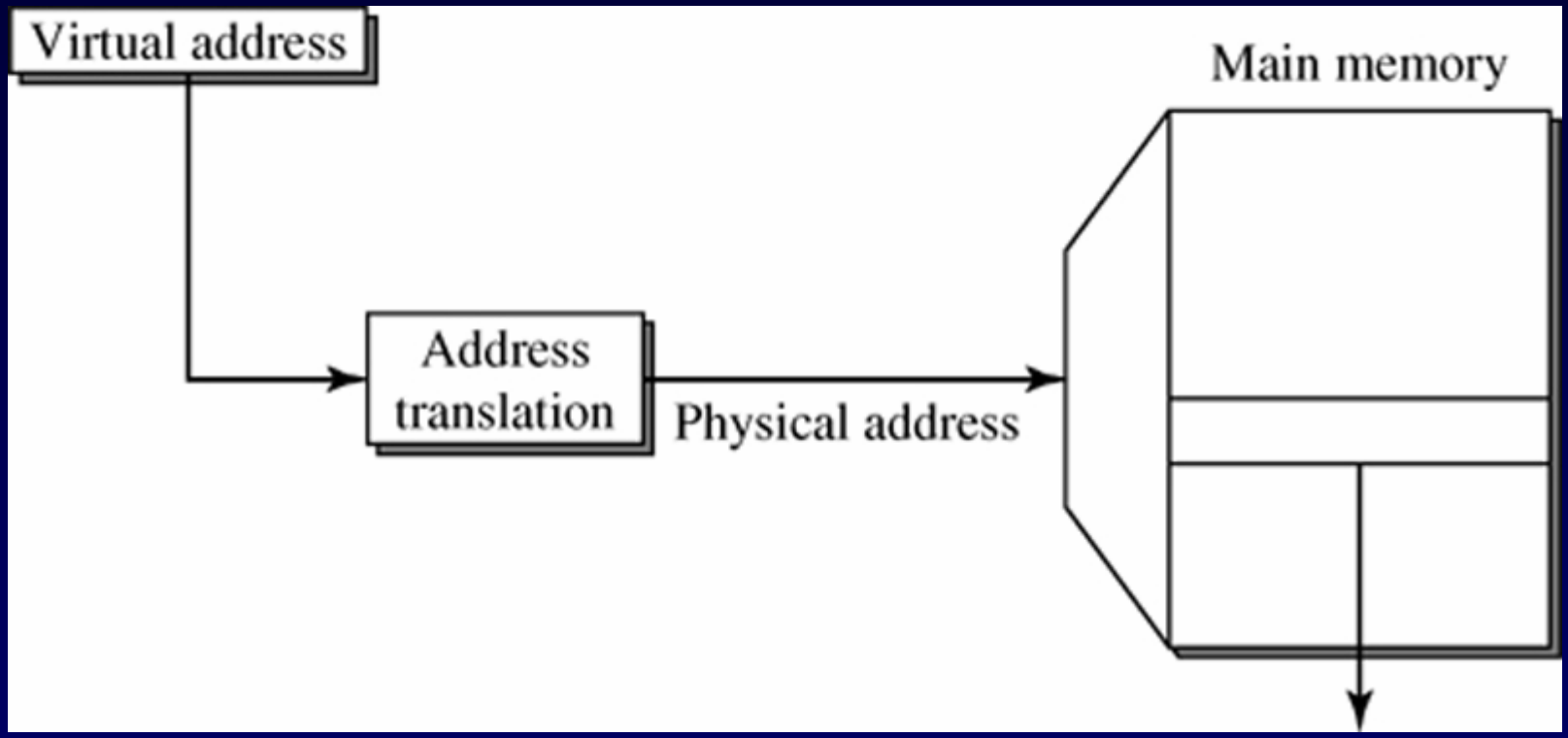
Main memory and I/O



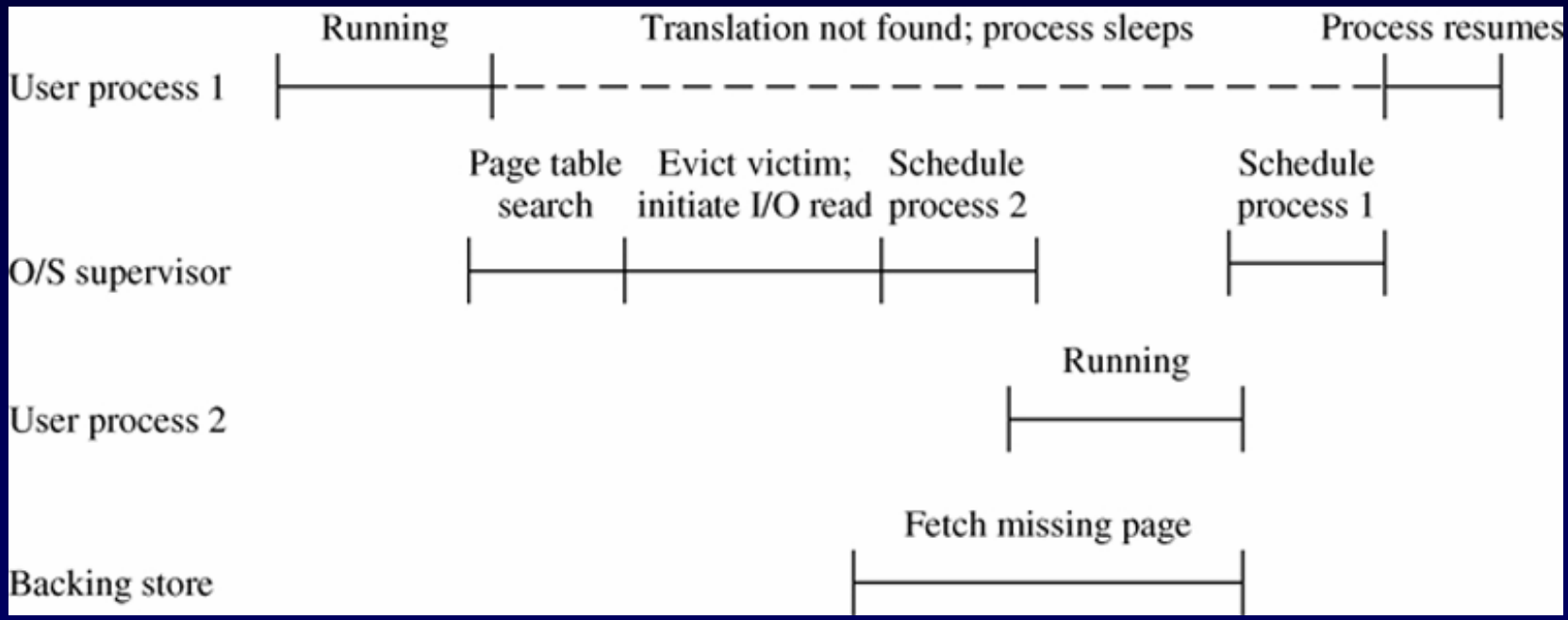
DRAM accesses



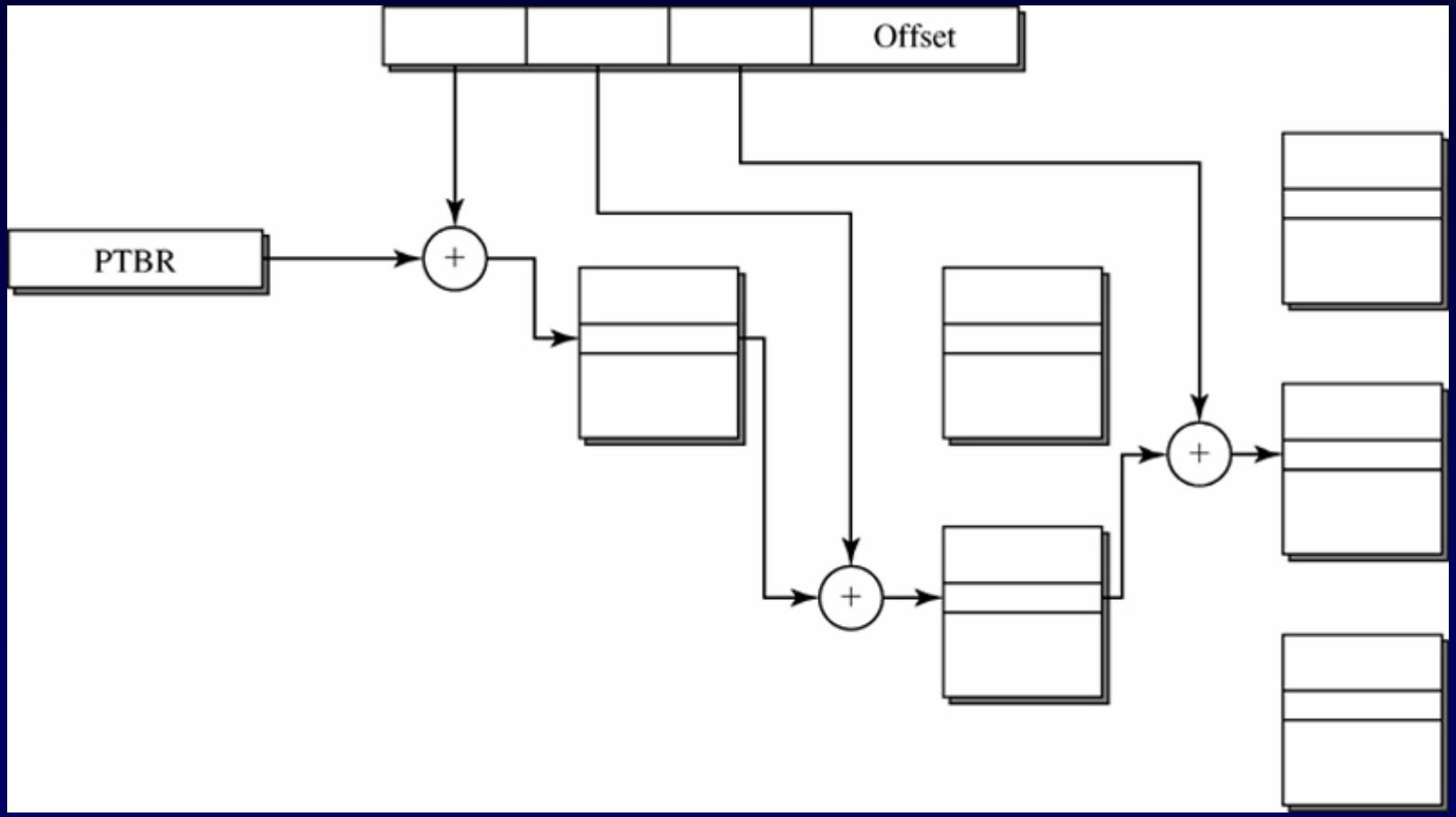
Virtual to physical address translation



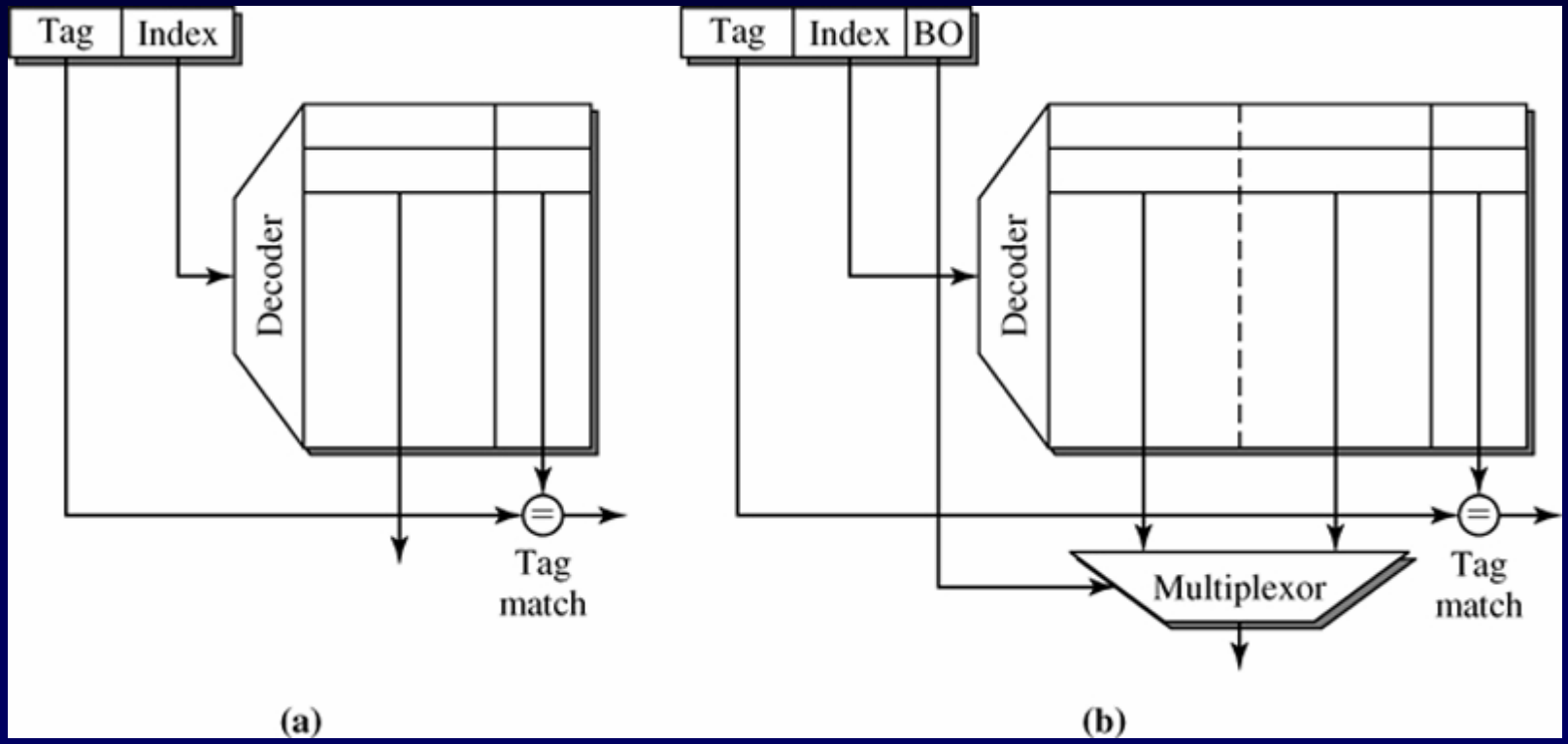
Processes switching



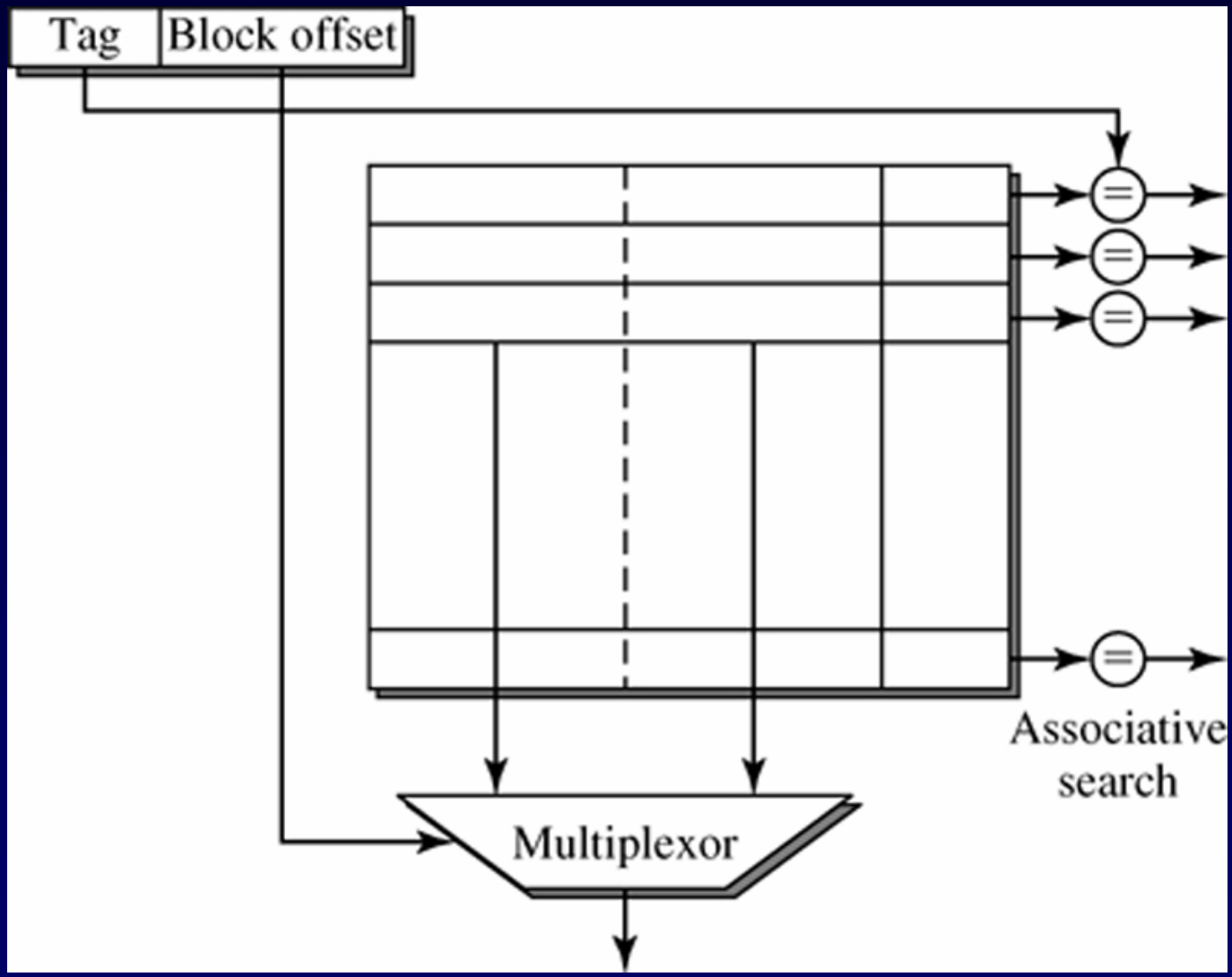
Paging tables



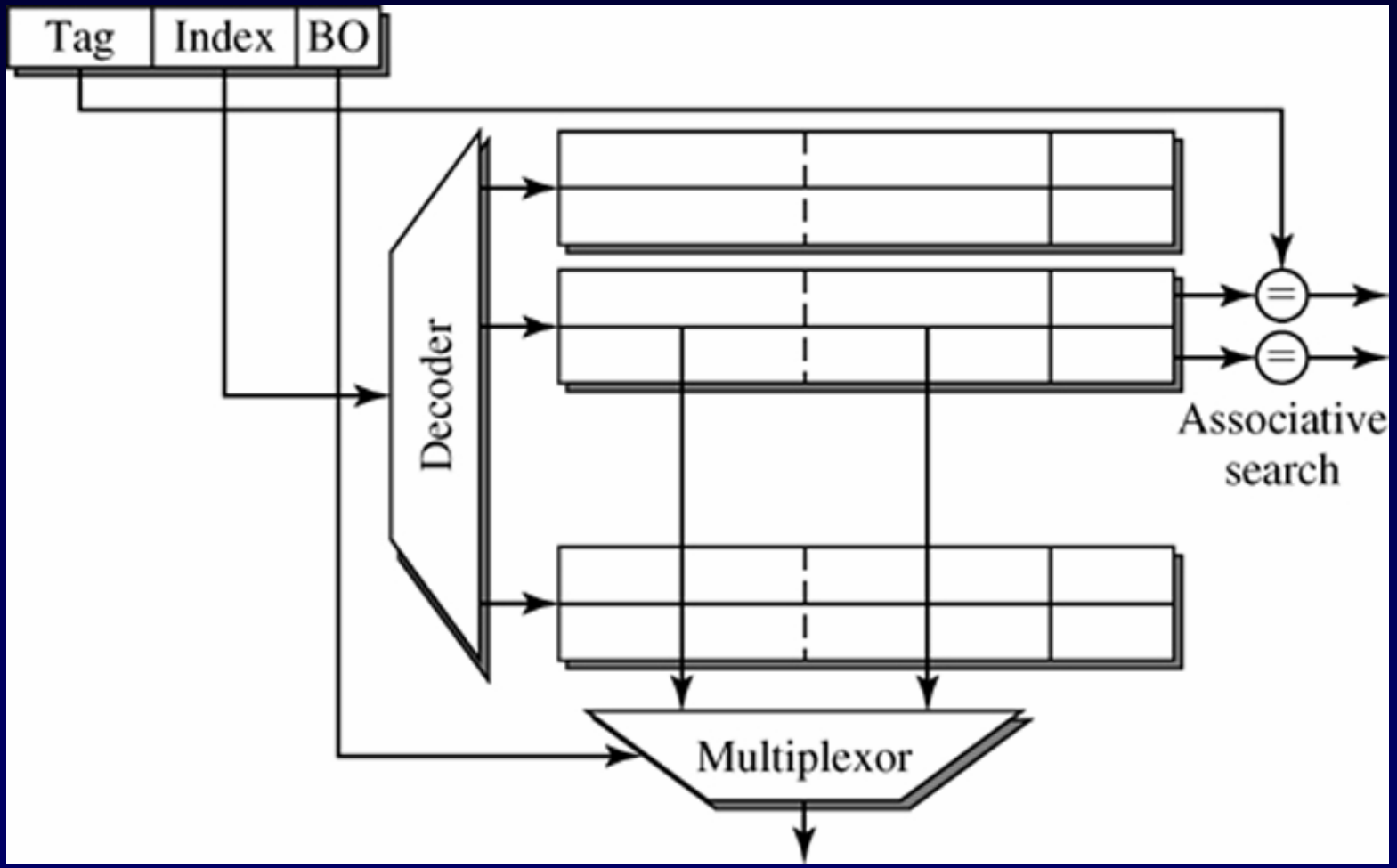
Direct cache



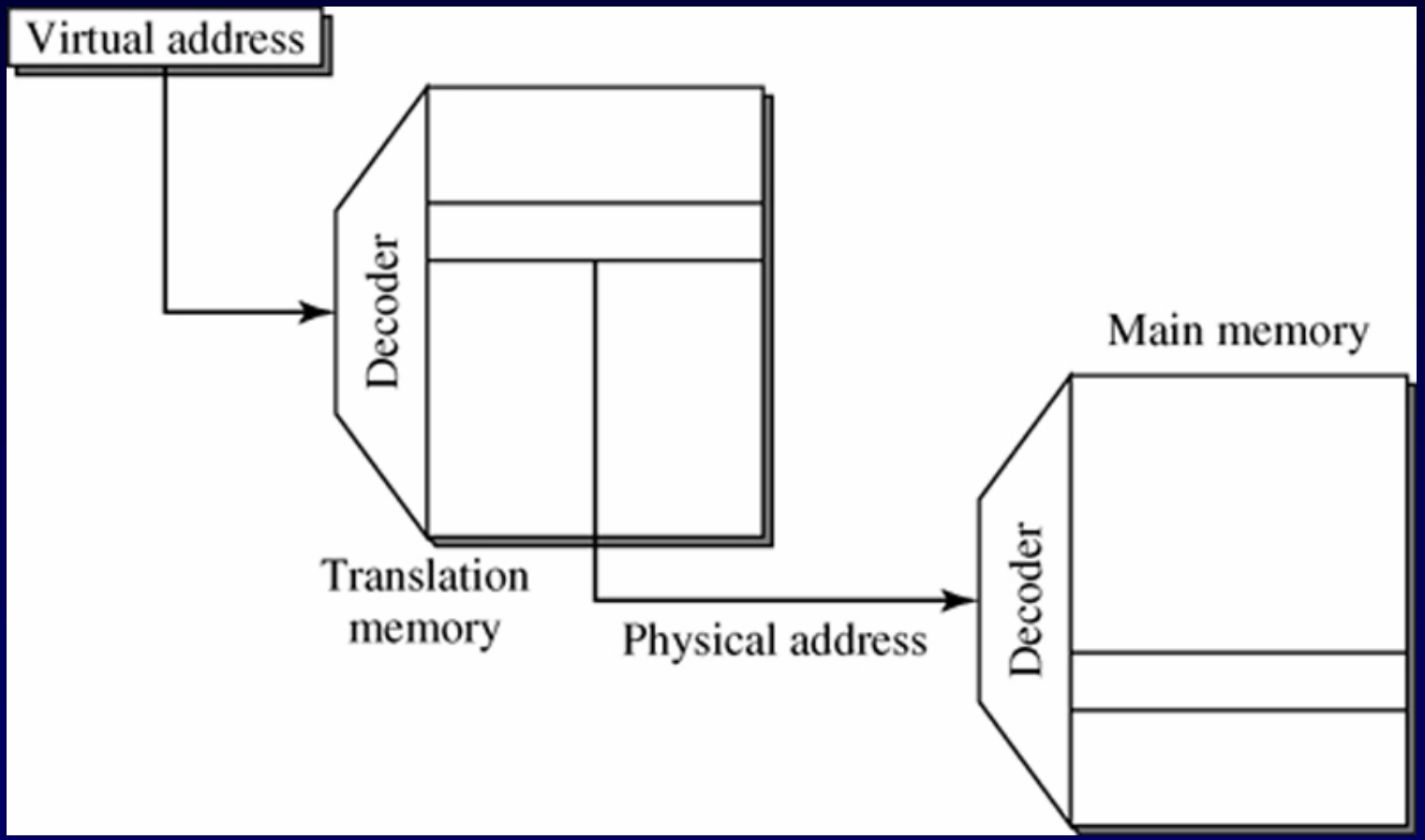
Associative cache



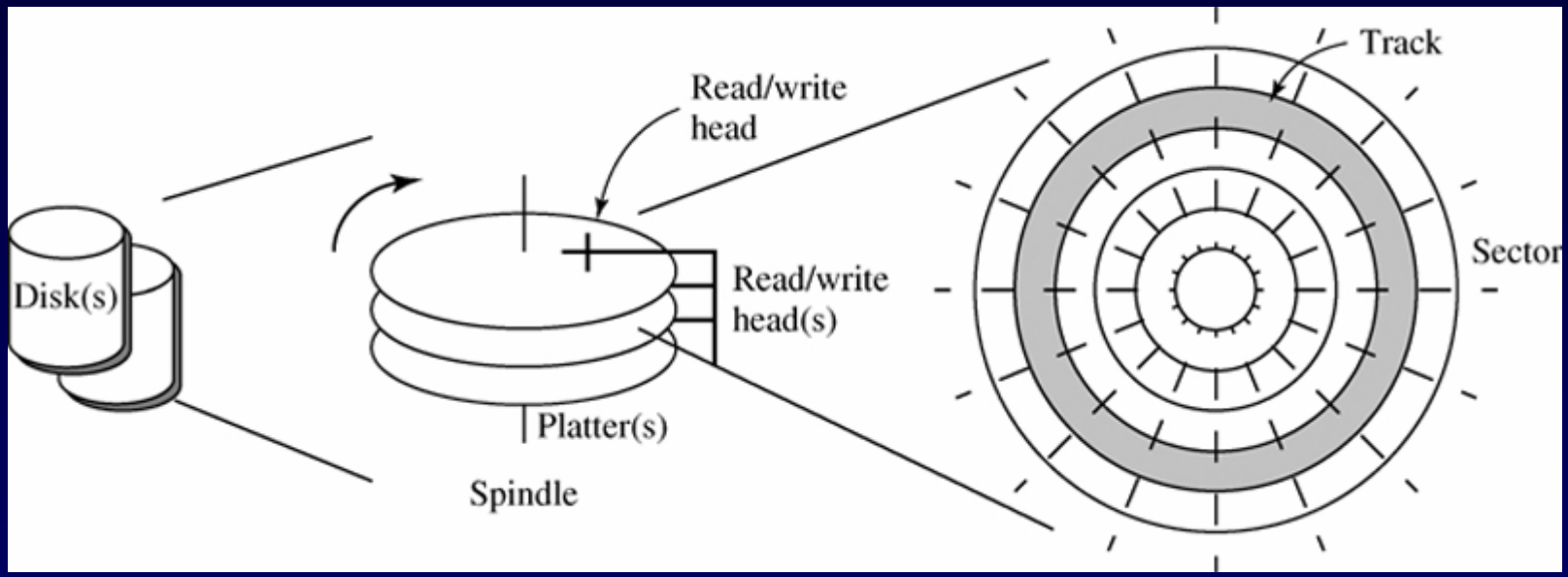
Set-associative cache



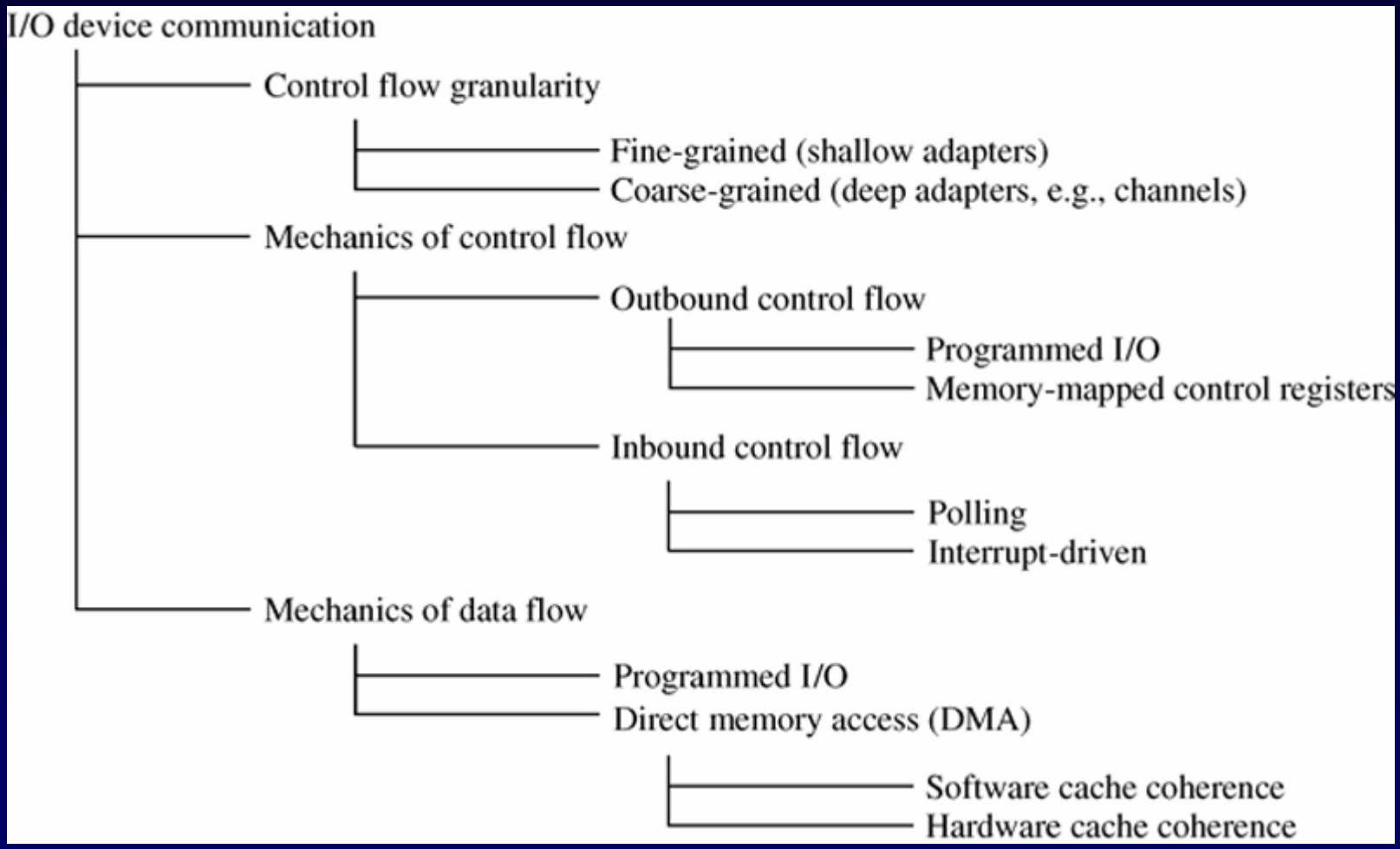
Virtual to physical address translation



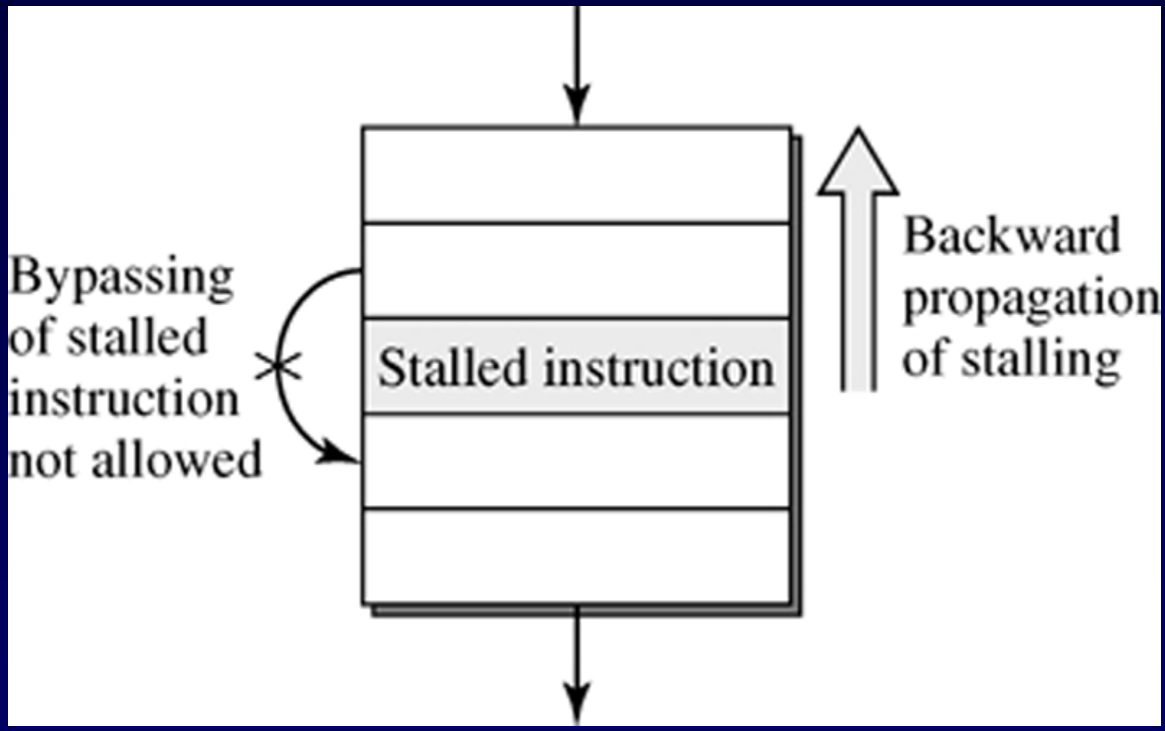
Disk


Track
Read/write
head
Sector
Disk(s)
Read/write
head(s)
Platter(s)
Spindle

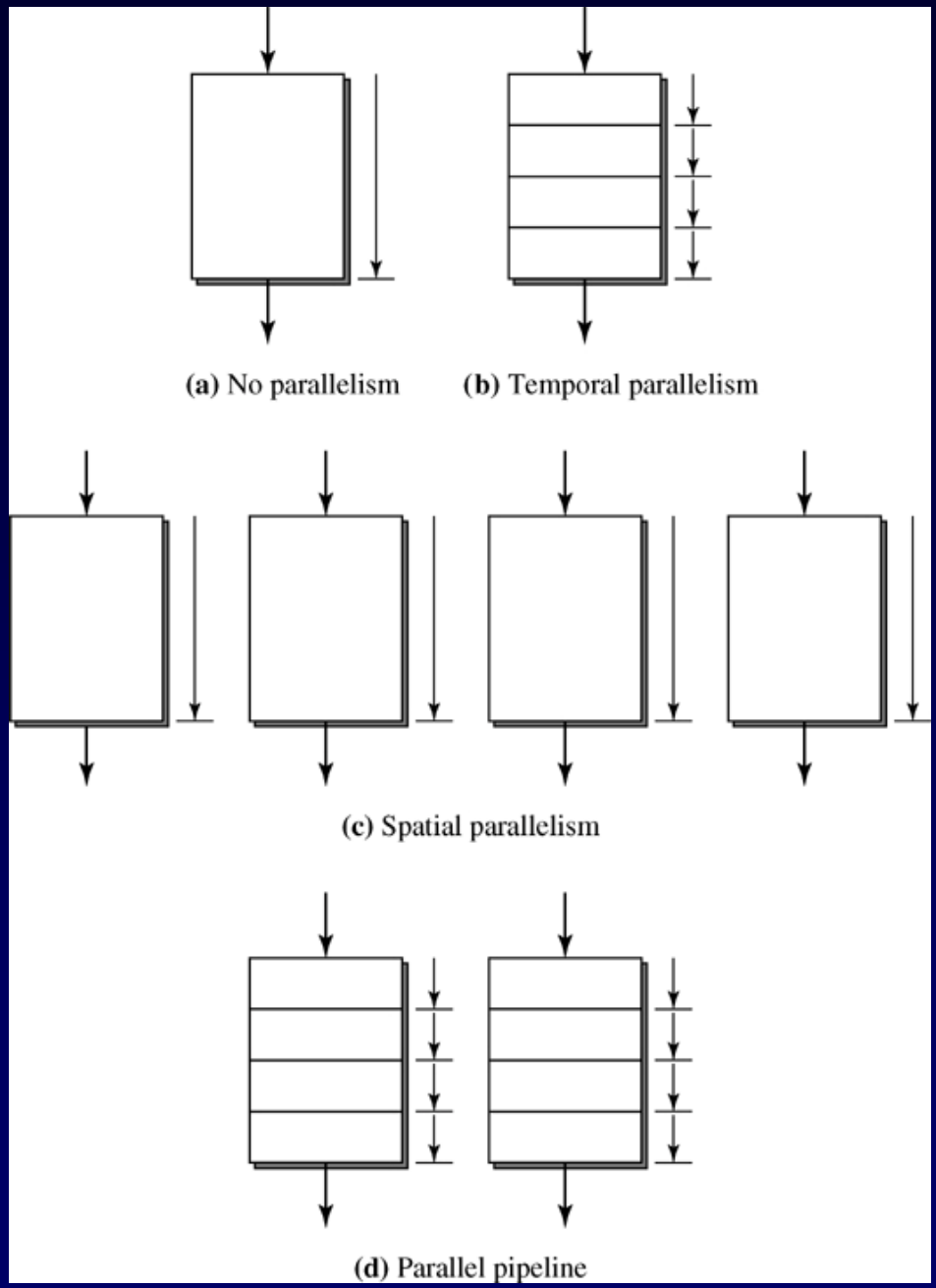
I/O device communication



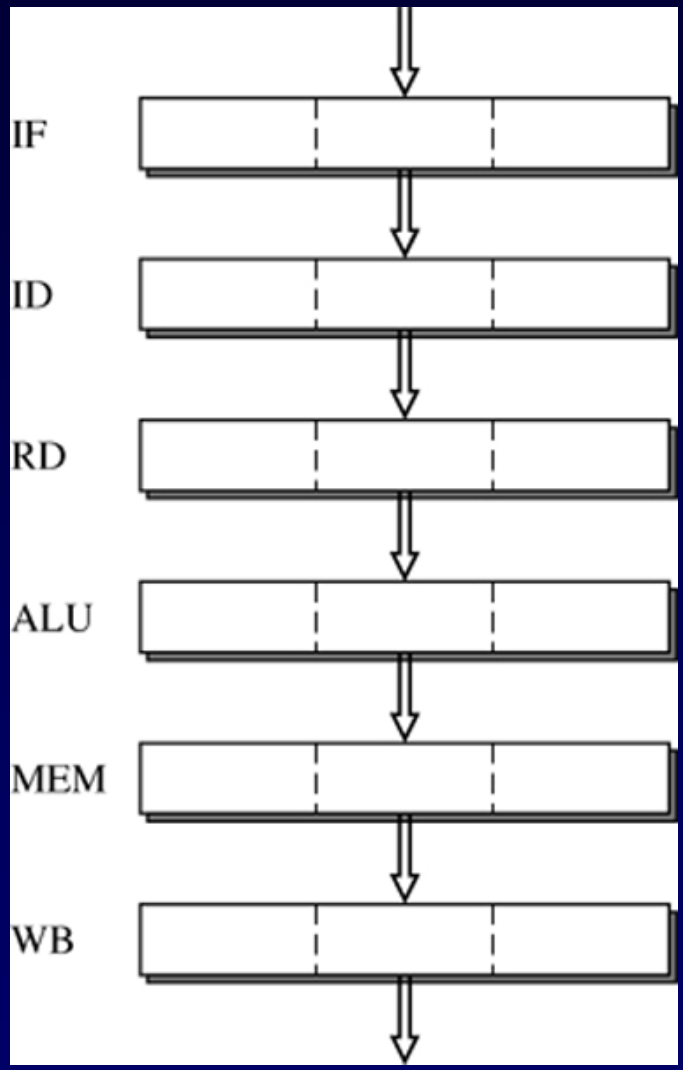
Stall cycle induced by backward propagation of stalling



Machine parallelism

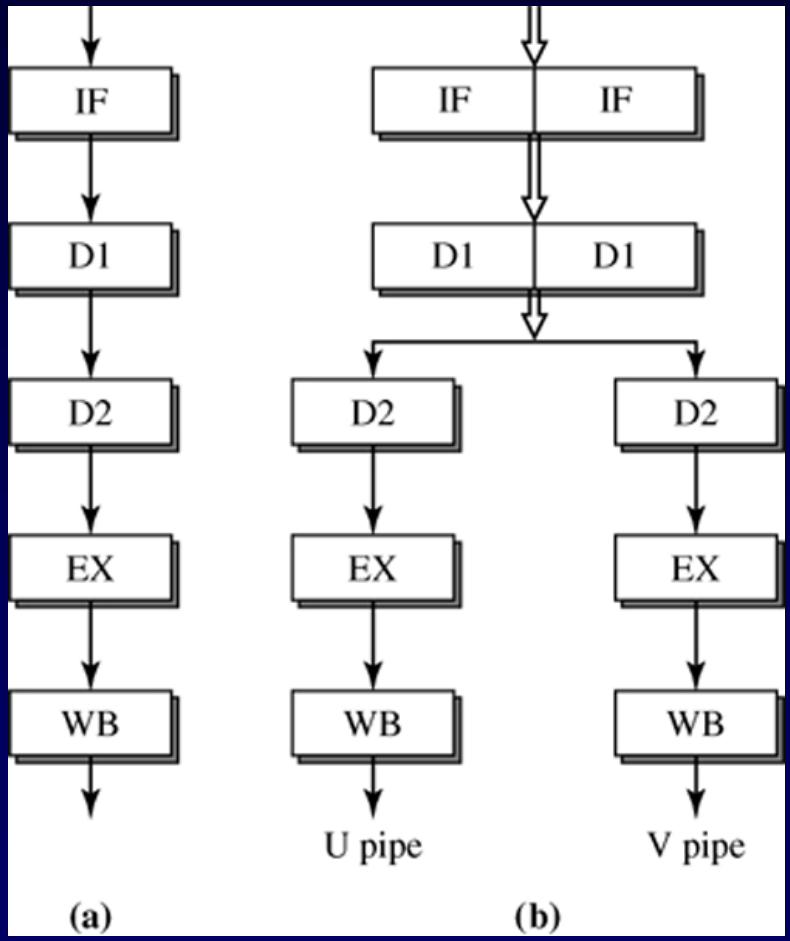


Parallel pipeline of width *s=3*



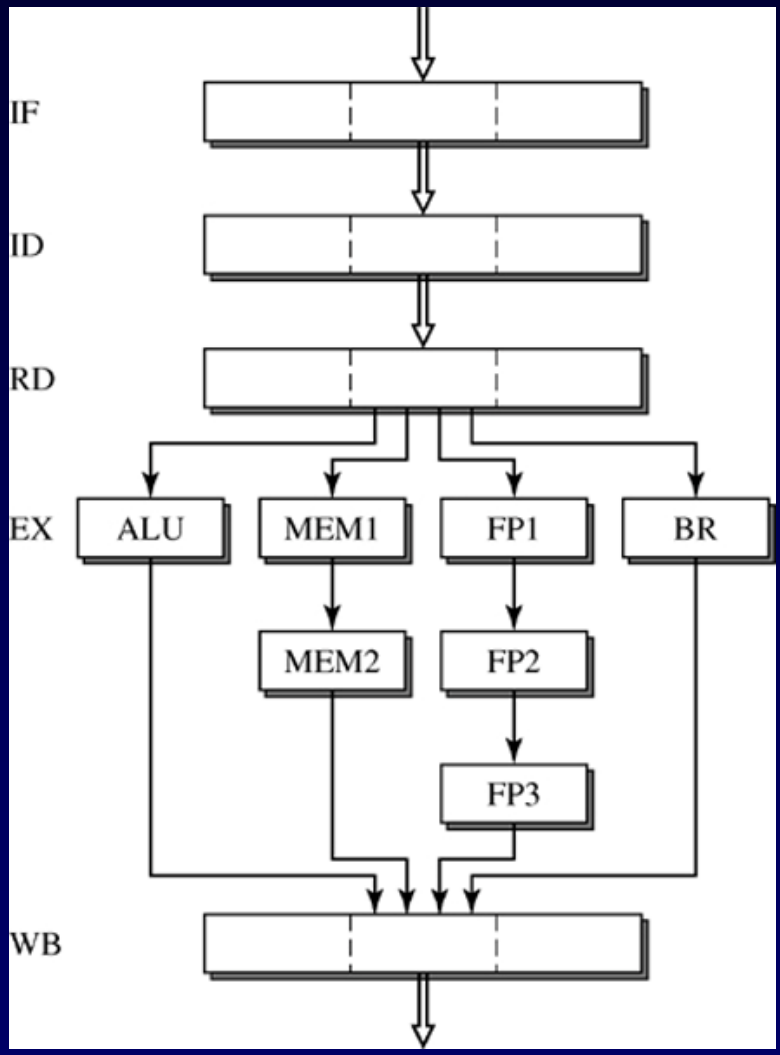
Parallel pipeline - examples

5-stage i486

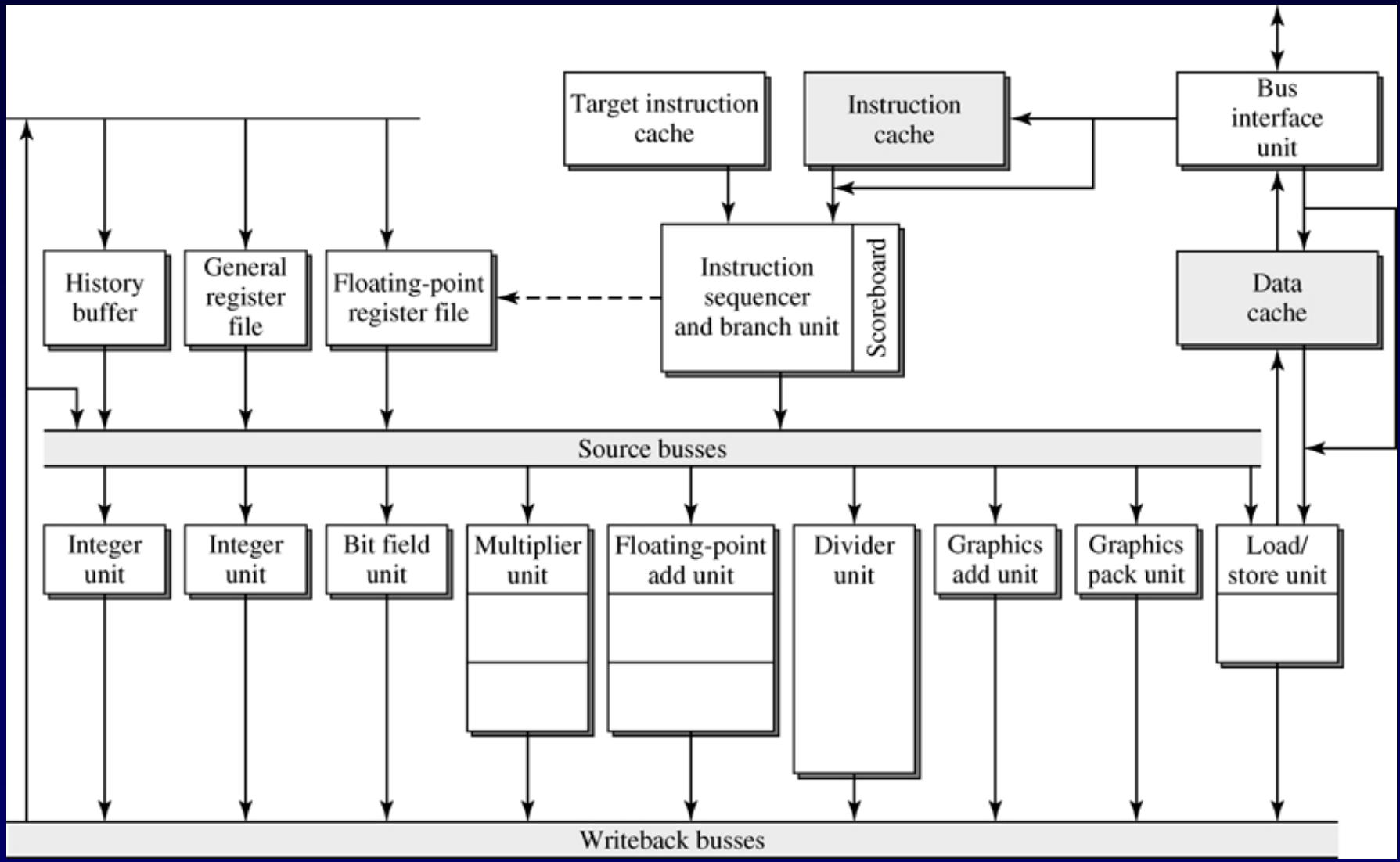


5-stage Pentium parallel pipeline *s*=2

Parallel pipeline with four execution pipes

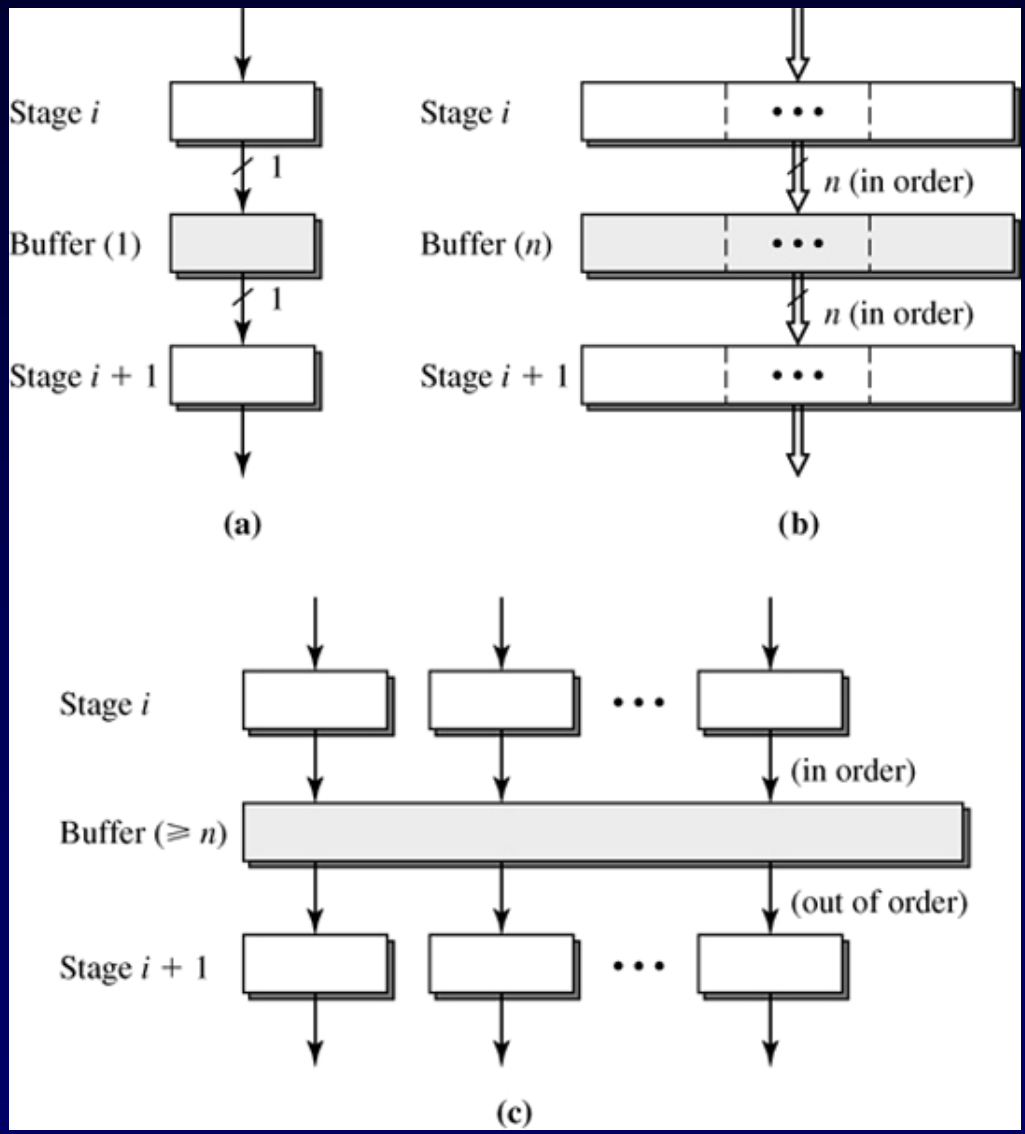


The Motorola 88110 superscalar microprocessor



Interpipeline-stage buffers

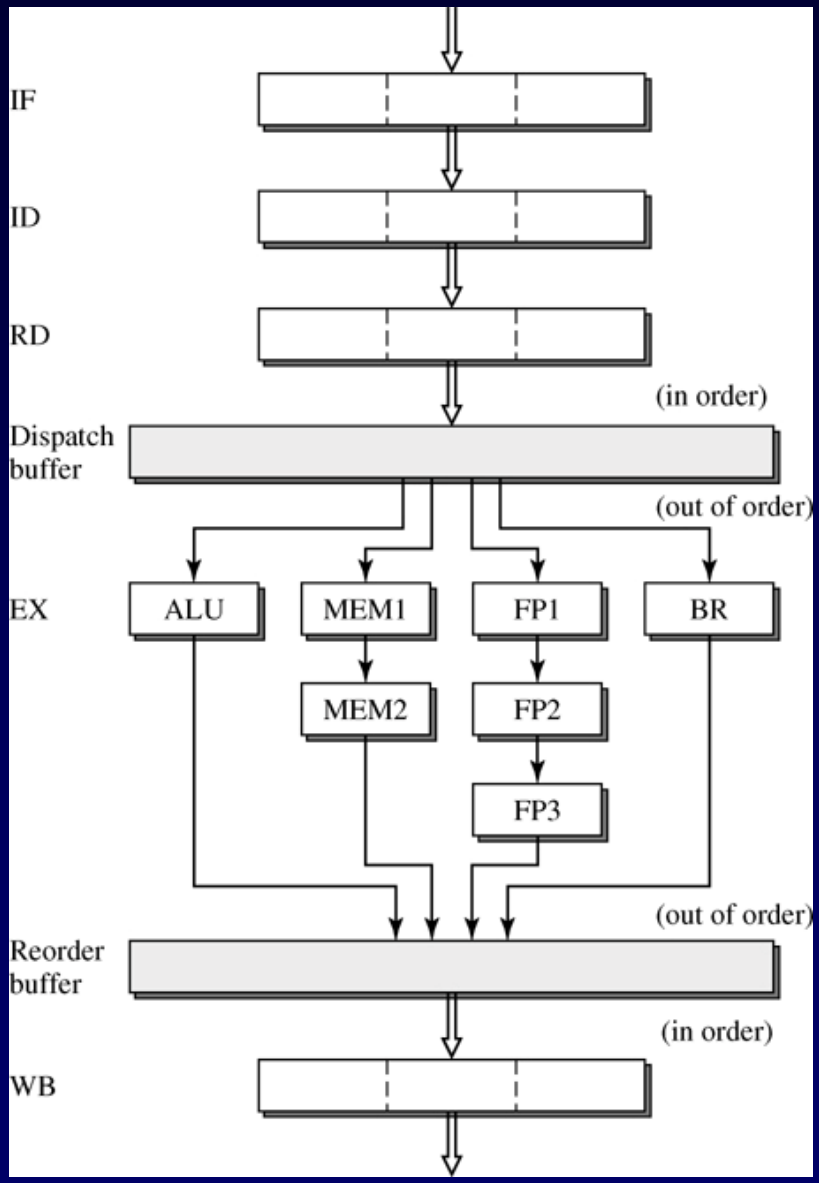
single-entry buffer



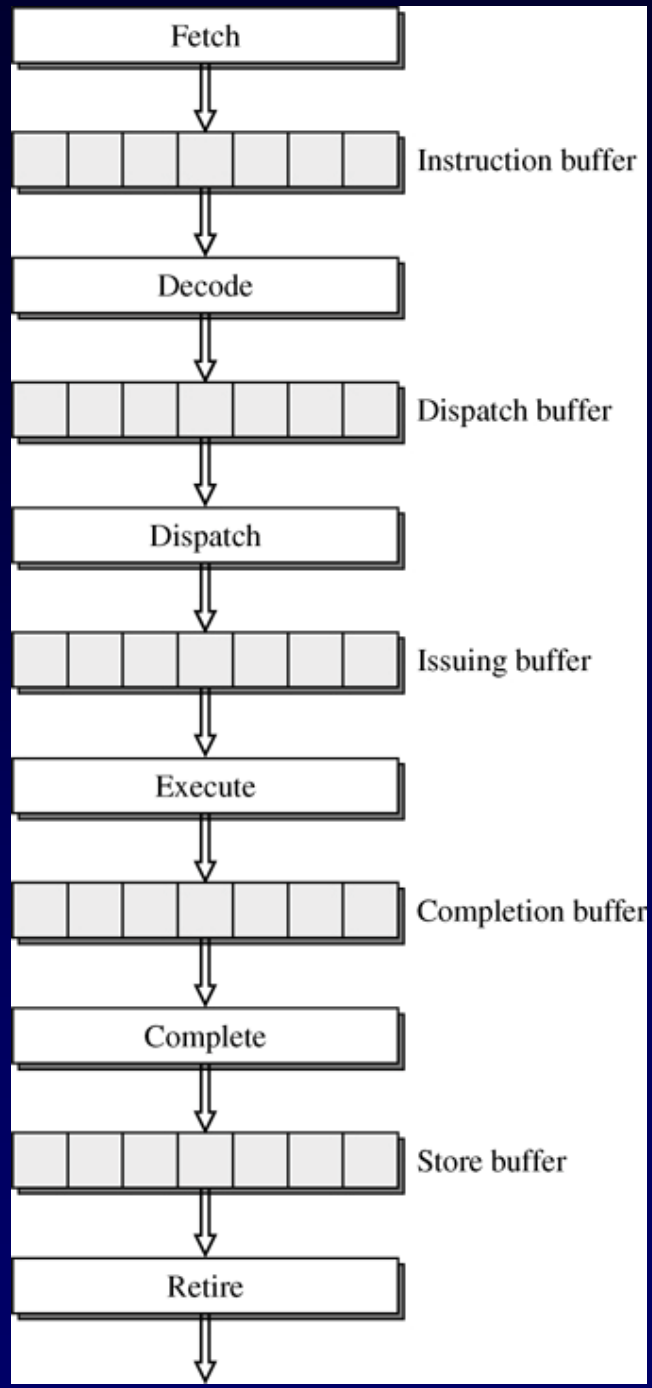
multi-entry buffer

multi-entry buffer with reordering

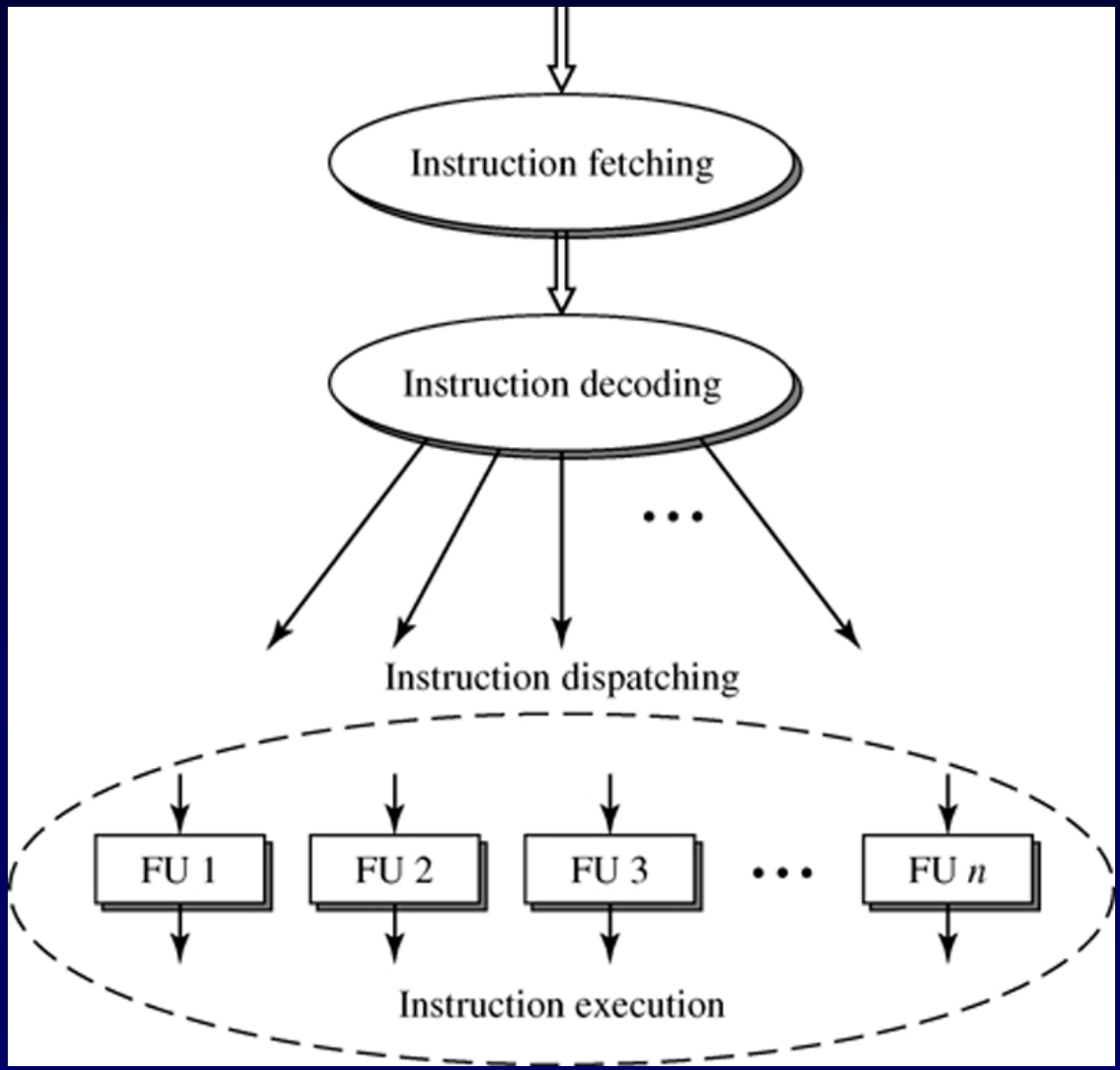
Dynamic pipeline of width *s=3*



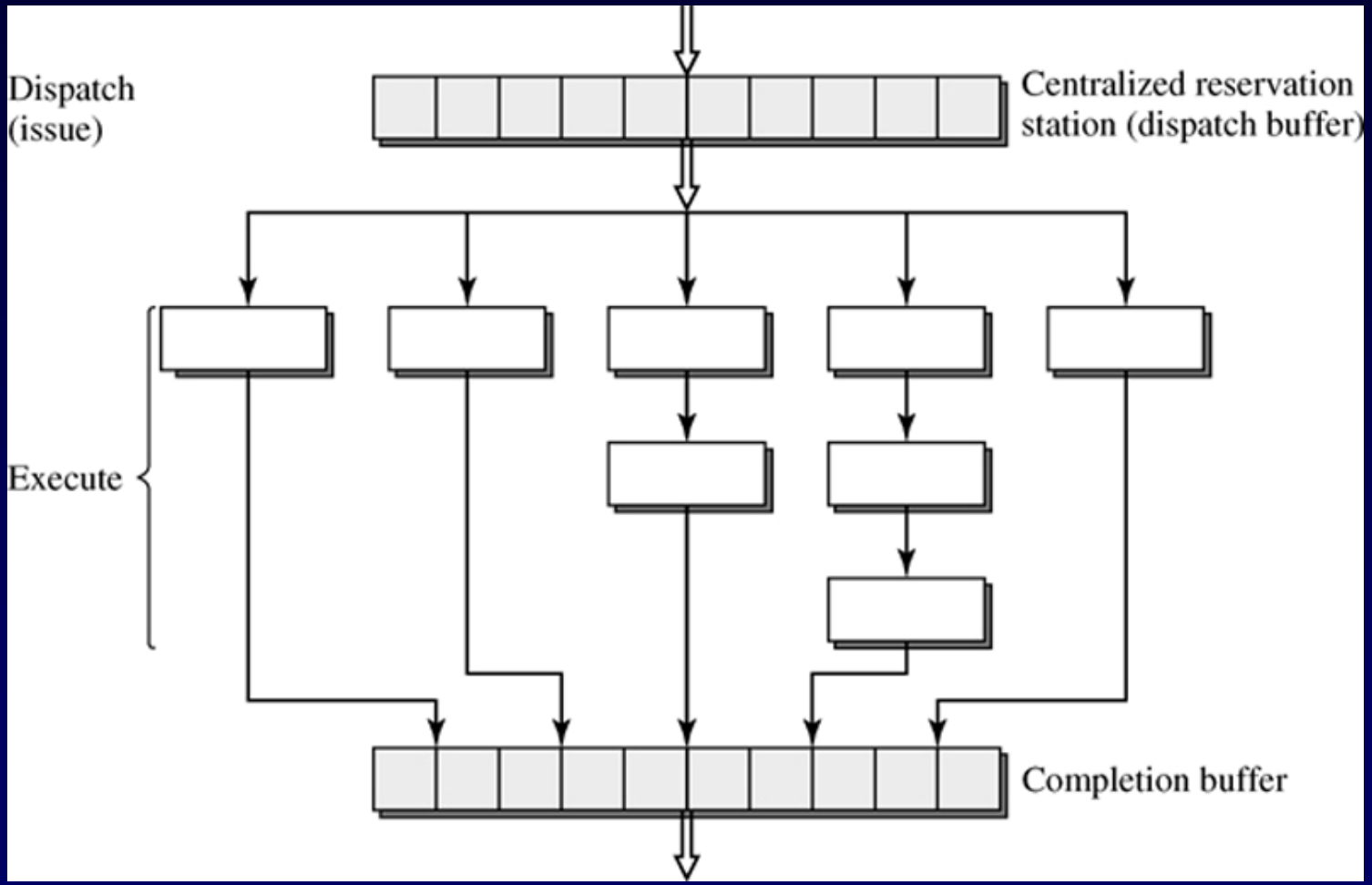
6-stage superscalar pipeline



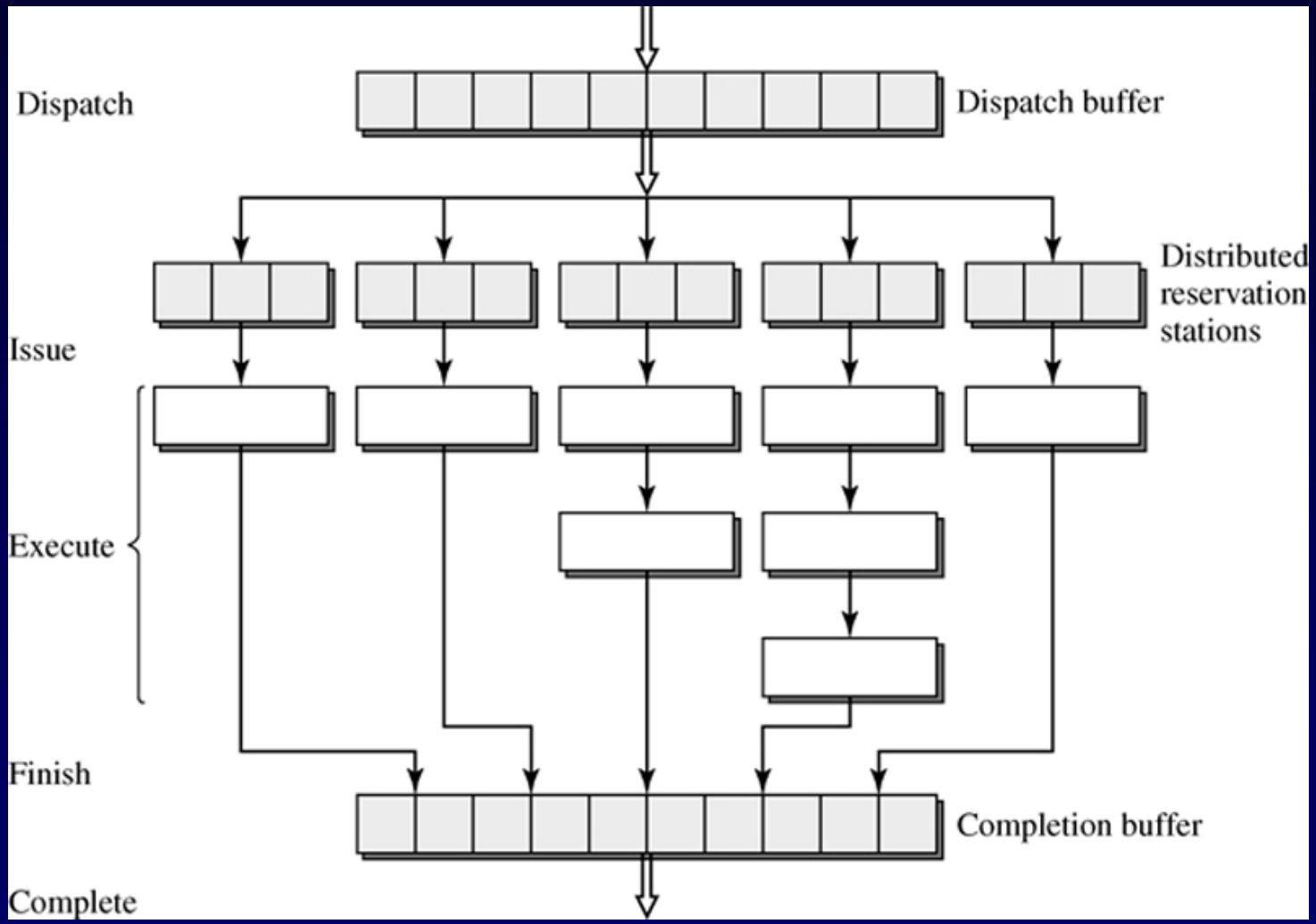
Instruction dispatching in superscalar pipeline



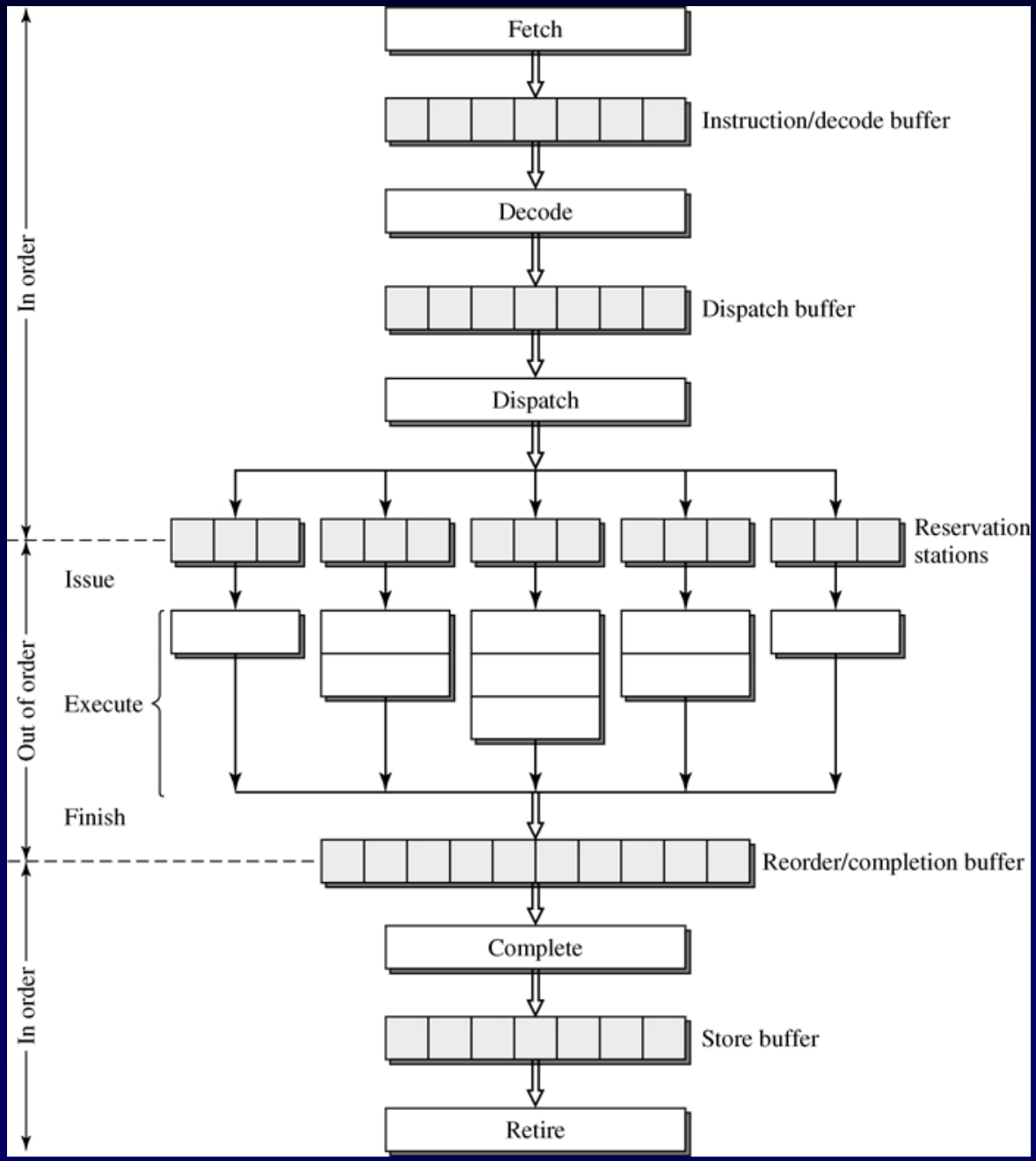
Centralized reservation station



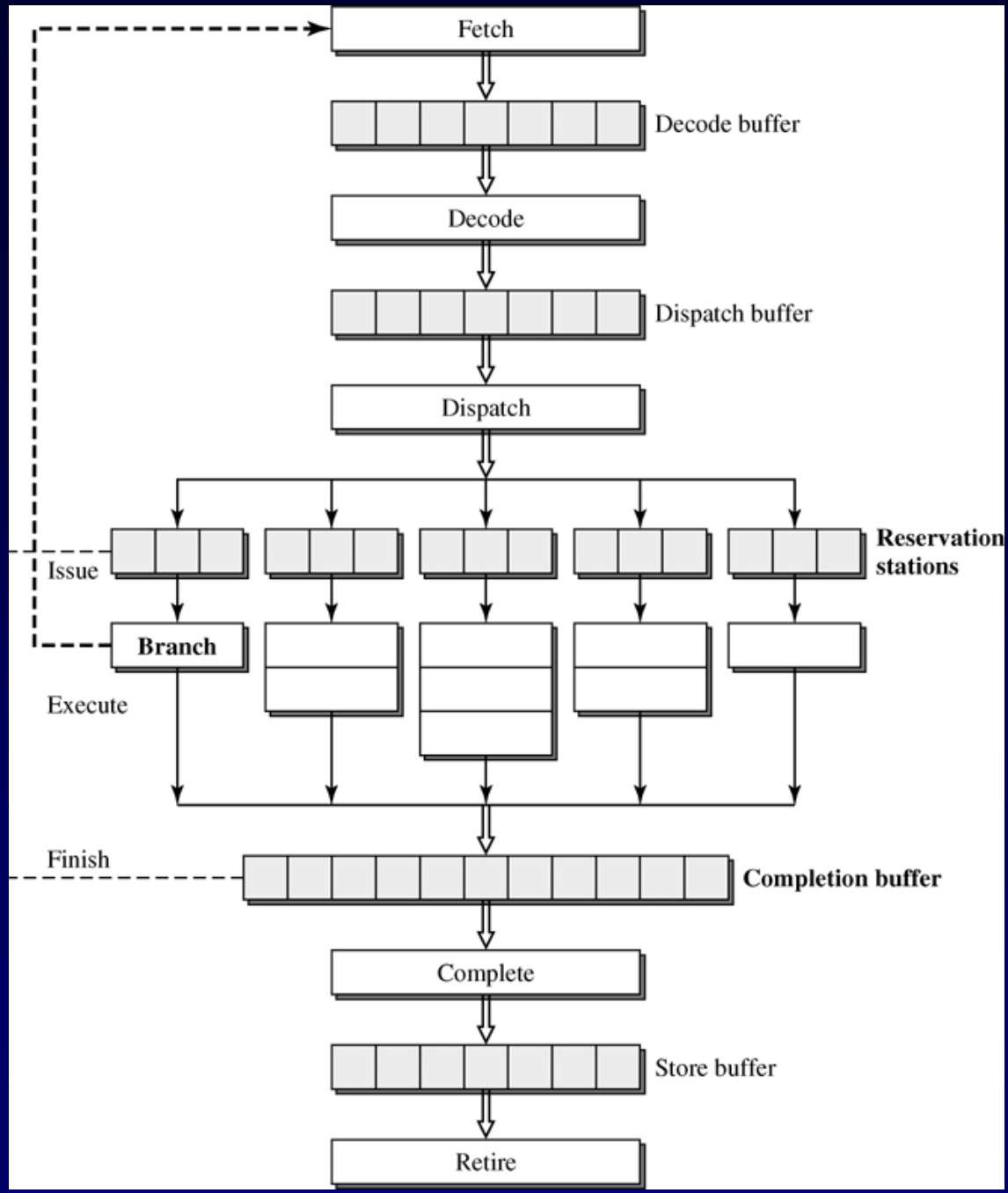
Distributed reservation stations



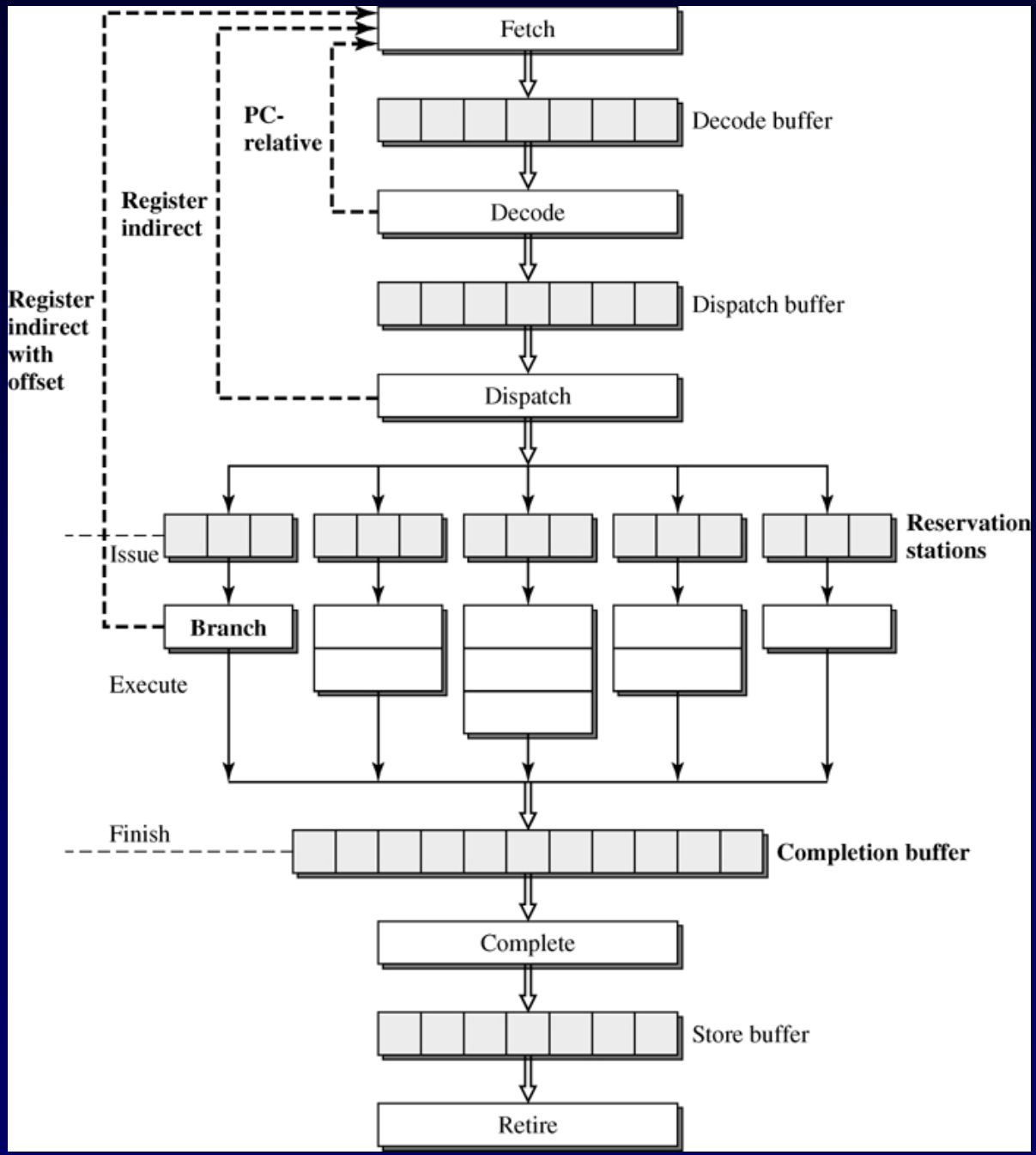
Dynamic pipeline with reservation stations and reorder buffer



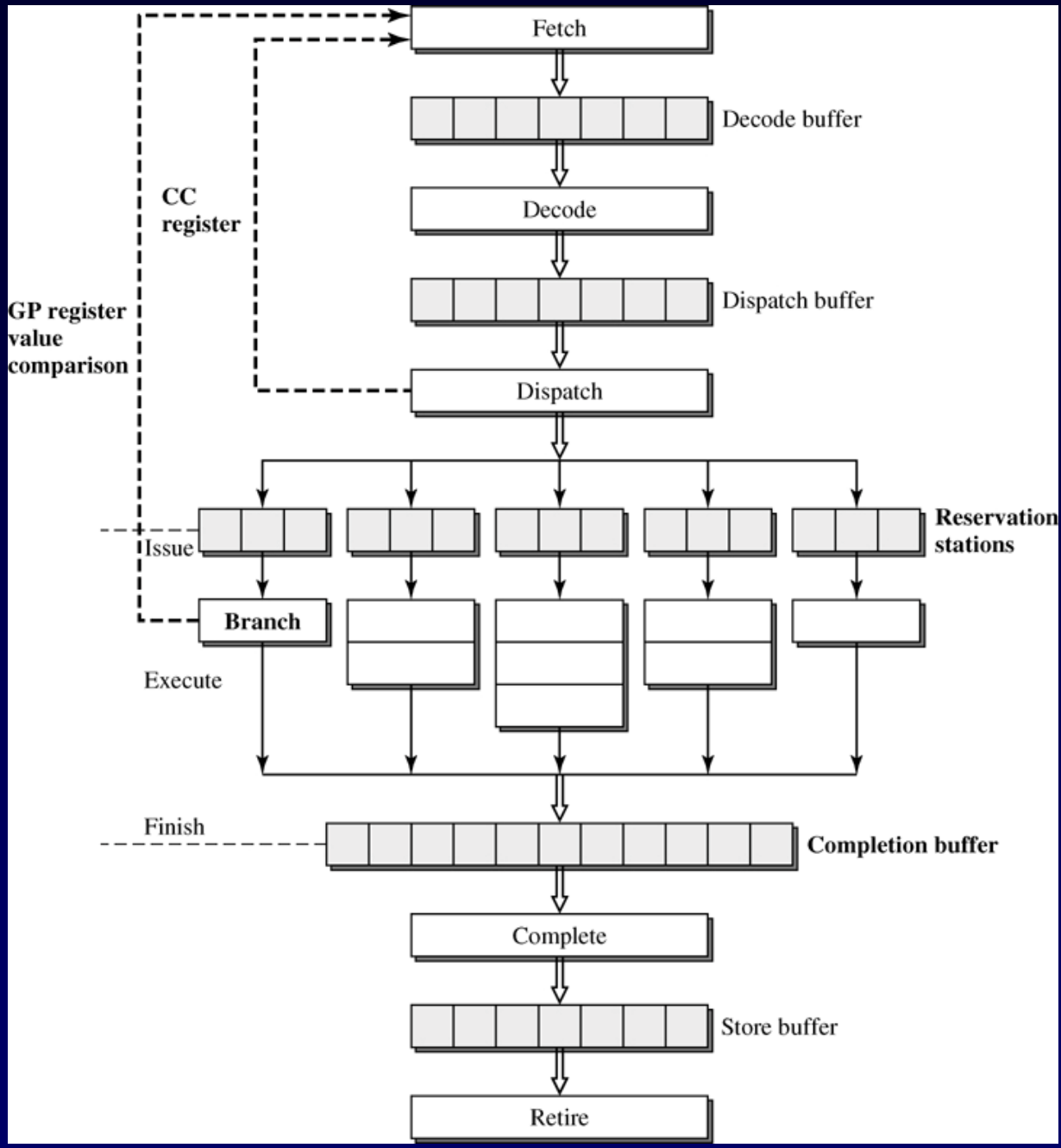
Disruption of sequential control flow by Branch instructions



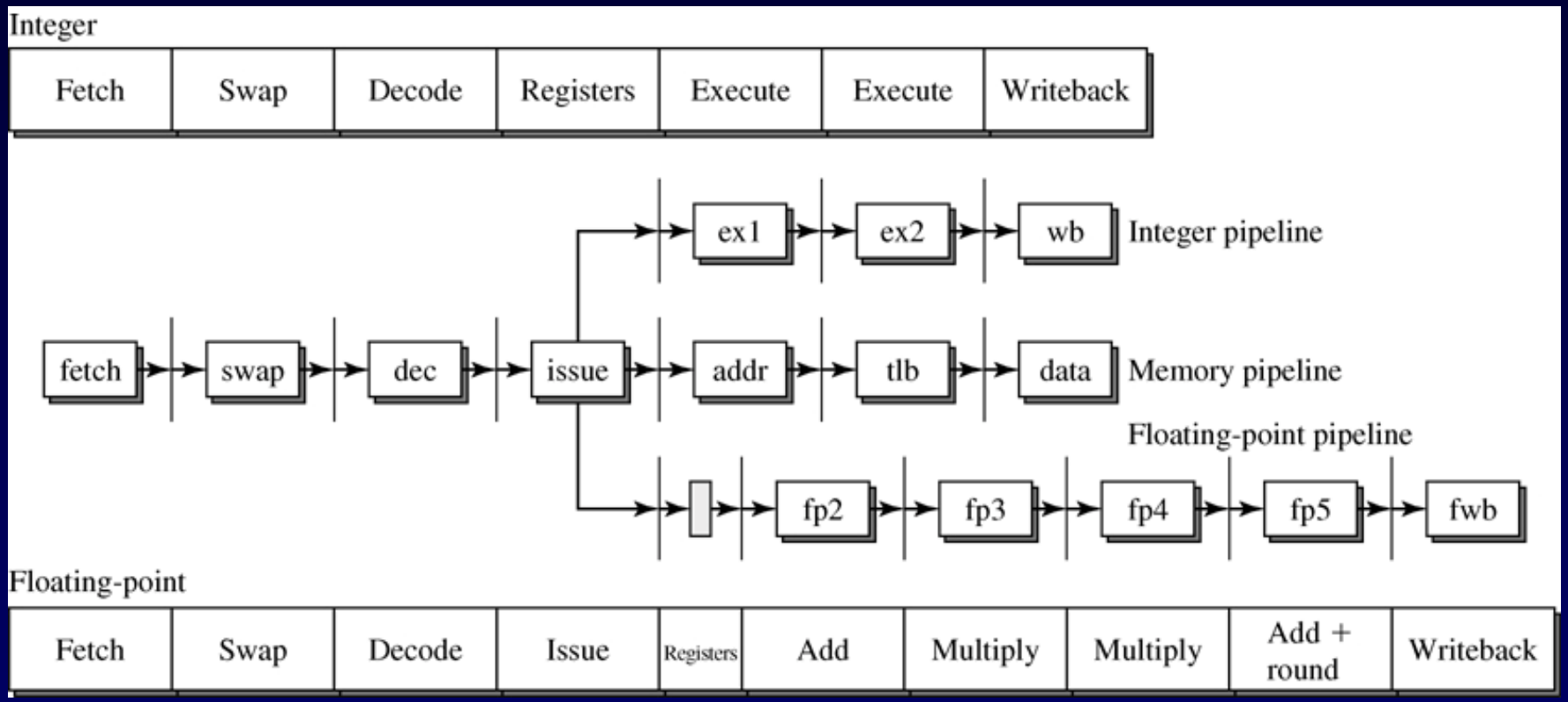
Branch target address generation penalties



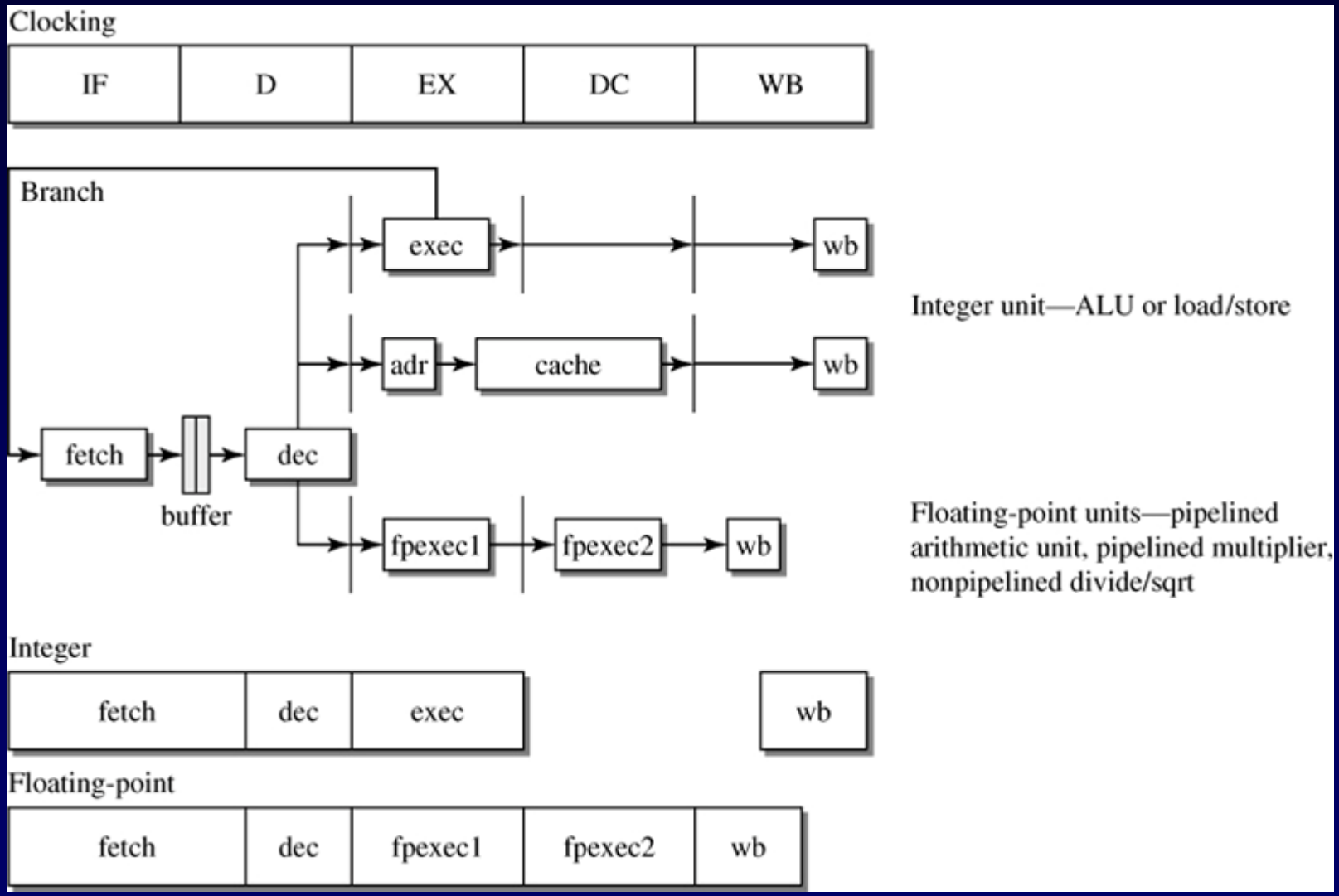
Branch condition resolution penalties



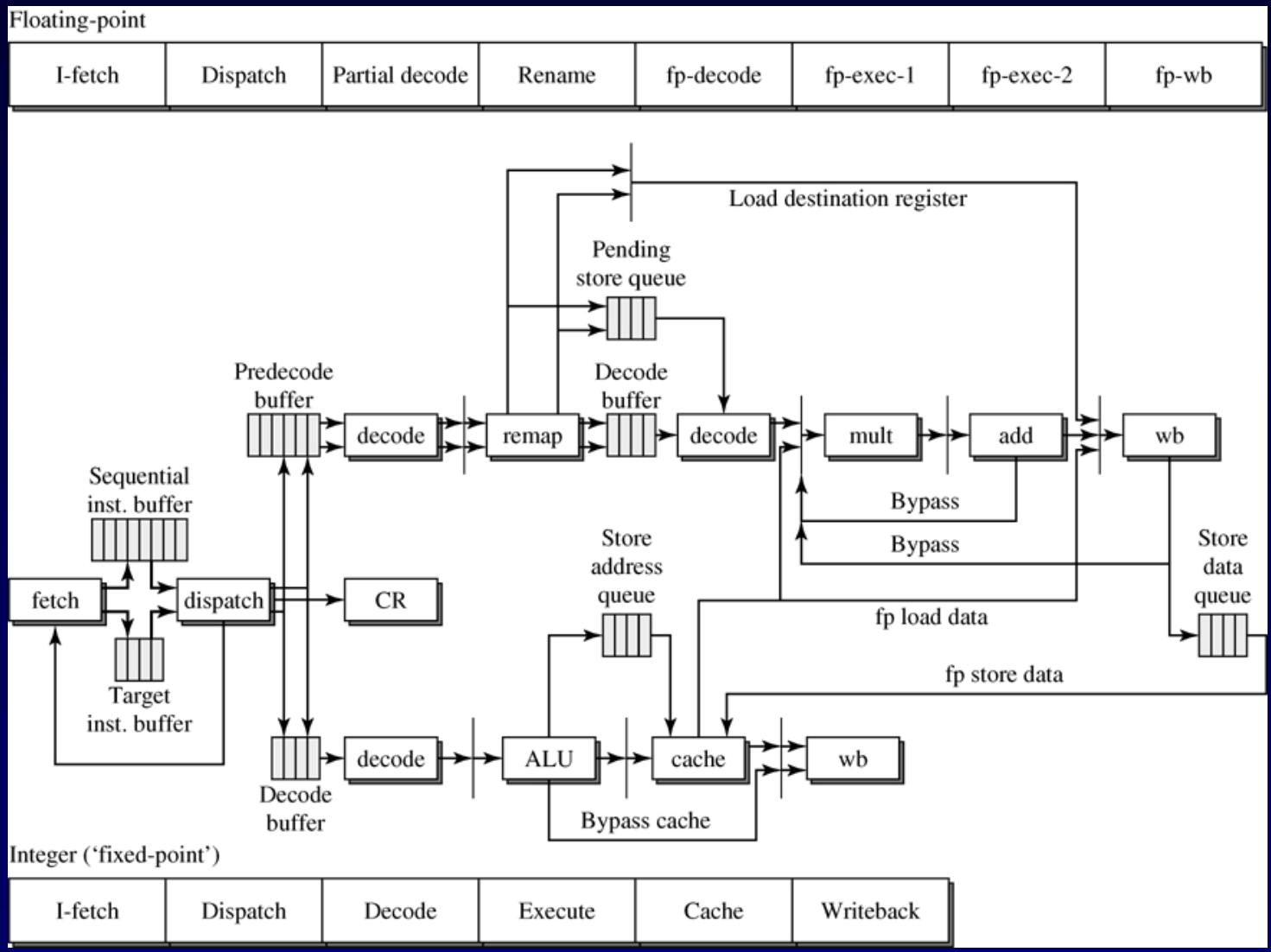
Alpha 21064 pipeline stages



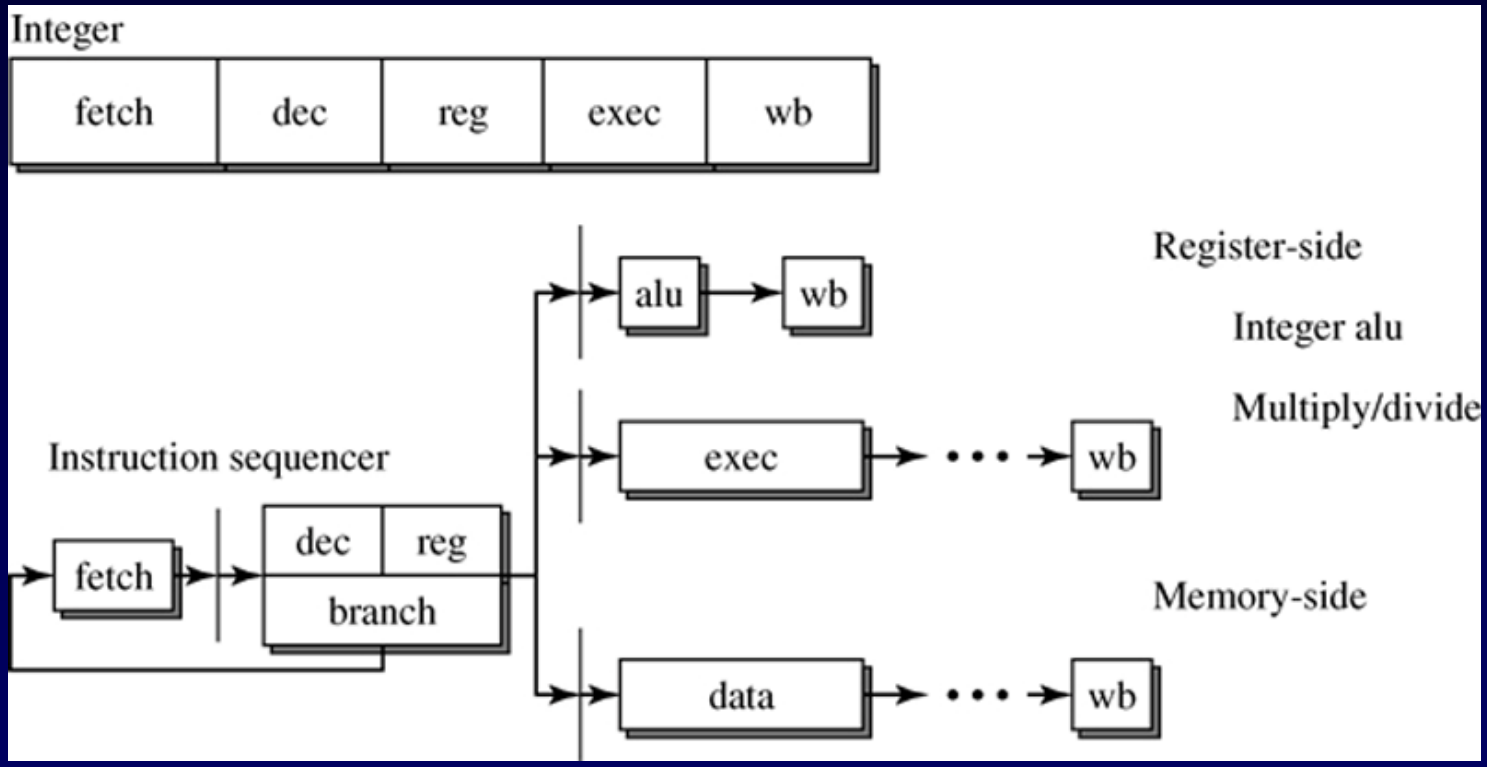
HP PA 7100 pipeline stages



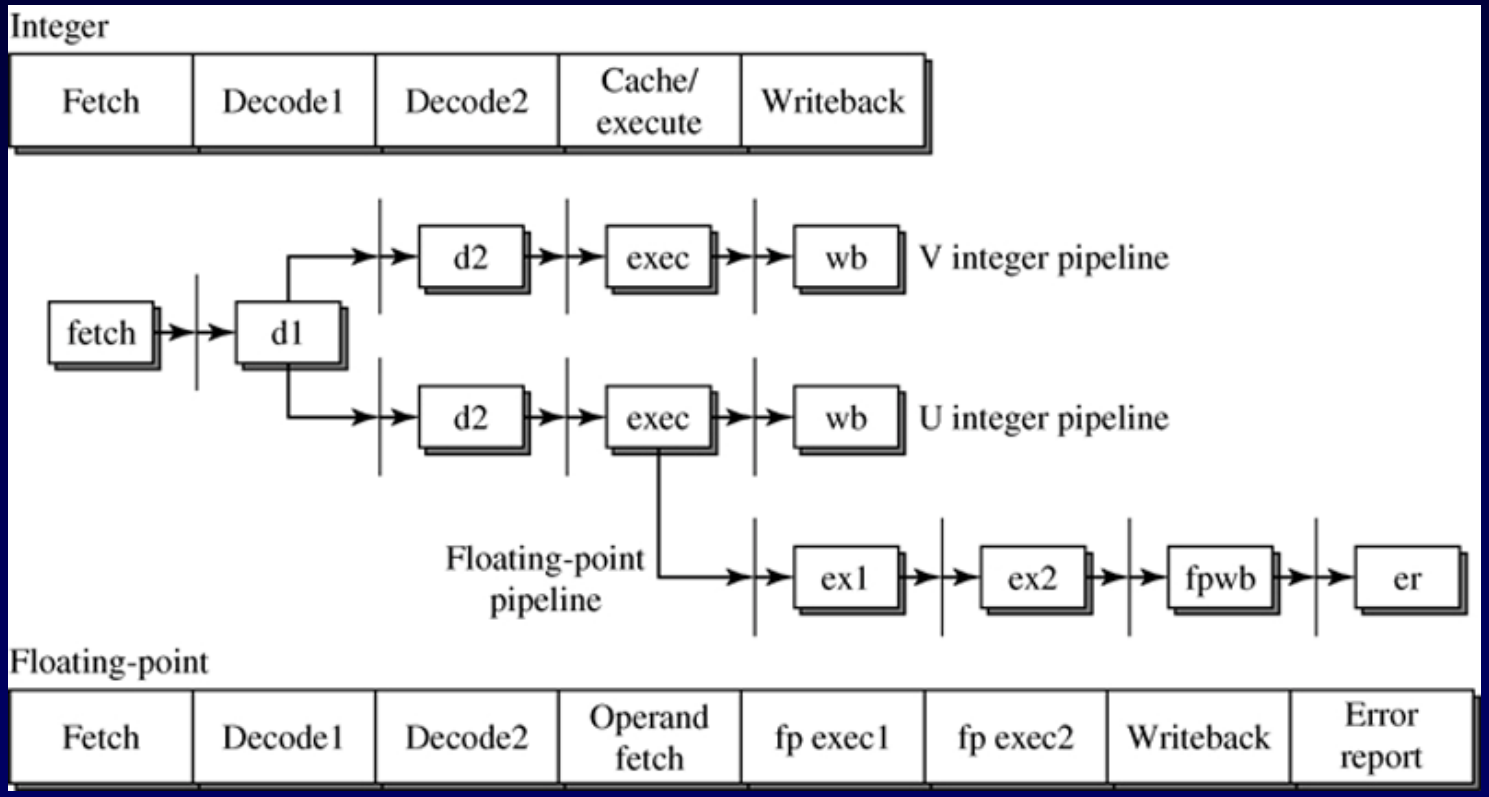
IBM POWER (RIOS) pipeline stages



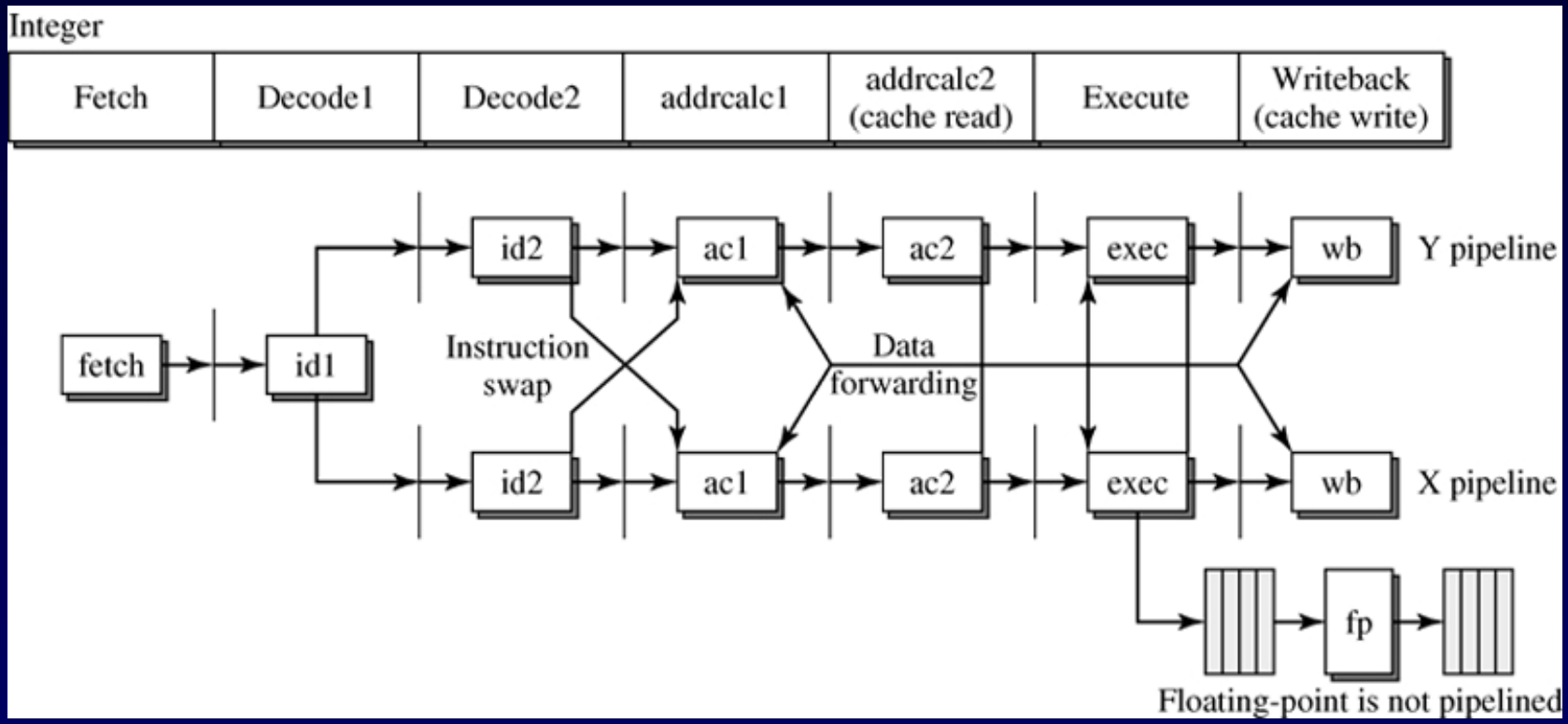
Intel i960CA pipeline stages



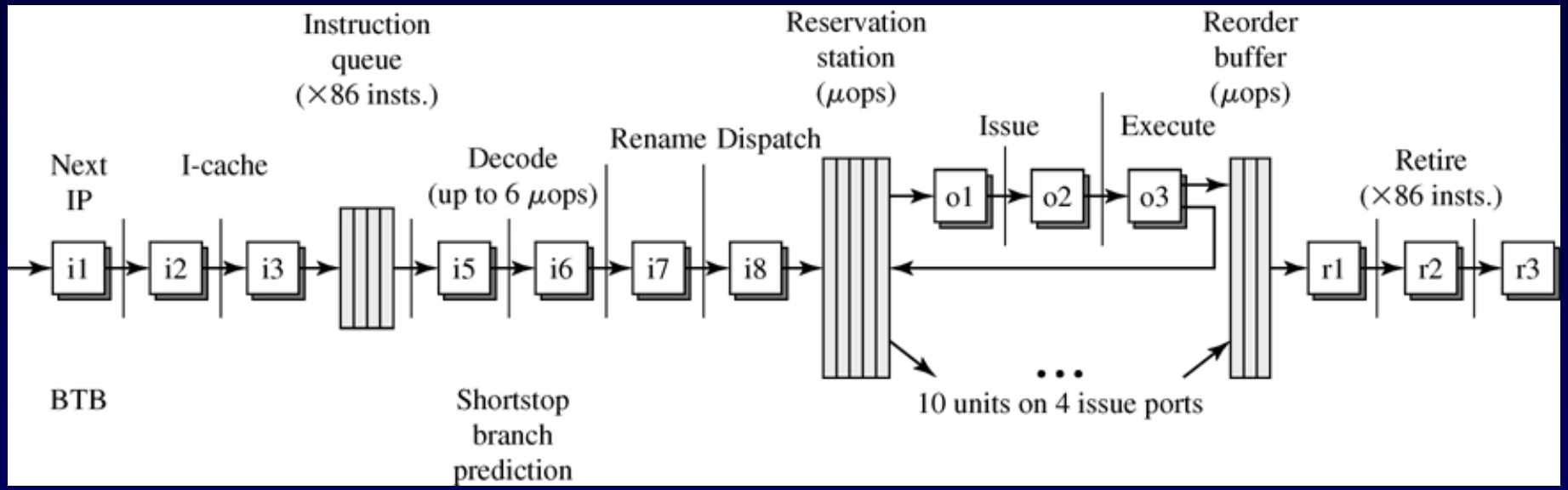
Intel Pentium pipeline stages



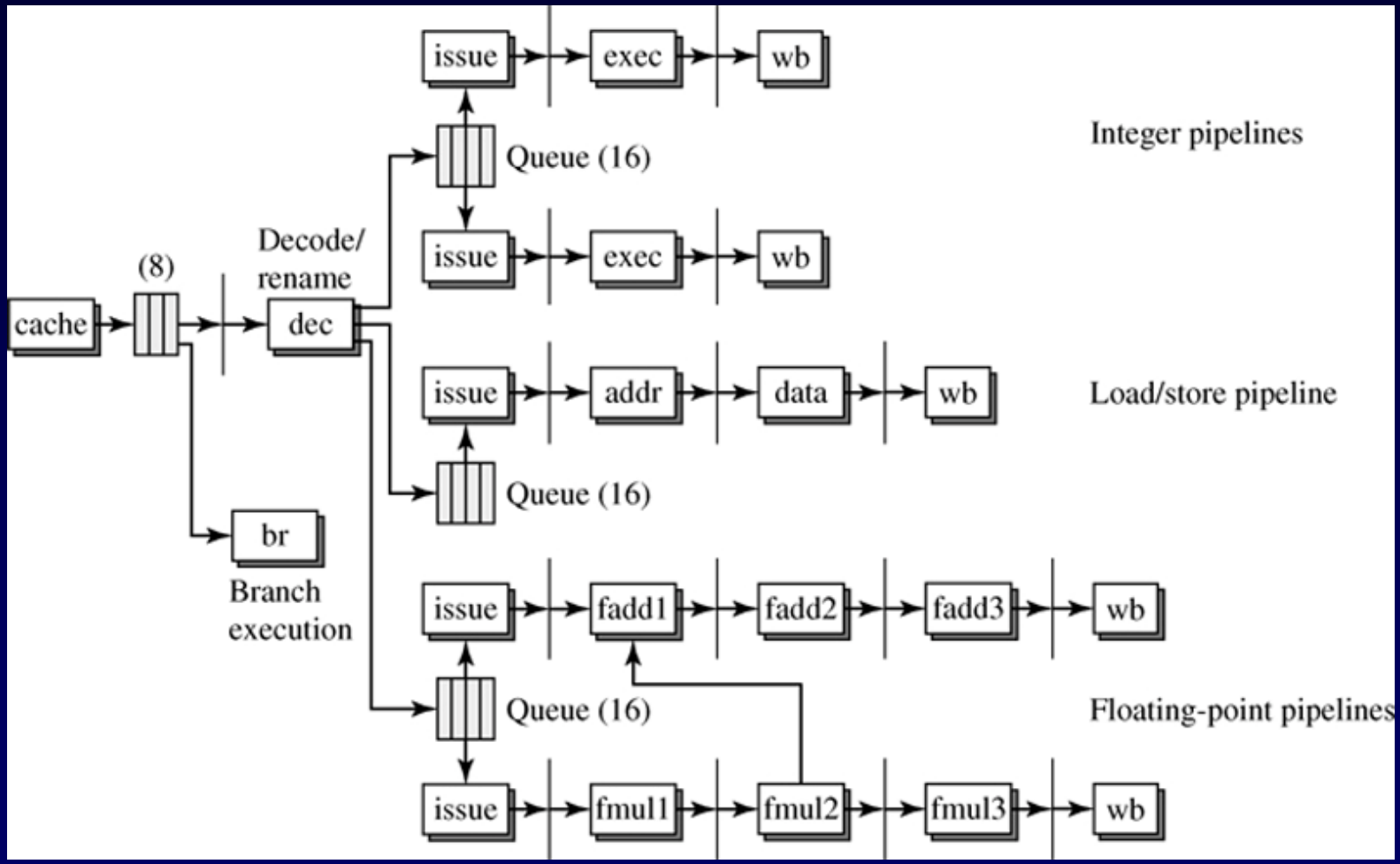
Cyrix 6x86 pipeline stages



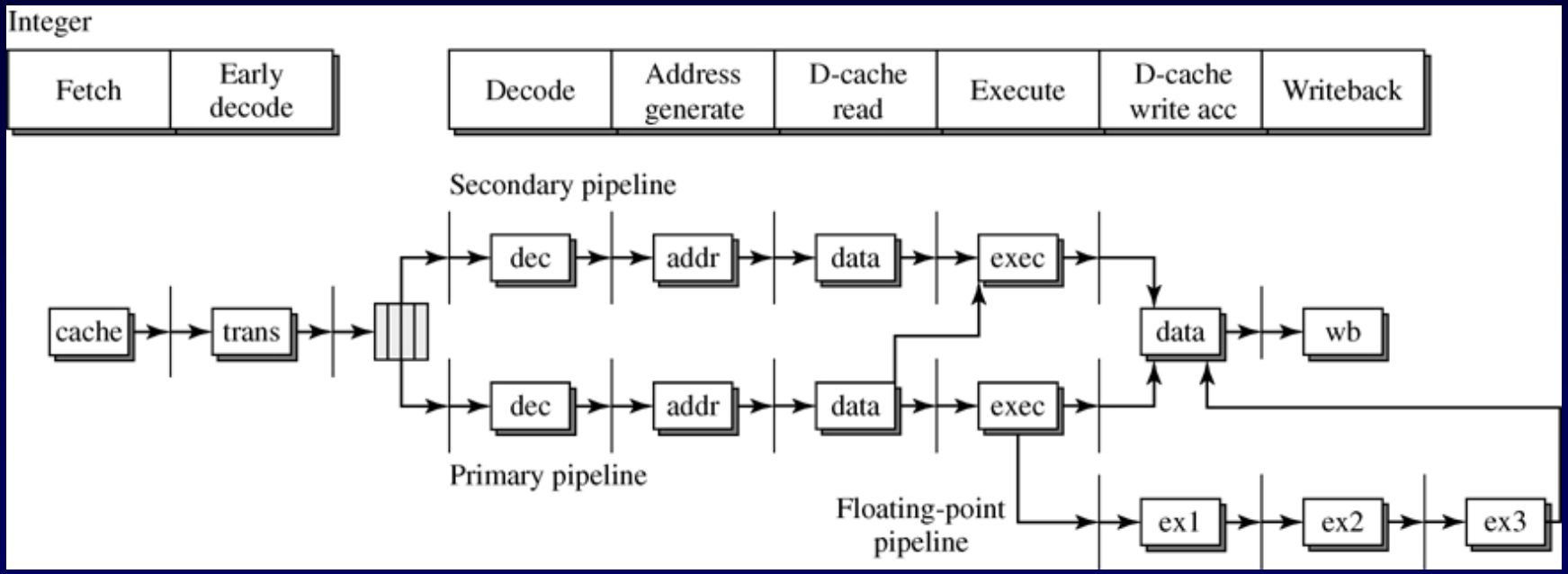
Intel P6 pipeline stages



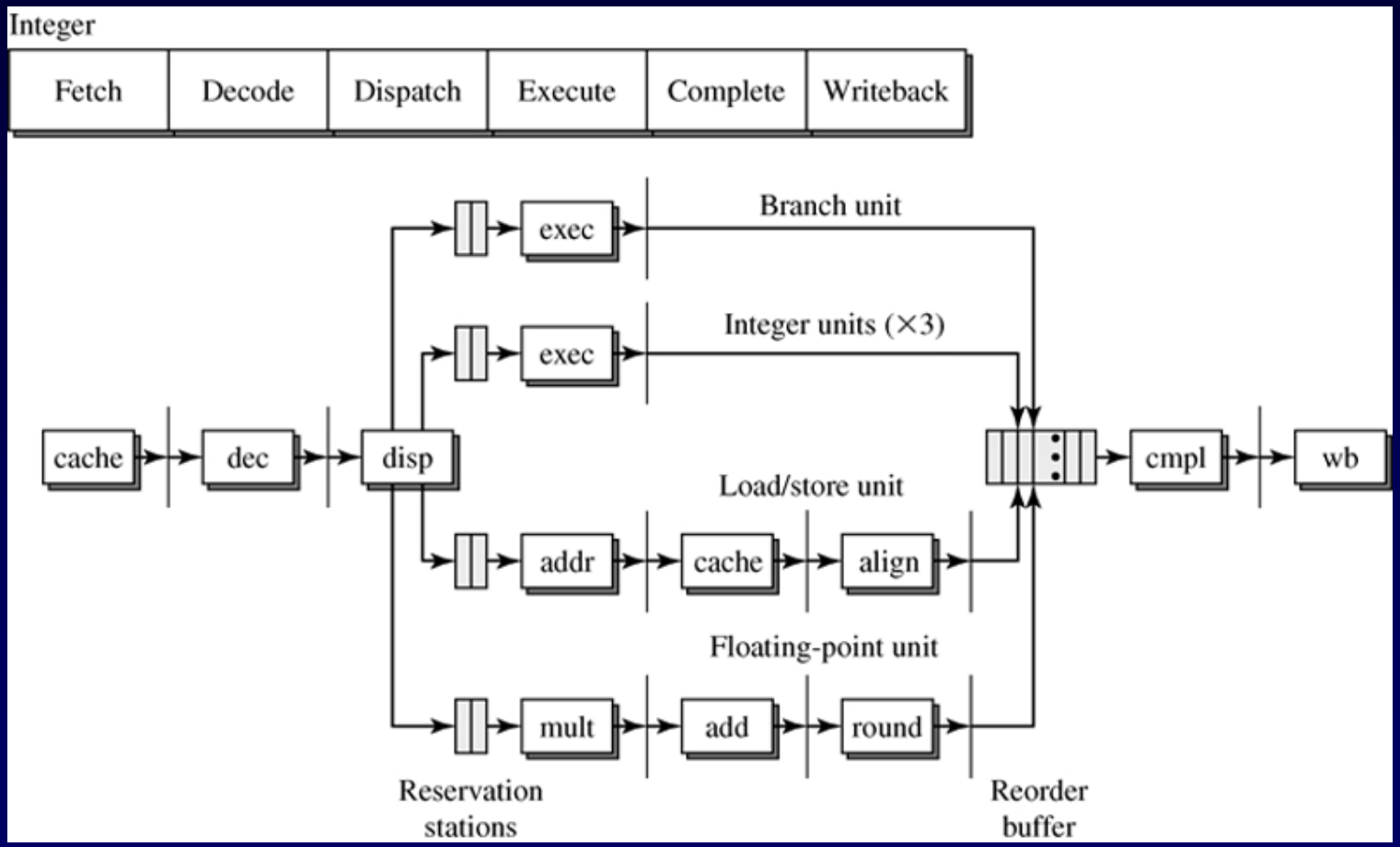
MIPS R10000 pipeline stages



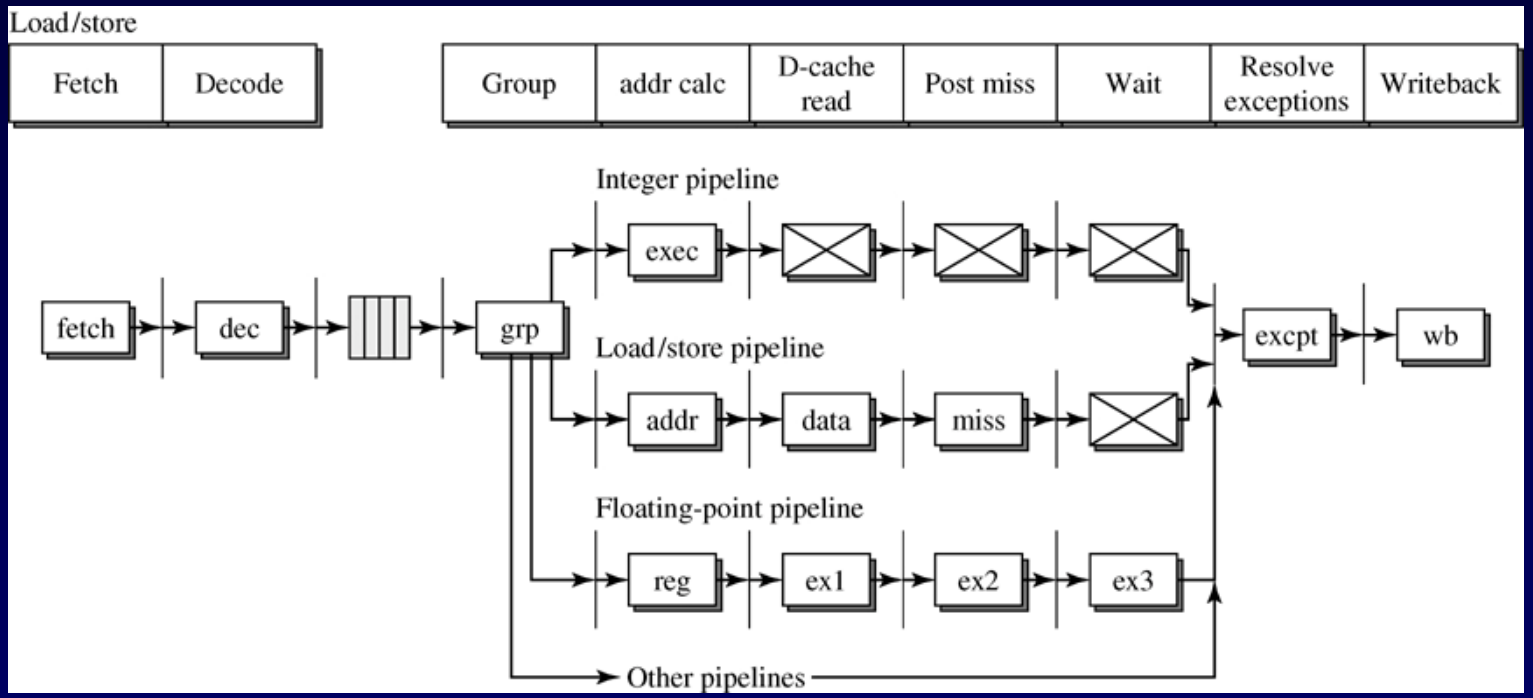
Motorola MC68060 pipeline stages



IBM/Motorola PowerPC 604 pipeline stages



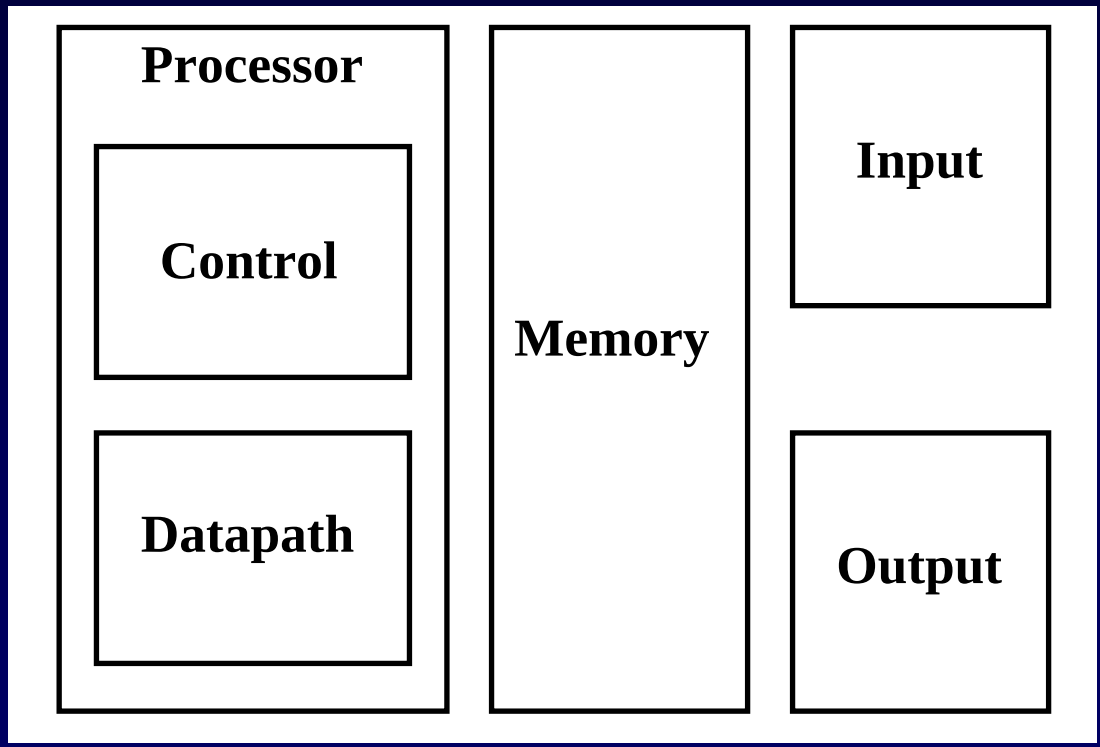
Sun UltraSPARC-I pipeline stages



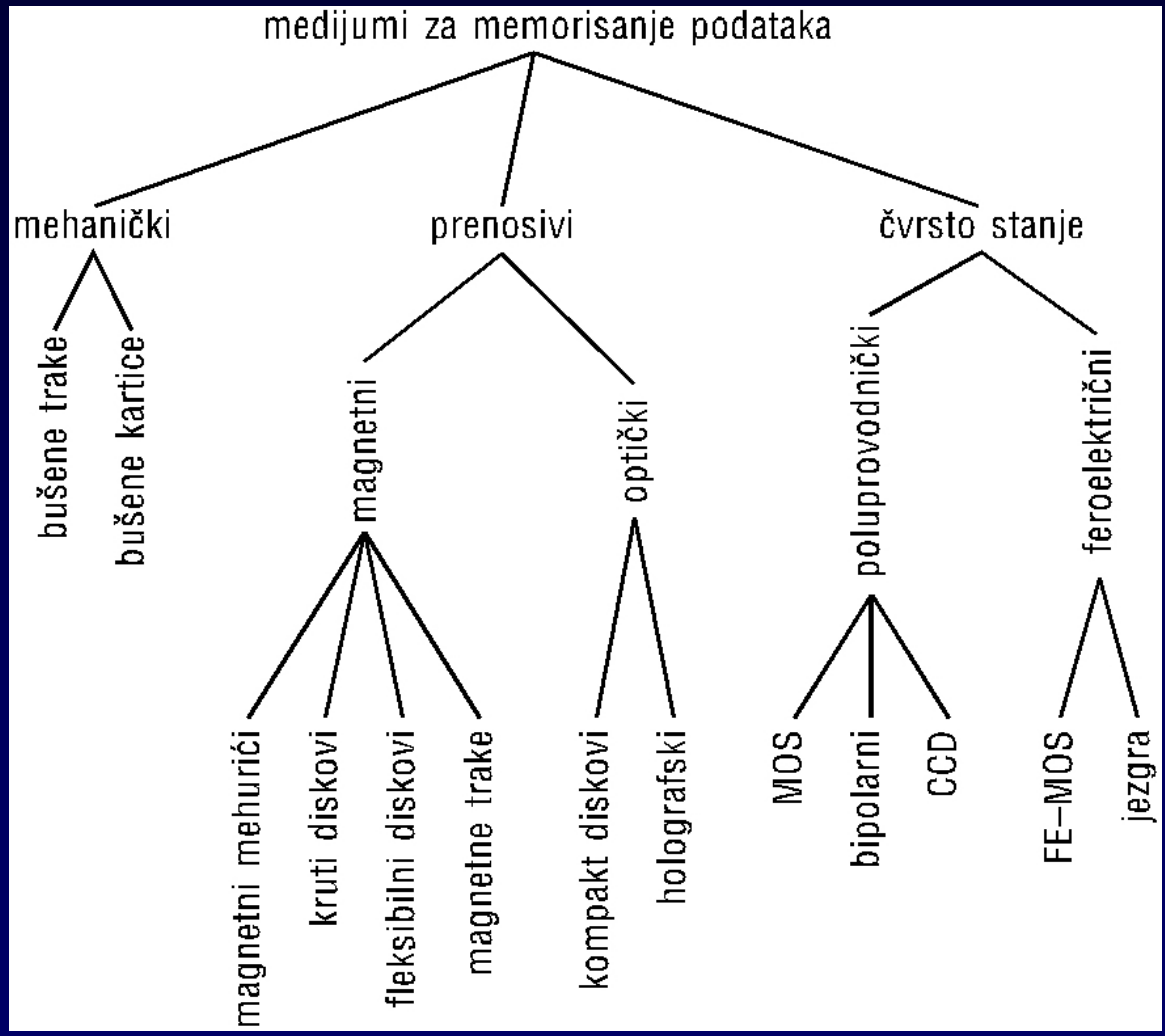
Memorijski sistem

The Big Picture

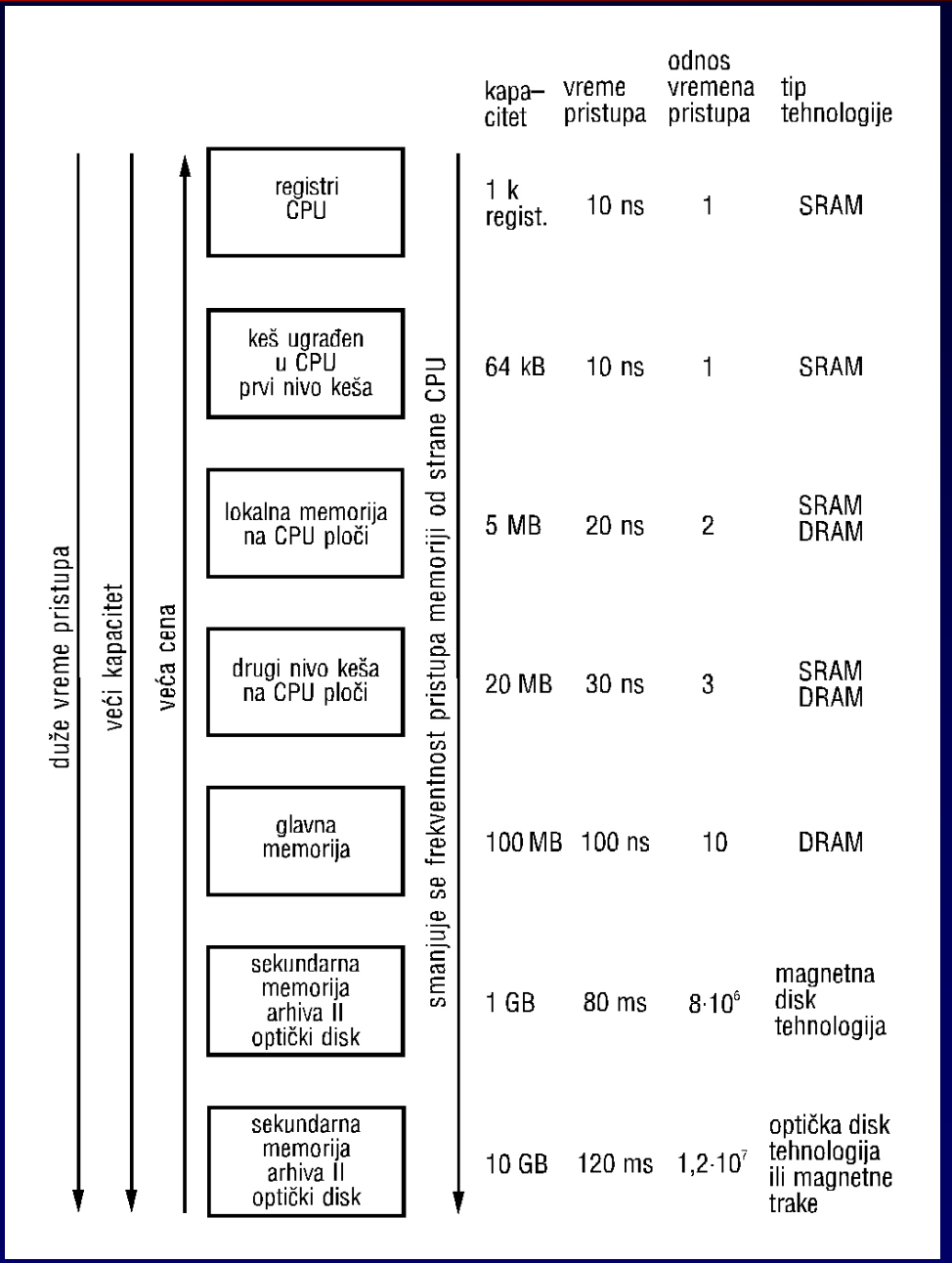
- Since 1946 all computers have had 5 components



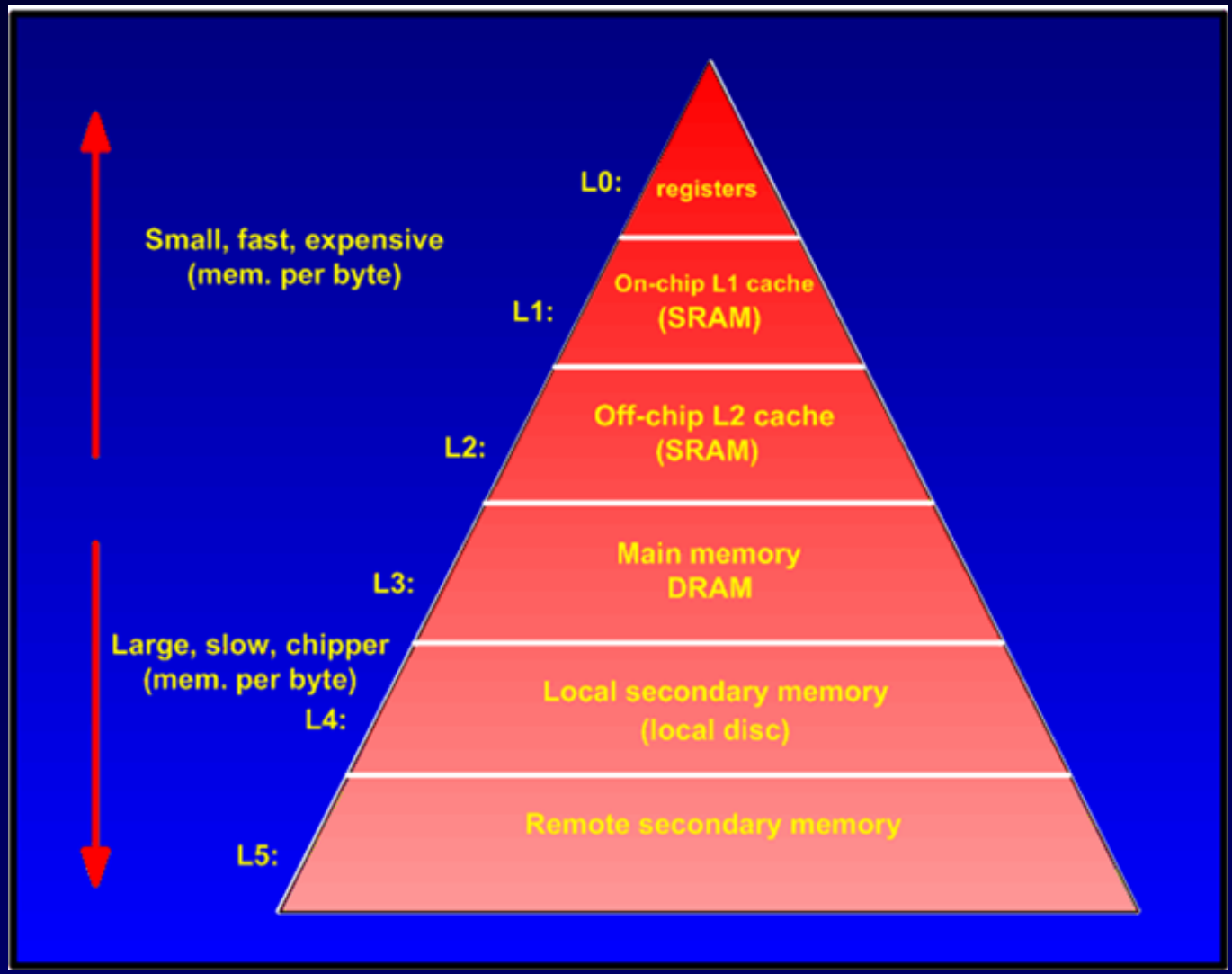
Tipovi memorija



Hijerarhijska organizacija memorije

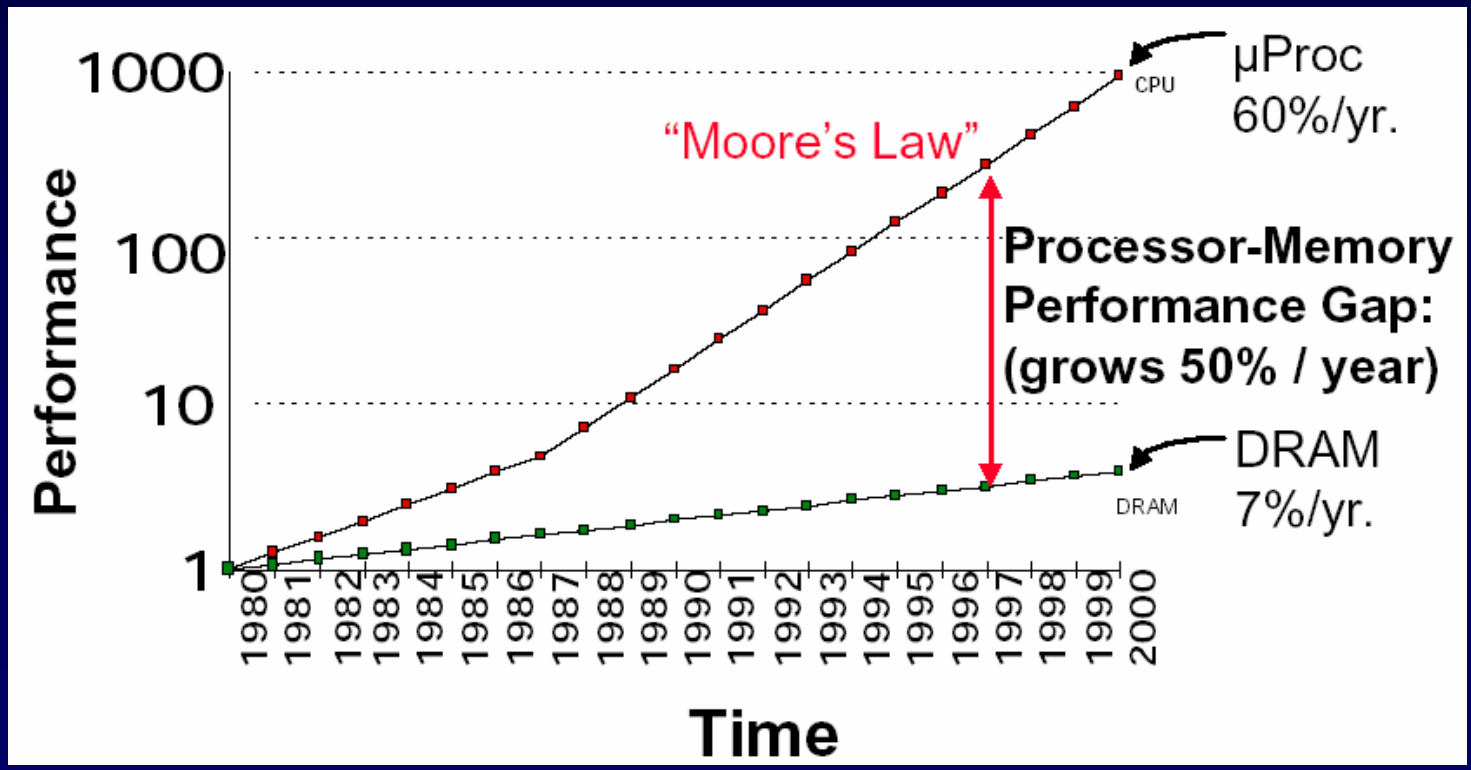


Levels in Memory Hierarchy

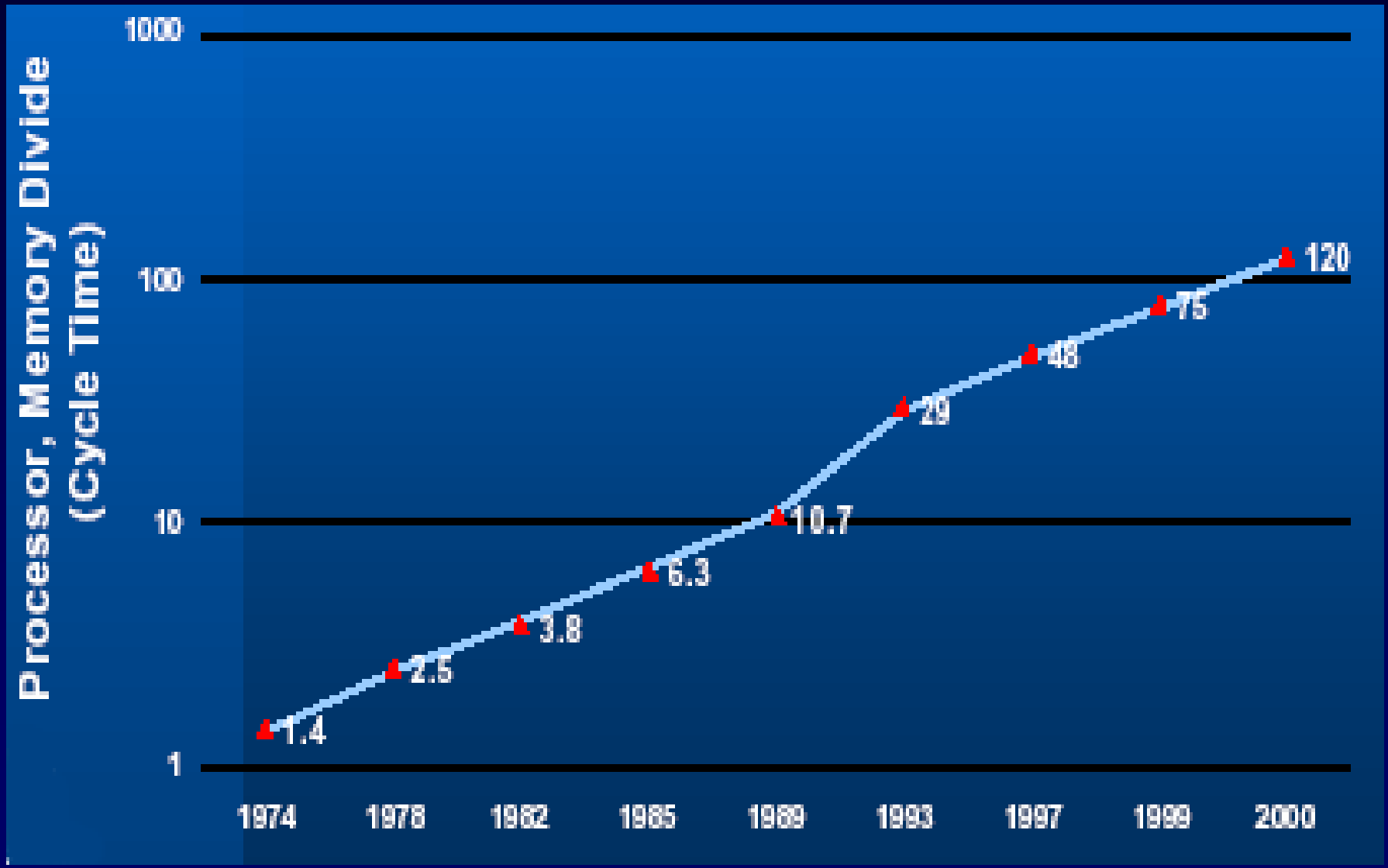


Processor-DRAM Gap

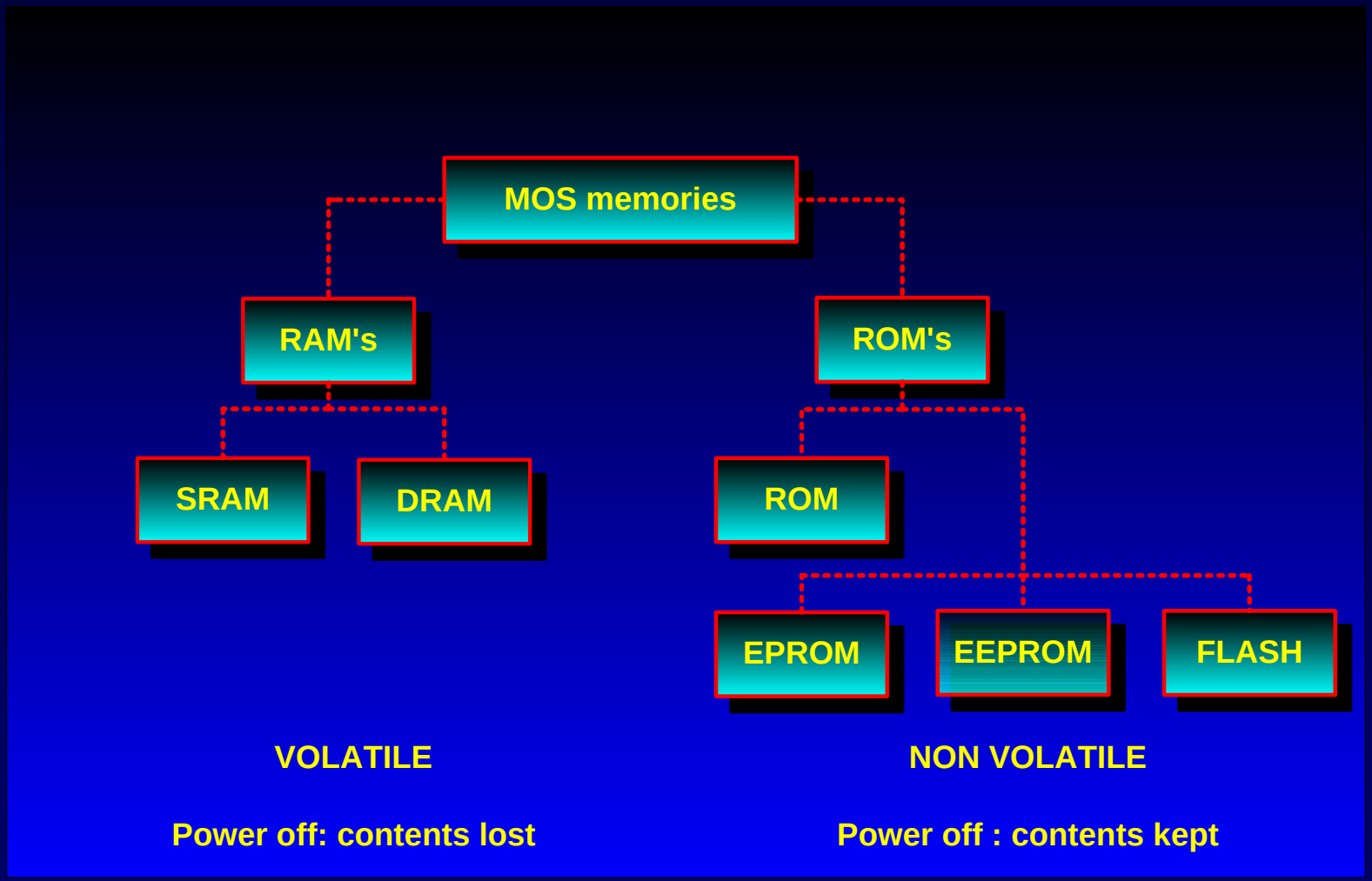
- Microprocessor performance improved 55% per year since 1987, and 35% per year until 1986
- Memory technology improvements aim primarily at increasing DRAM capacity not DRAM speed



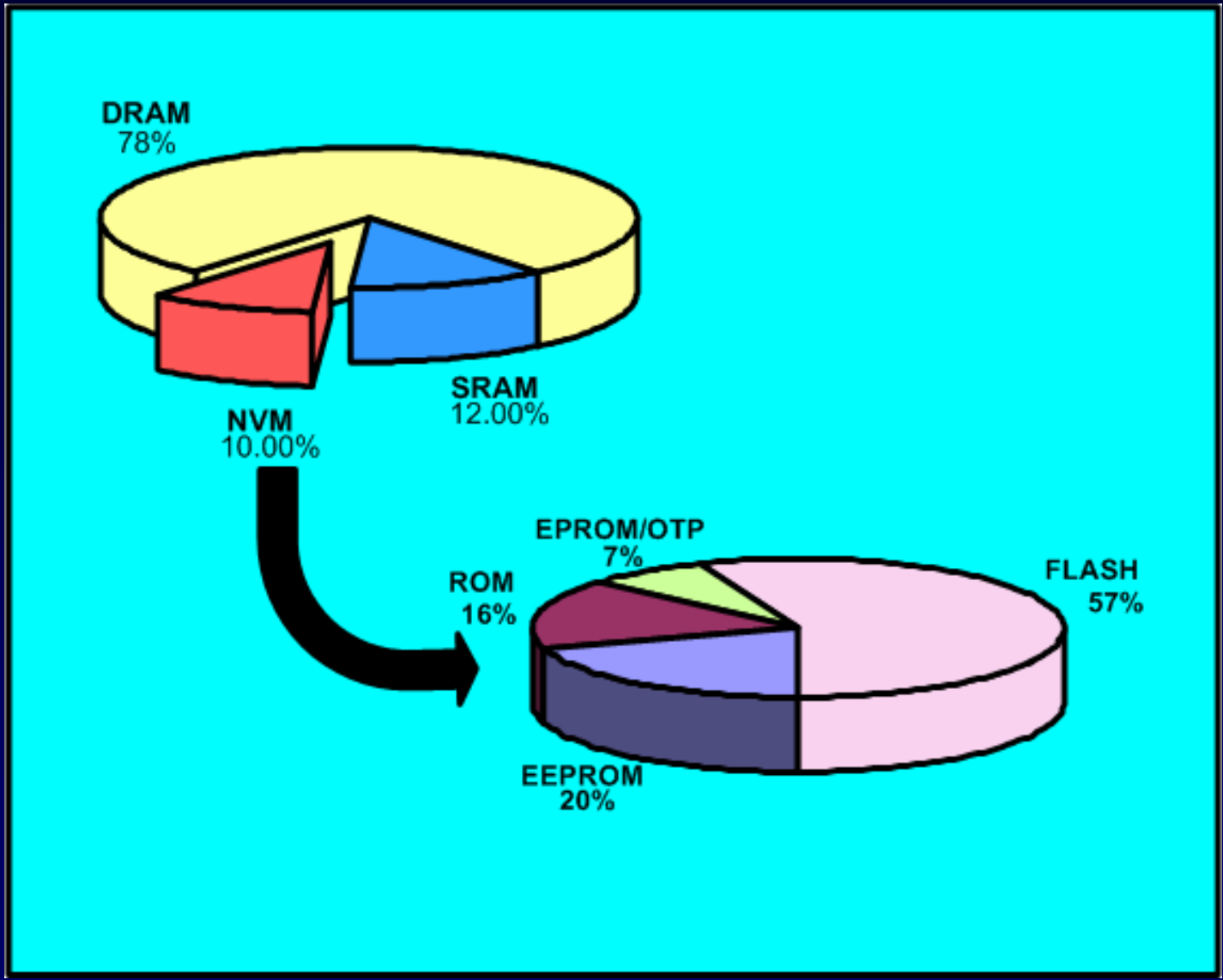
Relative processor/memory speed



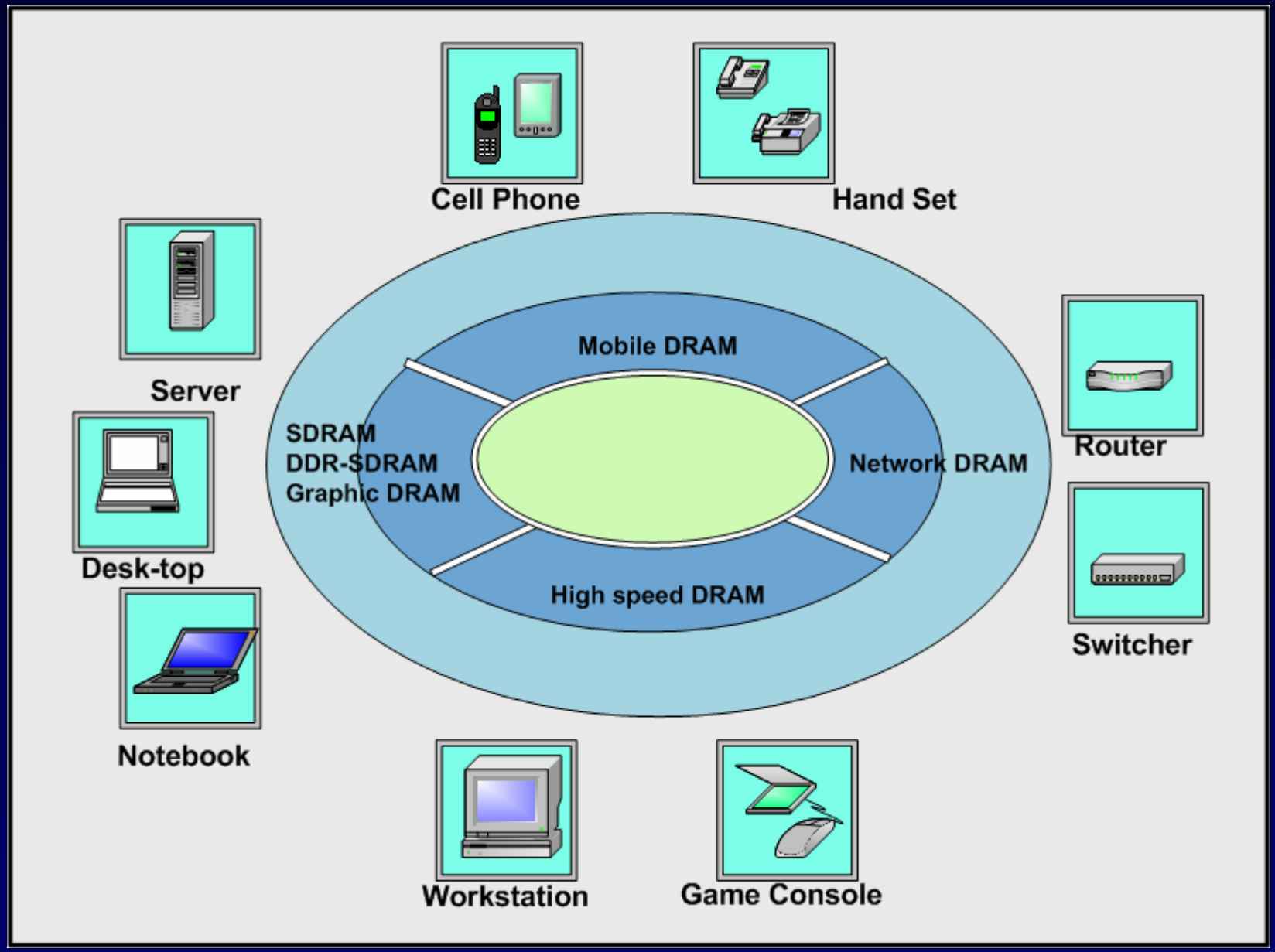
Volatile vs non Volatile Types of Memories



Percentage of Usage

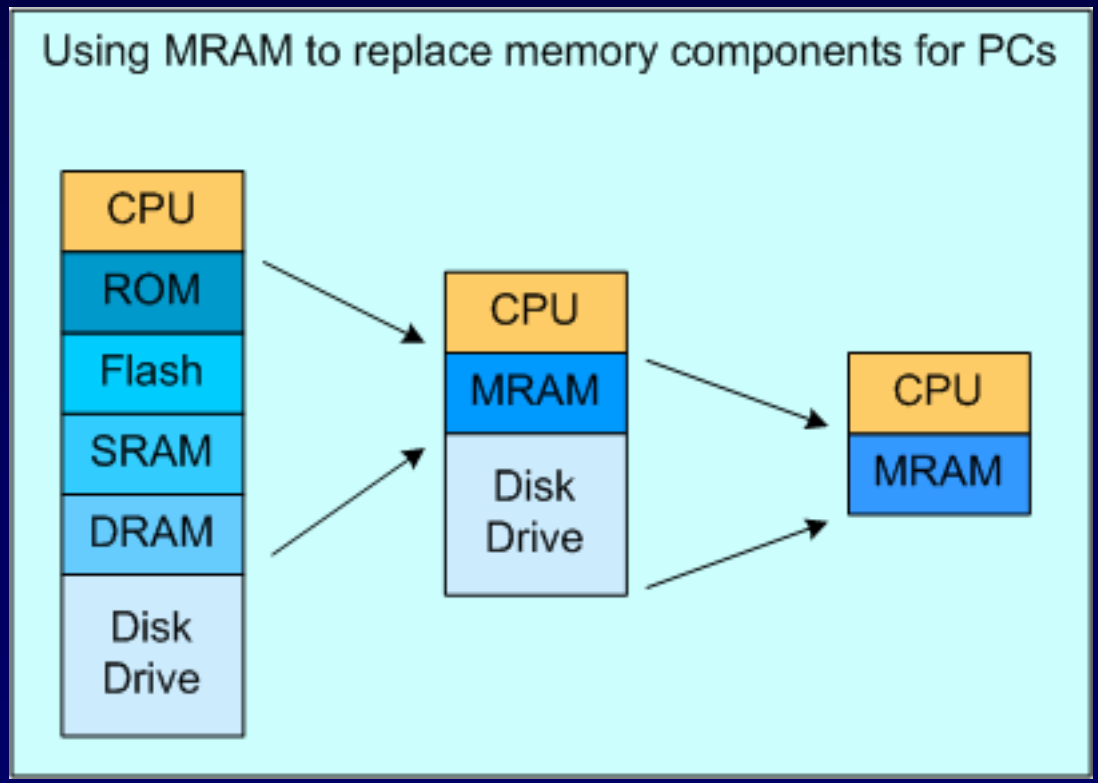


Typical Applications of DRAM

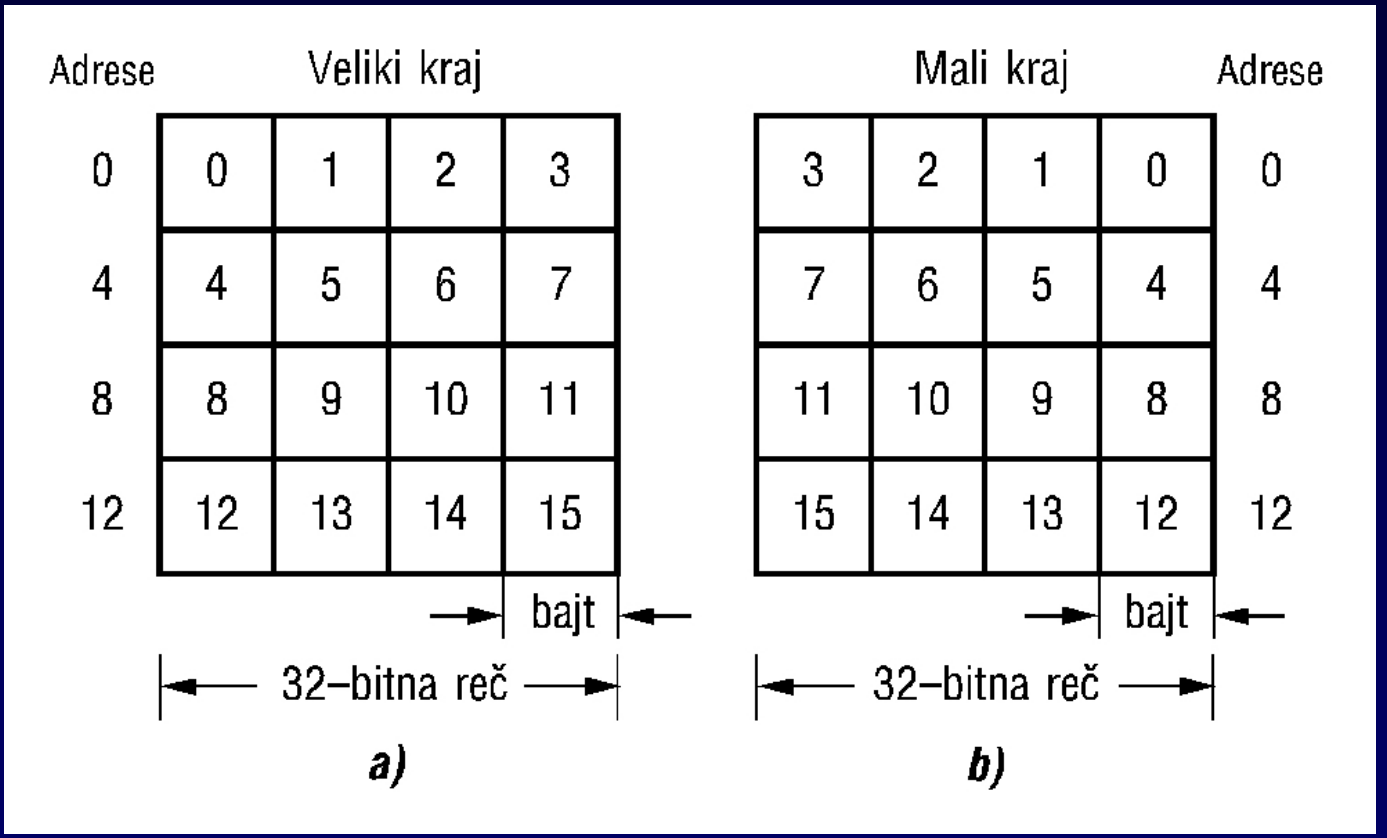


Magnetic RAM as Universal Memory

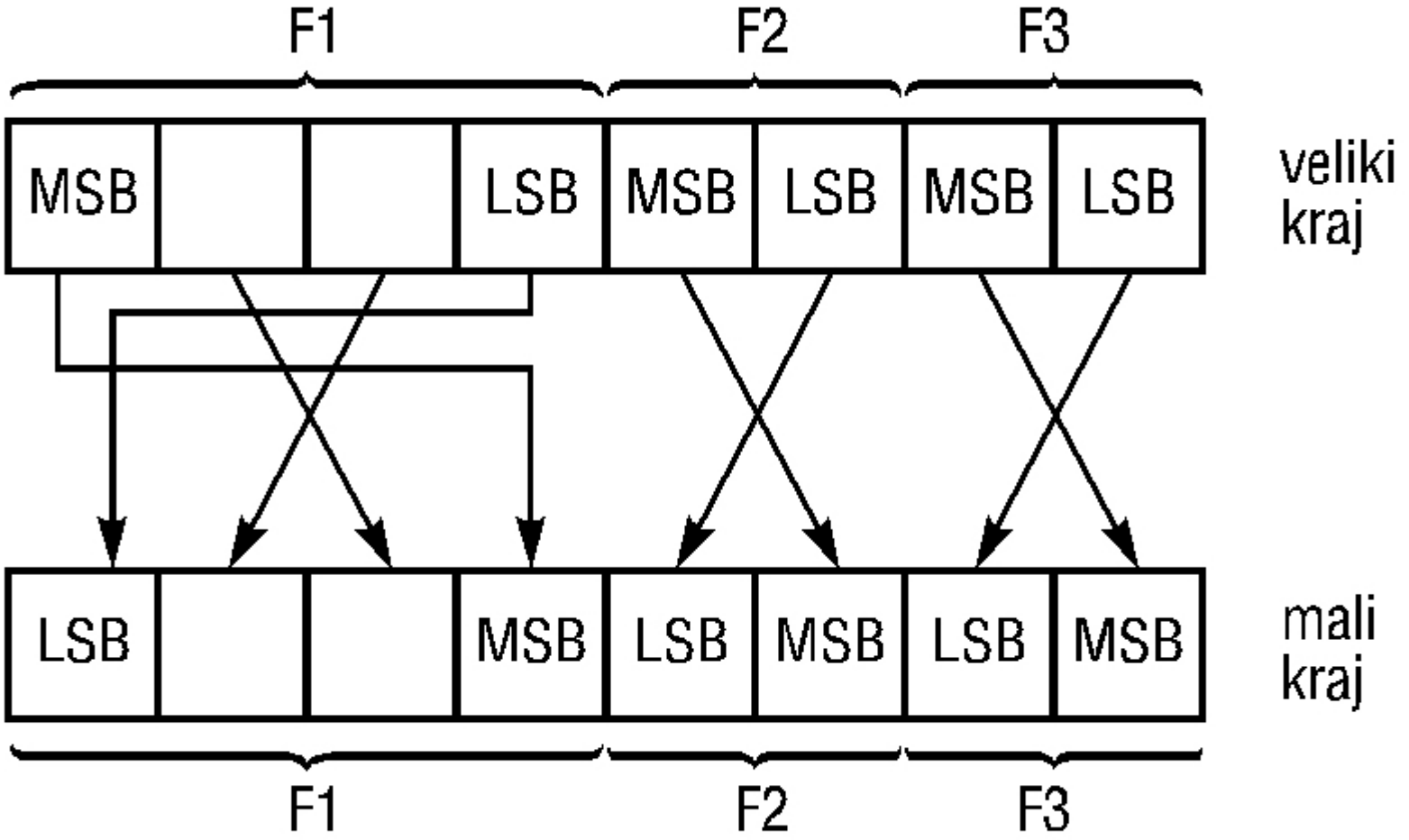
MRAM can replace many ofthers types of memory including SRAM, DRAM, ROM, EEPROM, Flash EEPROM, and feroelectric RAM (FRAM) . Prediction are crystalline structures that users grow on silicon.



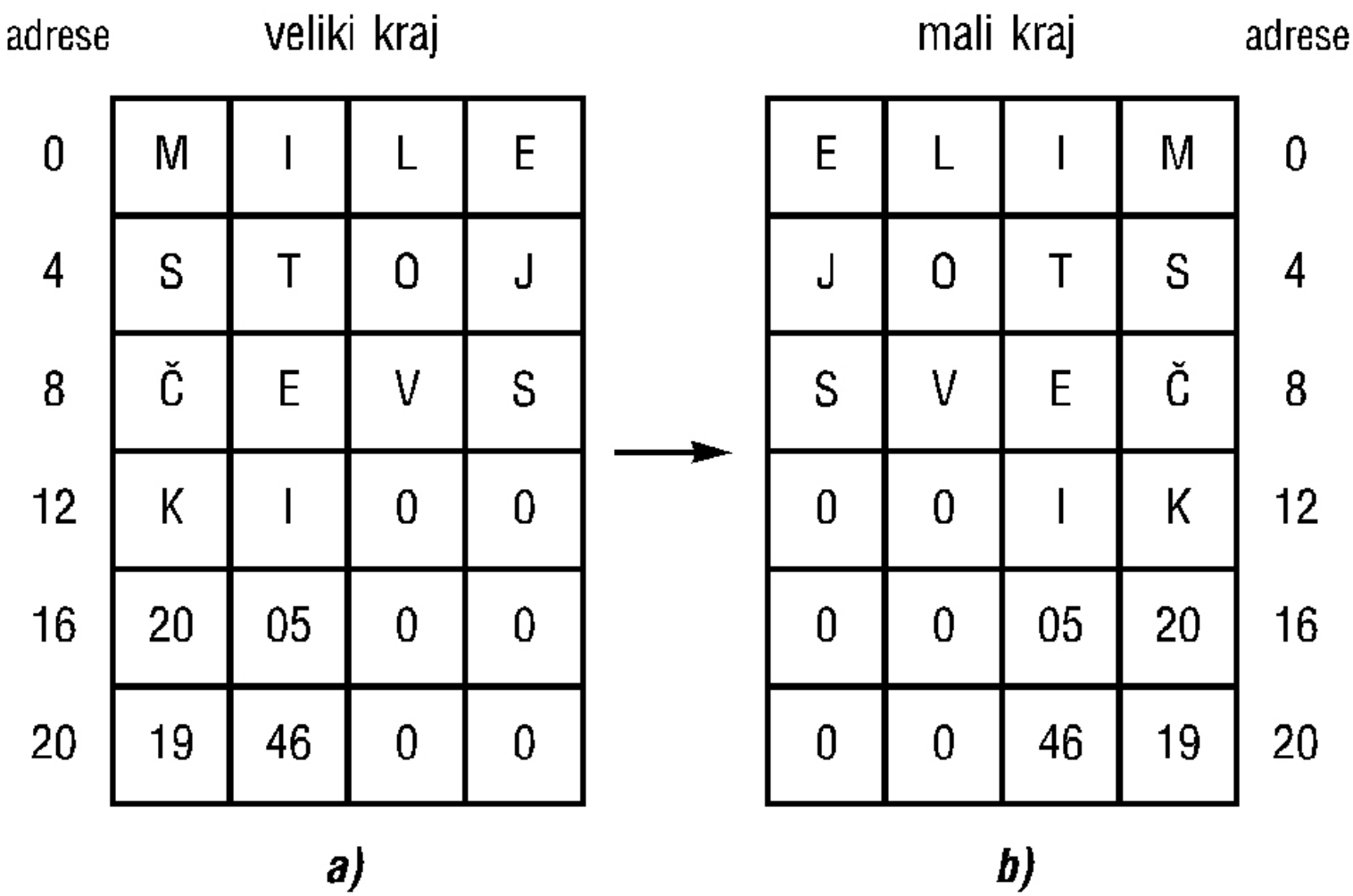
Veliki kraj u odnosu na mali kraj



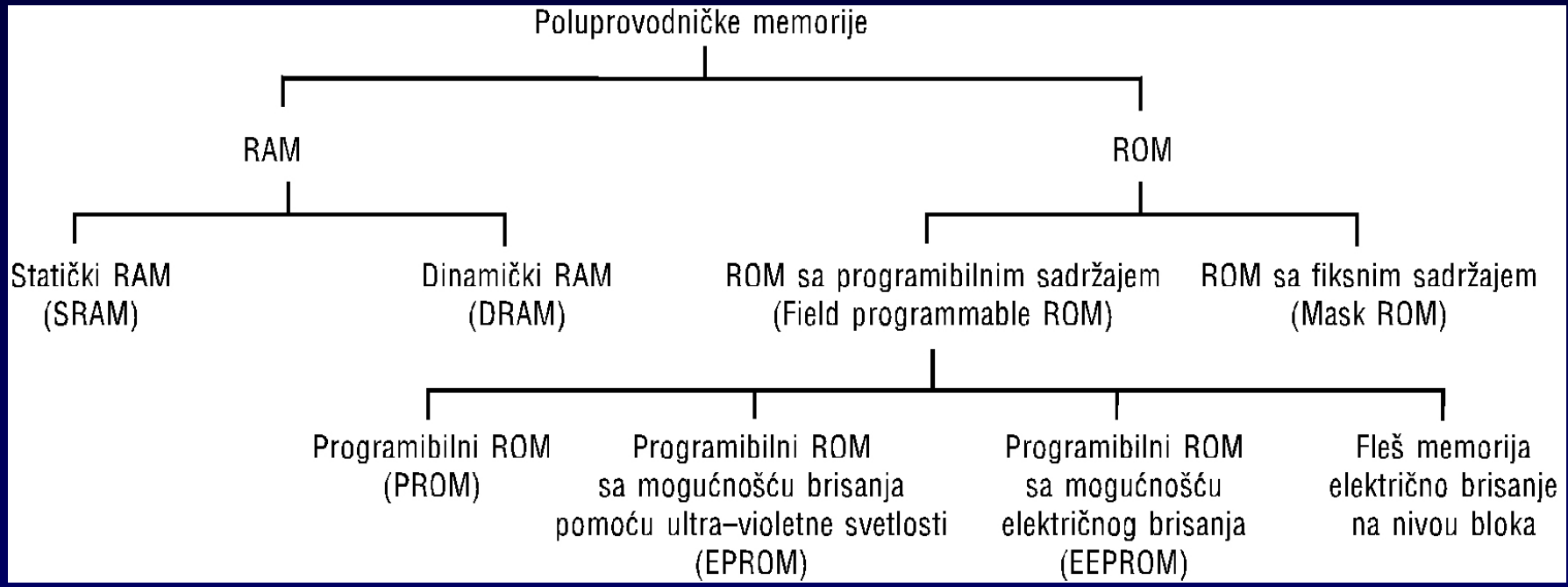
Konverzija BE u LE



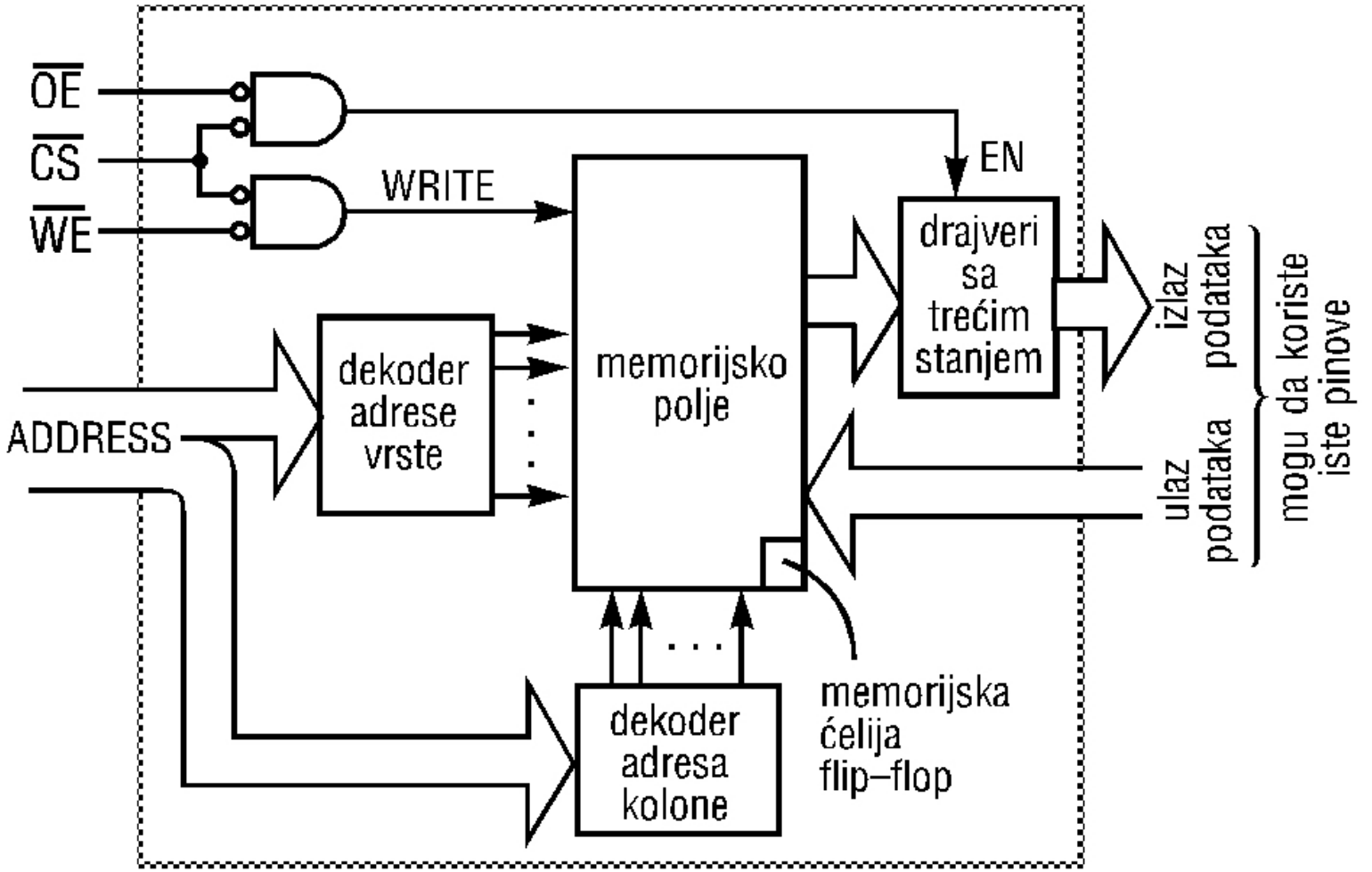
Prezentacija u BE i LE



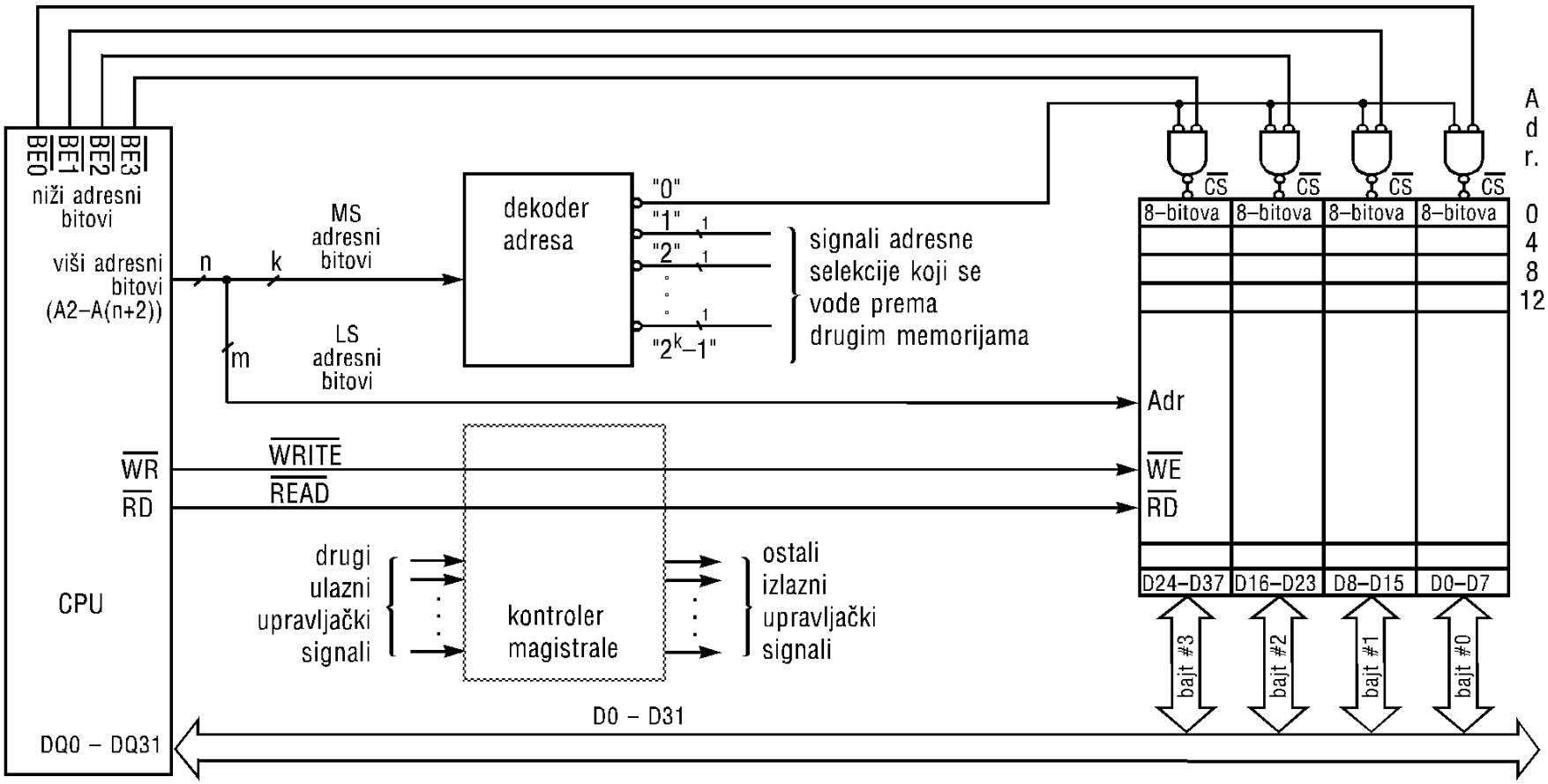
Tipovi poluprovodničkih memorija



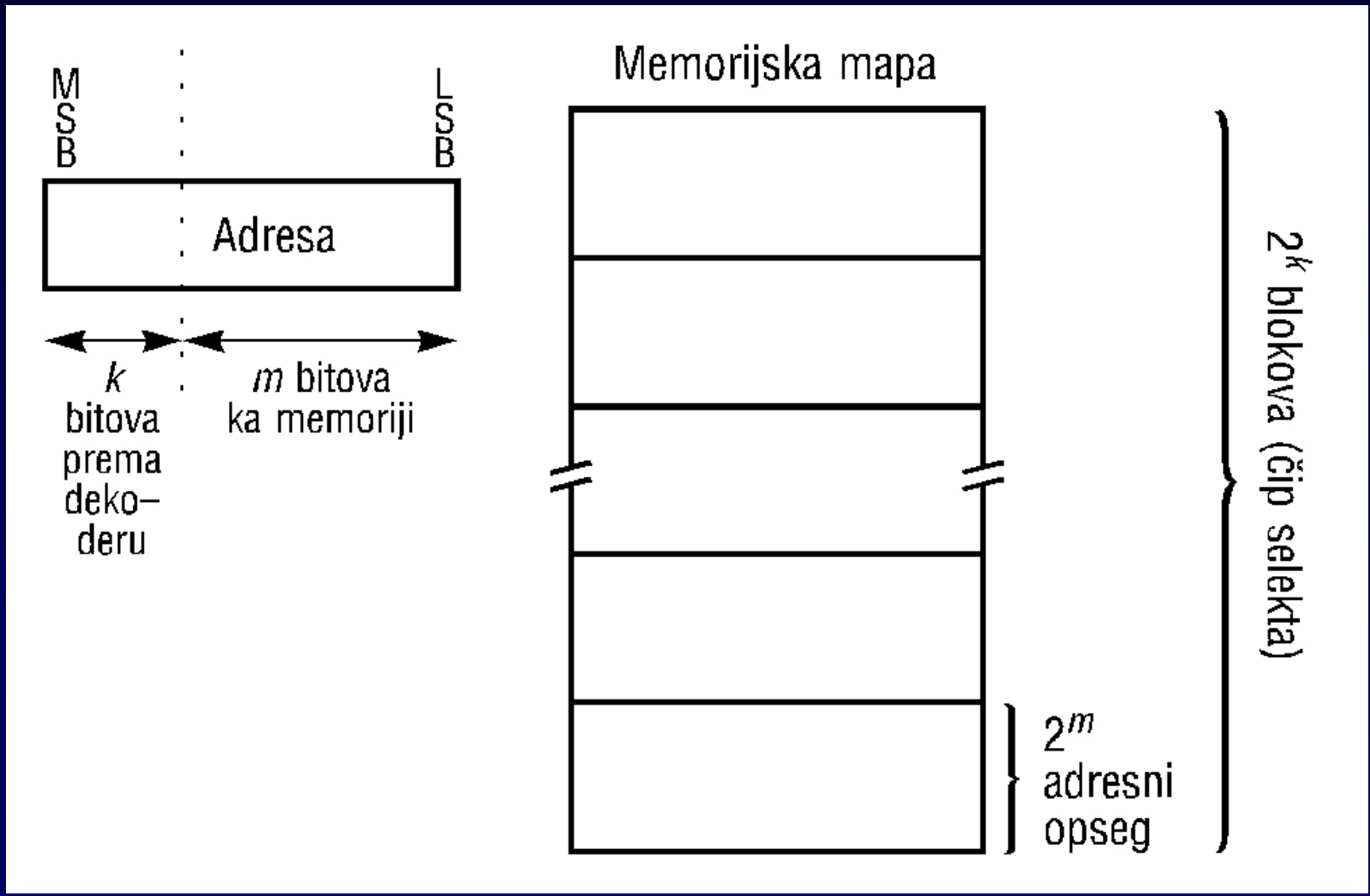
Struktura memorijskog čipa



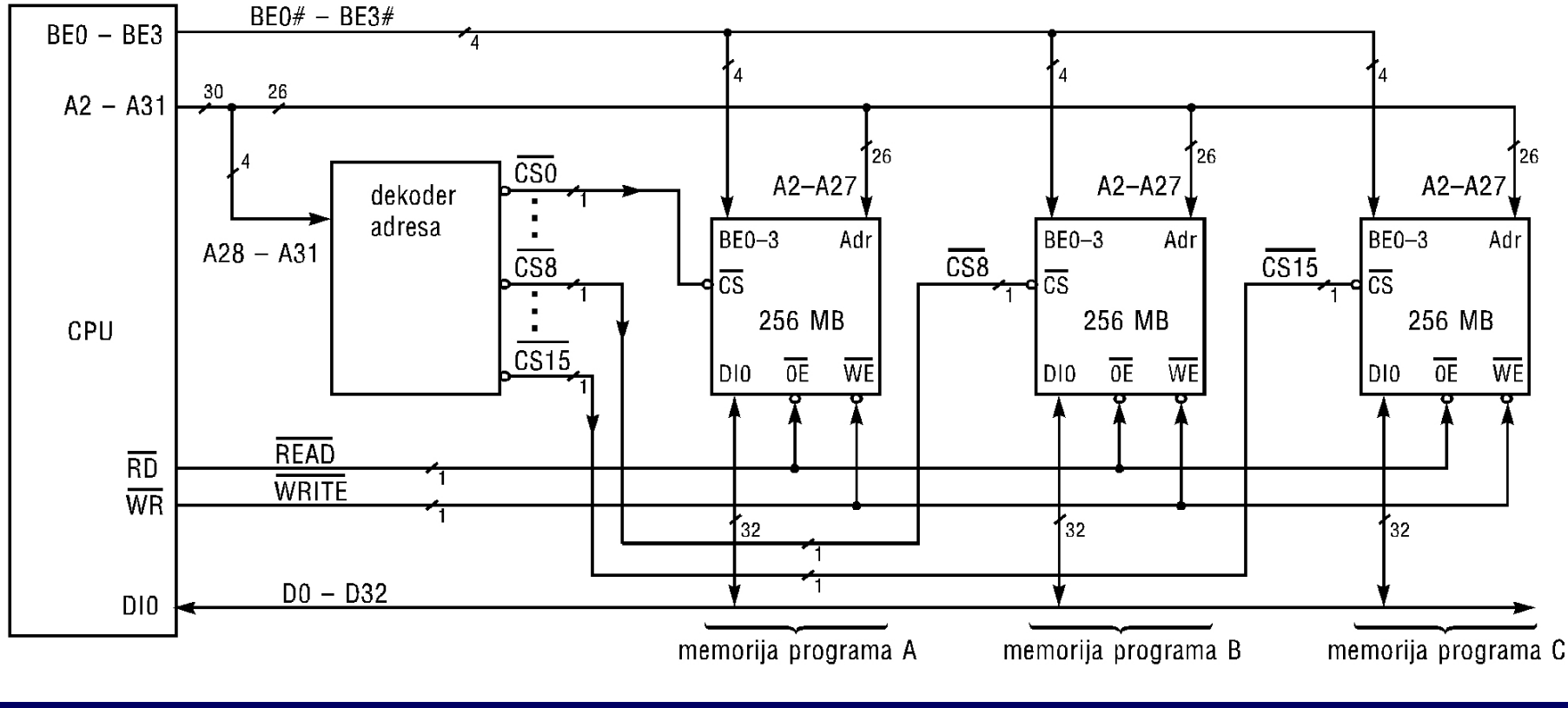
Povezivanje memorije kod 32-bitnih procesora



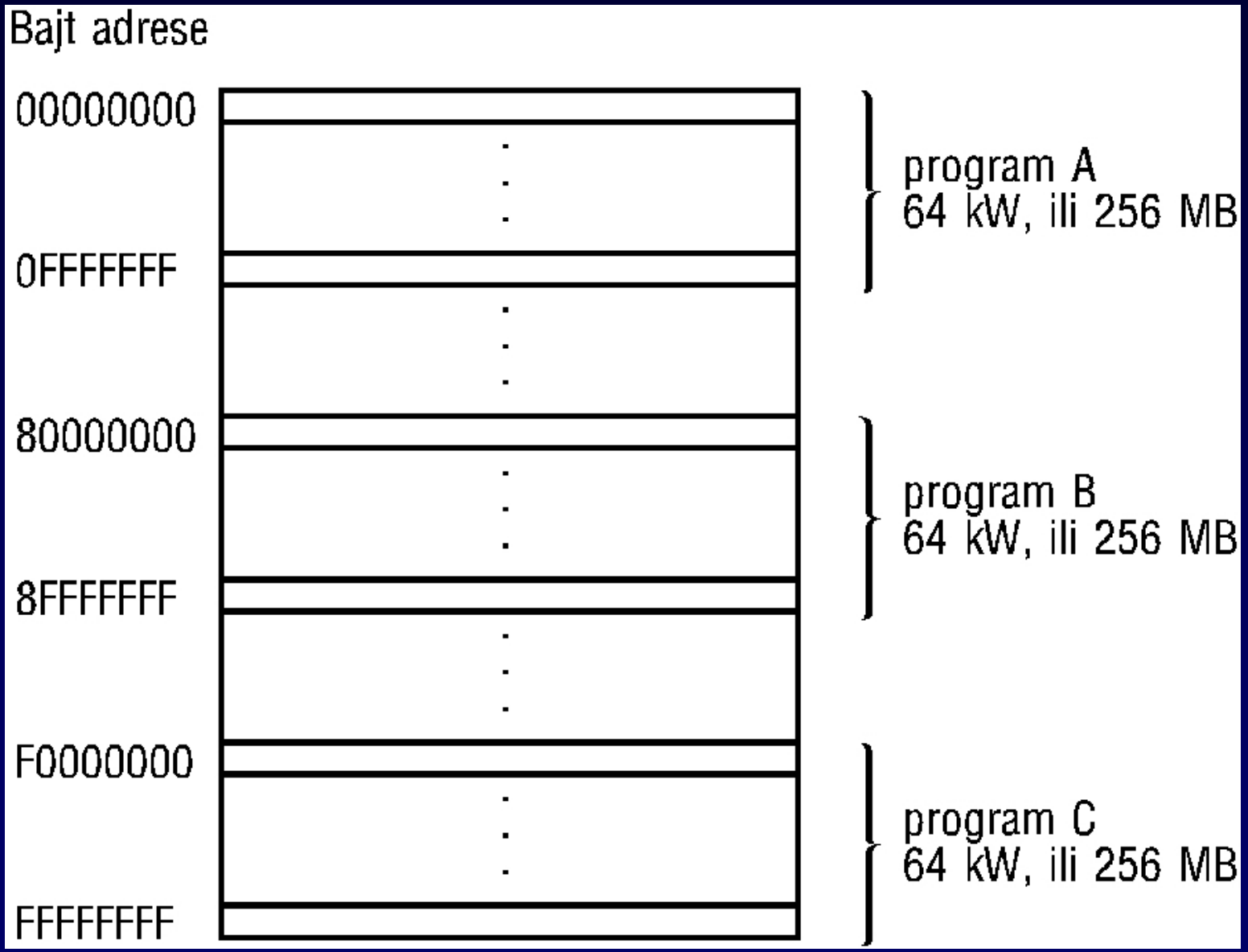
Memorijska adresa i memorijska mapa



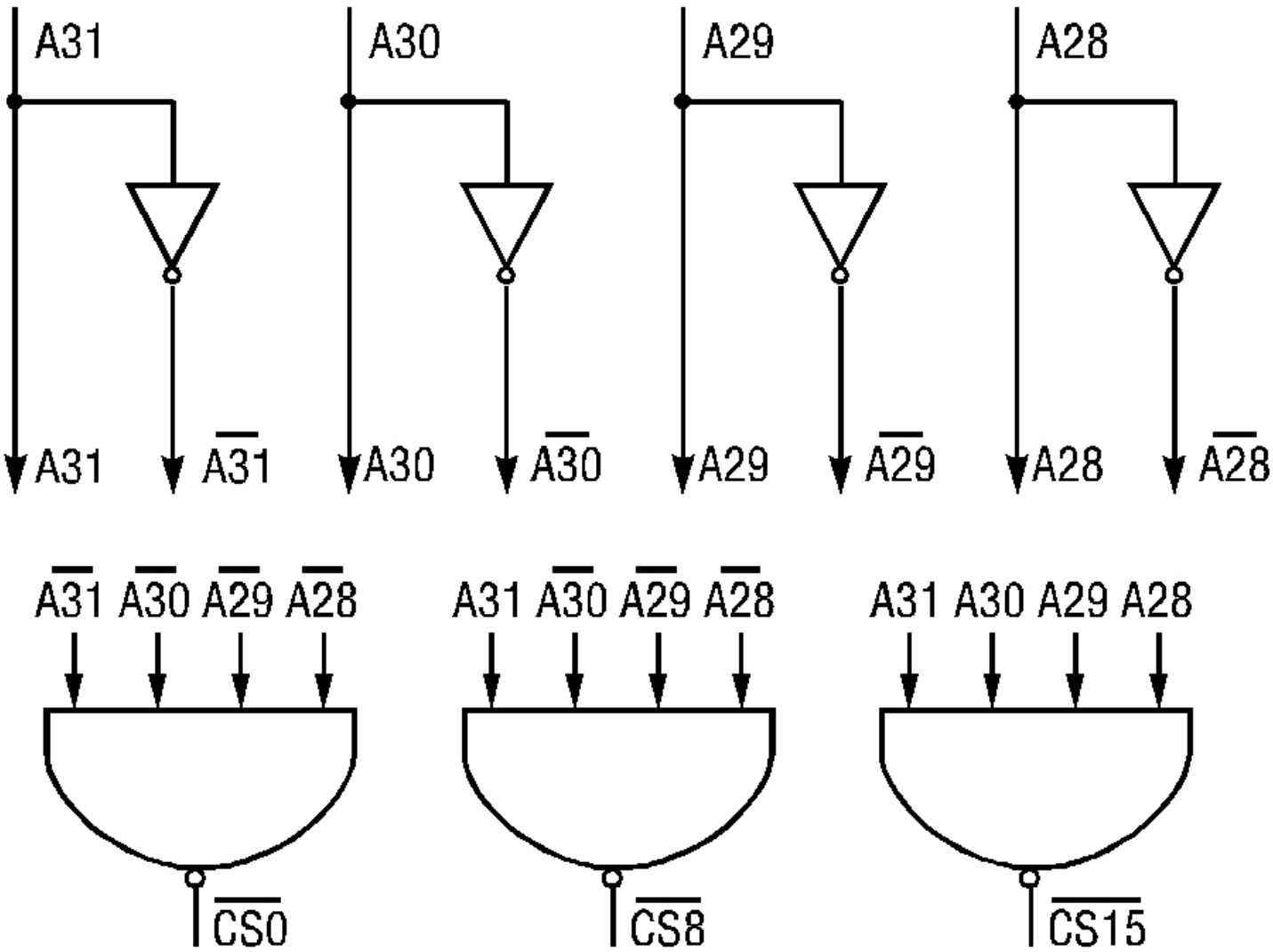
Primer 1: Povezivanja memorije kod 32-bitnog procesora



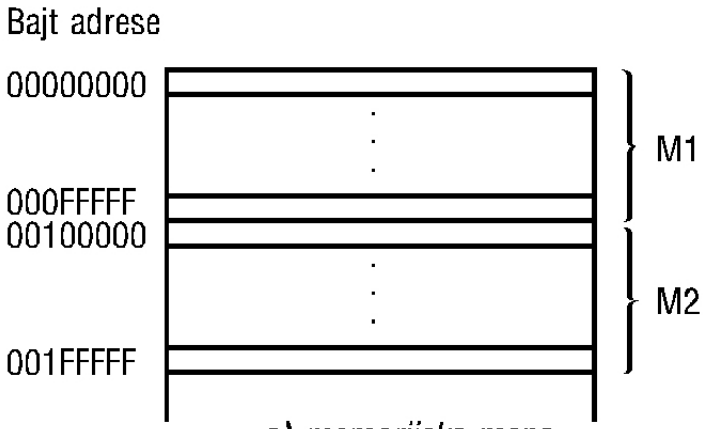
Primer 1: Memorijska mapa



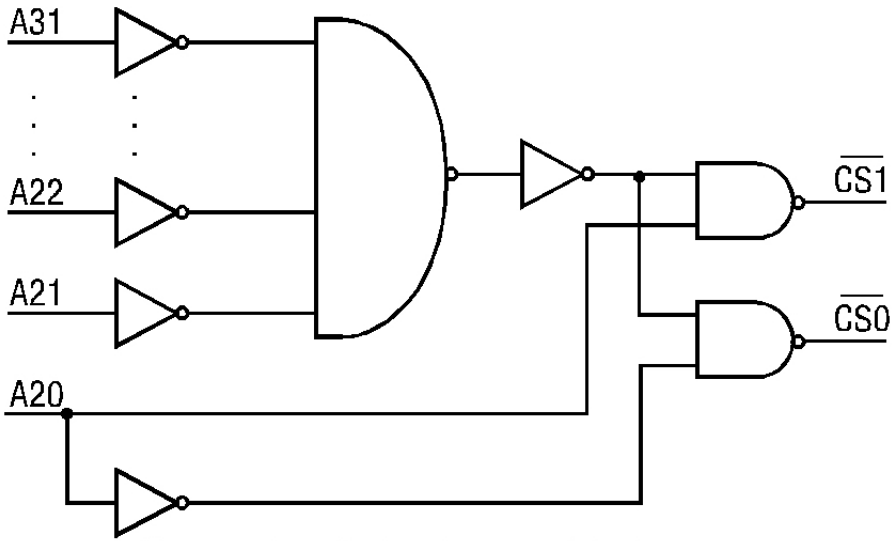
Primer 1: Realizacija dekodera adresa



Primer 2: Memorijska mapa i dekoder kada su M1 i M2 veličine 1 MB

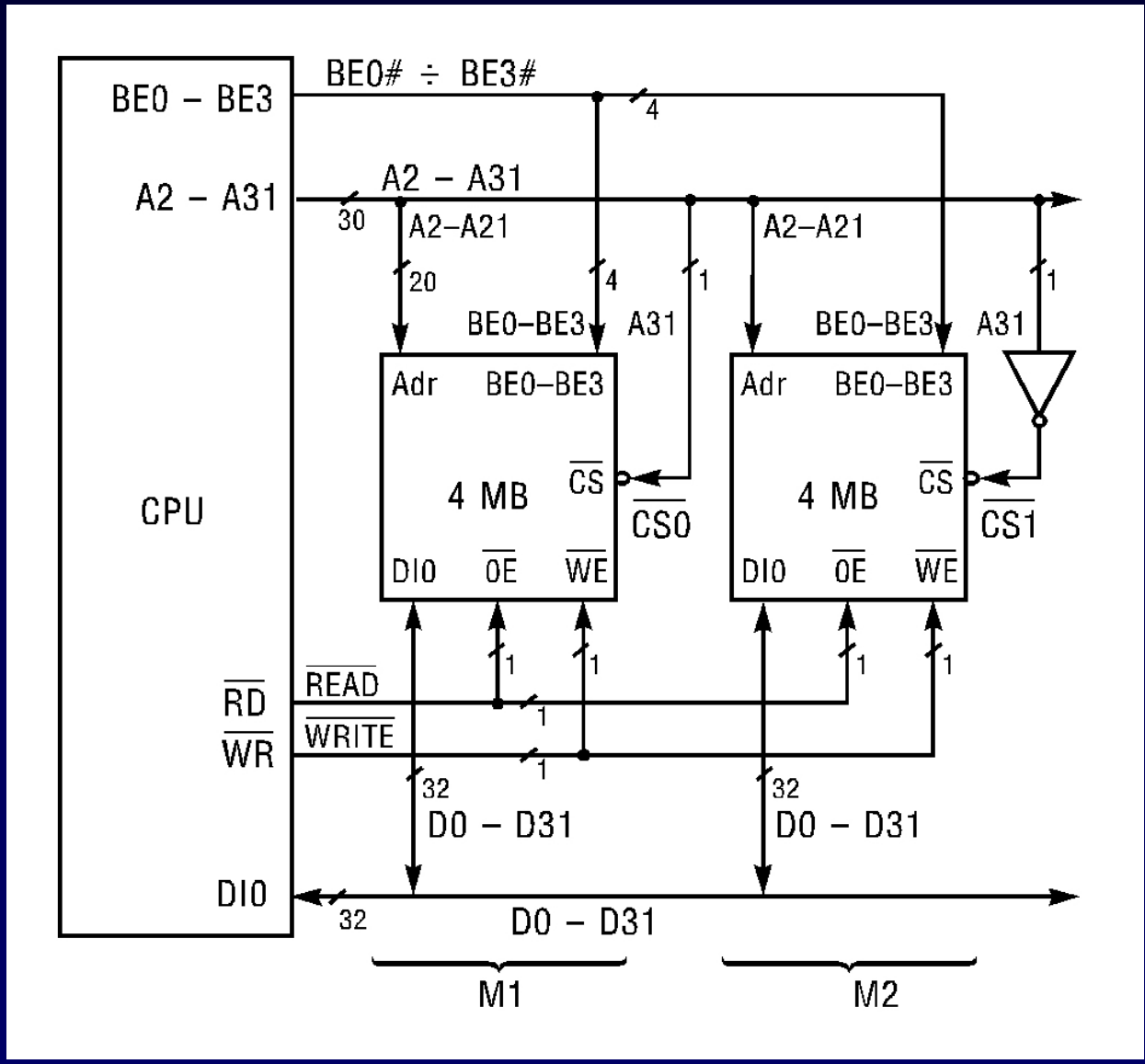


a) memorijska mapa

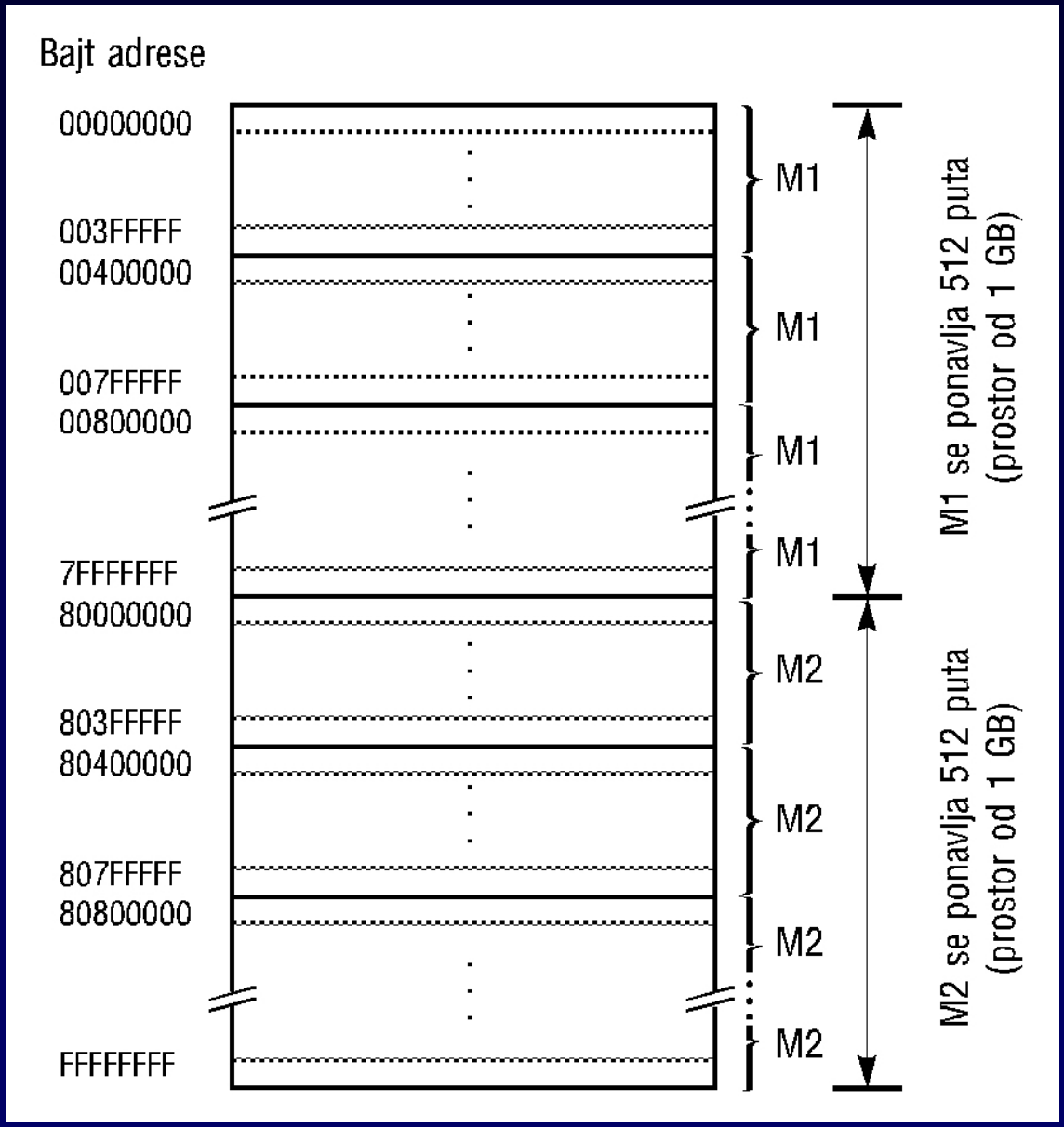


b) moguće rešenje adresnog dekodera

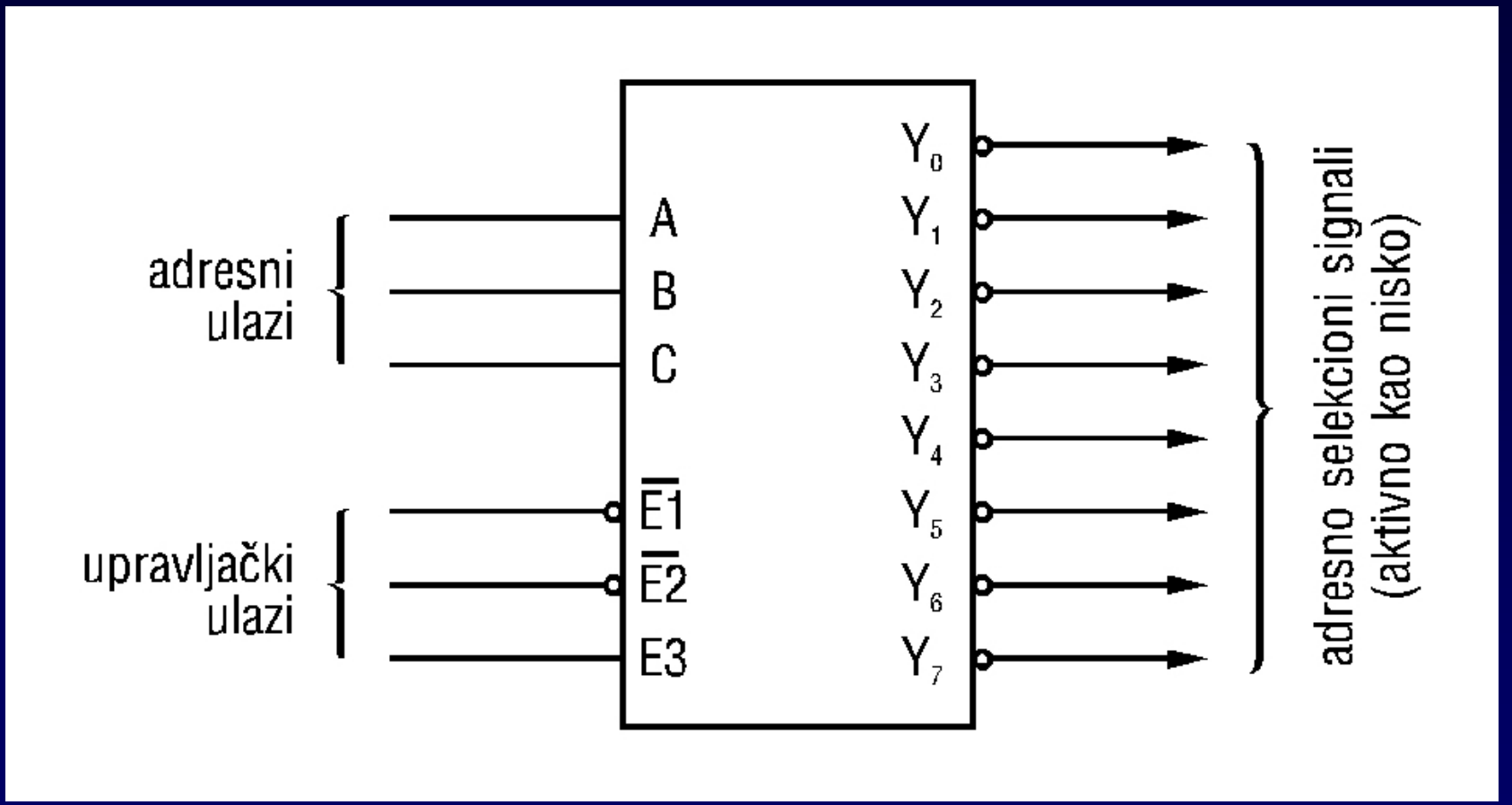
Primer 3: Selekcija na osnovu MS bita



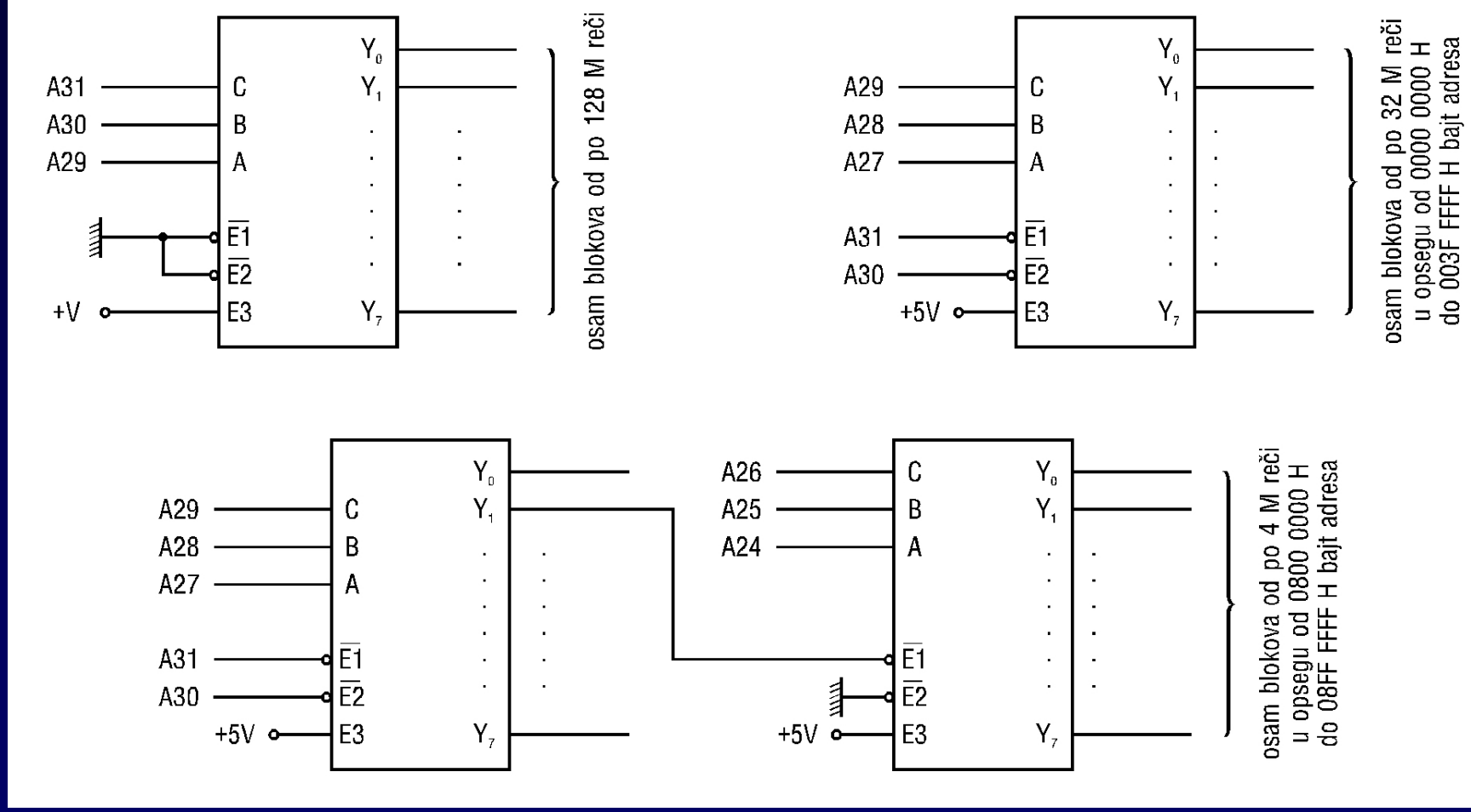
Preslikavanje memorijskih prostora



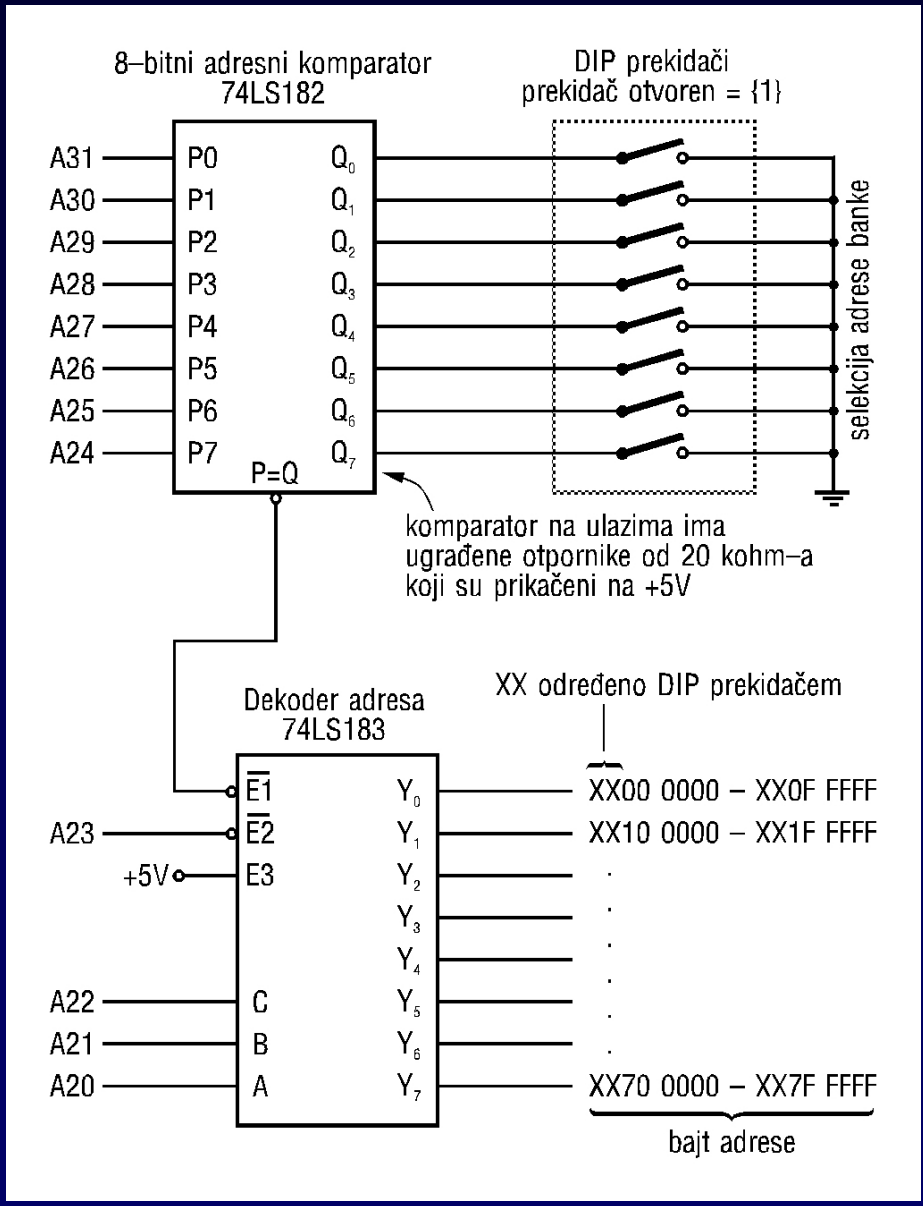
Tipičan adresni dekoder



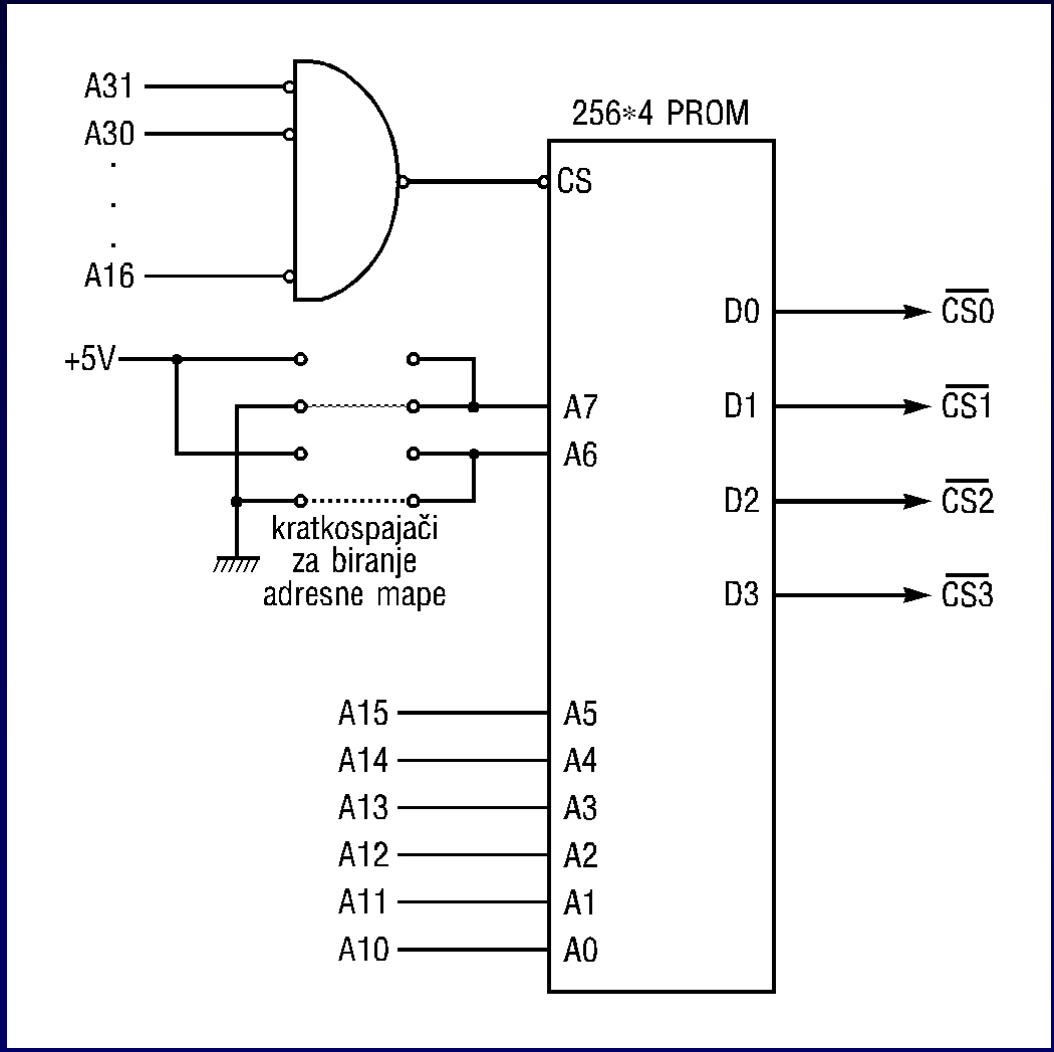
Realizacija jedne adresne šeme



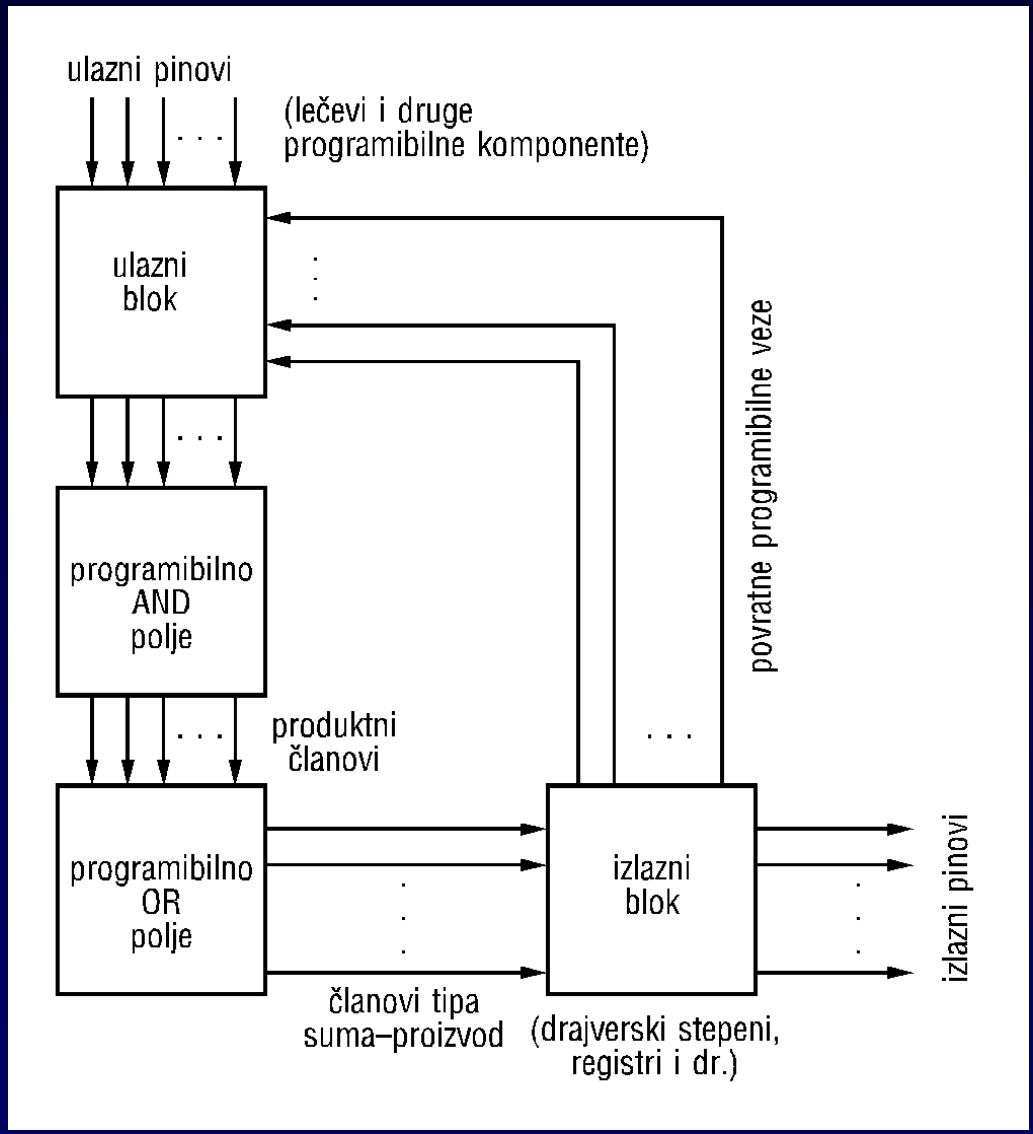
Realizacija dekodiranja adresa korišćenjem komparatora



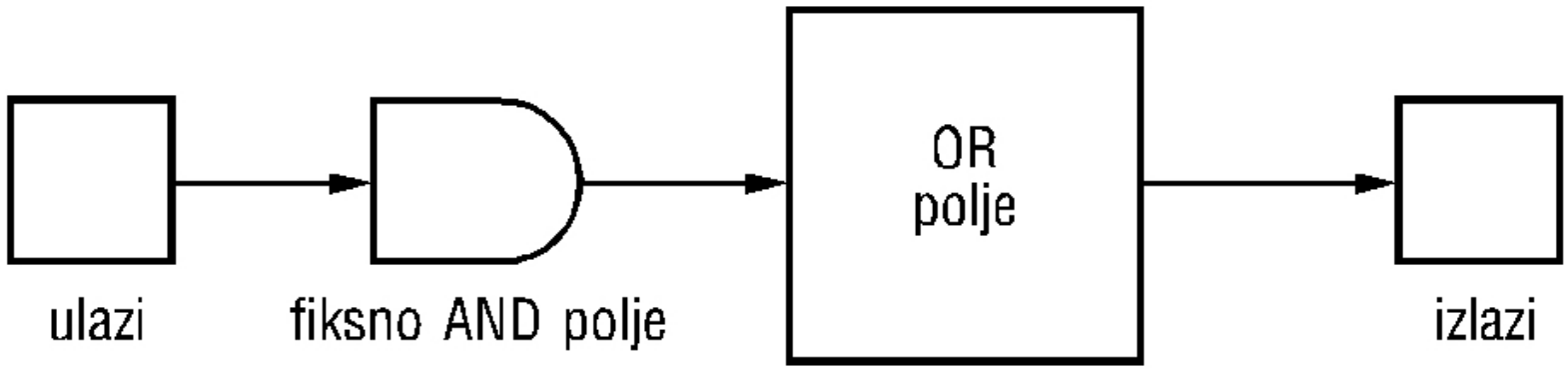
Realizacija dekodera adresa korišćenjem PROM-a



Struktura PLD-a

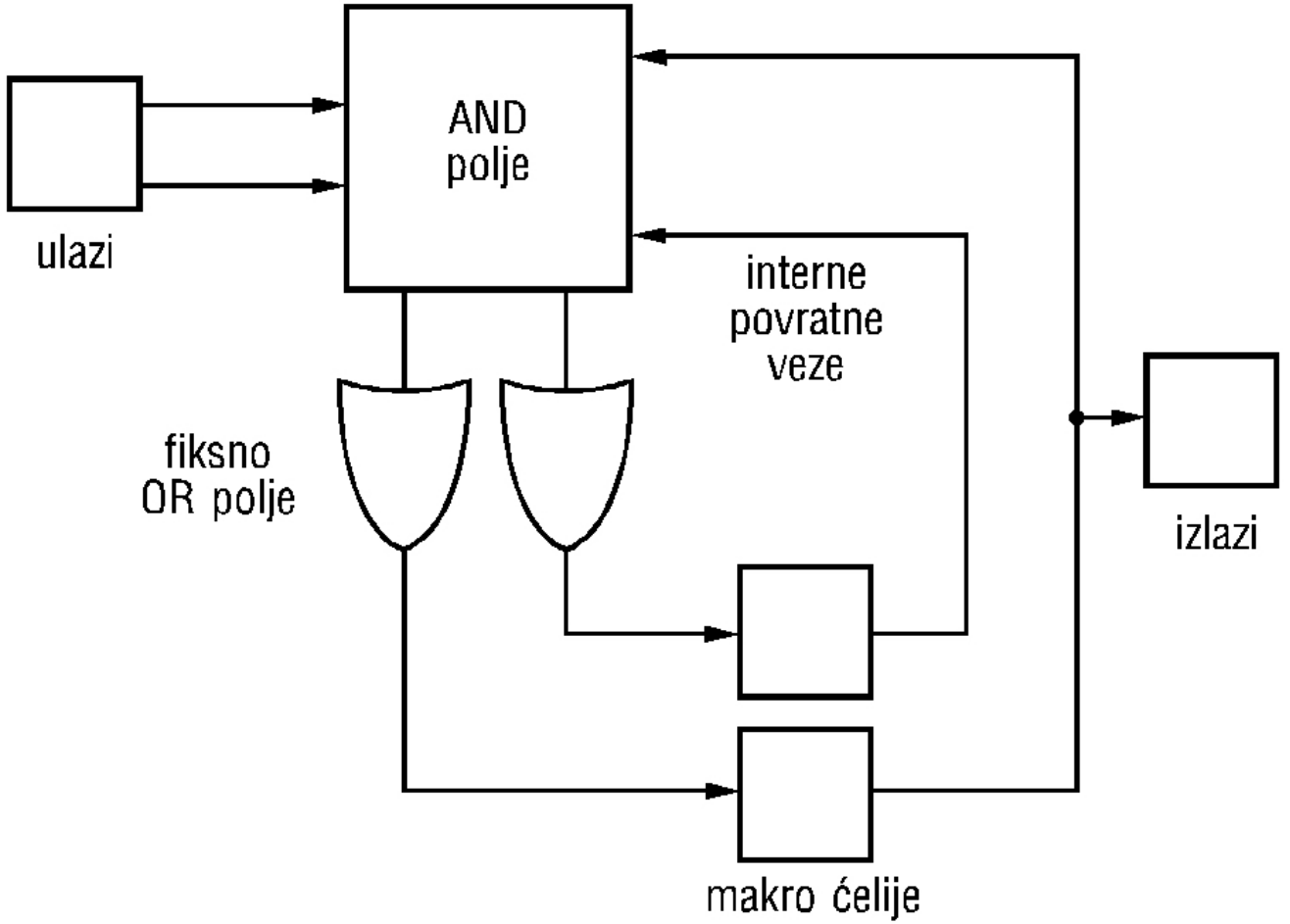


ROM model sa programibilnim OR poljem



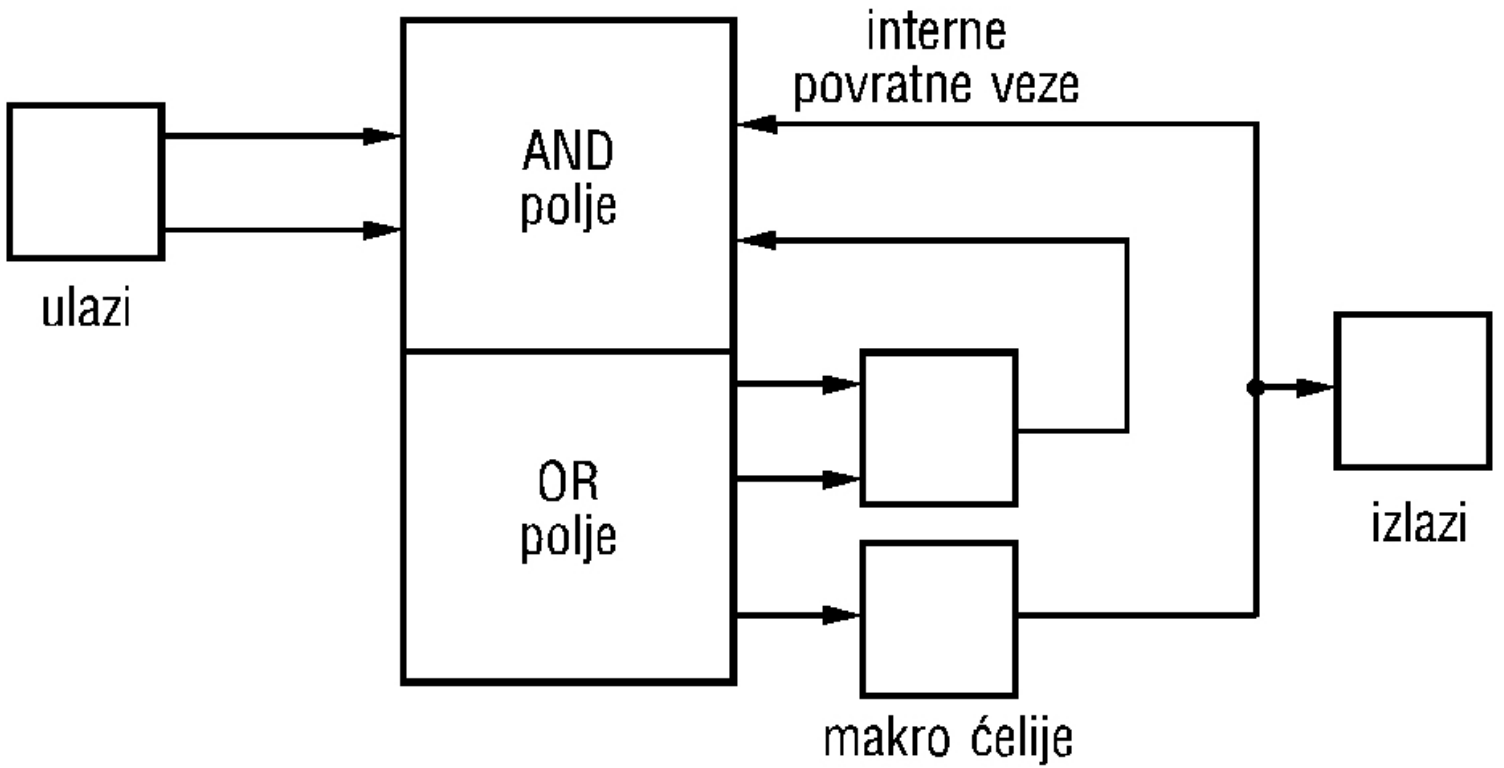
a) *ROM model sa programibilnim OR poljem*

PAL model sa fiksnim OR poljem



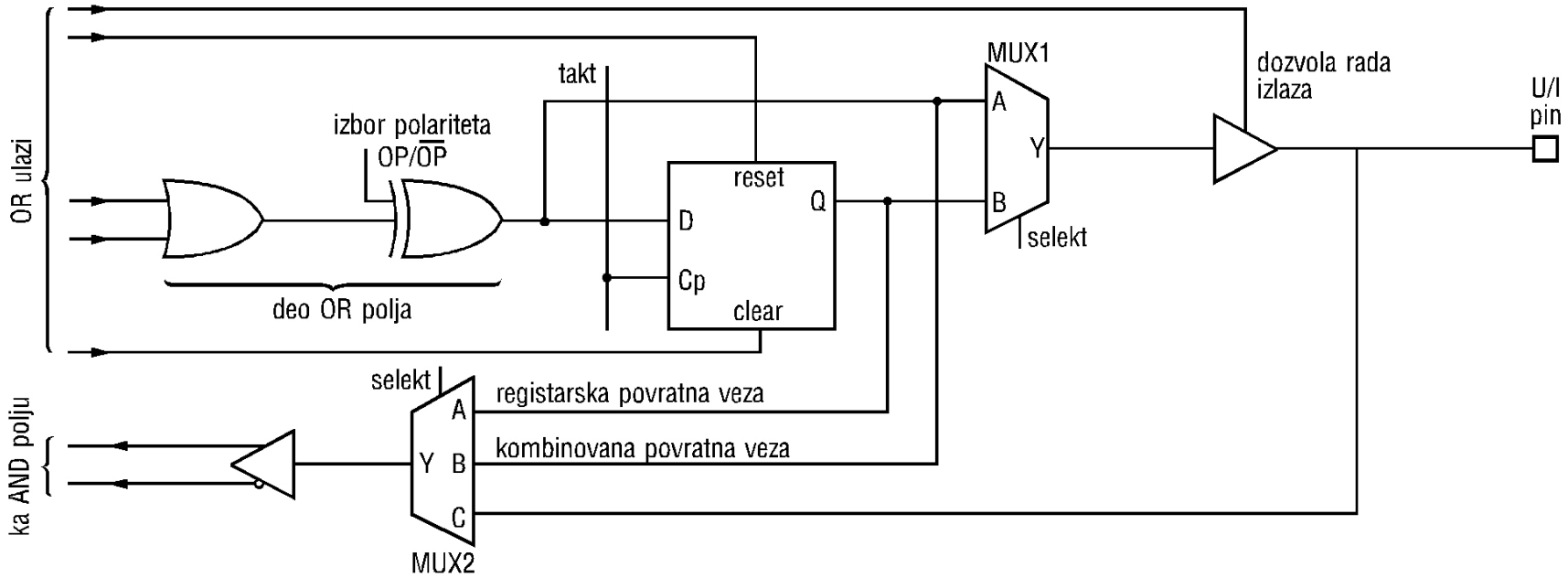
b) *PAL model sa fiksnim OR poljem.*
Svako OR kolo obično ima između šest do osam ulaza.

PAL model sa programibilnim AND i OR poljima



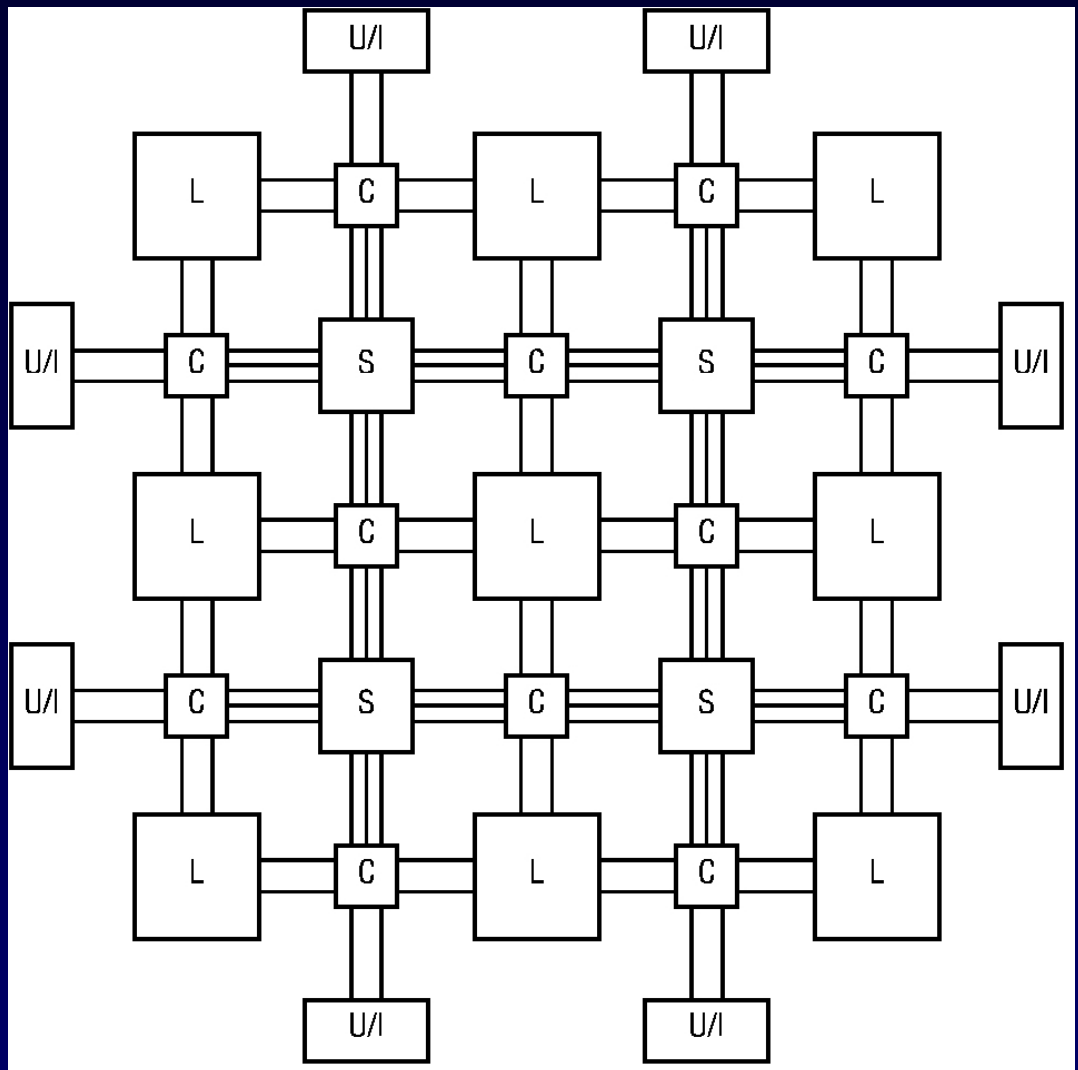
c) PAL model sa programibilnim AND i OR poljima

Izgled makroćelije



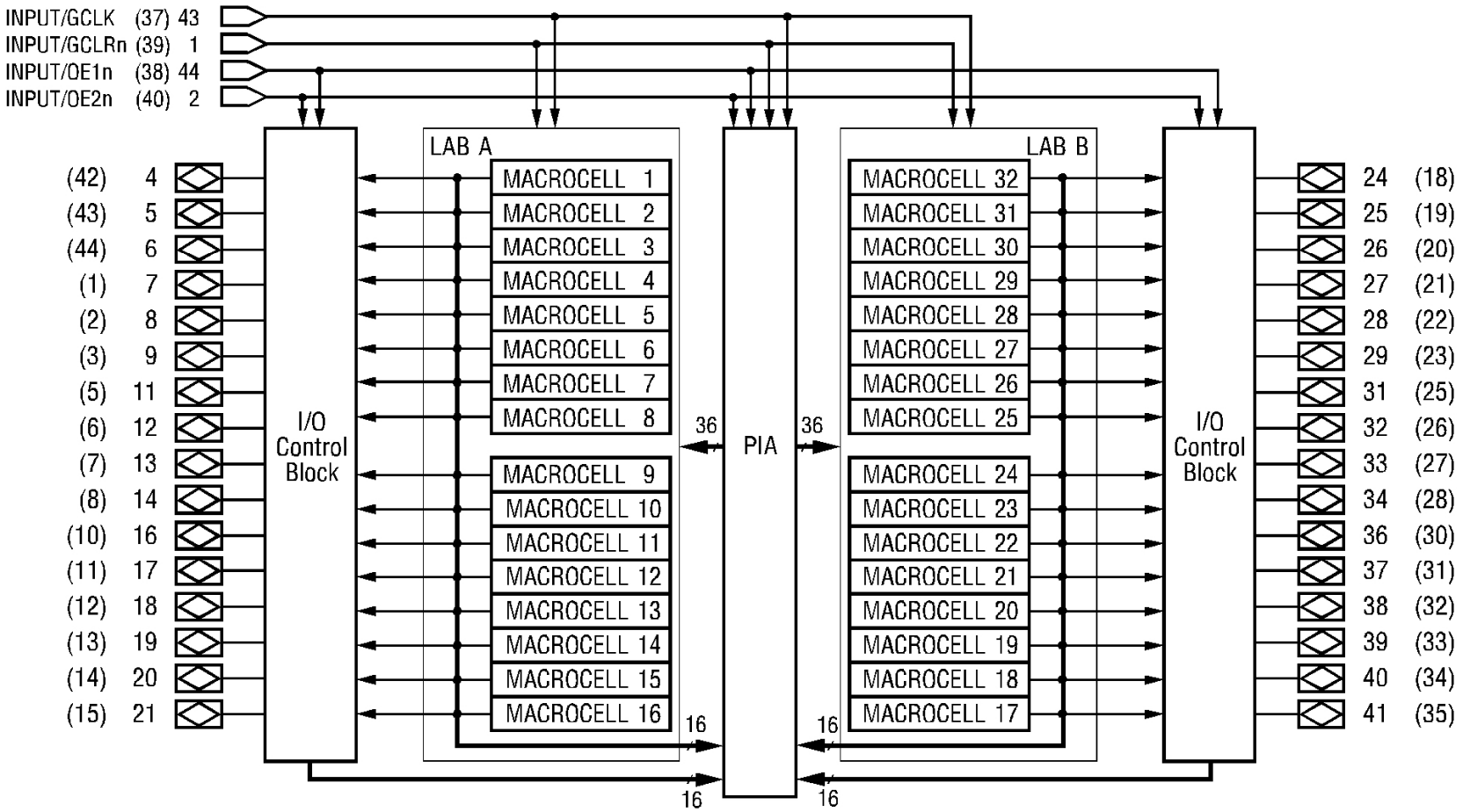
d) *Organizacija jednostavne izlazne makroćelije koja sadrži kombinacione i registarske izlaze i povratne veze, izlaze u pravoj i komplementarnoj formi, dozvolu rada izlaza, deo OR polja i dr.*

Izgled FPGA kola

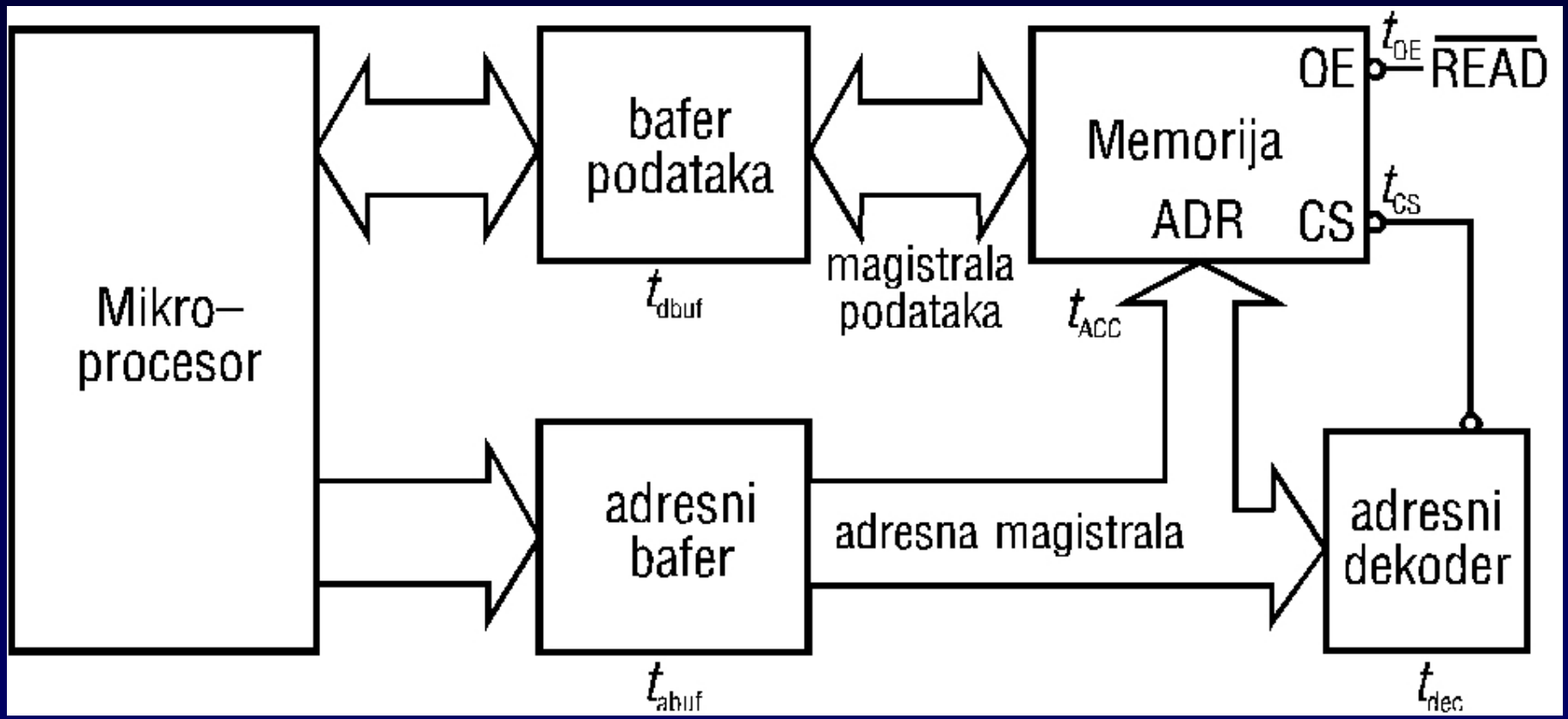


Legenda: *L – logički blok; S – komutatorski (prekidački) blok; C – blok za povezivanje; U/I – Ulazno/Izlazi blok; ═ – kanal između blokova S i C; ═ – kanal između blokova L i C i između C i U/I*

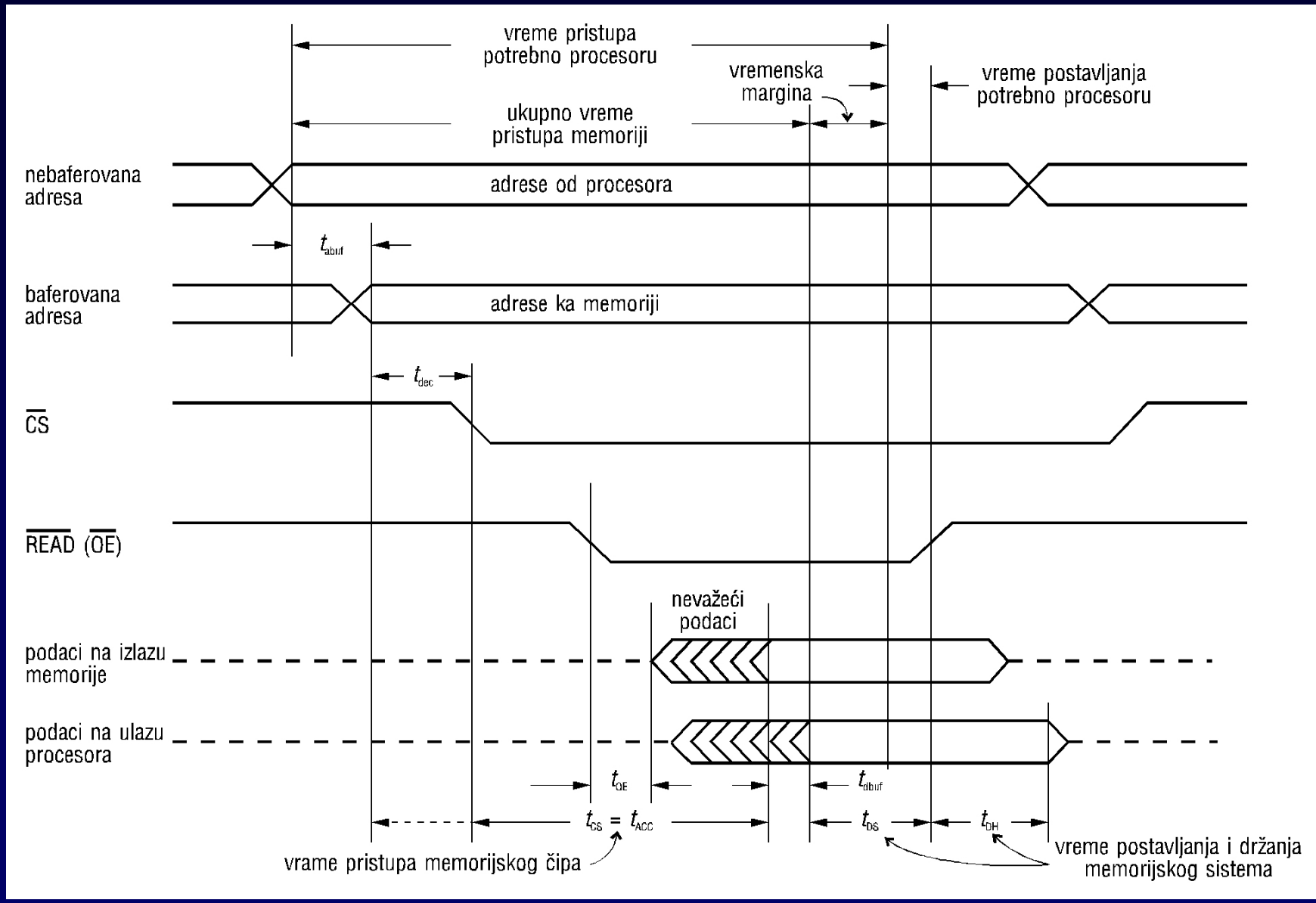
Izgled CPLD ćelije



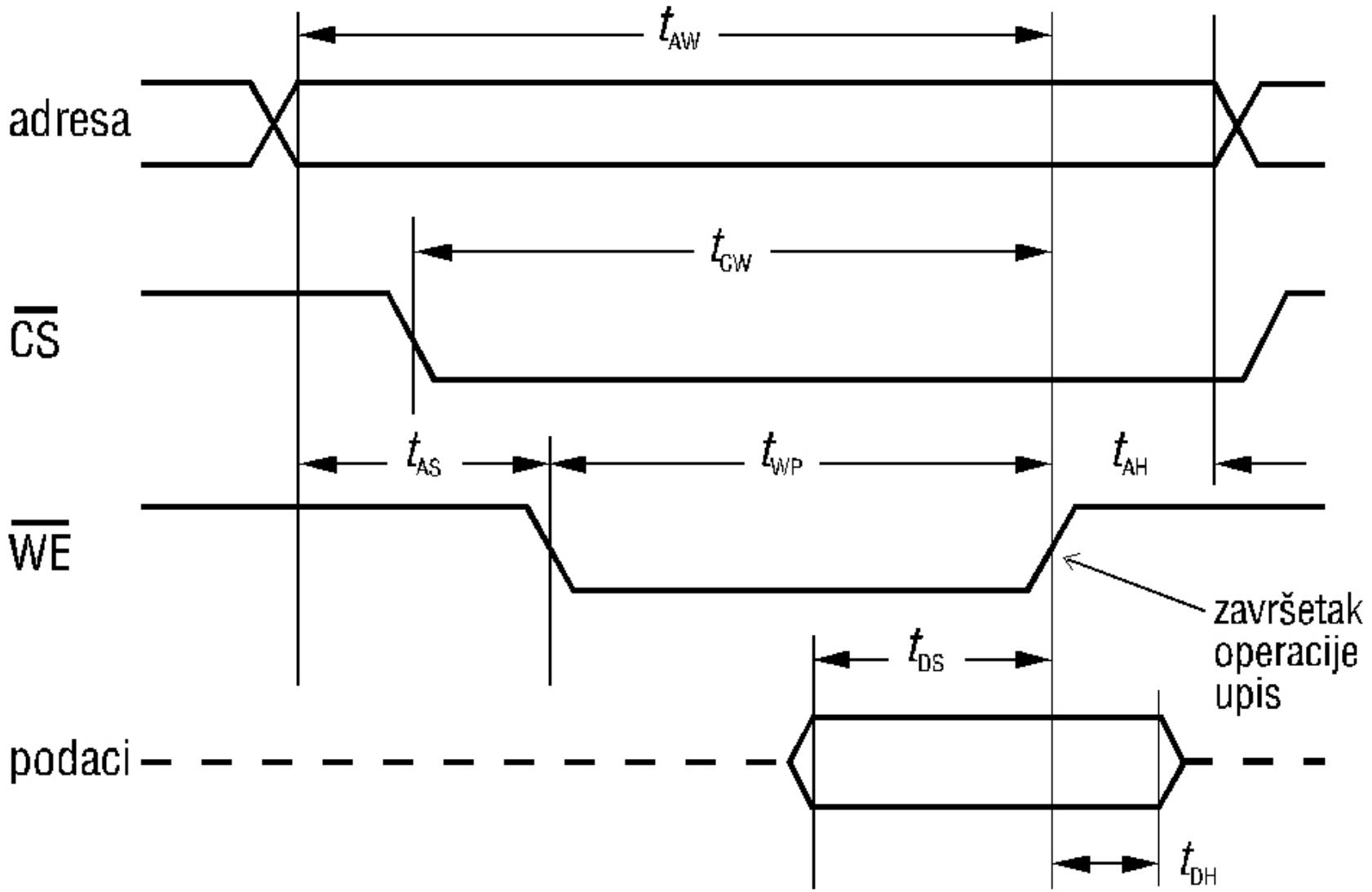
Tipičan način povezivanja memorije



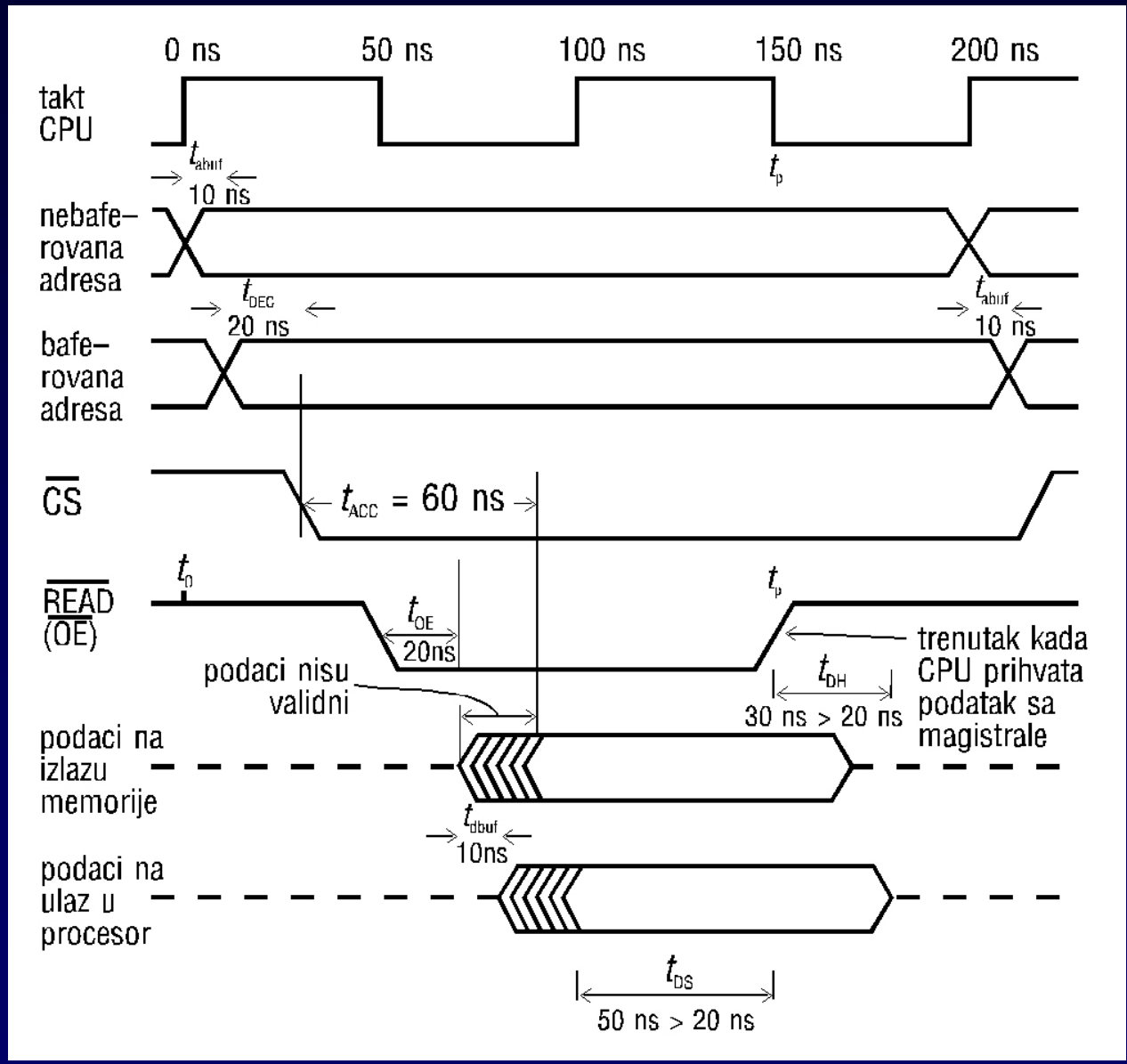
Tajming memorije



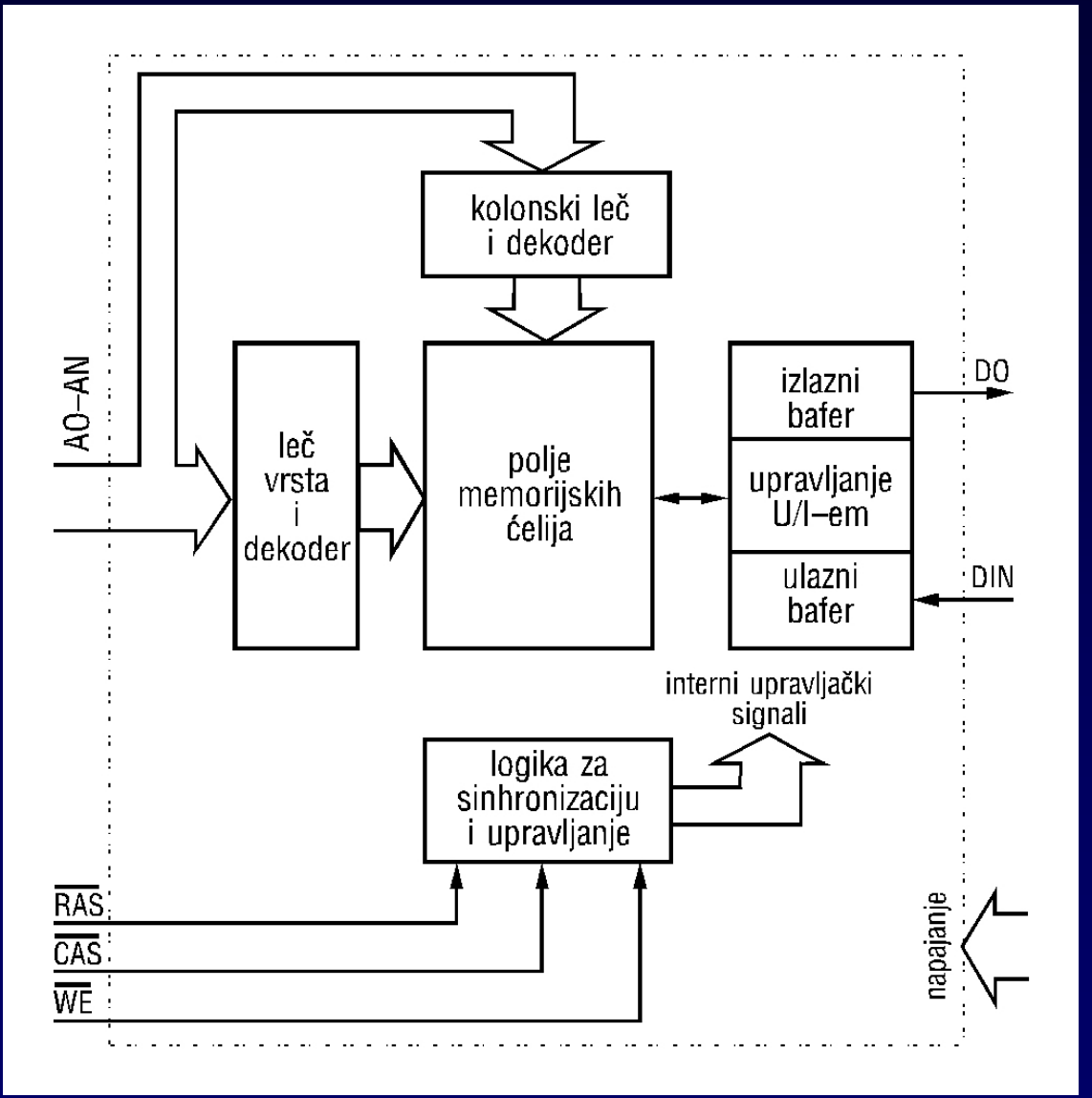
Tajming ciklusa upisa u memoriju



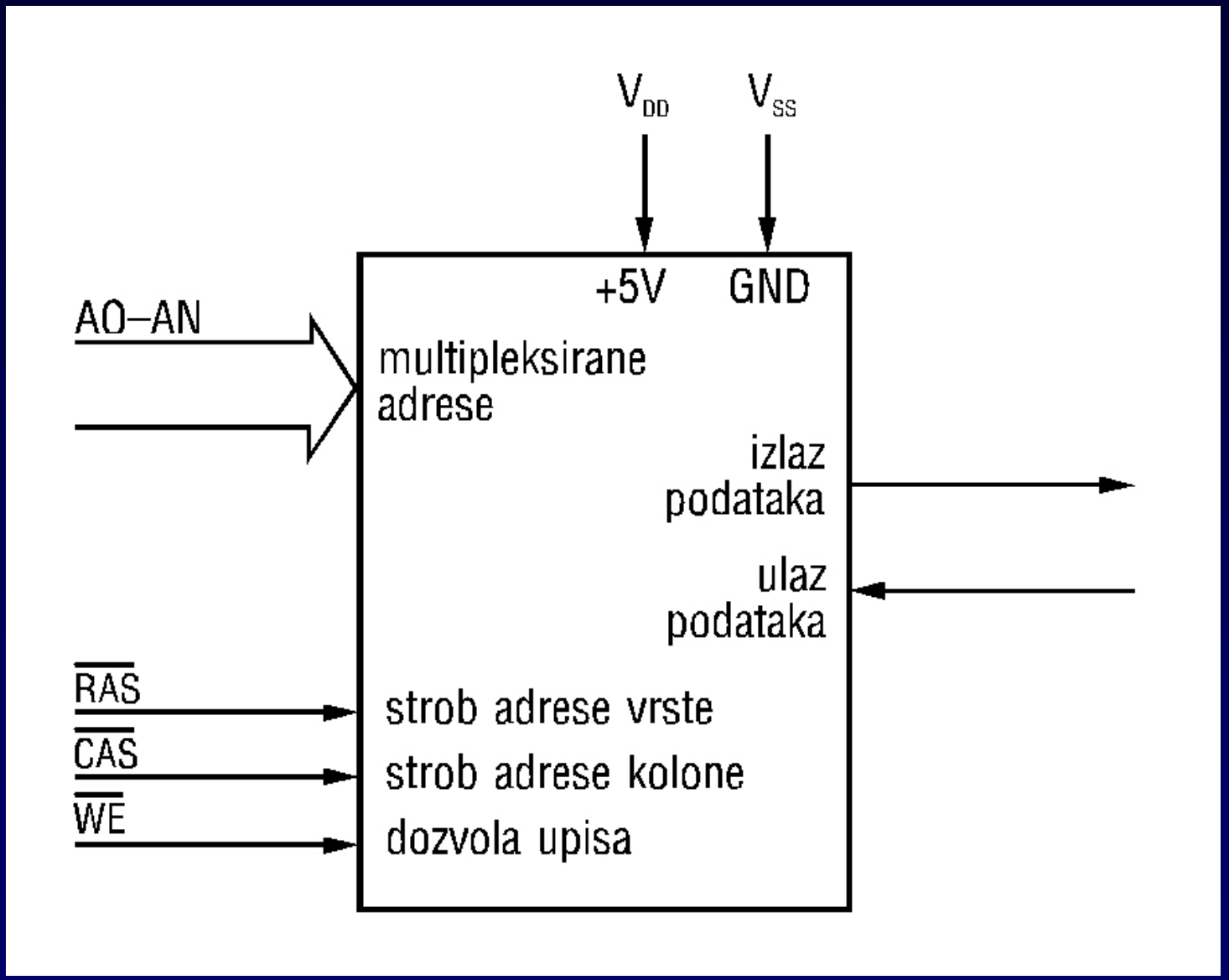
Tajming ciklusa čitanja iz memorije



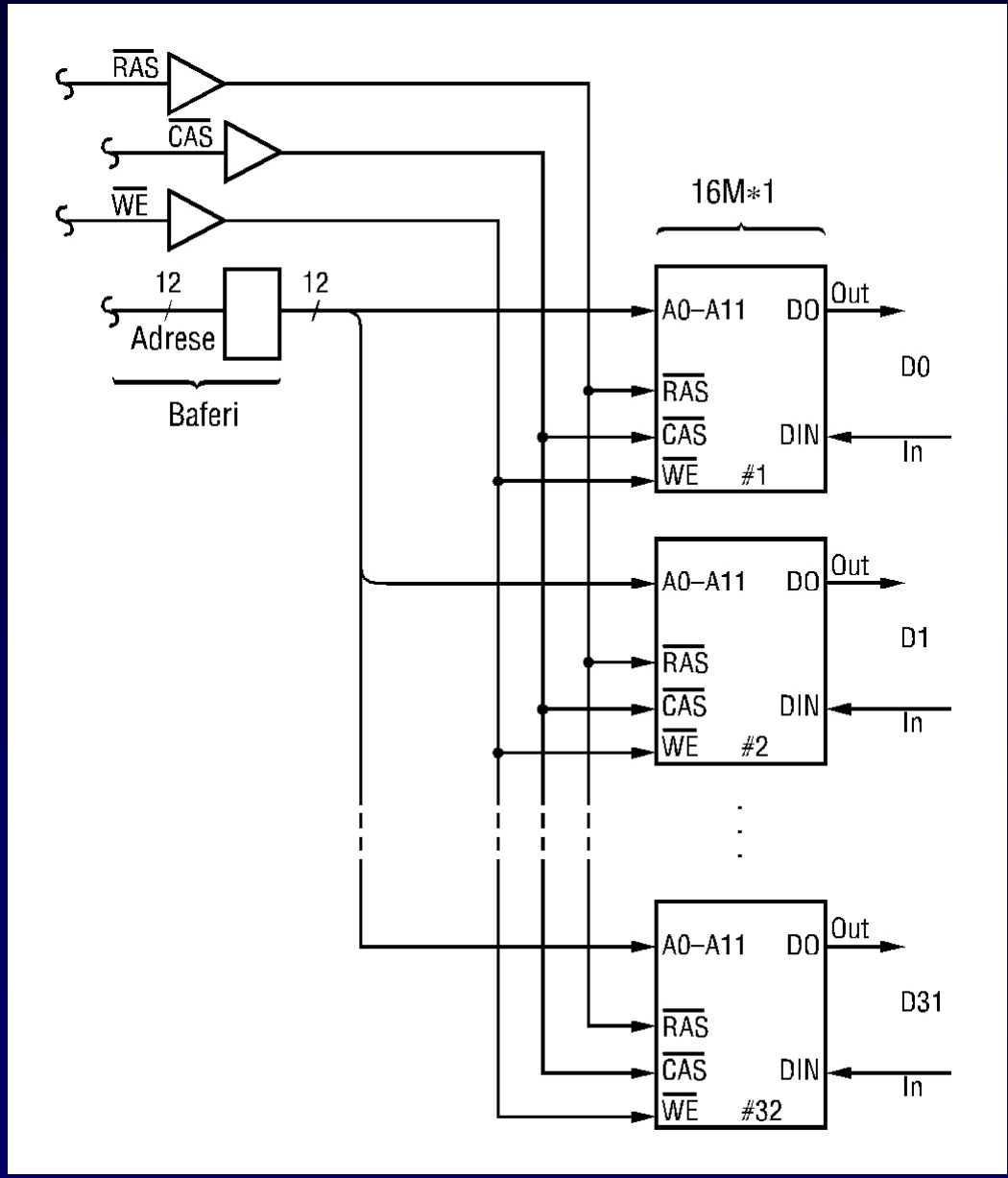
Struktura dinamičke memorije



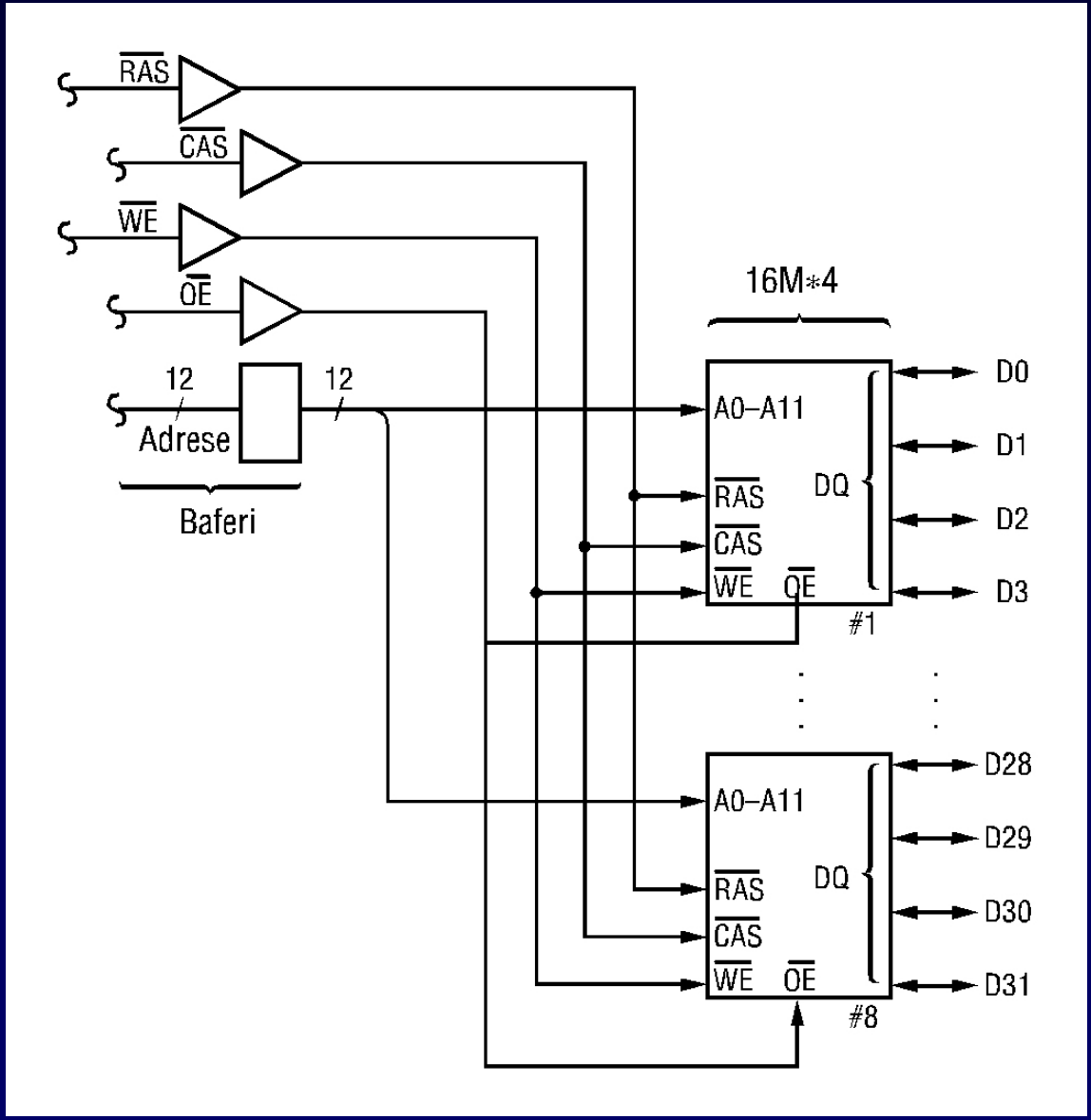
Dinamička memorija – logička šema



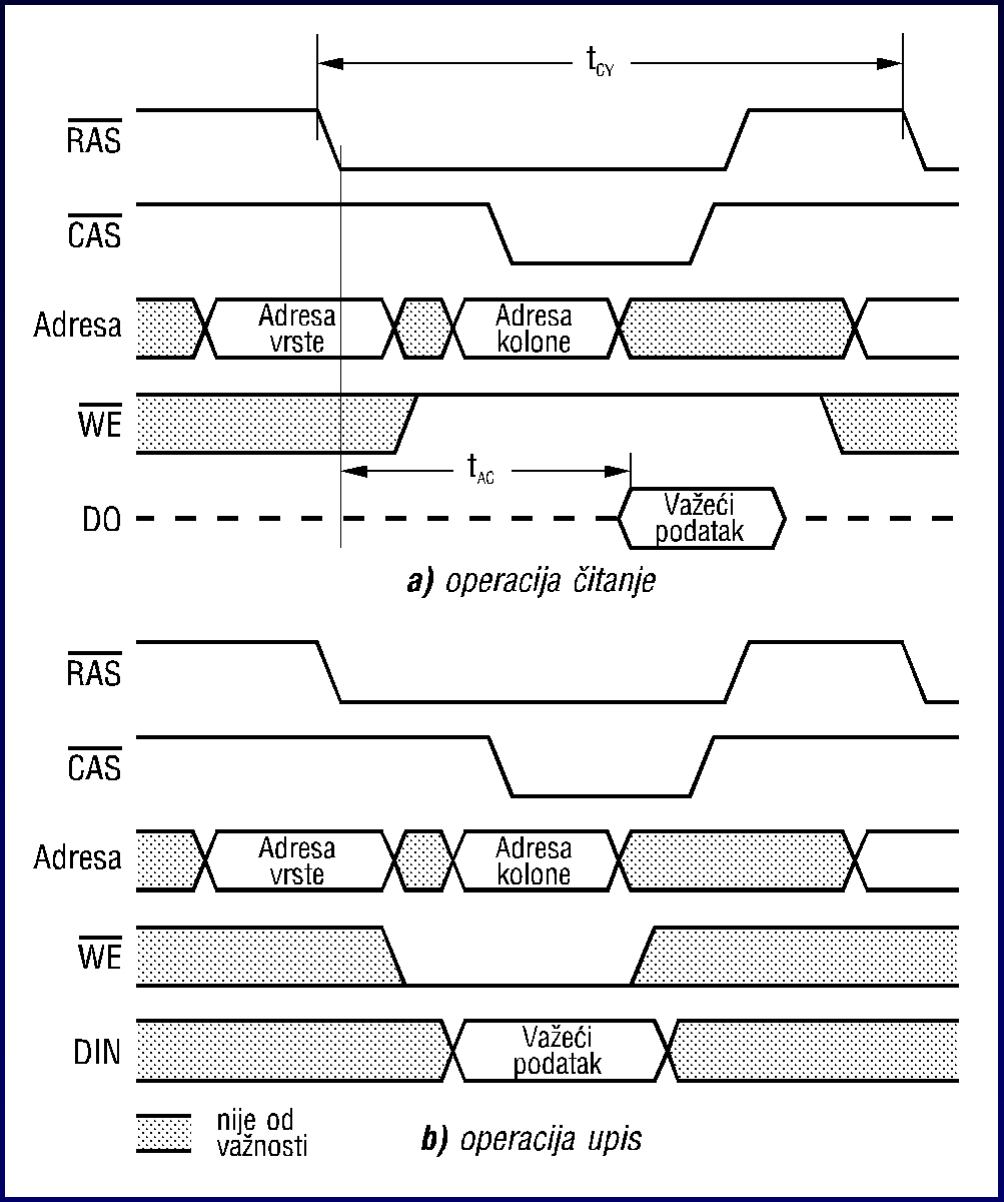
Povezivanje DRAM-ova



Povezivanje dinamičkih memorija



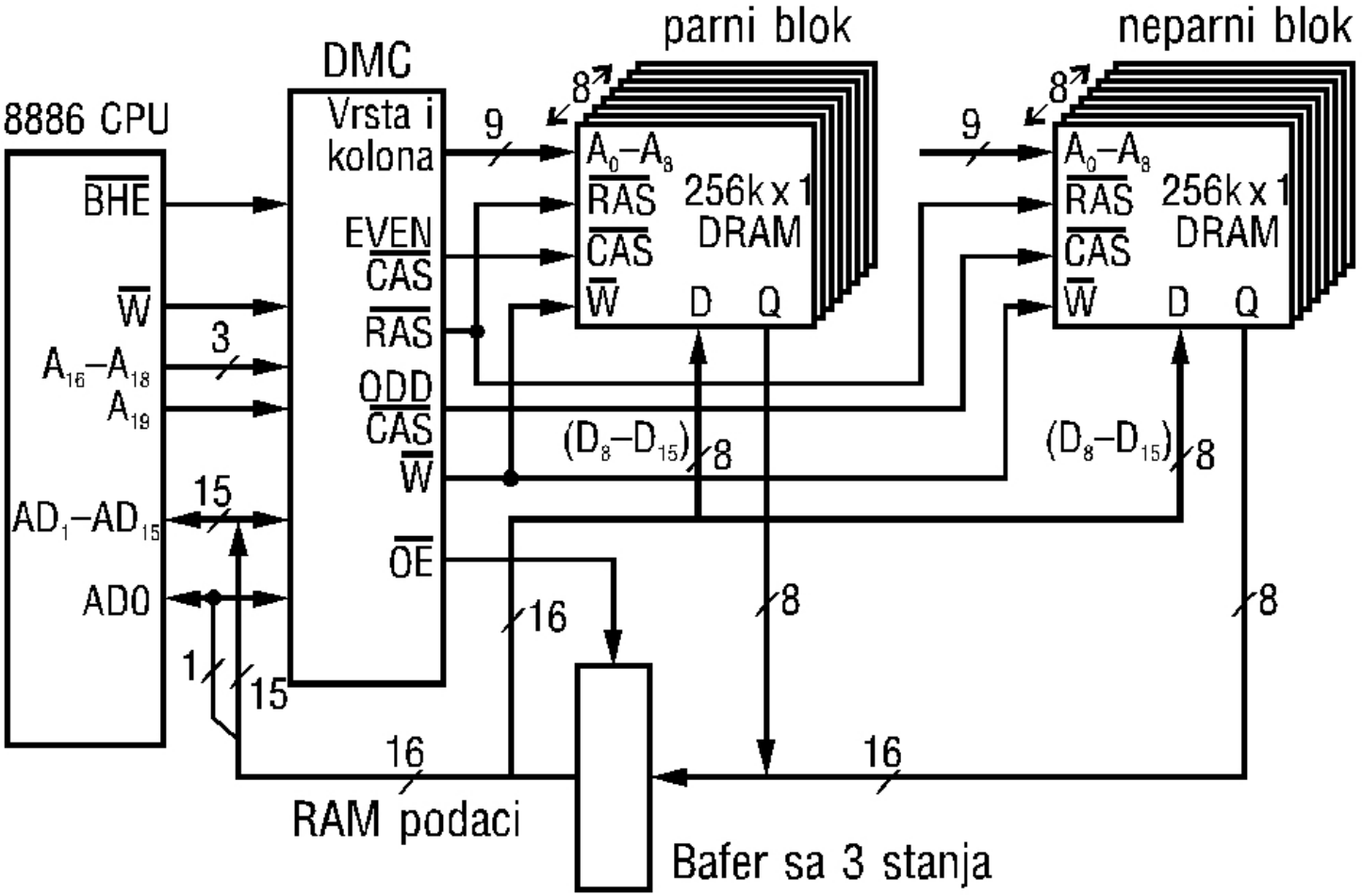
Tajming kod dinamičkih memorija



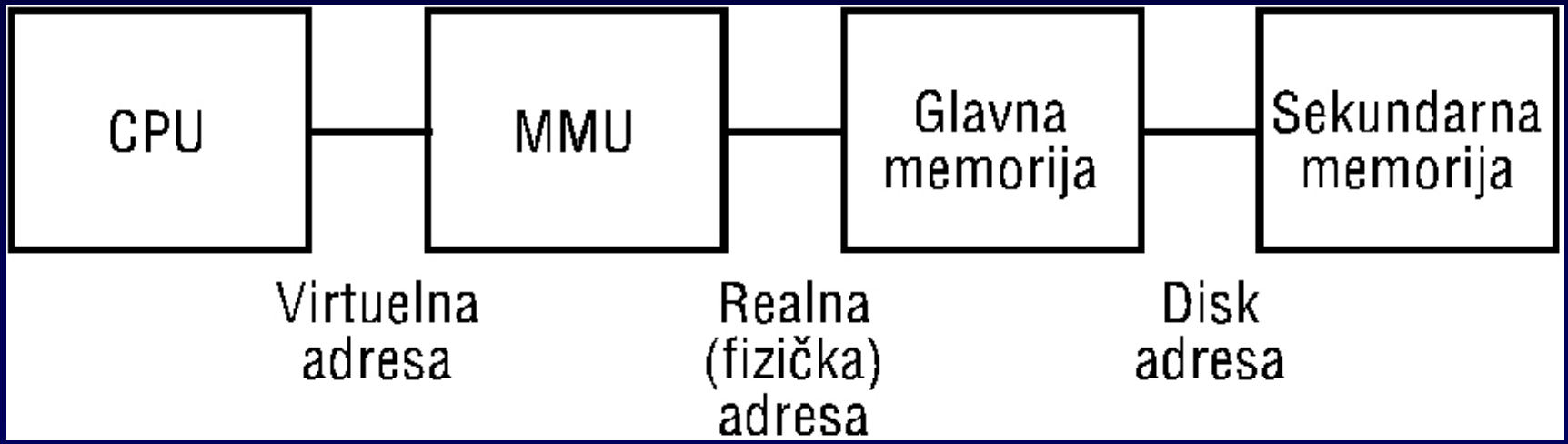
a) *operacija čitanje*

b) *operacija upis*

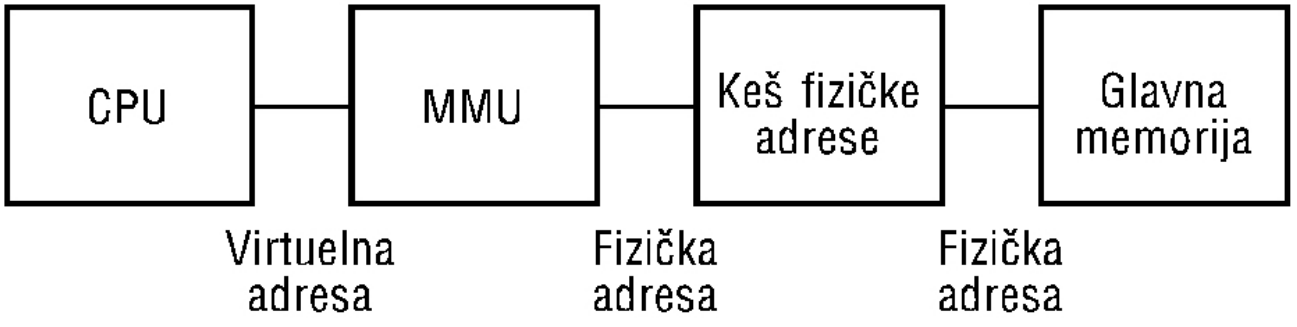
Povezivanje dinamičkih memorija



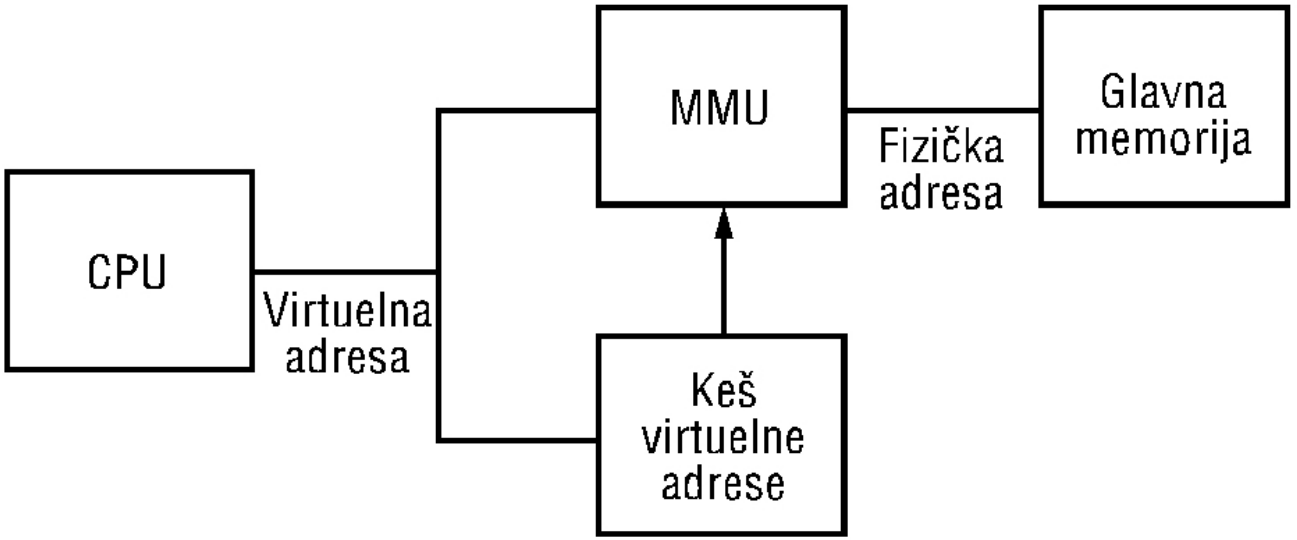
Virtuelna – fizička adresa



Položaj keša u memorijskoj hijerarhiji

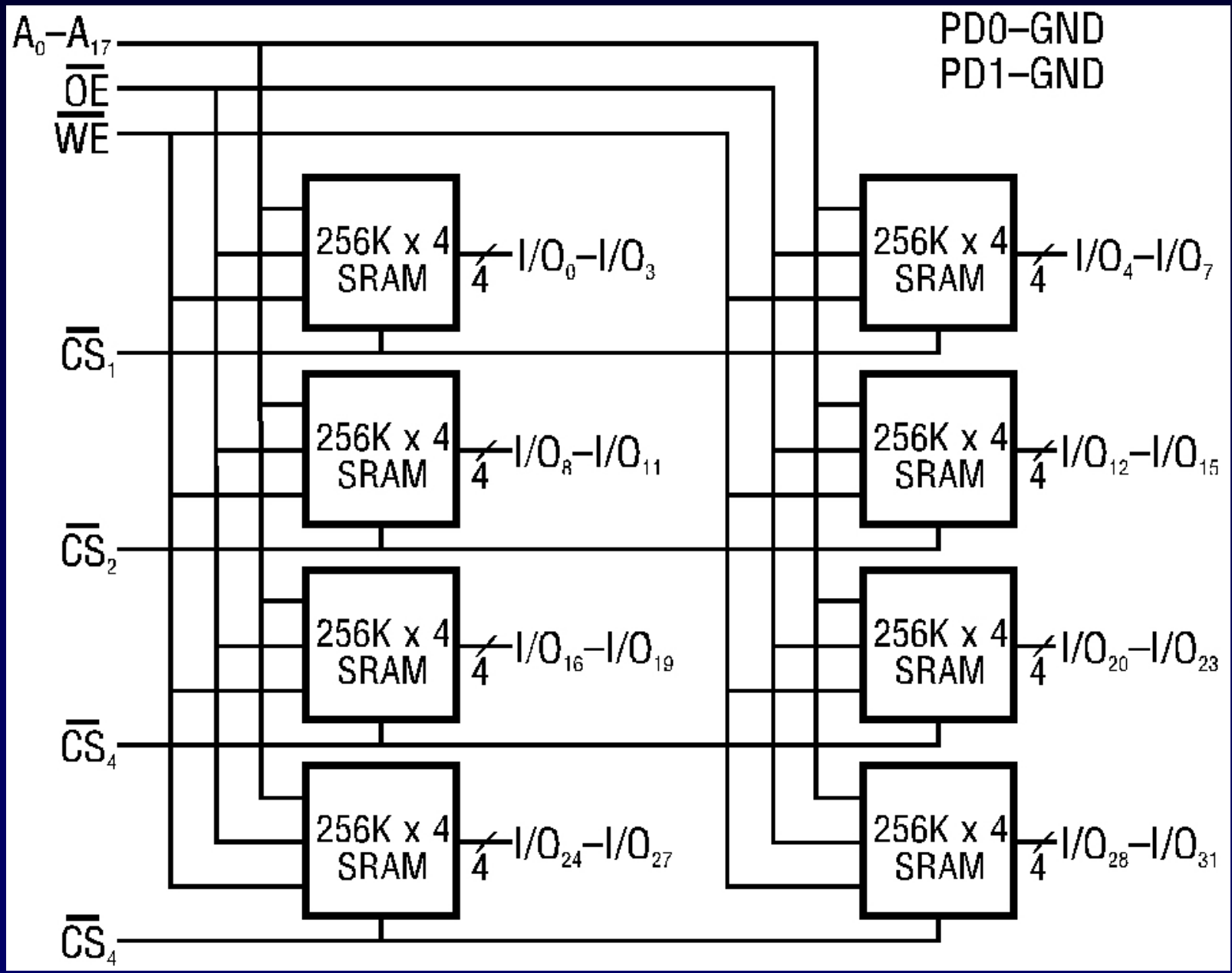


a) *keš na fizičkom putu*

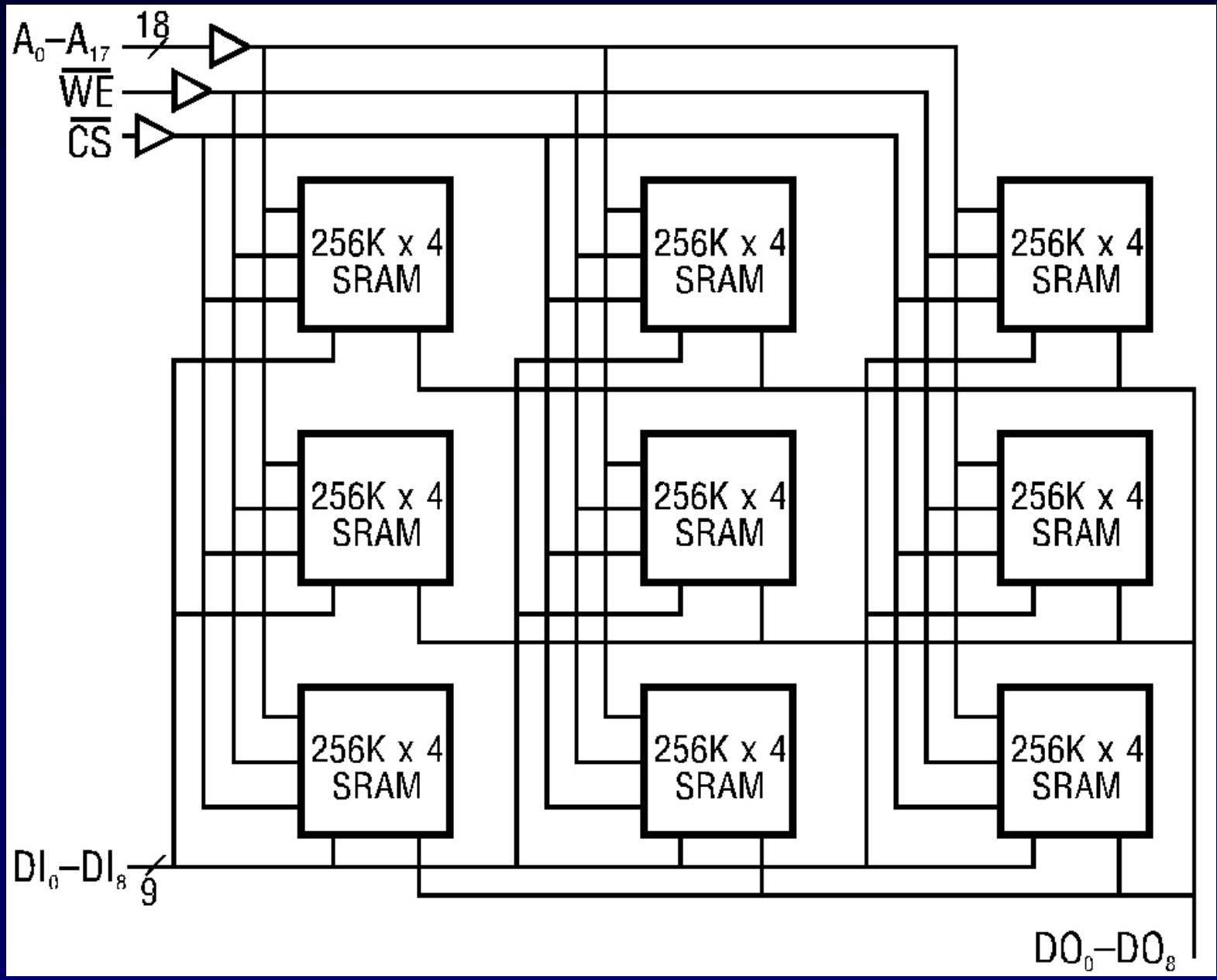


b) *keš na virtuelnom putu*

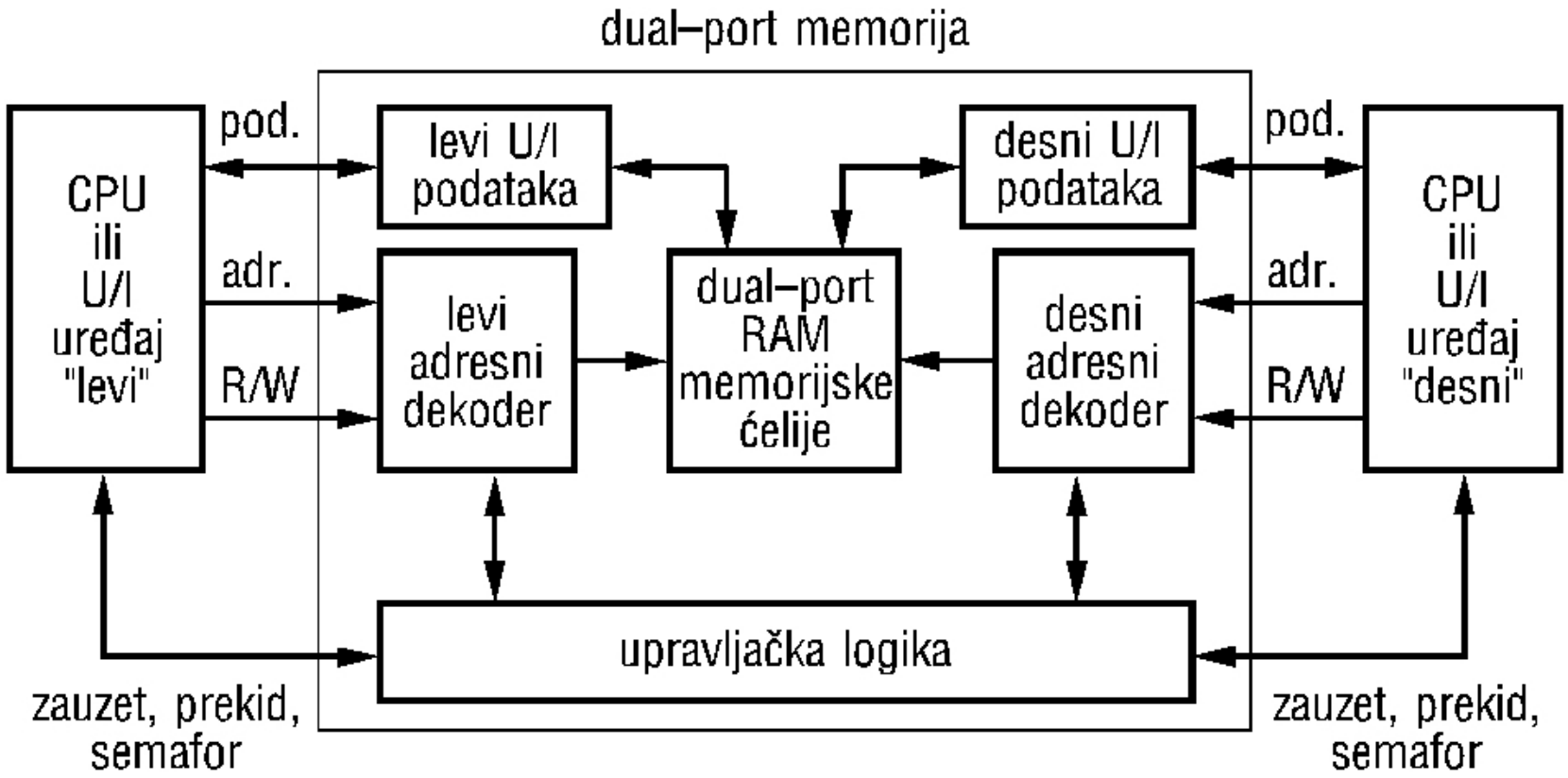
Memorijski moduli



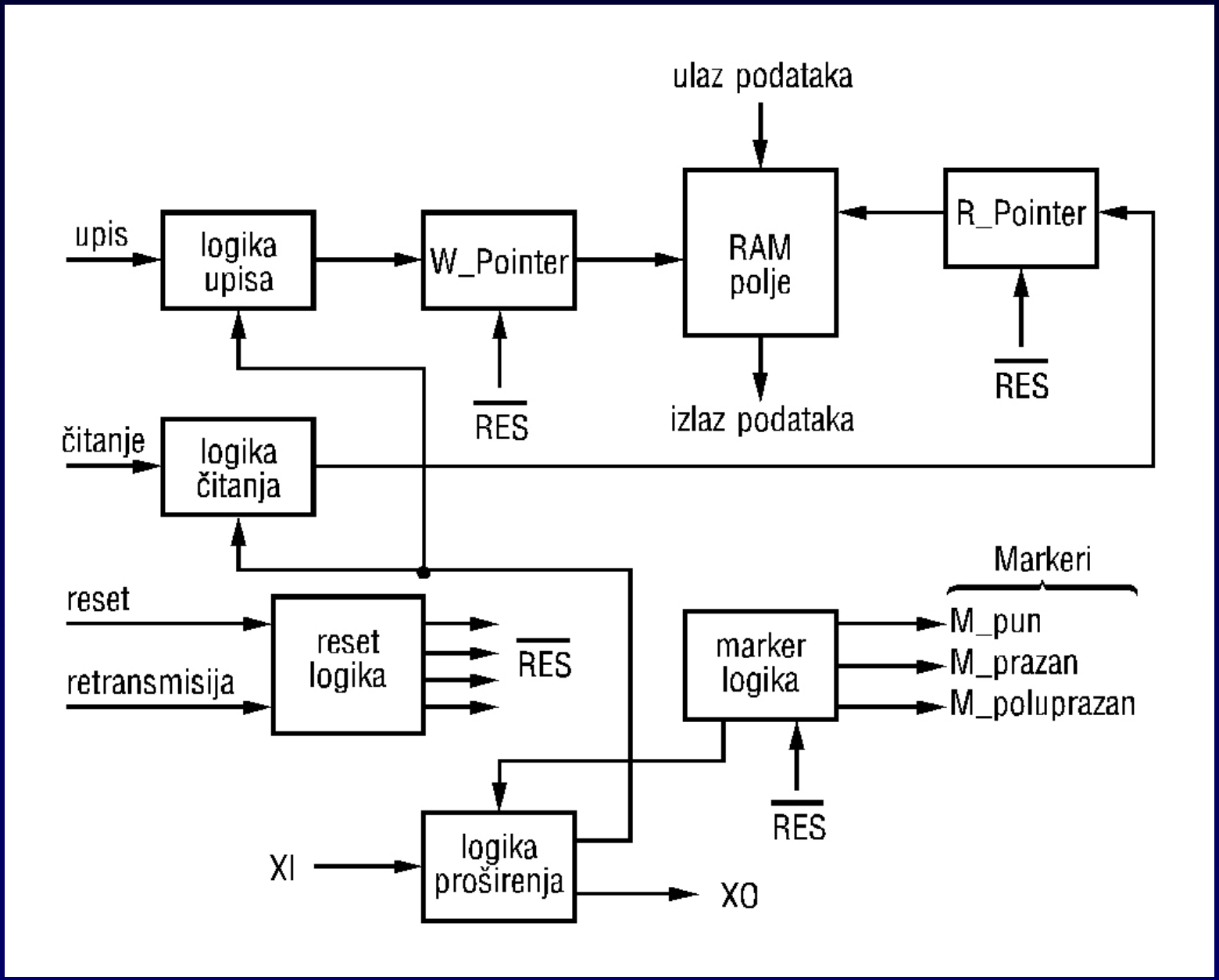
Memorijski moduli



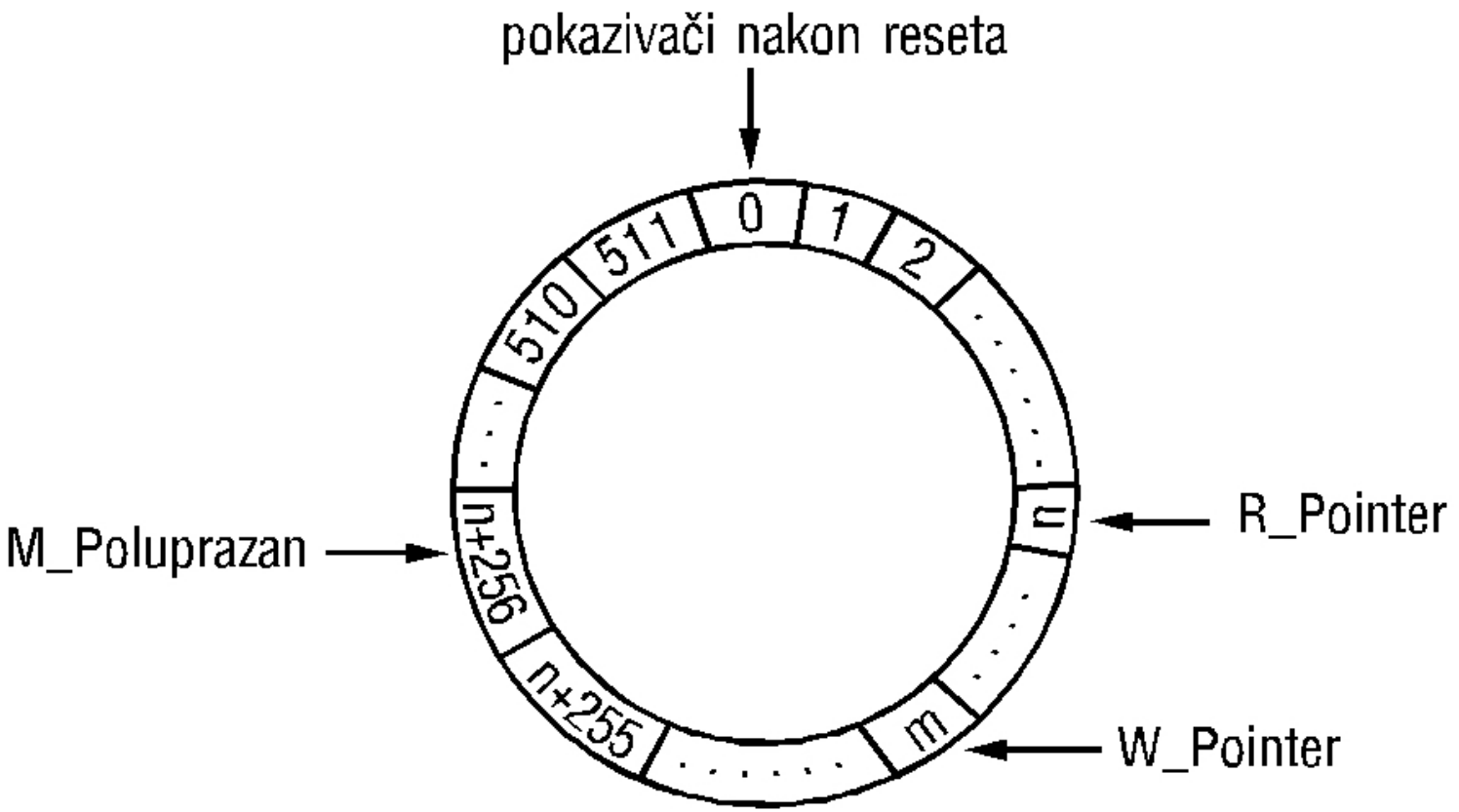
Dual-port memorija



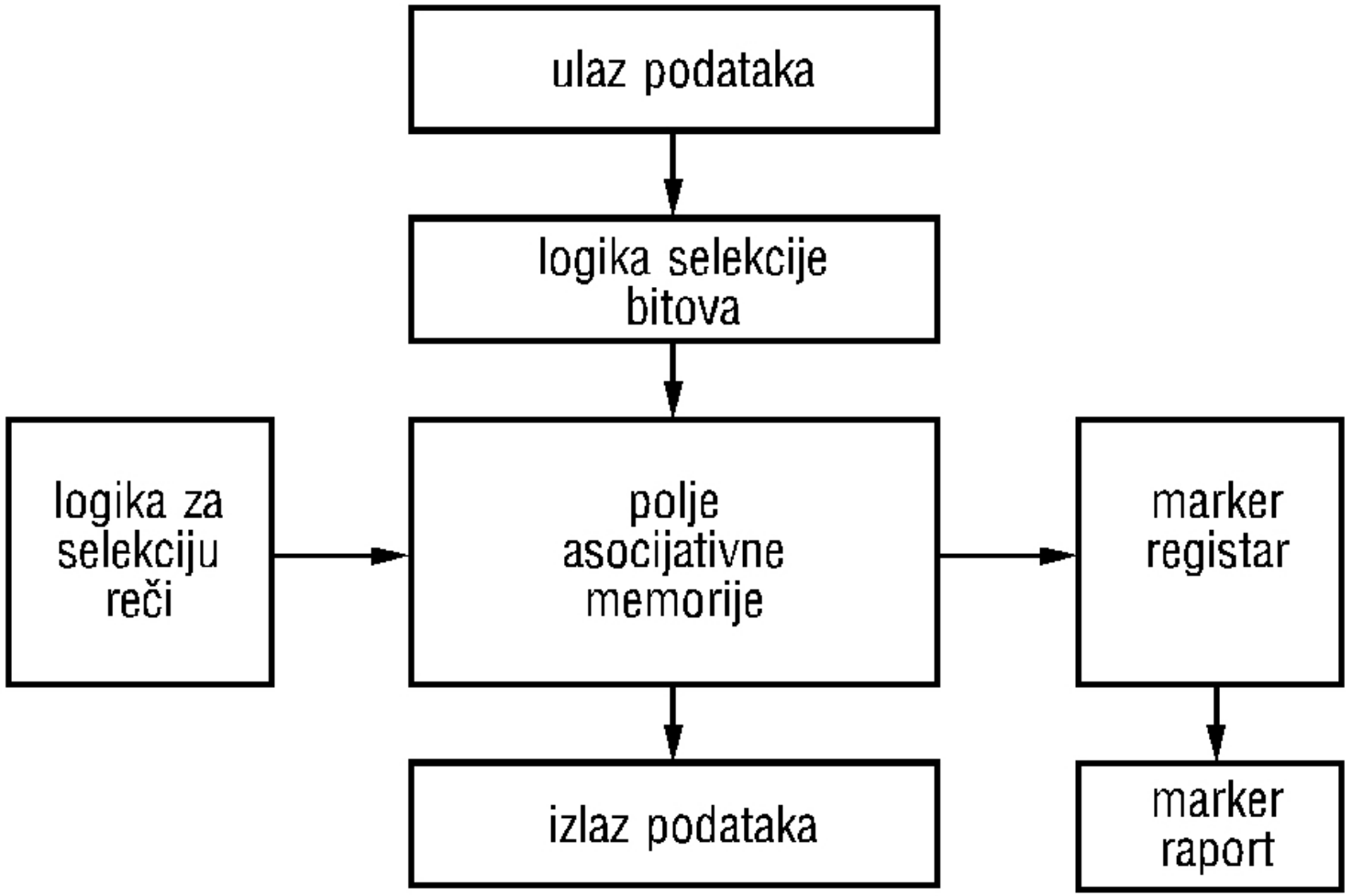
Struktura kružnog bafera



Princip rada kružnog bafera

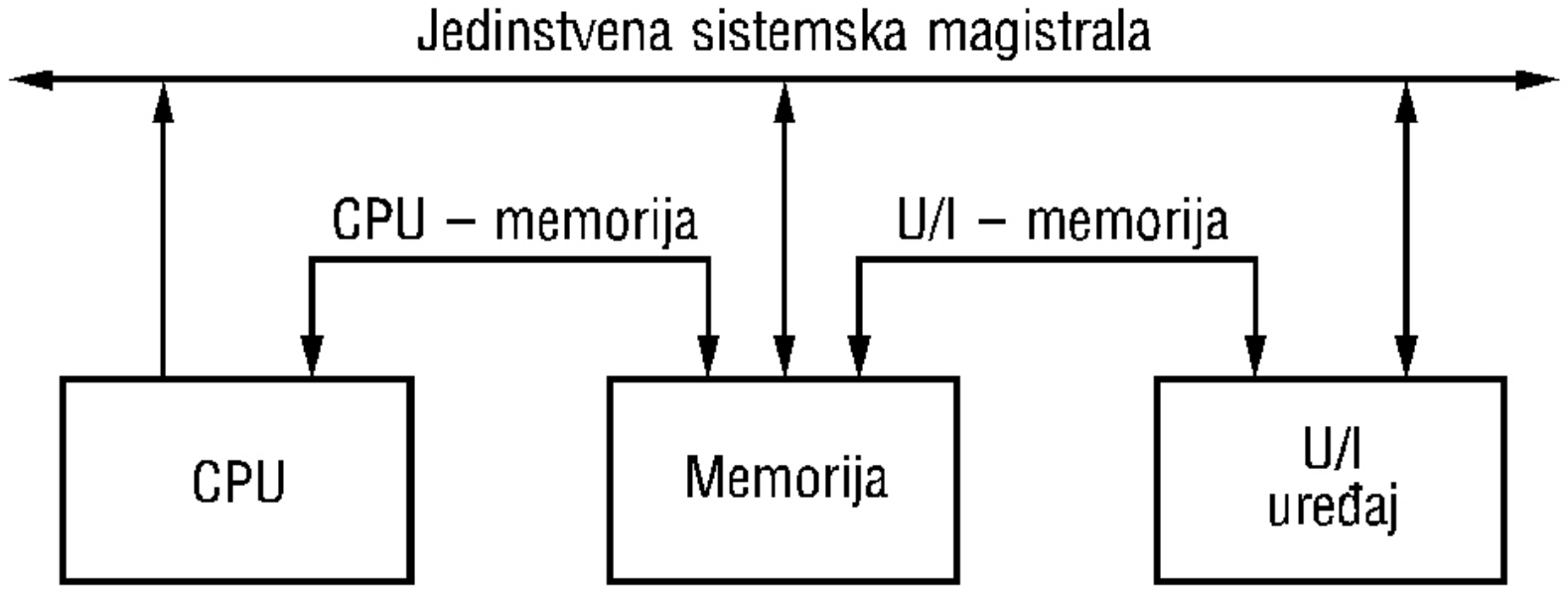


Asocijativna memorija

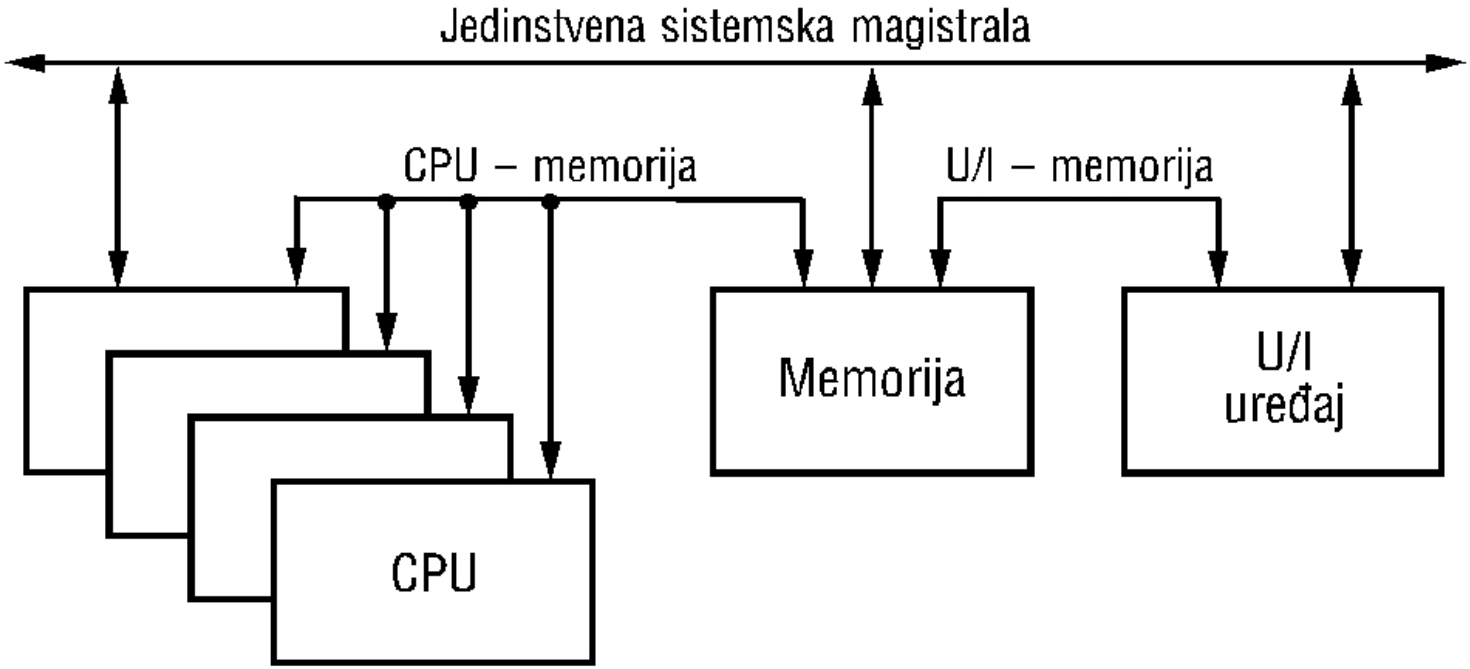


Asocijativna memorija

Memorija u sistemu



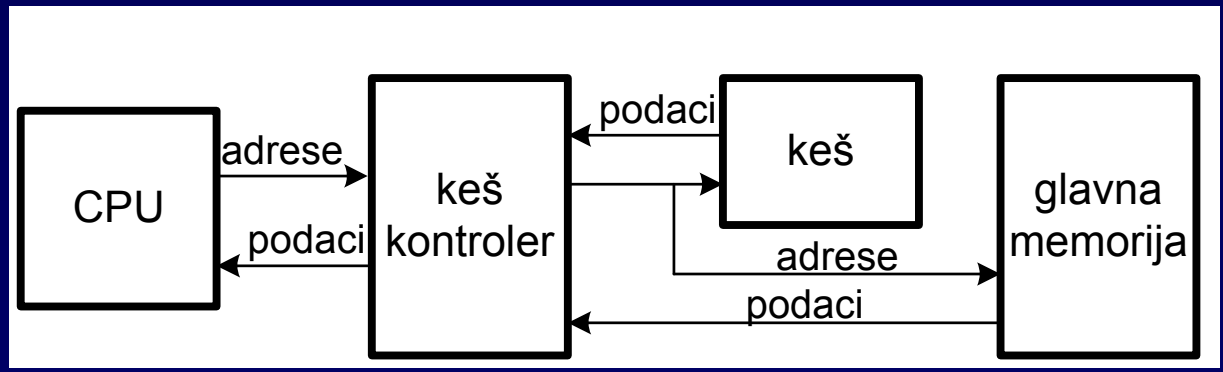
Memorija u sistemu sa većim brojem procesora



Keš memorija

- Princip rada keša se zasniva na lokalnosti- koja može biti:
  - vremenska lokalnost
  - prostorna lokalnost

Keš kontroler posreduje izmedju CPU-a i memorijskog sistema koga čine keš i glavna memorija.



Temporal- and spatial-locality of reference

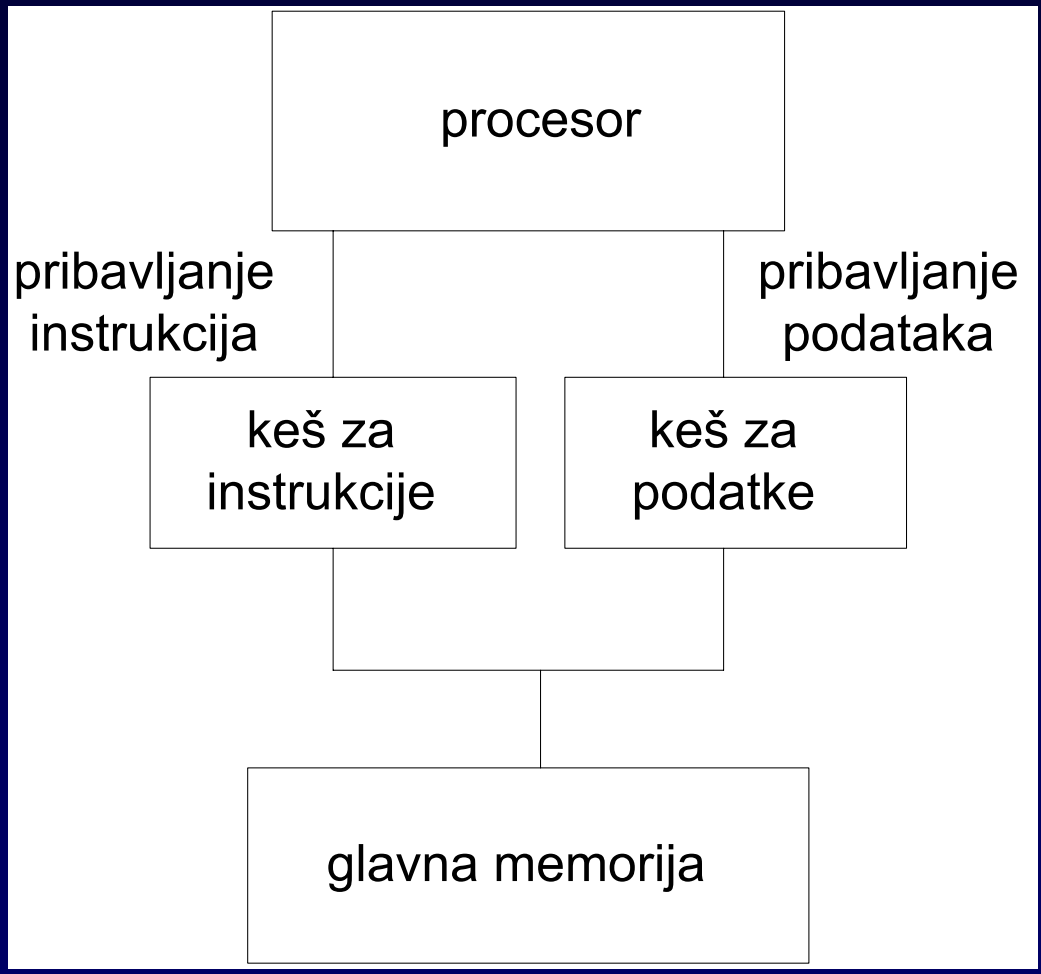
- When exhibiting spatial locality, a program accesses neighboring memory locations.
- When displaying temporal locality of reference a program repeatedly accesses the same memory location during a short time period.

Both forms of locality occur in the following Pascal code segment:

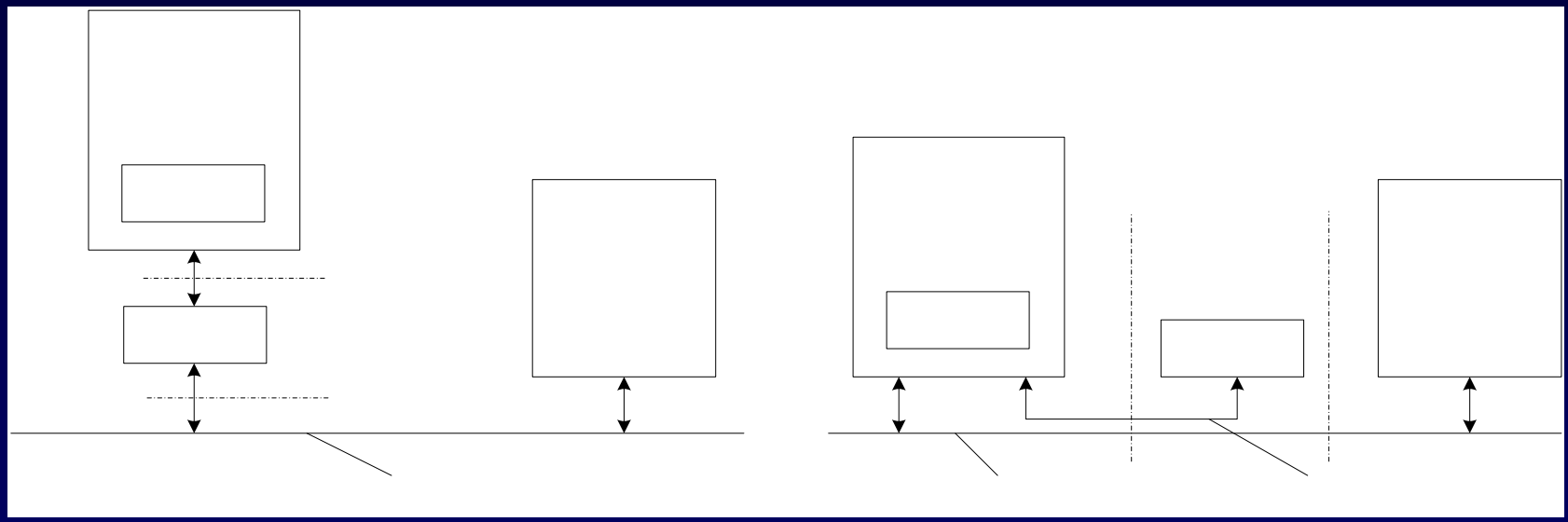
```
for i := 0 to 10 do
        A [i] := 0;
```

- Temporal locality - the CPU accesses i at three points in a short time period.
- Spatial locality - assuming that Pascal stores the elements of A into consecutive memory locations6, each loop iteration accesses adjacent memory locations.

Harvard arhitektura



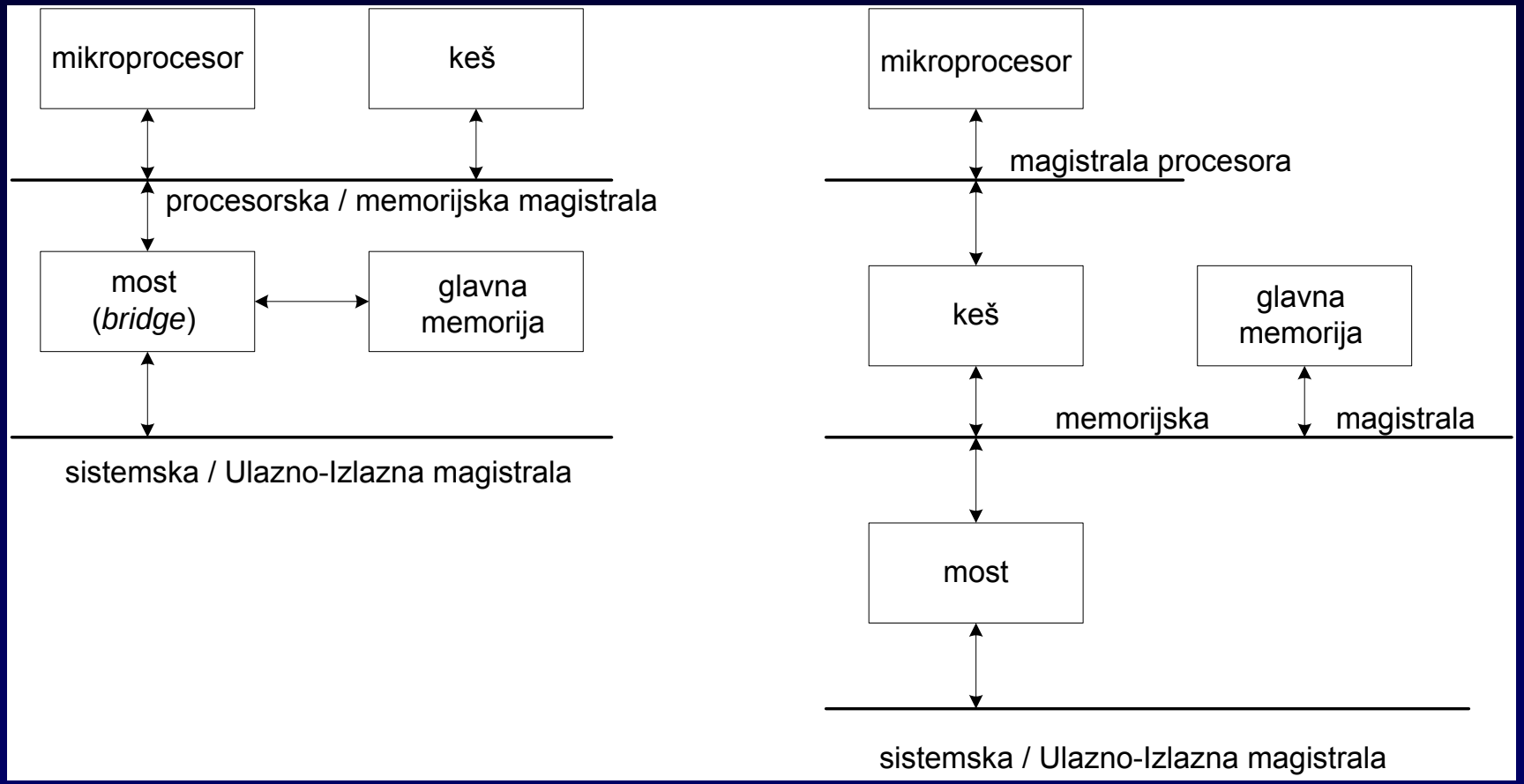
L1 i L2 keš



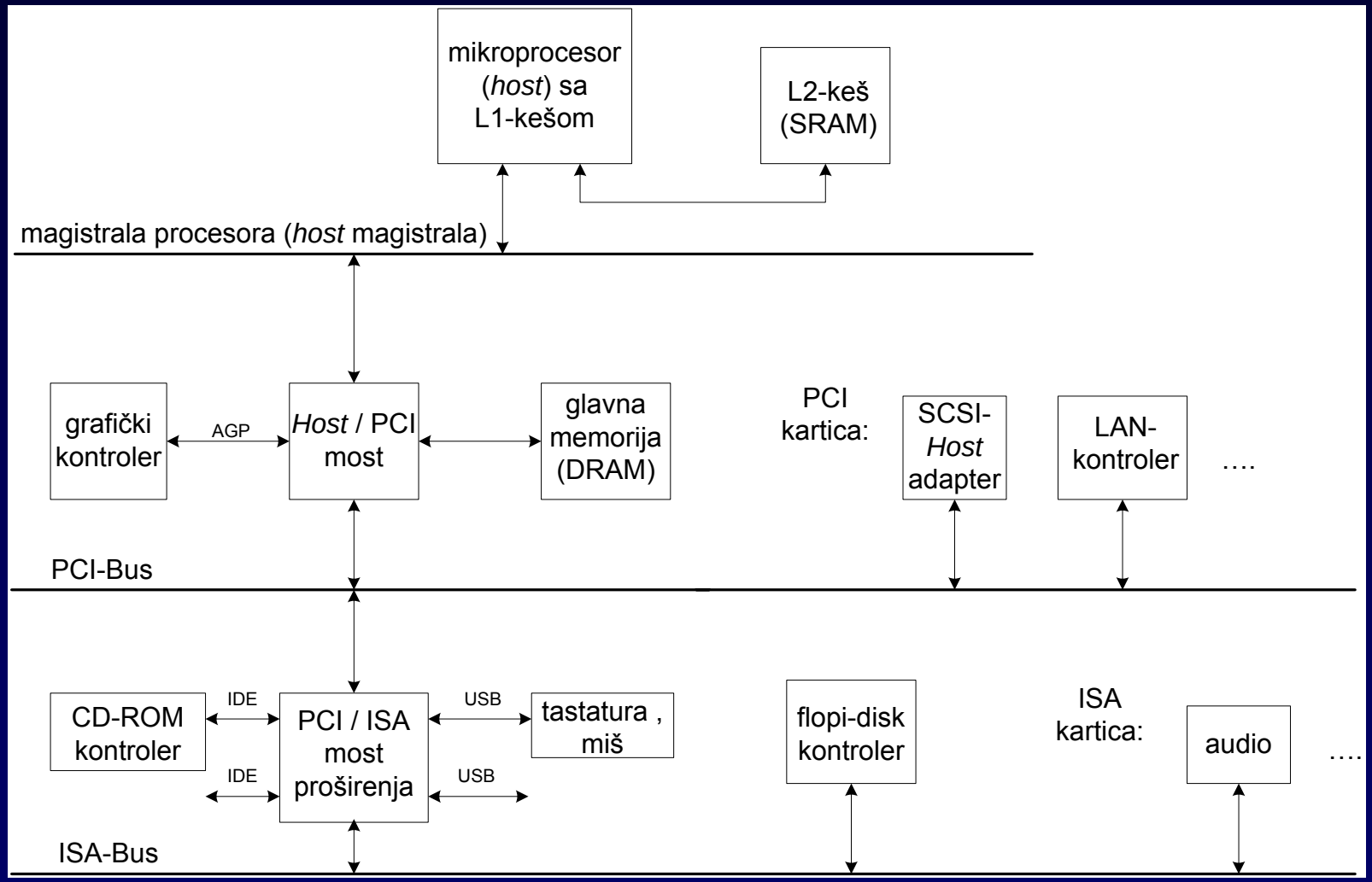
mikroprocesor

L 1-keš

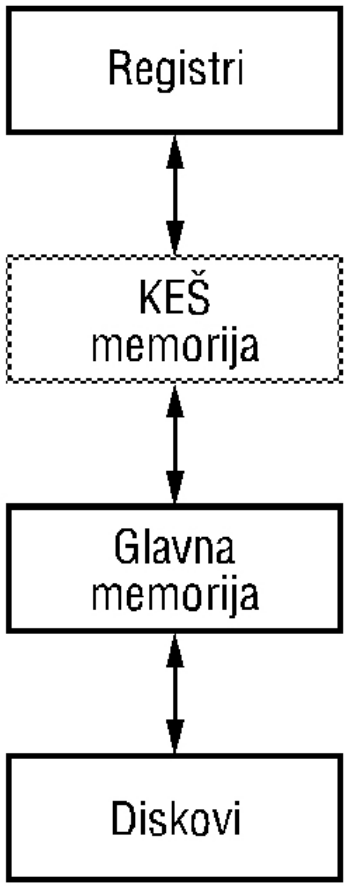
Položaj keša u sistemu



Položaj keša u sistemu PC-a

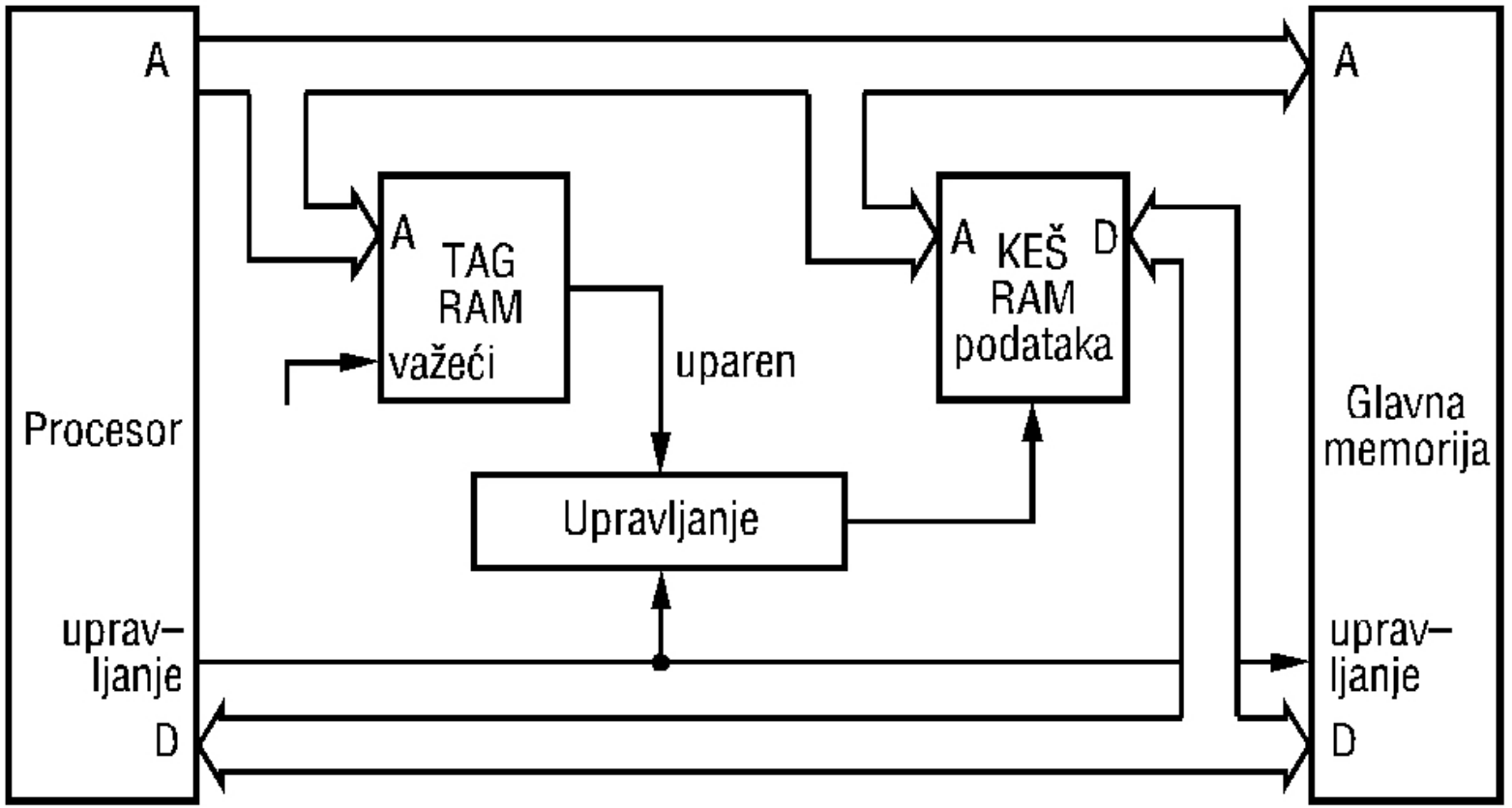


Razlika u vremenima pristupa

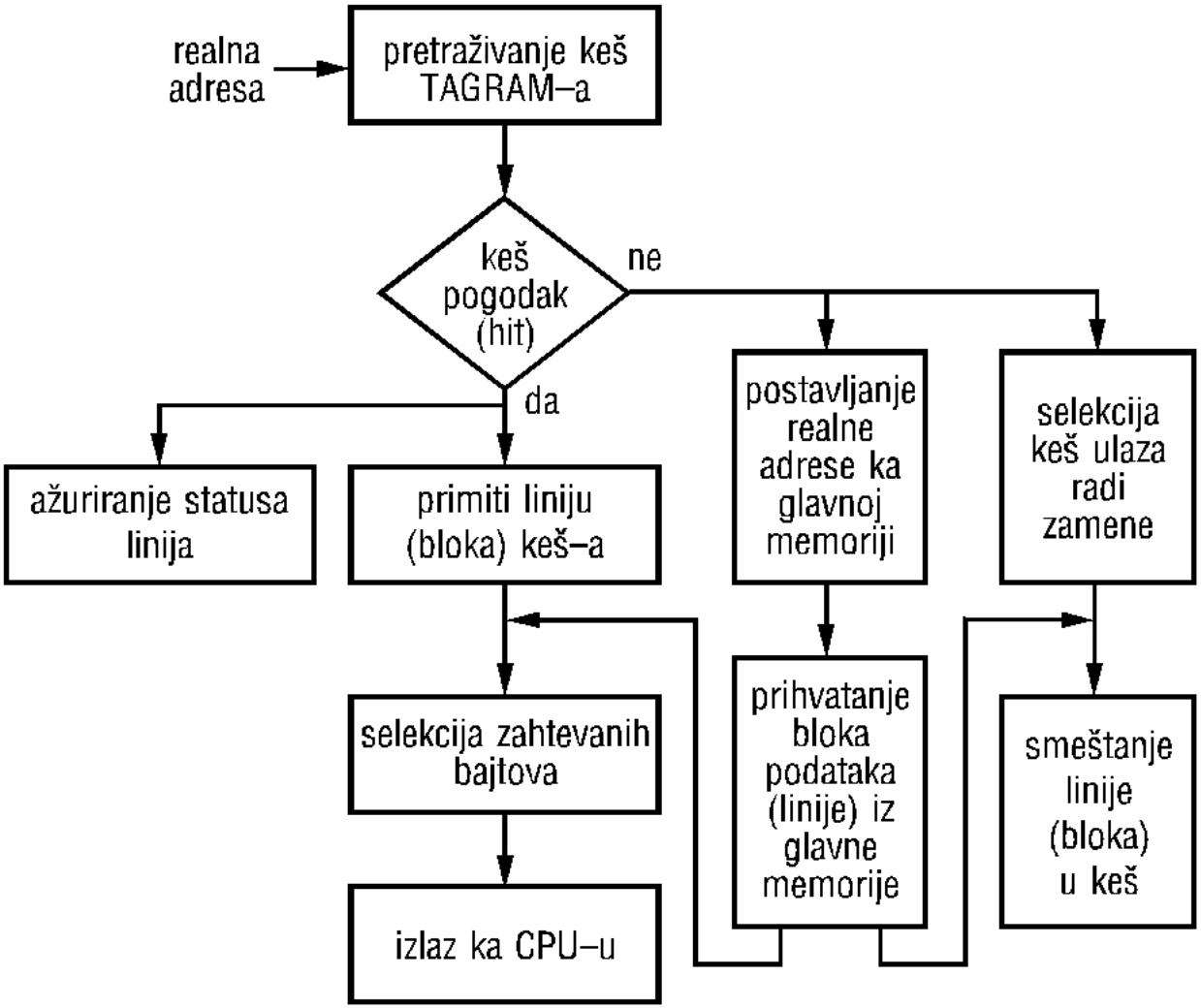


	Vreme pristupa	Kapacitet
Registri	5–25 ns	512 B
KEŠ memorija	25–90 ns	1–512 kB
Glavna memorija	90–500 ns	640 kB–128 MB
Diskovi	5–30 ms	40 MB–2 GB

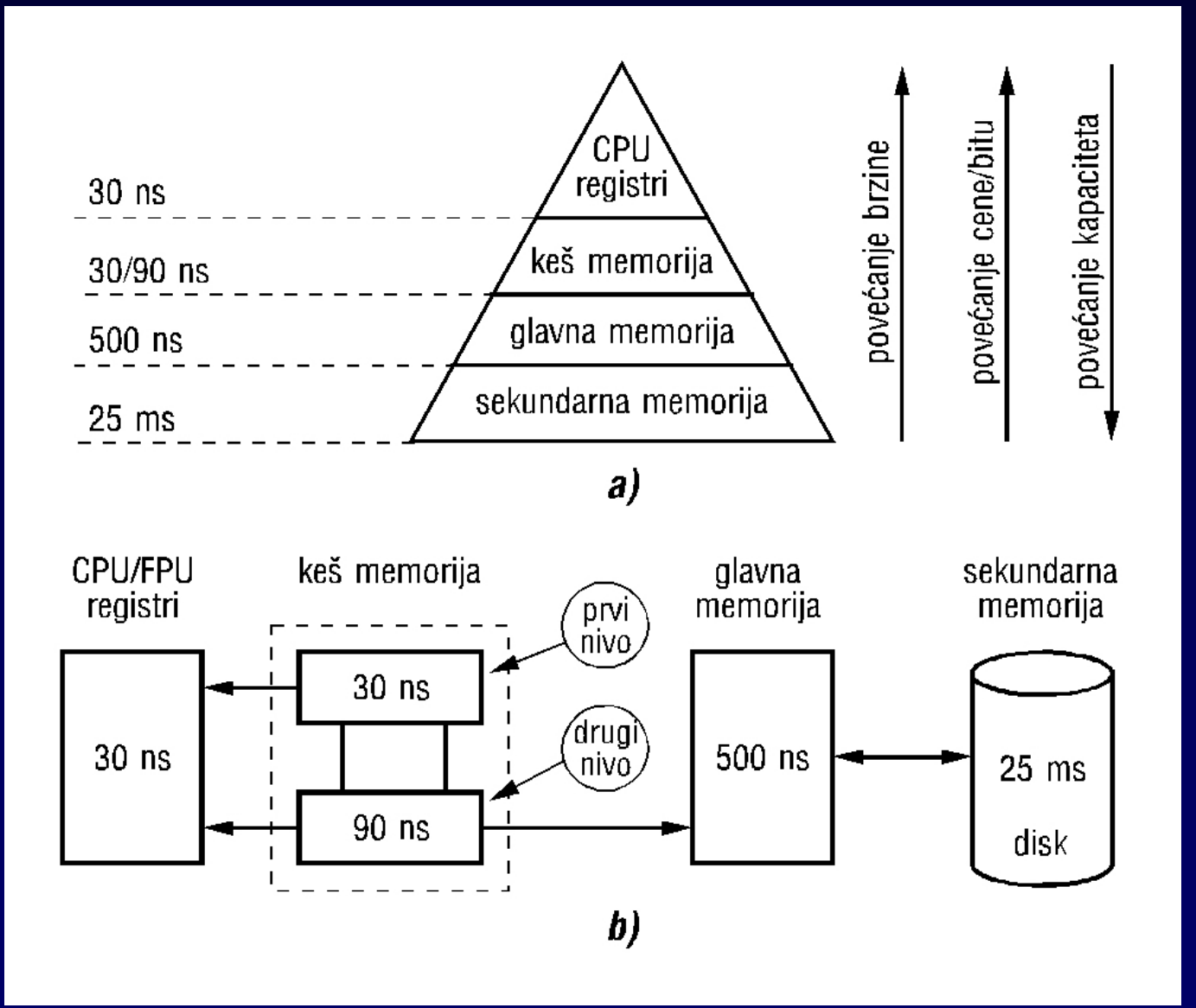
Gradivni delovi keš memorije



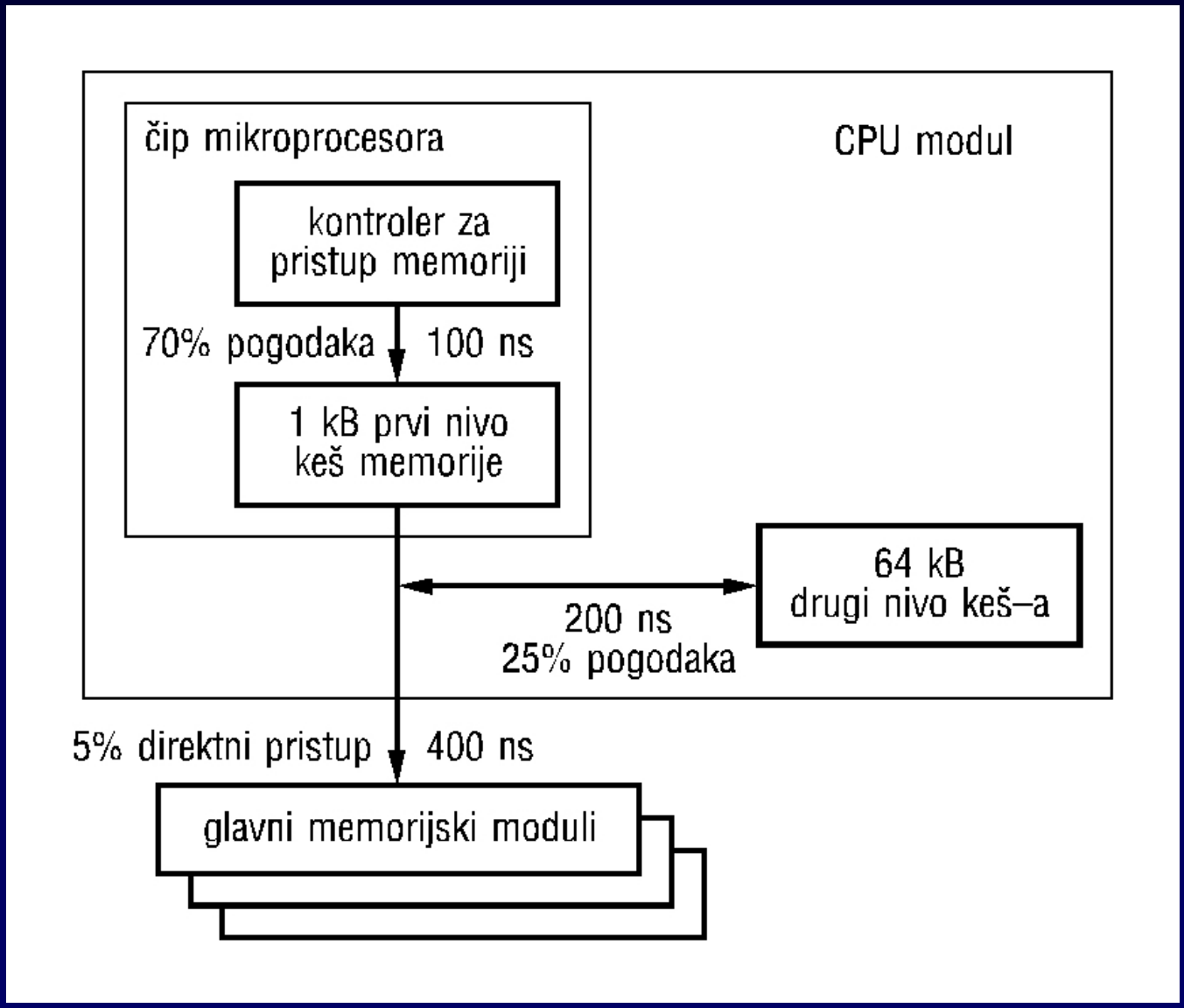
Princip rada keš memorije



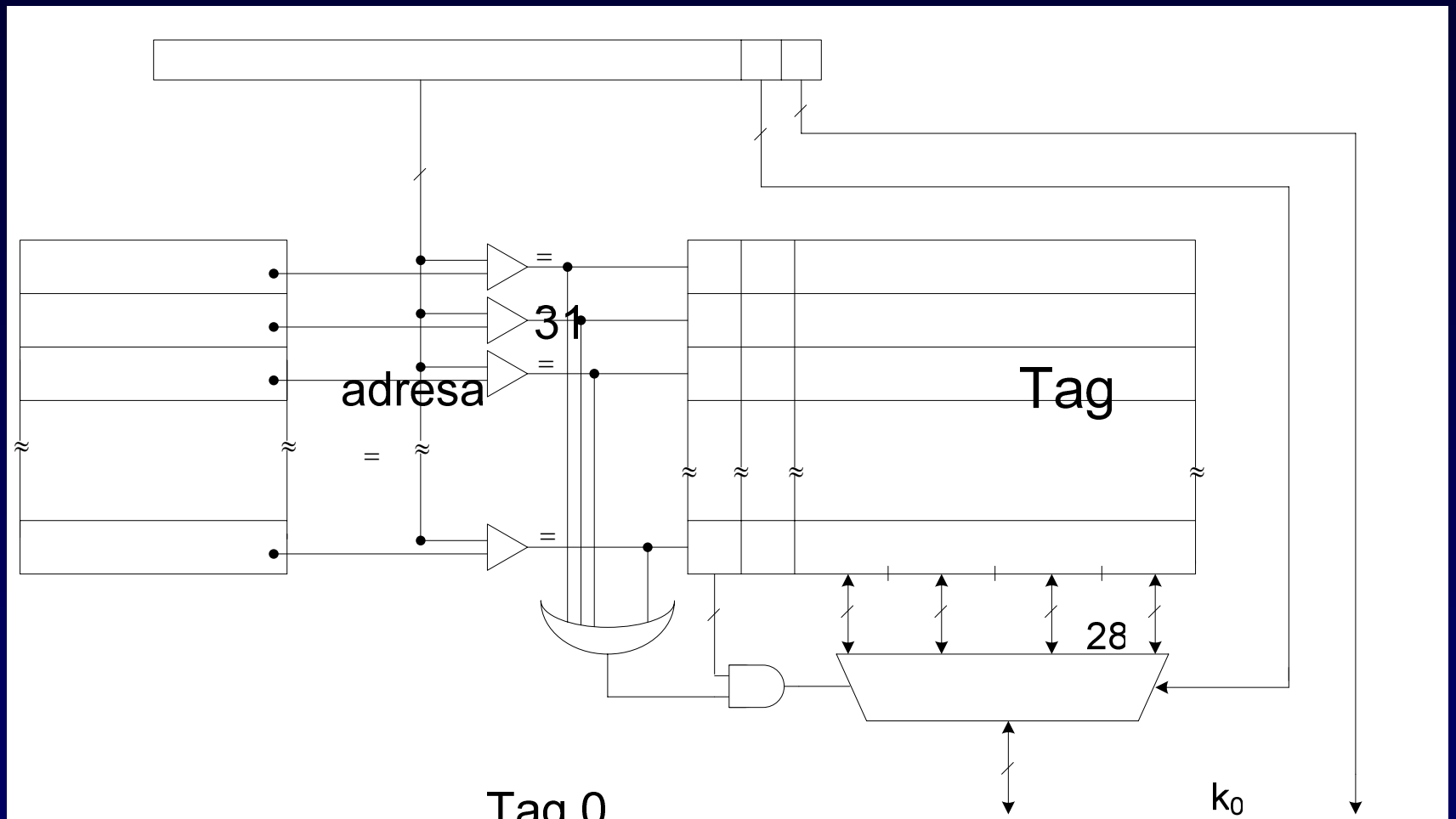
Keš memorija – primer organizacije



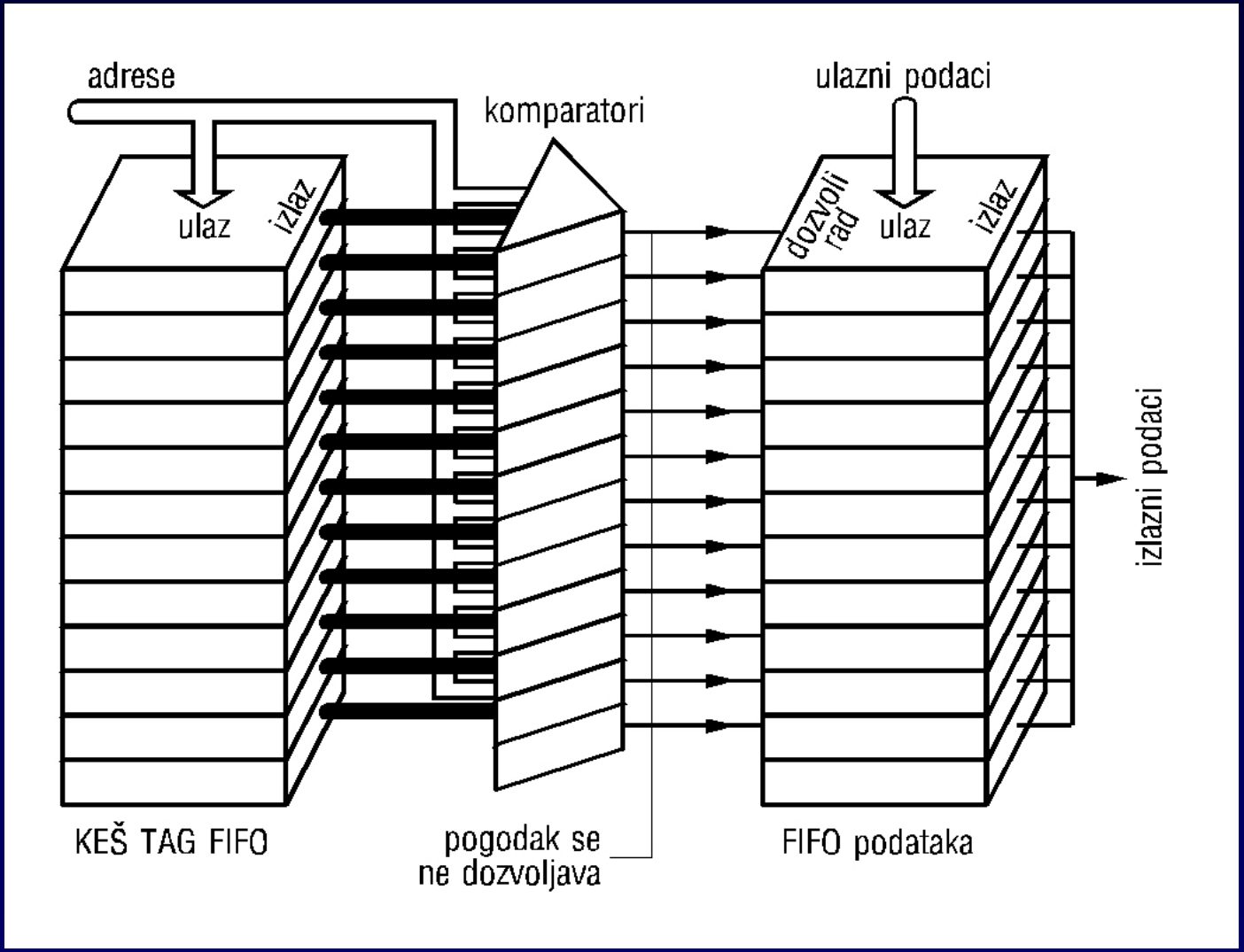
Keš memorija – primer realizacije u dva nivoa



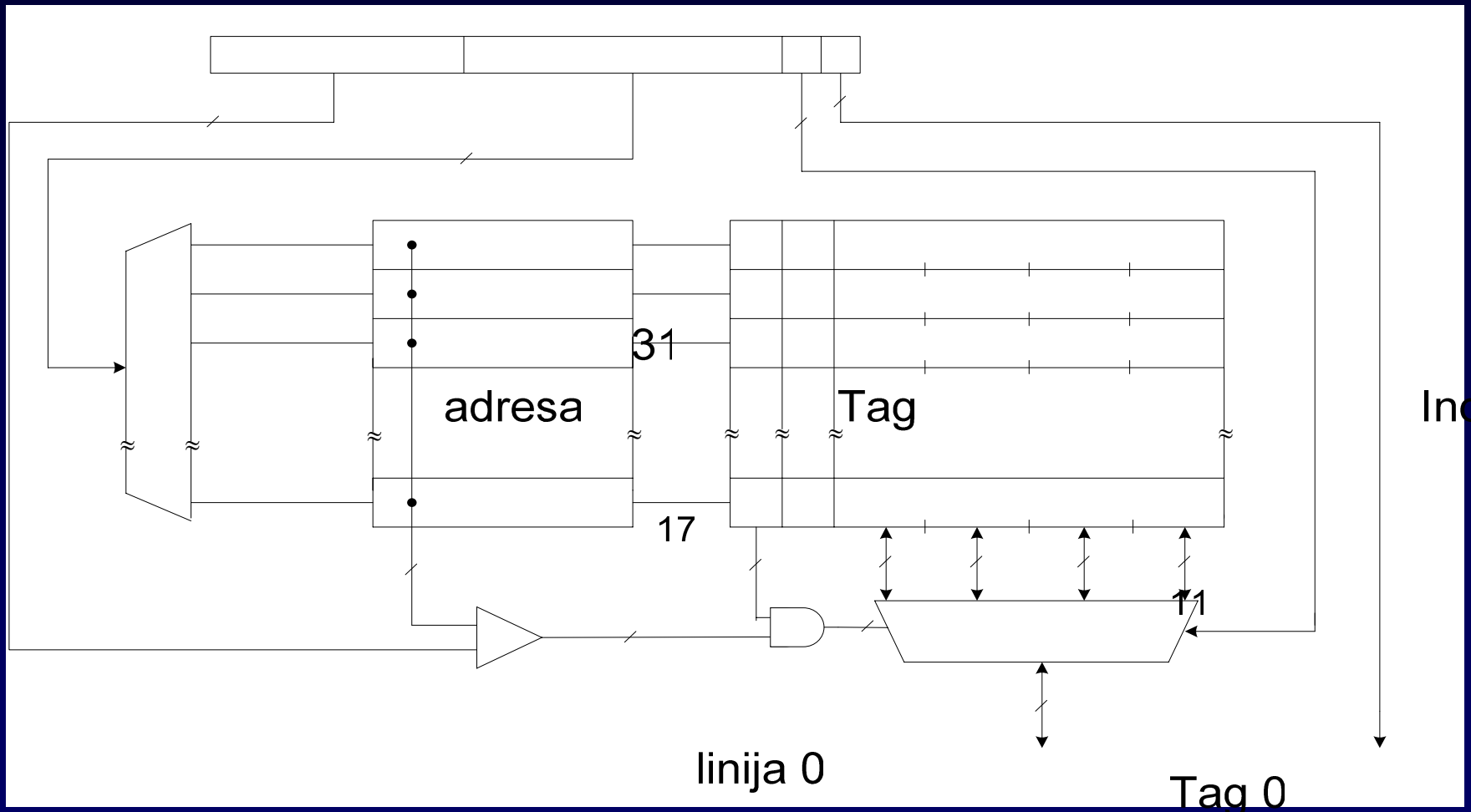
Asocijativna keš memorija



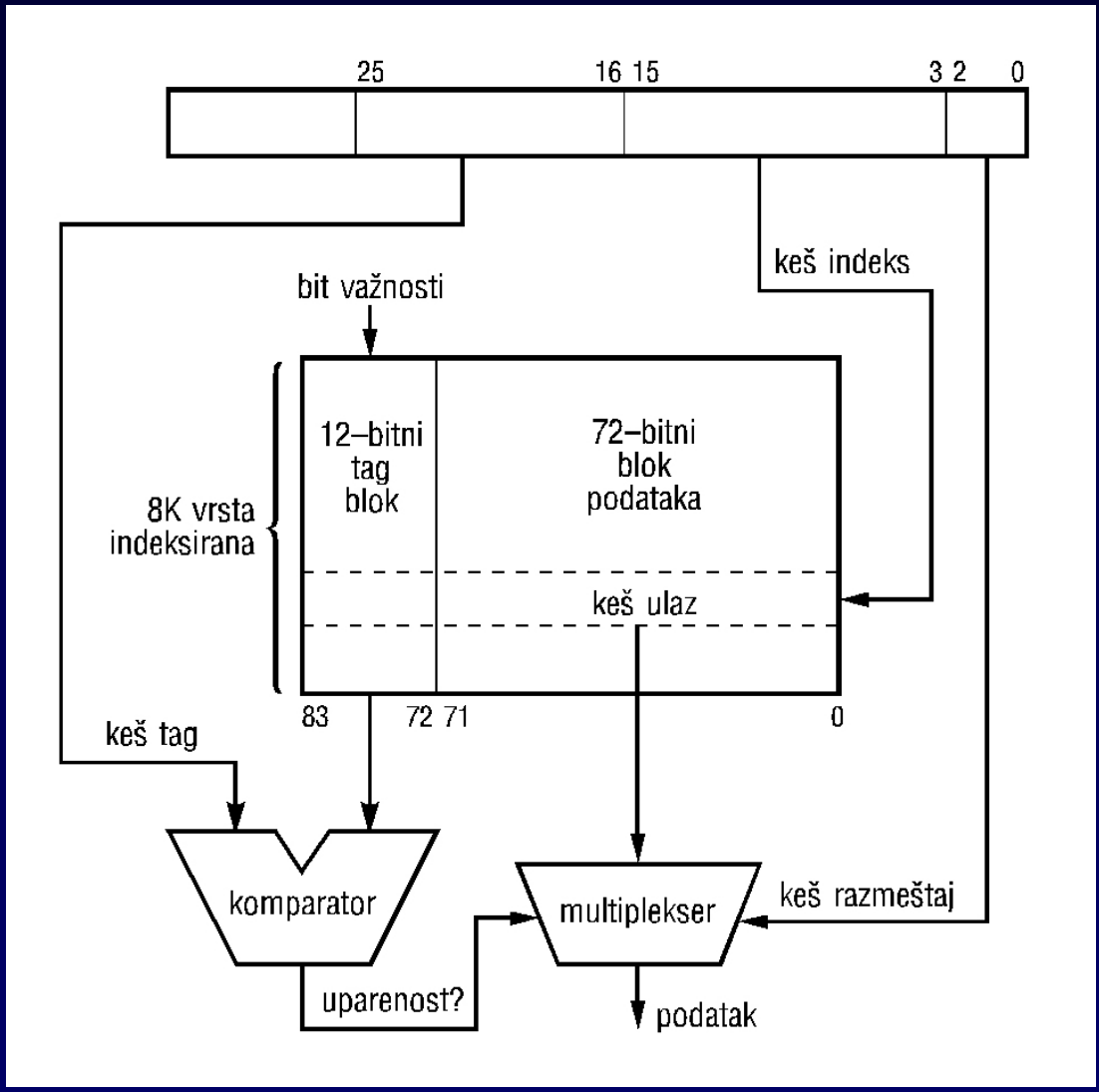
Asocijativna keš memorija sa FIFO politikom zamene



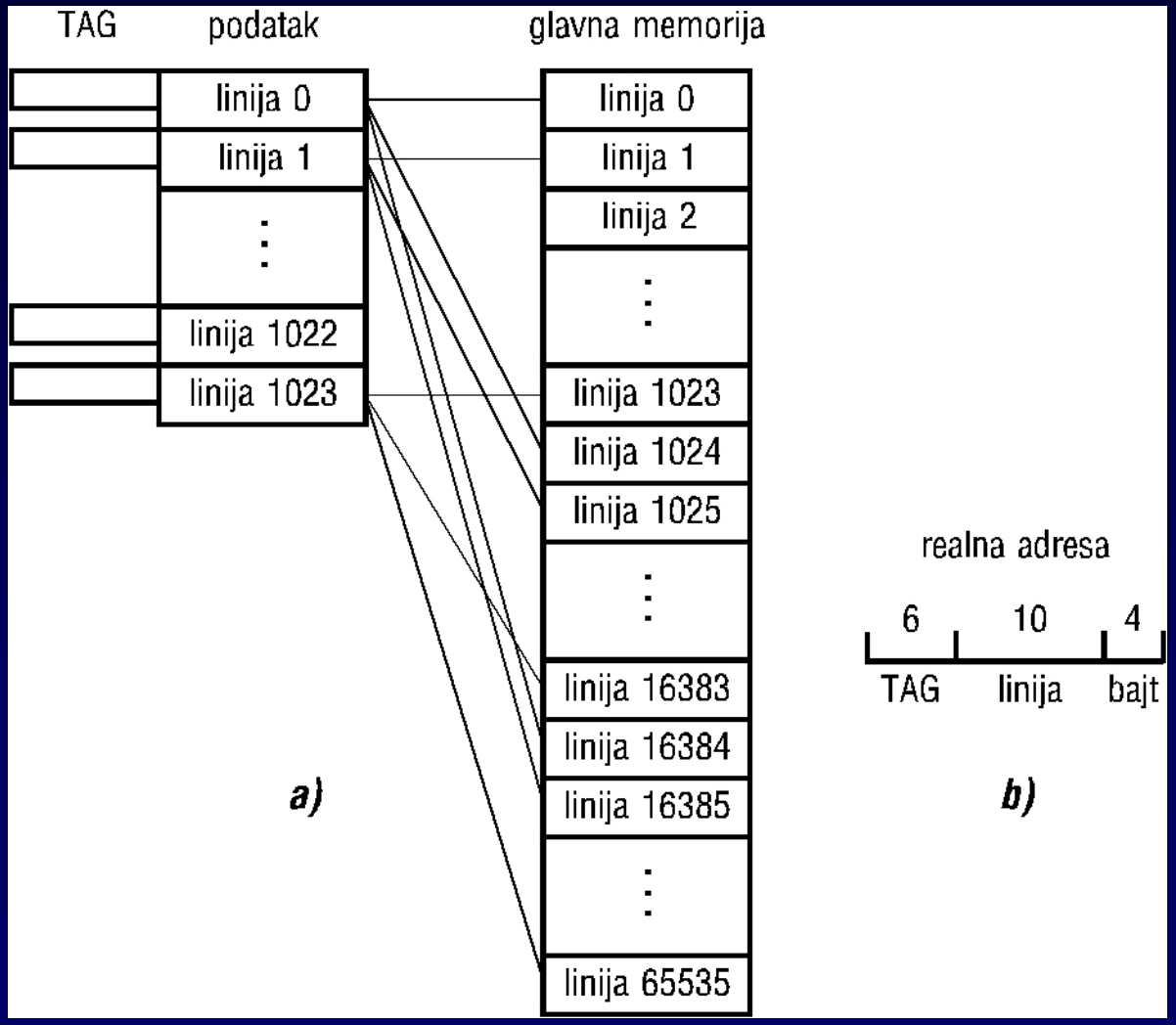
Direktno preslikana keš memorija



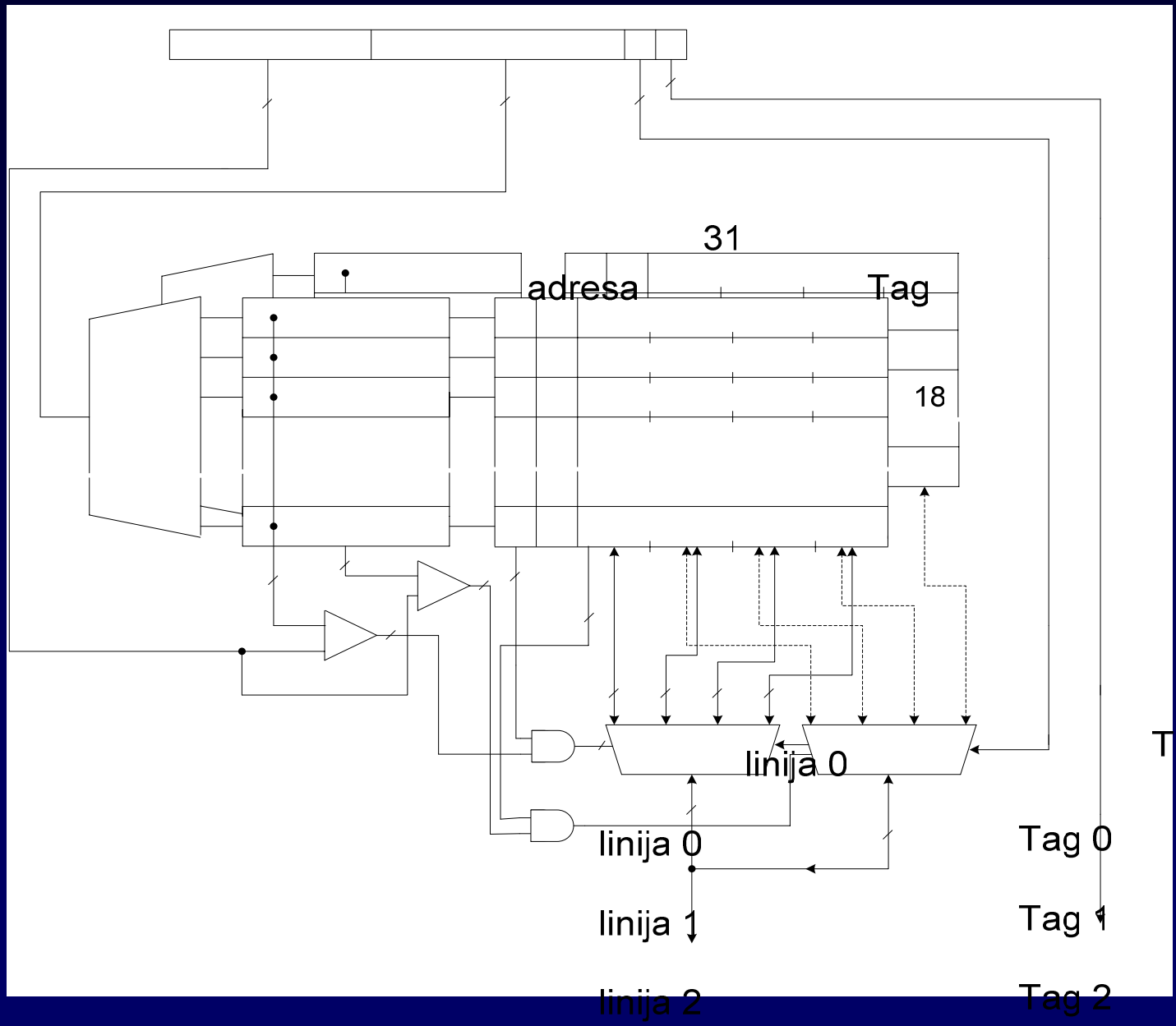
Primer direktno preslikane keš memorije



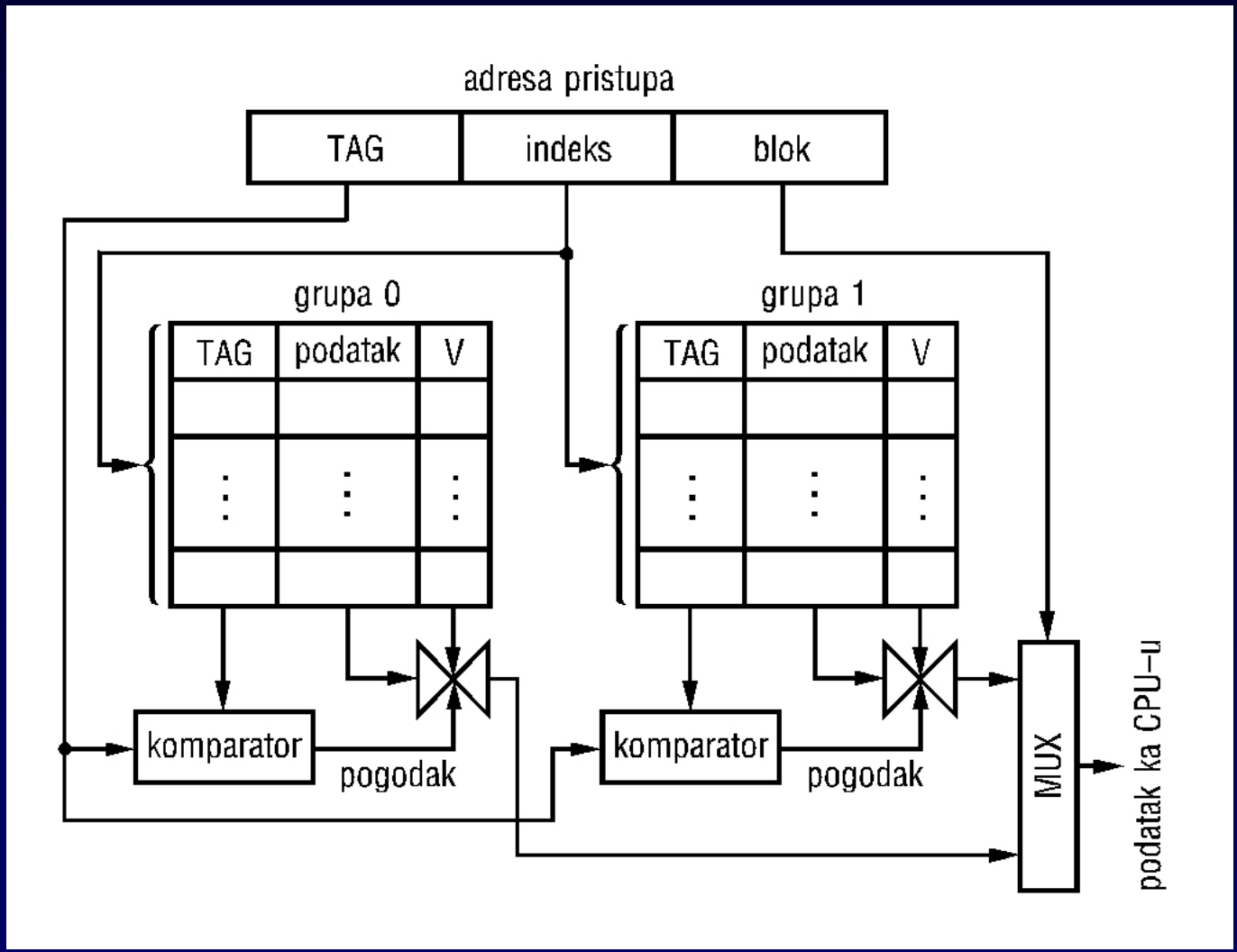
Preslikavanje adresa kod direktno preslikane keš memorije



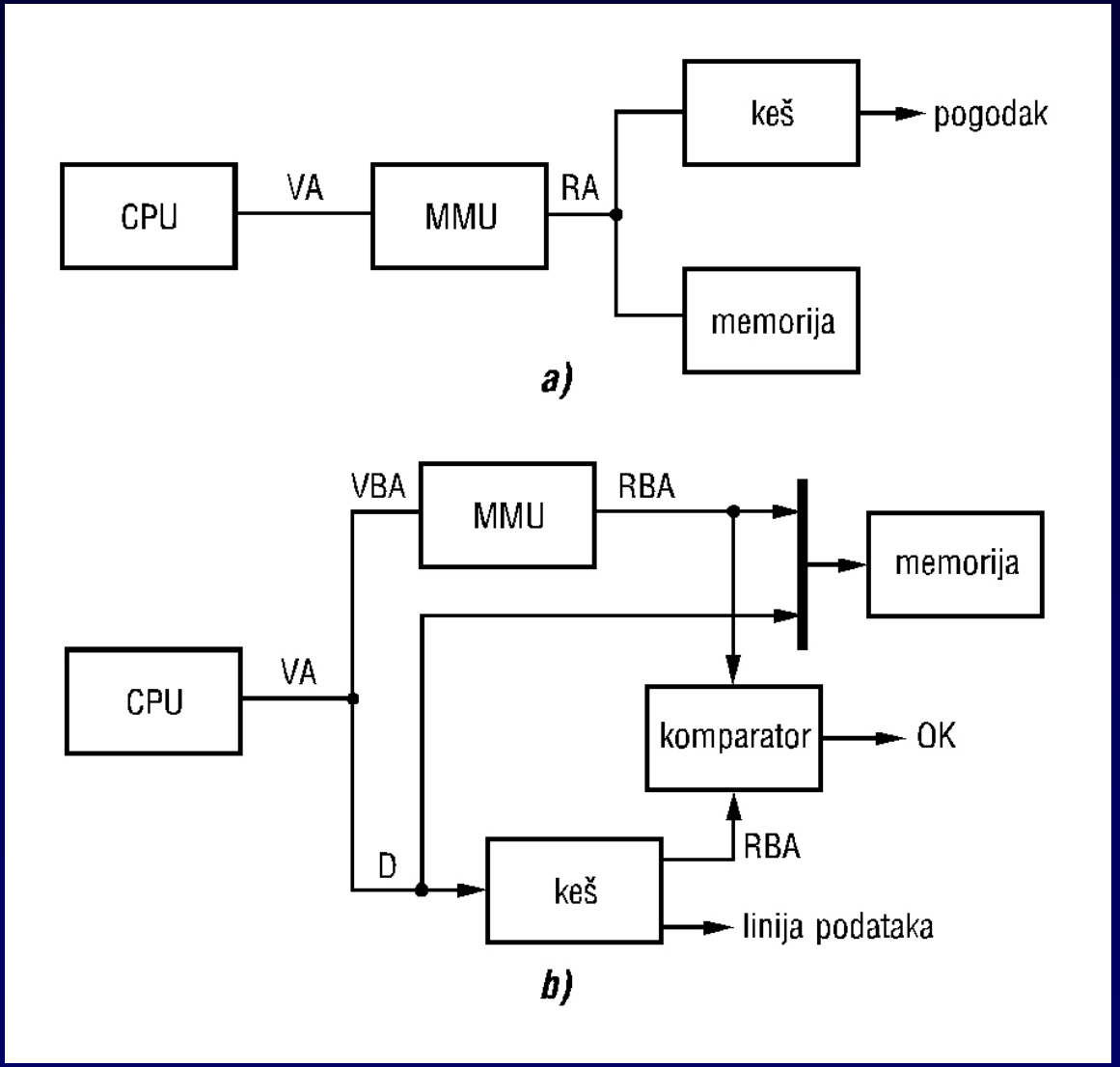
Skupno-asocijativna keš memorija



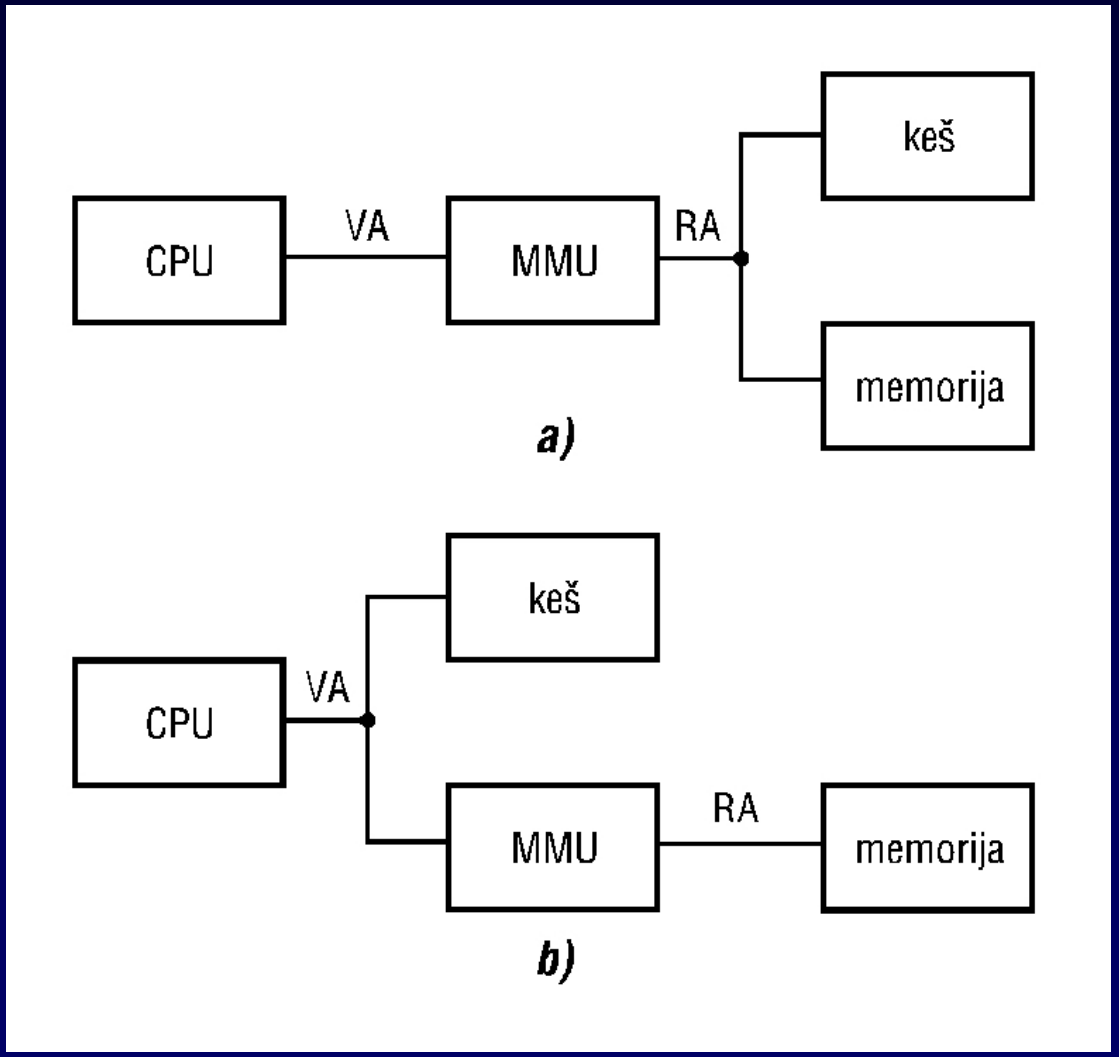
Primer dvostruko skupno-asocijativne keš memorije



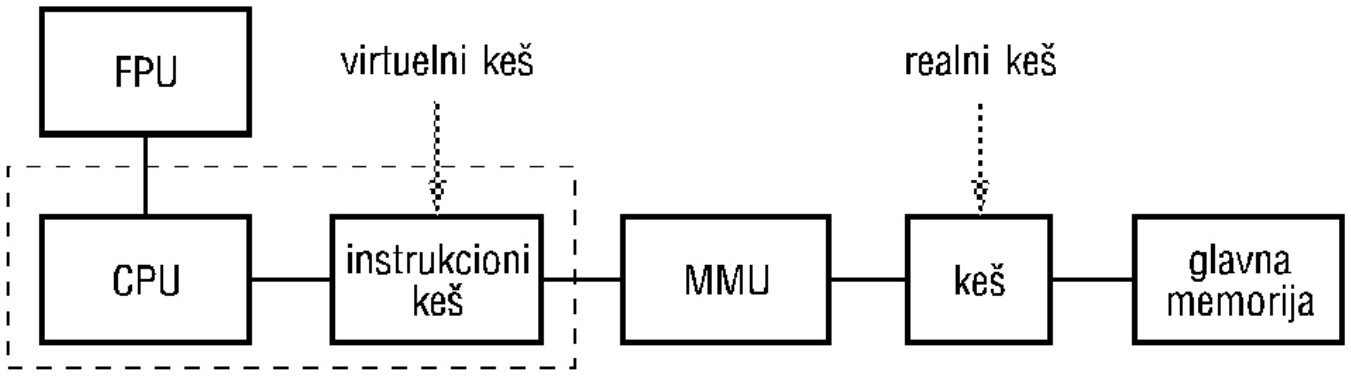
Položaj keša u sistemu



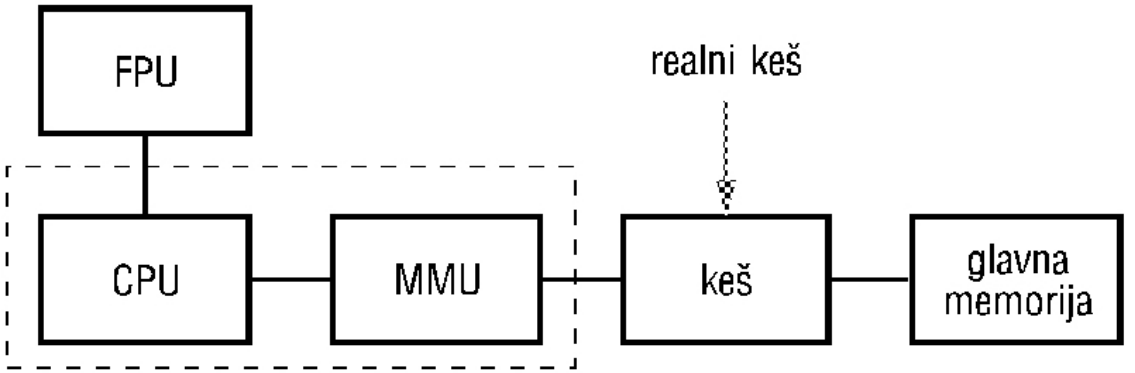
Položaj keša u sistemu – prod.



Virtuelni i realni keš

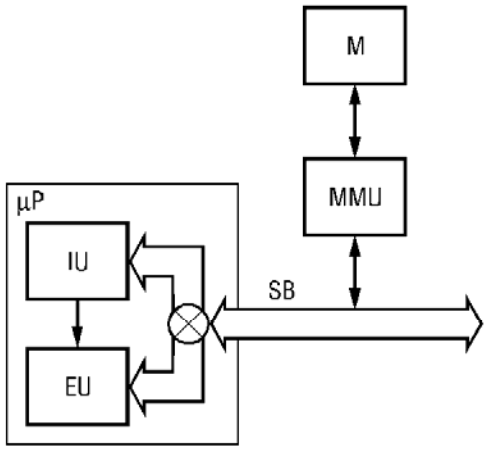


a)

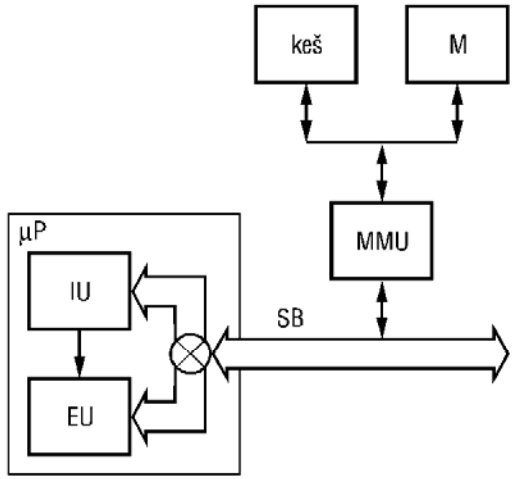


b)

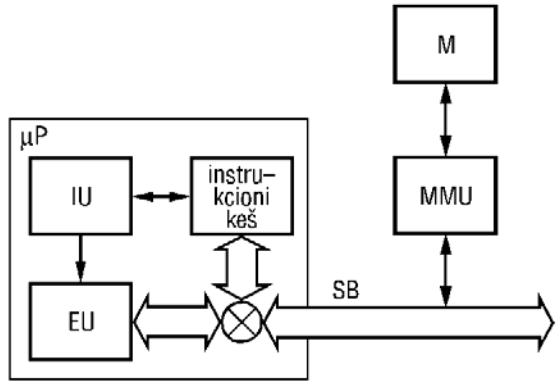
Interni i eksterni keš



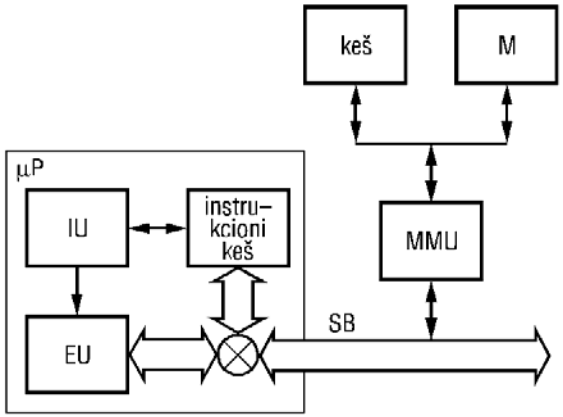
a) *tipični savremeni mikroračunarski sistem*



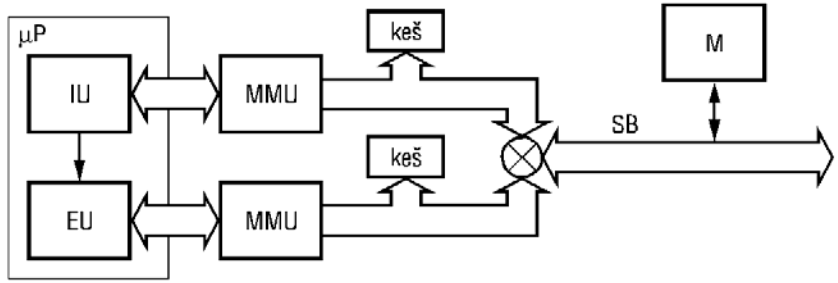
b) *mikroprocesorski sistem sa brzom keš memorijom ugrađenom van mikroprocesora*



c) *mikroprocesorski sistem kod koga je instrukcioni keš ugrađen u keš memoriju*

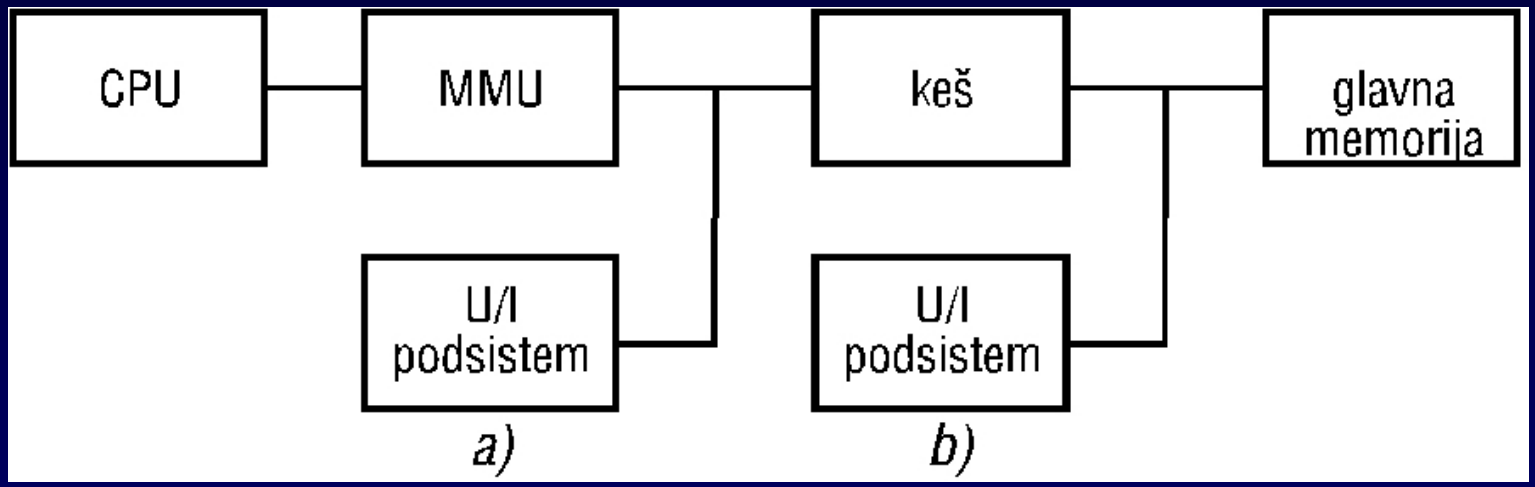


d) *mikroprocesorski sistem kod koga je keš malog kapaciteta delimično ugrađen u mikroprocesor, a keš za podatke/instrukcije van mikroprocesora*

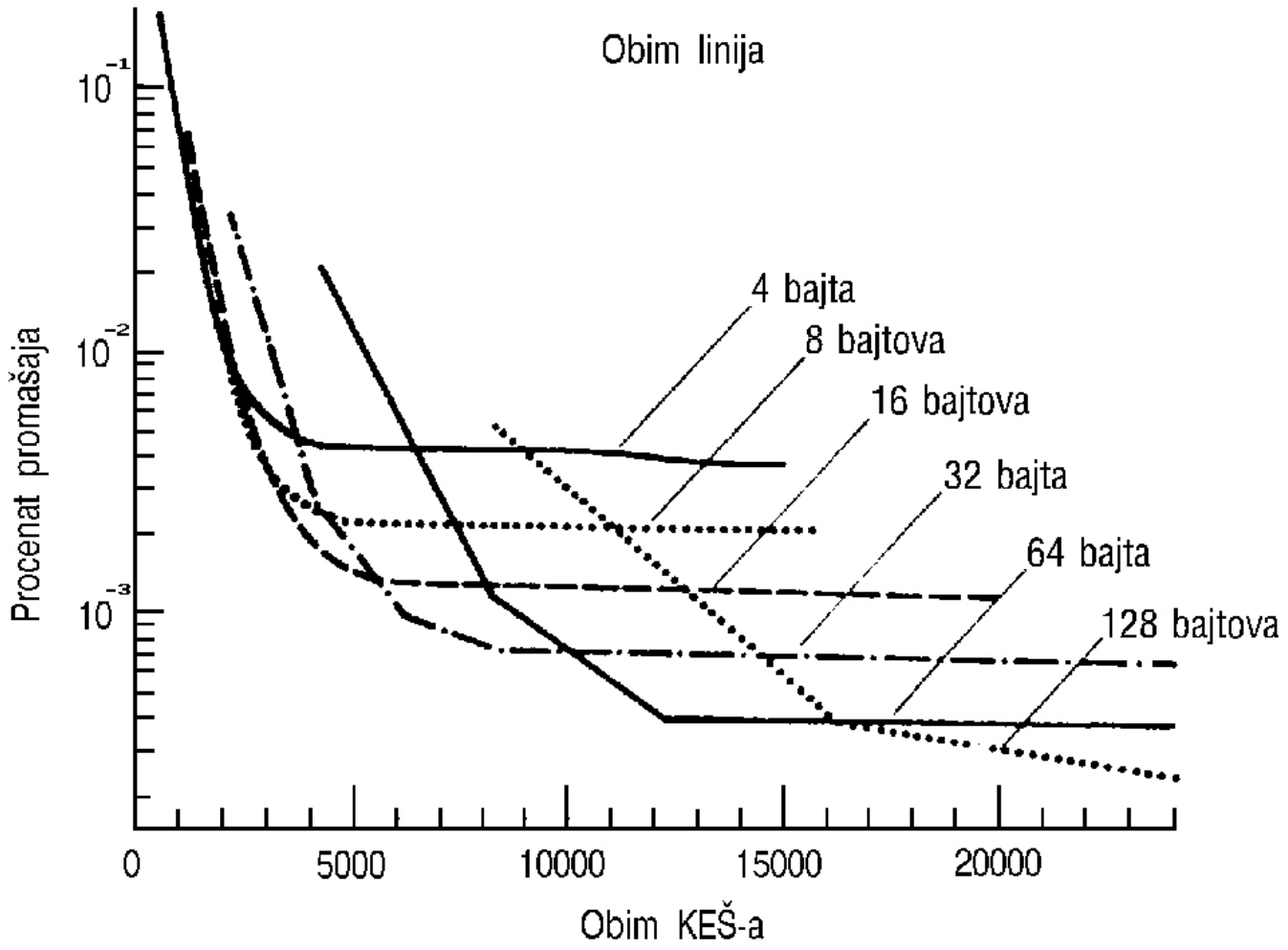


e) *keš šema kod sistema sa dve magistrale*

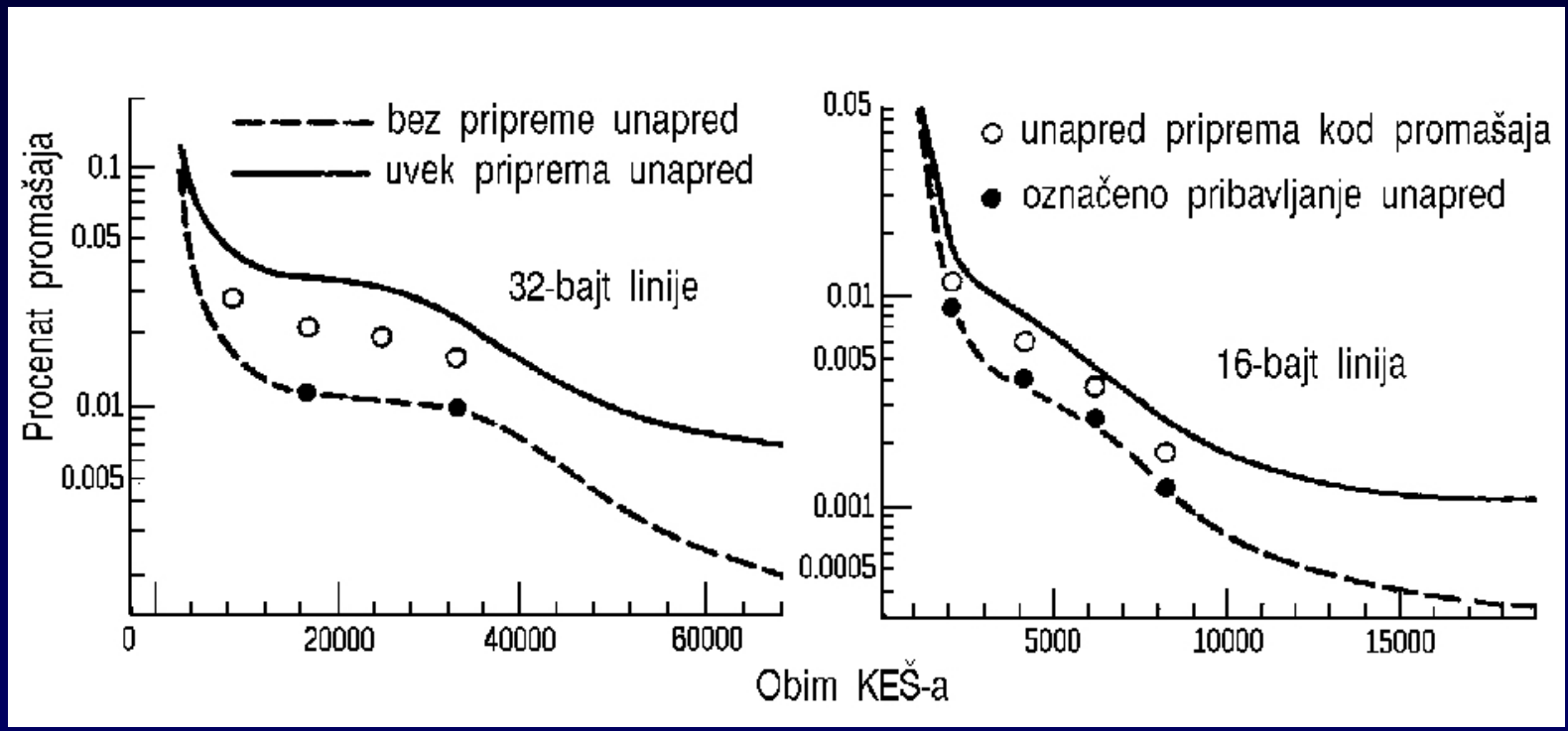
Keš u odnosu na U/I podsistem



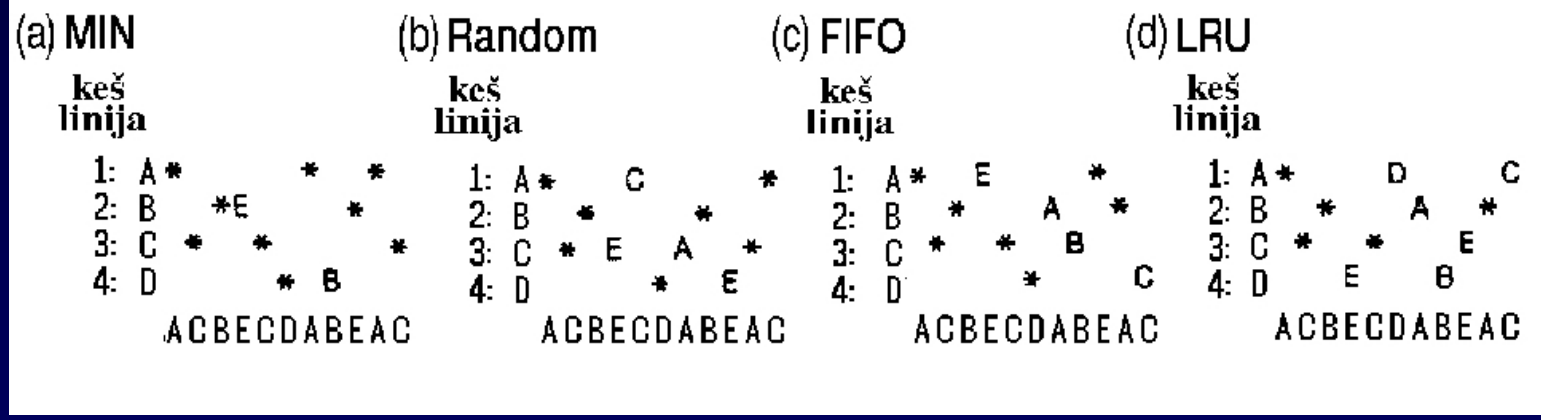
Procenat promašaja u funkciji od obima keša



Procenat promašaja u funkciji od obima keša



Politike zamene kod keša



MAGISTRALE

Šta je magistrala ?

- Magistrala je komunikacioni put koji se koristi za povezivanje dva ili više uredjaja
- Magistrala je deljivi prenosni medijum
- Na magistralu je povezan veći broj uredjaja, a signal koji se predaje od strane jednog uredjaja dosrupan je za prijem svim ostalim uredjajima koji su povezani na tu magistralu.
- Ako dva uredjaja vrše istovremeno prenos dolazi do kolizije
- U datom trenutku samo jedan uredjaj može uspešno da vrši predaju
- Magistrala se može posmatrati kao skup linija (veza)

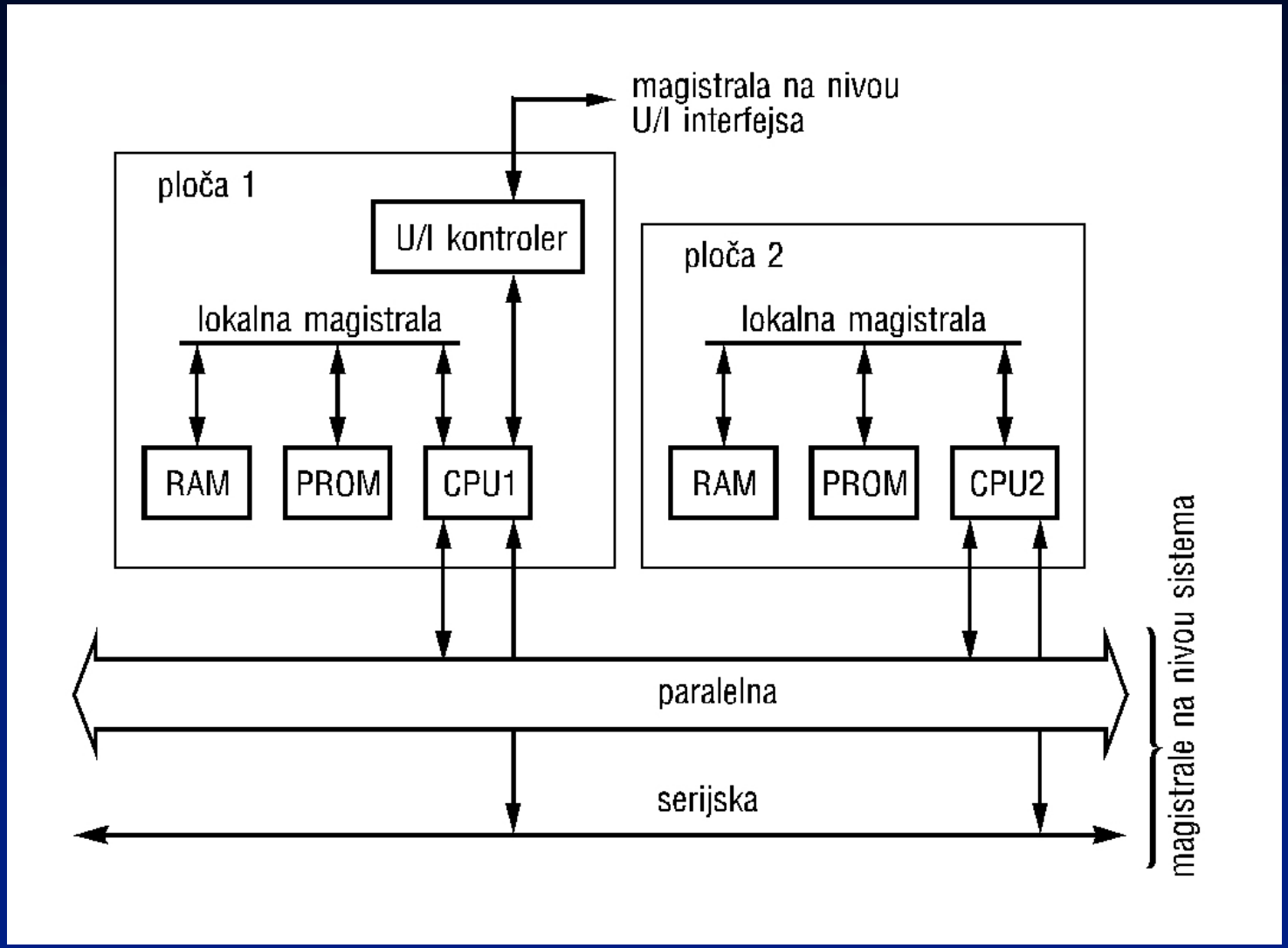
Sistem magistrala

- Predstavlja skup većeg broja magistrala koji se koriste za povezivanje različitih sistemskih elemenata u okviru računarskog sistema.

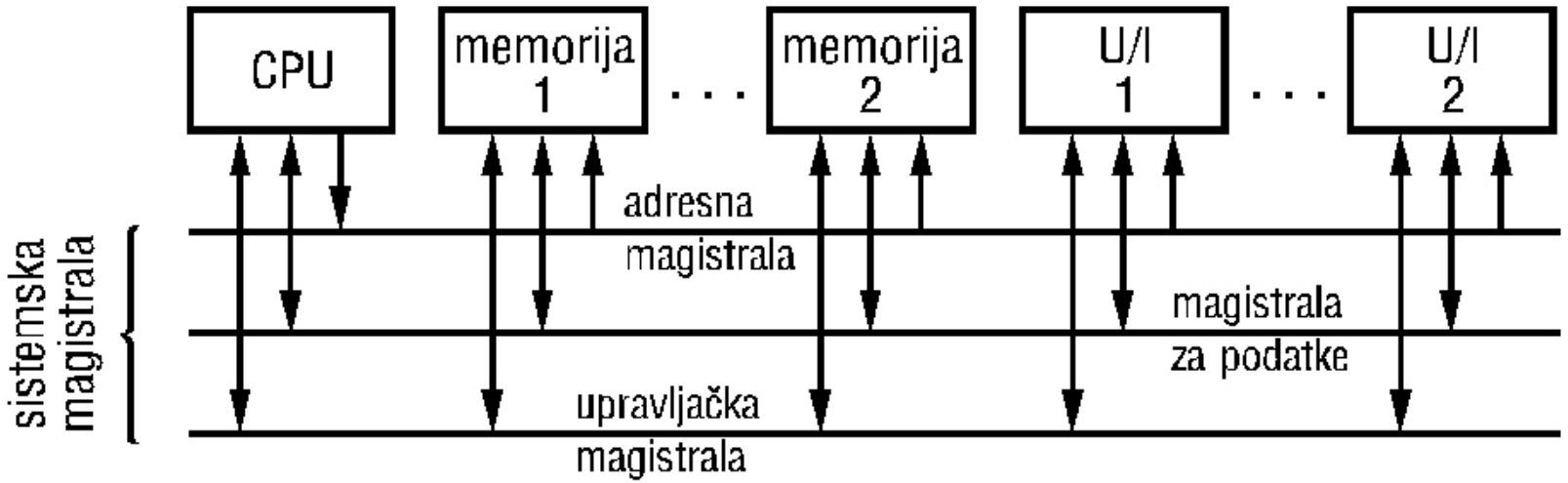
- Sistem magistrala čine magistrale koje su hijerarhijski organizovane po sledećim nivoima:

  - nivo ploča – obično se ova magistrala zove lokalna
  - zadnja ploča – *motherboard* realizova je kao štampana ploča sa većim brojem identičnoh konektora koji su paralelno povezani (VME, Multibus I i II, Nubus, Fastbus, Futurebus, itd.)
  - nivo interfejsa – ostvaruje se komunikacioni put izmedju U/I uredjaja i ostatka sistema (SCSI, GPIB, itd.)

Tipičan primer sistemske magistrale

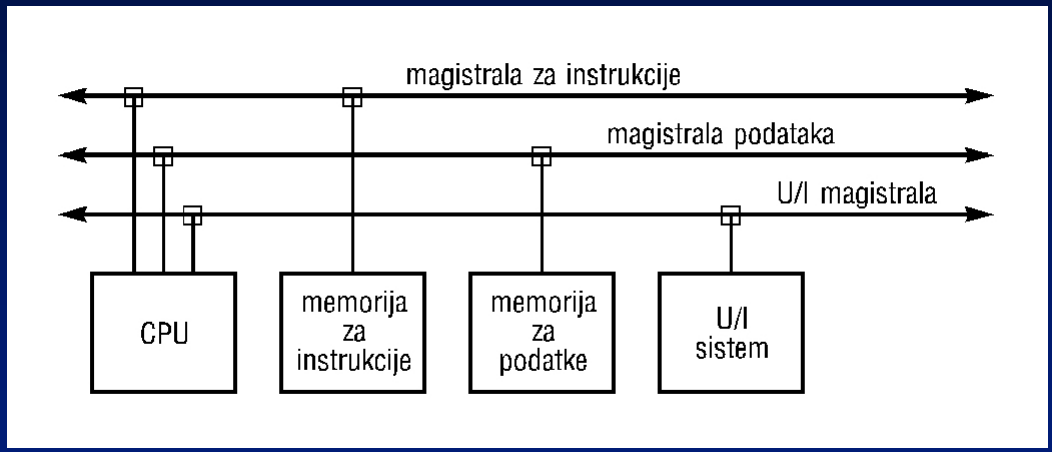


Struktura sistemske magistrale



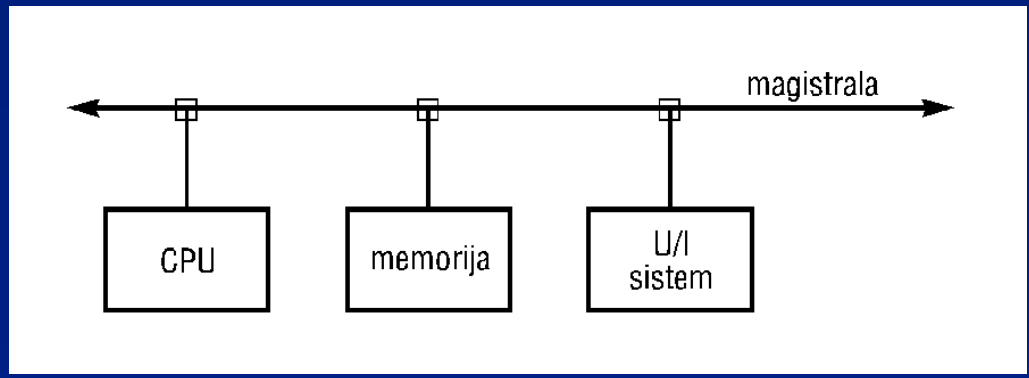
Podela magistrala u zavisnosti od namene

- **Magistrale se mogu podeliti na:**
  - **nameske – koriste se za obavljanje specifičnih funkcija**
  - **nenameske – zovu se deljive magistrale i koriste se za obavljanje većeg broja funkcija**



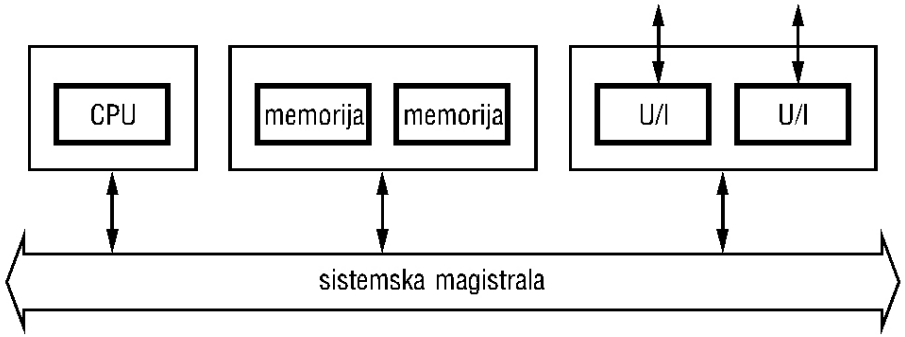
nameske

nenameske



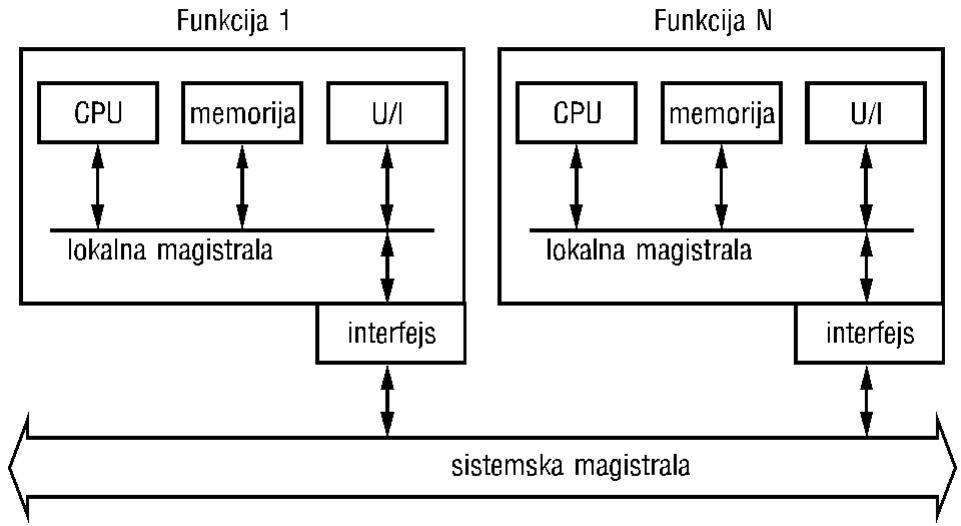
Klasifikacija deljivih magistrala

- Deljive magistrale možemo svrtati u zavisnosti od toga kako je izvršena :
  - podela finkcija po pločama
  - podela tipova resursa



sistem zasnovan na povezivanju svih tipova resursa

sistem zasnovan na povezivanju većeg broja ploča koje obavljaju različite funkcije



Termini vezani za magistrale

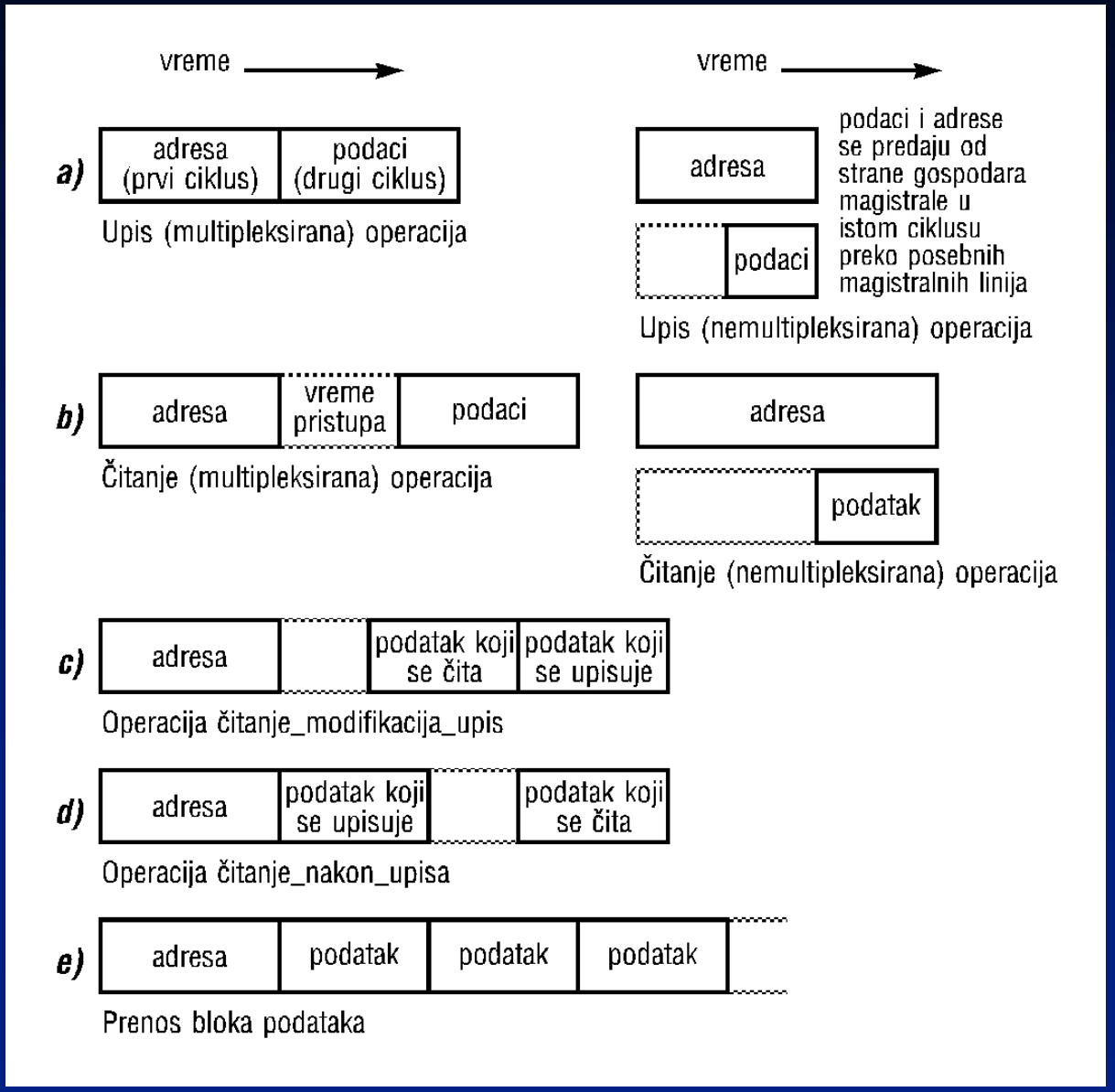
- Ploče koje se vezuju na magistralu mogu biti tipa:
  - gospodar – može inicirati ciklus na magistrali
  - sluga – odaziva se gospodaru
- Neki od uredjaja mogu biti i gospodari i sluge, ali ne istovremeno
- Kada prenos podataka počne ploča koja predaje podatak se zove izvorište, a ploča koja prima podatak odredište

Kompletna sekvenca predaje

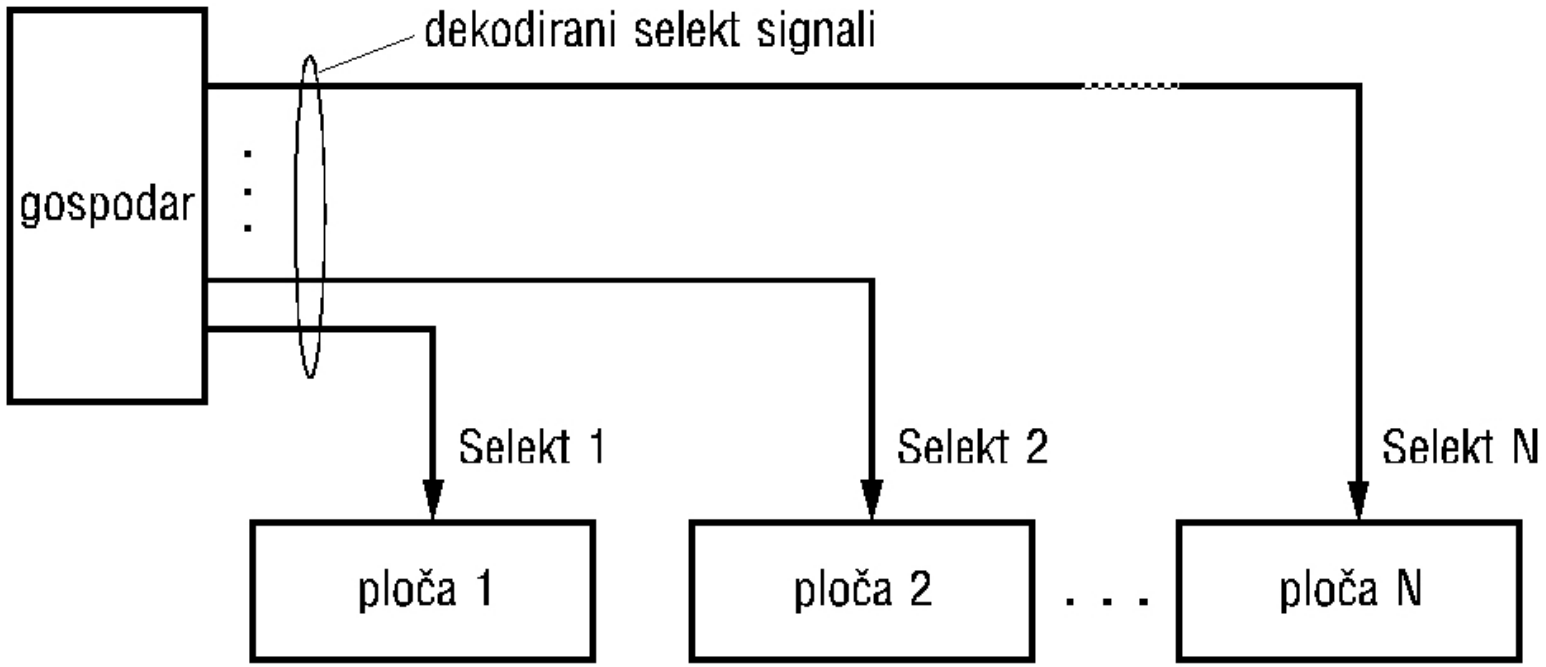
- Sekvenca dogadjaja od trenutka kada se magistrala zahteva do trenutka završetka prenosa podataka zove se transakcija na magistrali

- Da bi se prenos po magistrali obavio, neophodno je da se obave sledeće operacije:
  - zahtev
  - arbitraža
  - adresiranje
  - prenos podataka
  - detekcija i signalizacija greške

Tipovi prenosa podataka preko magistrale

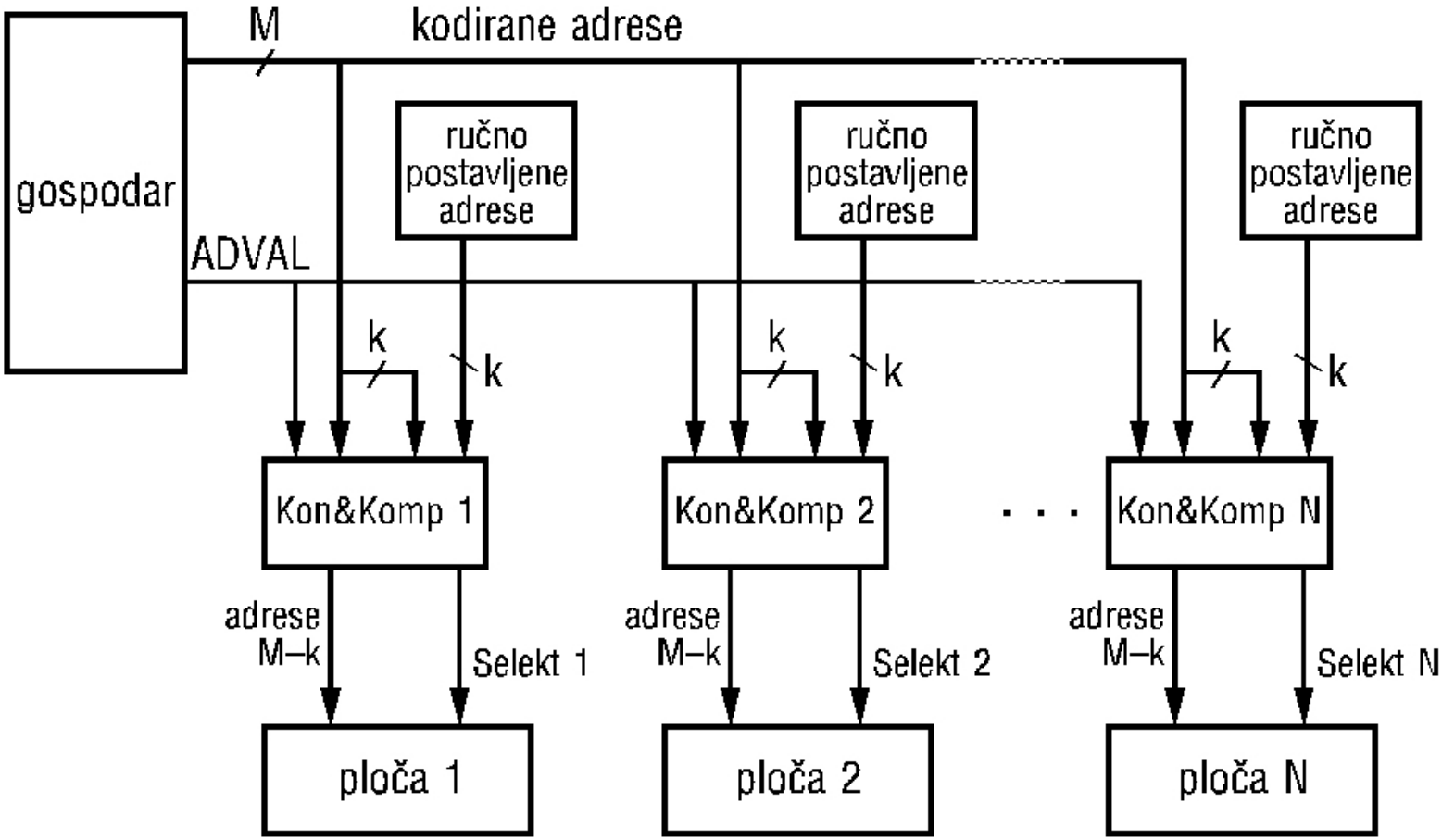


Načini organizacije adresiranja



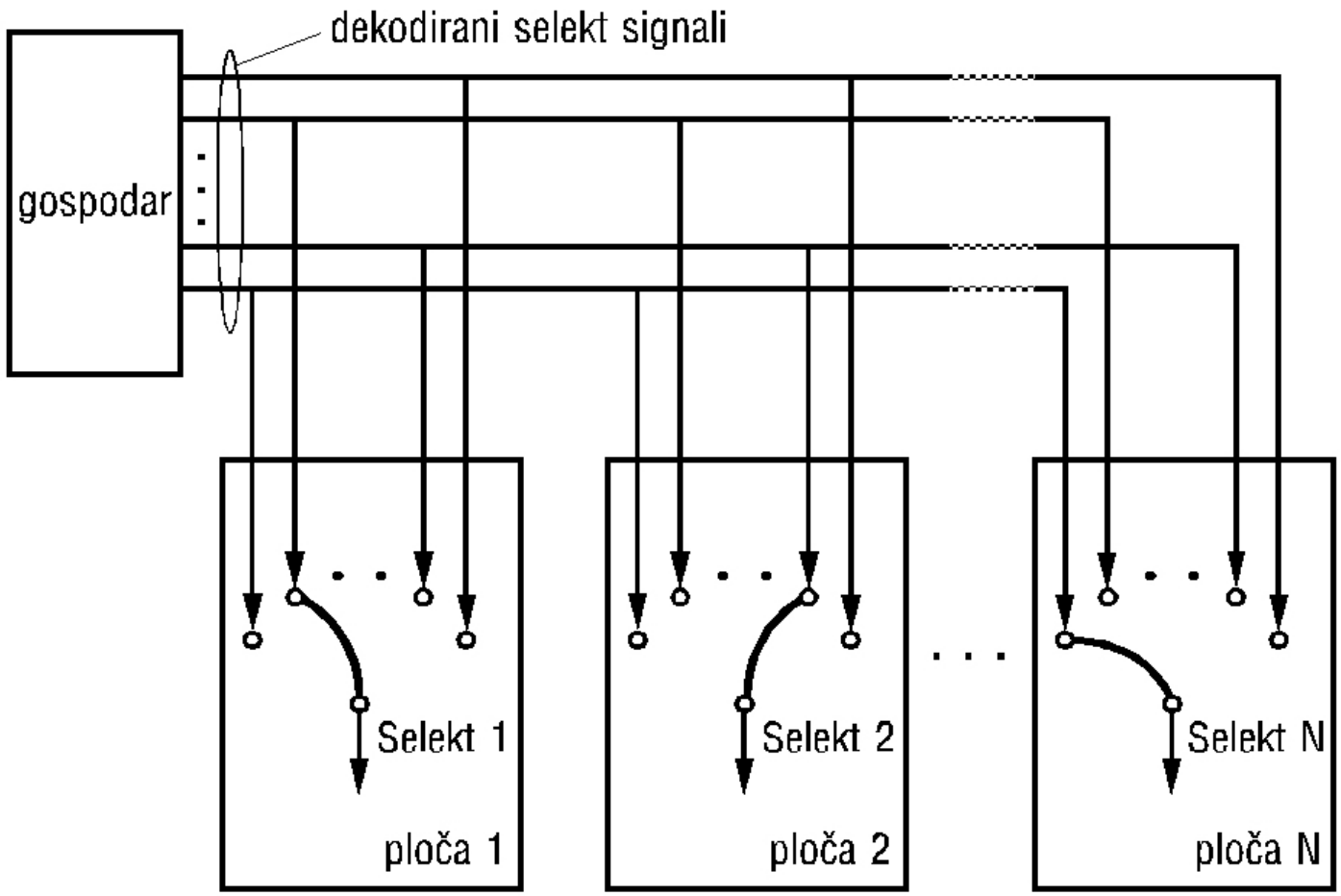
a) *geografsko dekodirano*

Načini organizacije adresiranja - prod



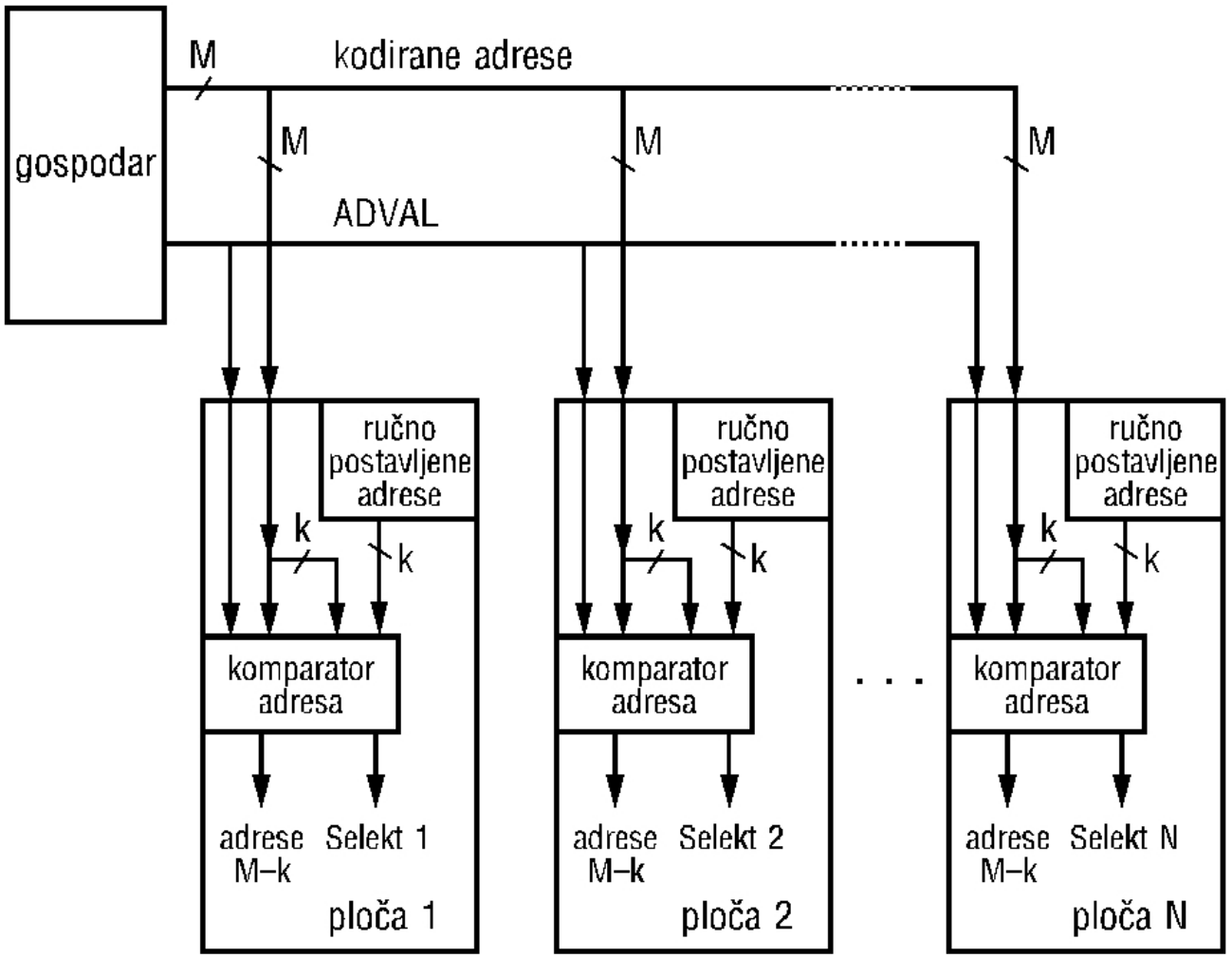
b) *geografsko kodirano*

Načini organizacije adresiranja - prod



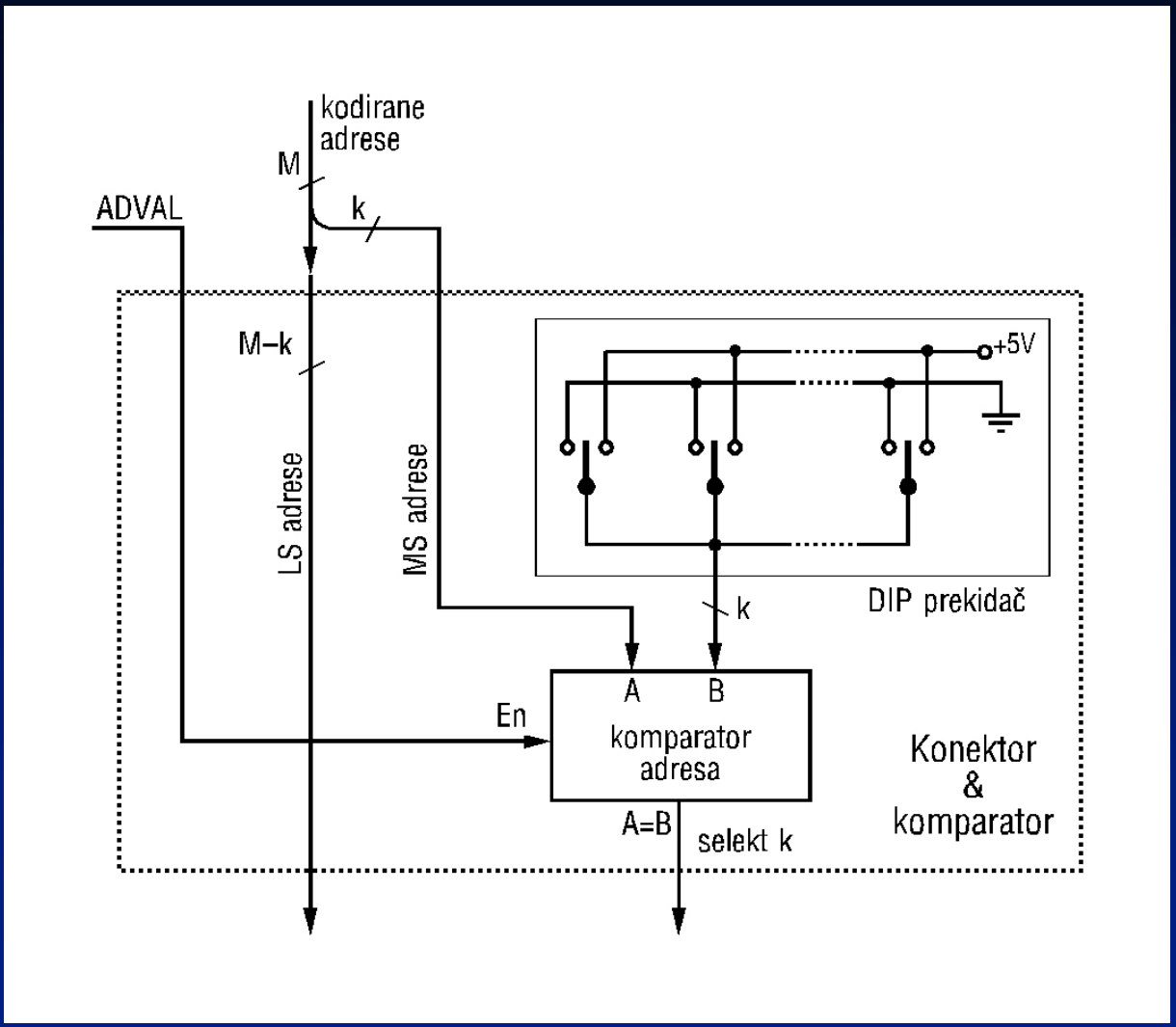
c) dekodirano logički

Načini organizacije adresiranja - prod



d) kodirano logički

Načini organizacije adresiranja - prod

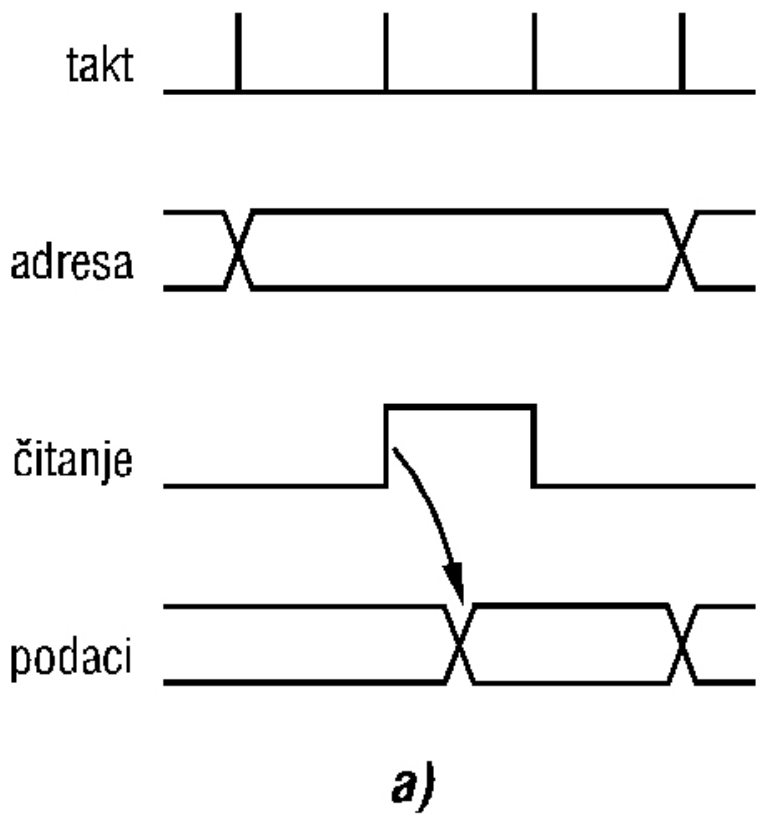


Logika bloka konektora i komparatora

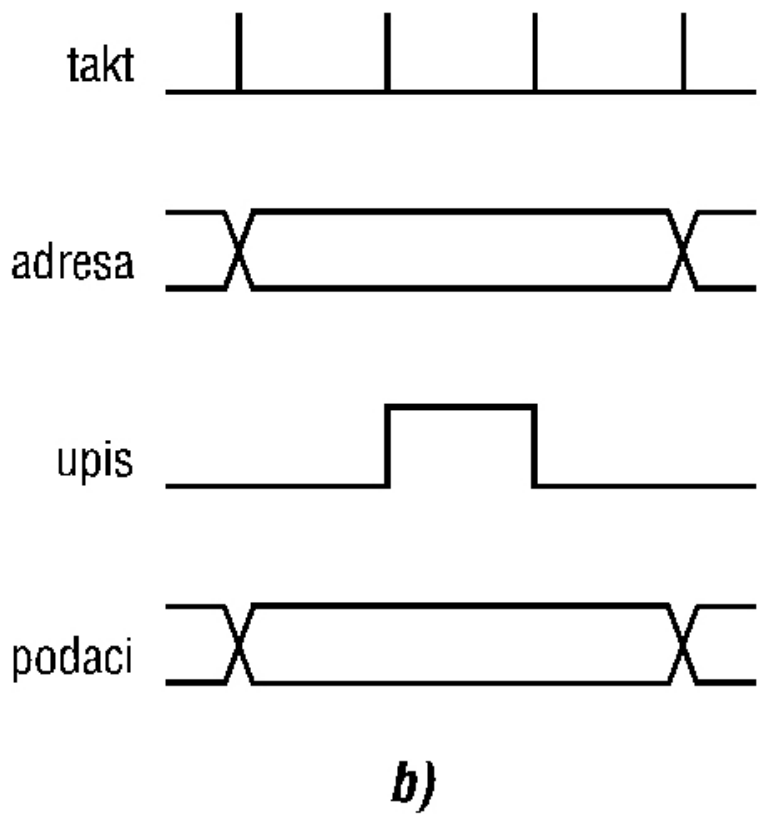
Podela magistrala – vremensko vodjenje dogadjaja

- Sa tačke gledišta vremenkog vodjenja dogadjaja magistrale delimo na:
  - sinhrone – svi dogadjaji se dešavaju u fiksnim vremenskim intervalima
  - asinhrone – signali u sekvenci se generišu u proizvoljnim vremenskim intervalimja
  - polusinhrone – signali u sekvenci se generišu u fiksnim trenucima u odnosu na taktni signal

Mašinski ciklus Čitanje i Upis kod sinhrone magistrale

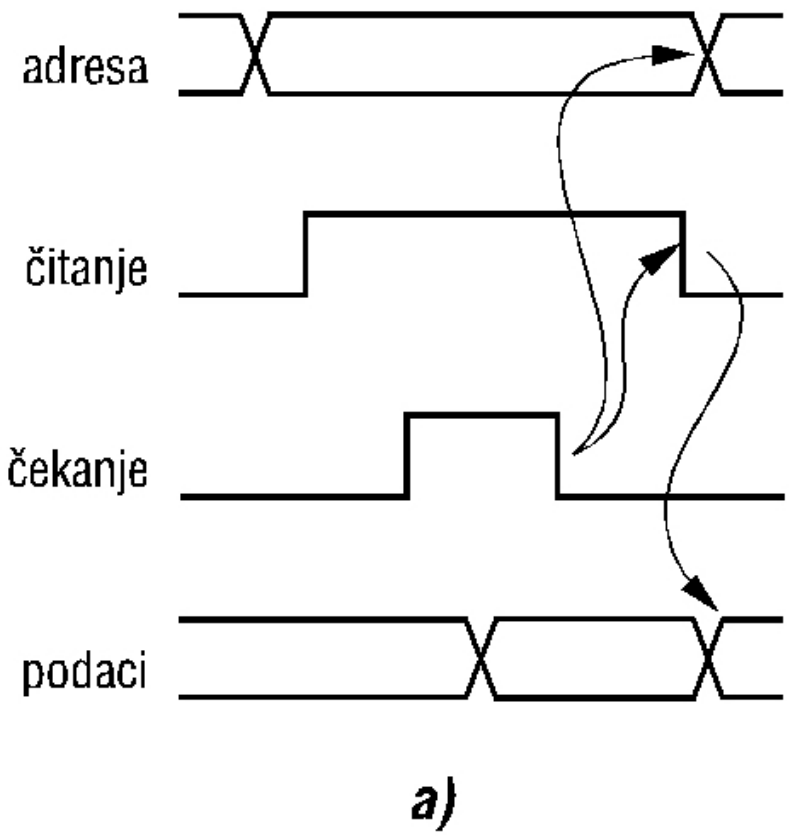


a)

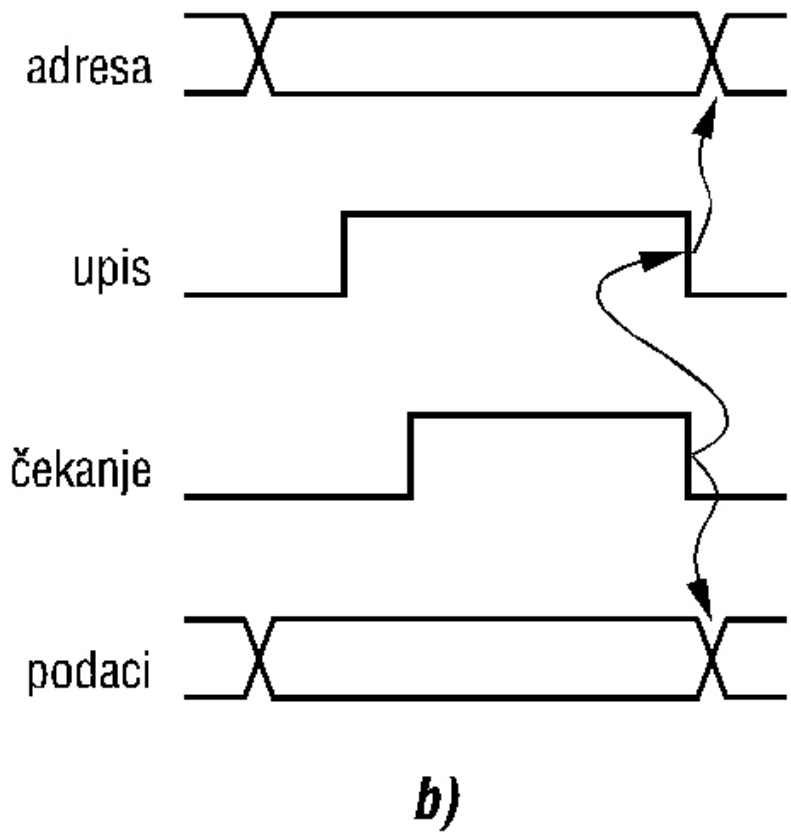


b)

Mašinski ciklus Čitanje i Upis kod asinhrone magistrale

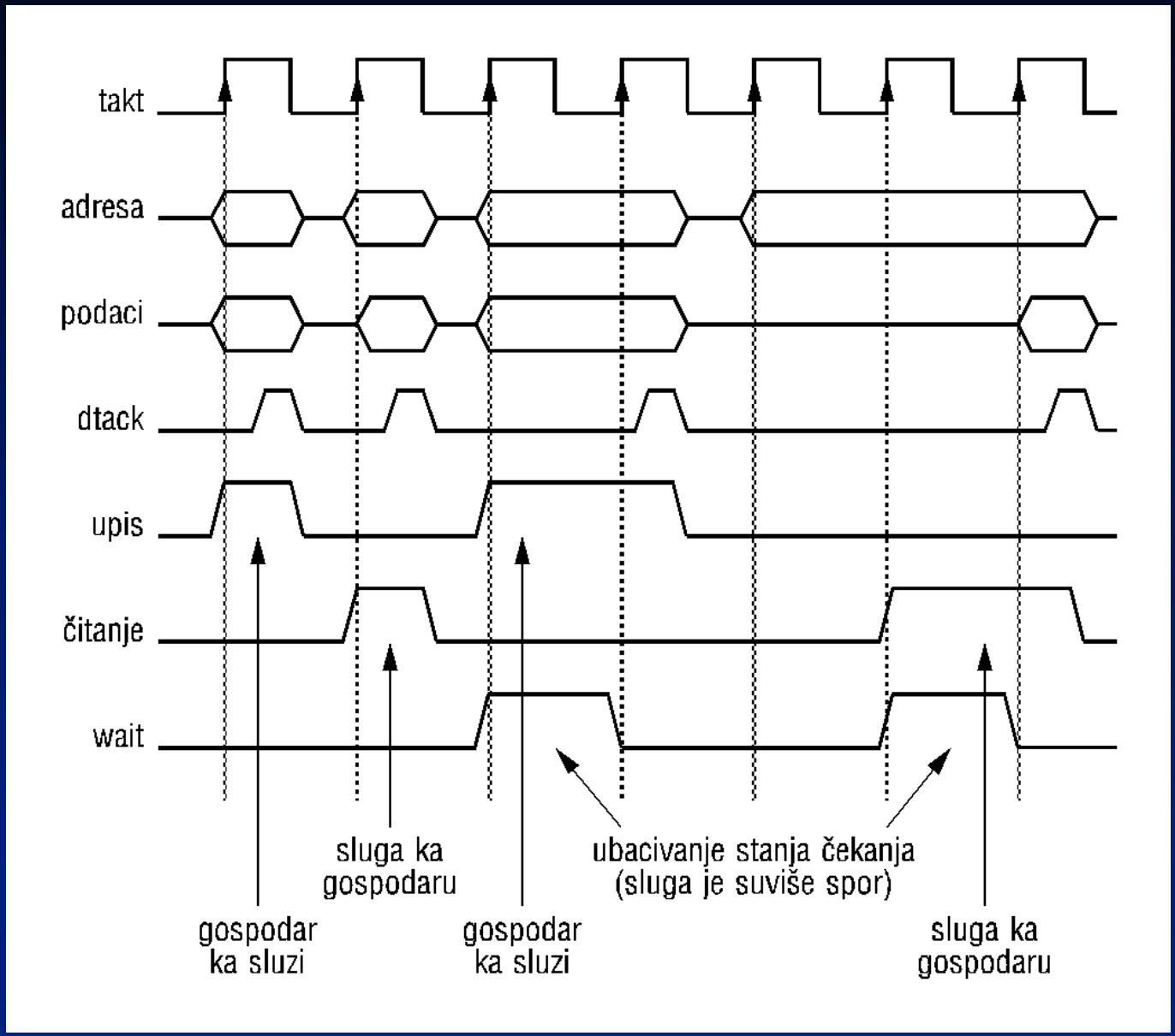


a)



b)

Princip rada polusinhrone magistrale



Arbitraža na magistrali

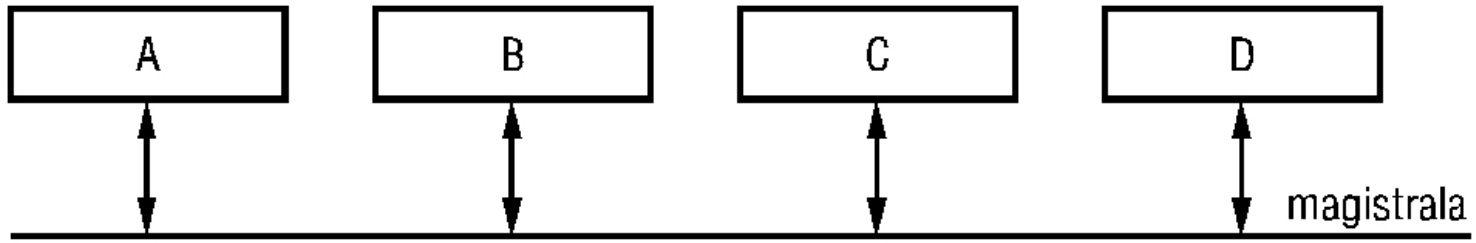
- Arbitražni mehanizam garantuje da se dodela magistrale ostvaruje bez konflikta

- U datom trenutku će samo jedan gospodar upravljati magistralom, a ostali se takmiče da bi dobili pravo upravljanja nad njom

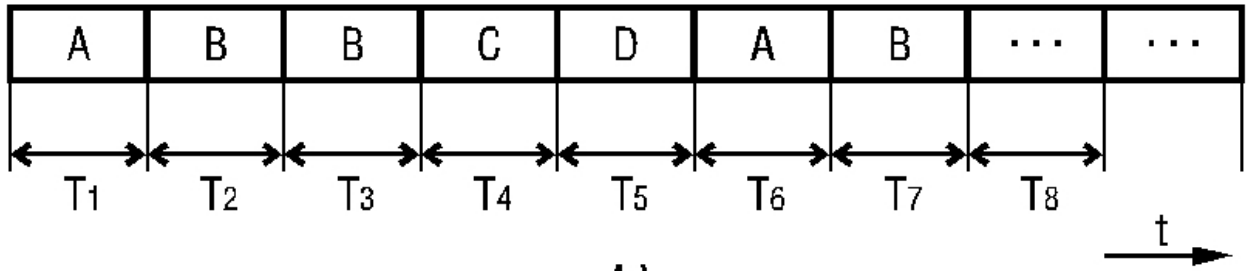
- Arbnitraža se može izvesti kao:

  - statička
  - dinamička

Statička dodela magistrale



a)

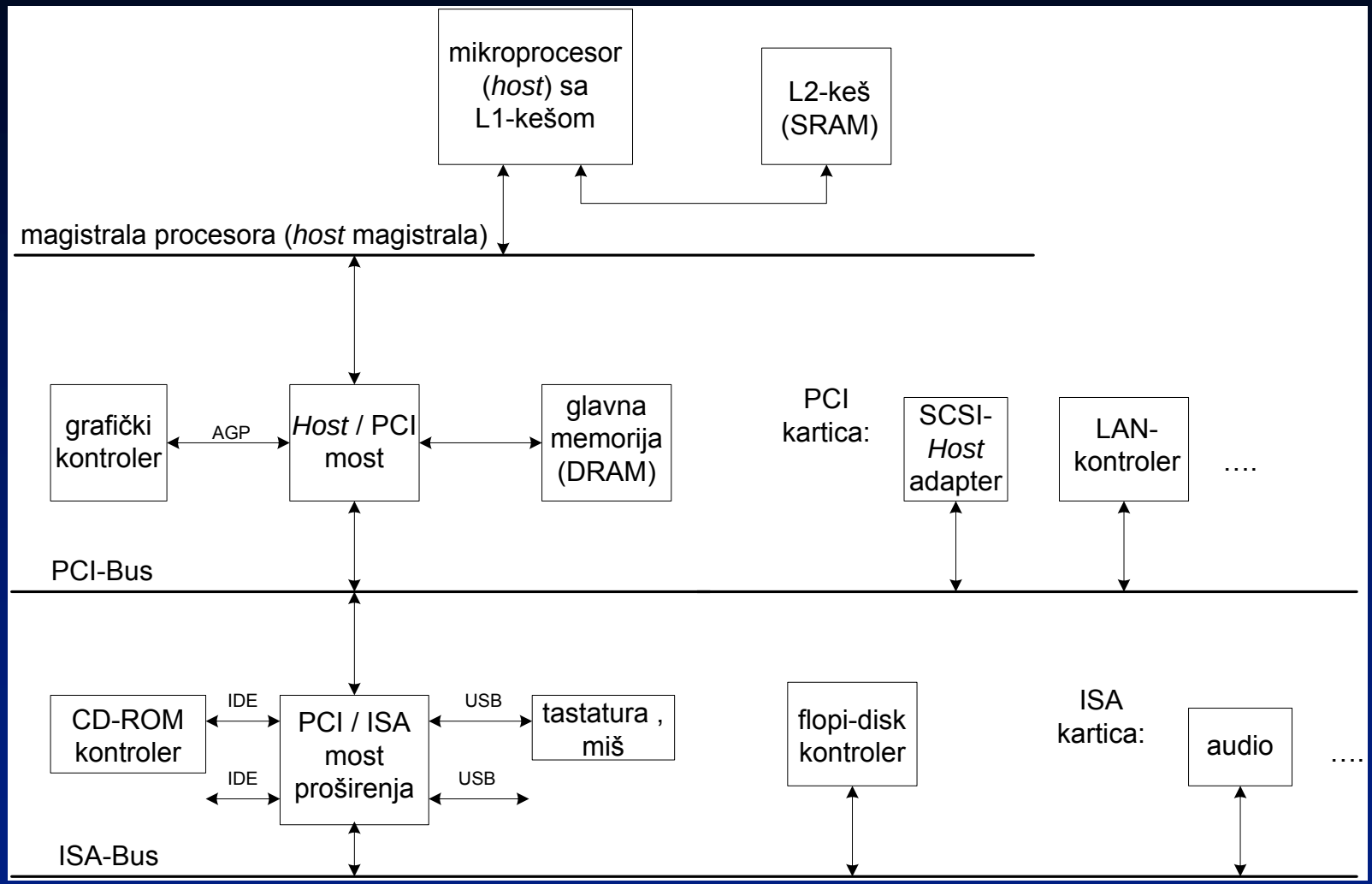


b)

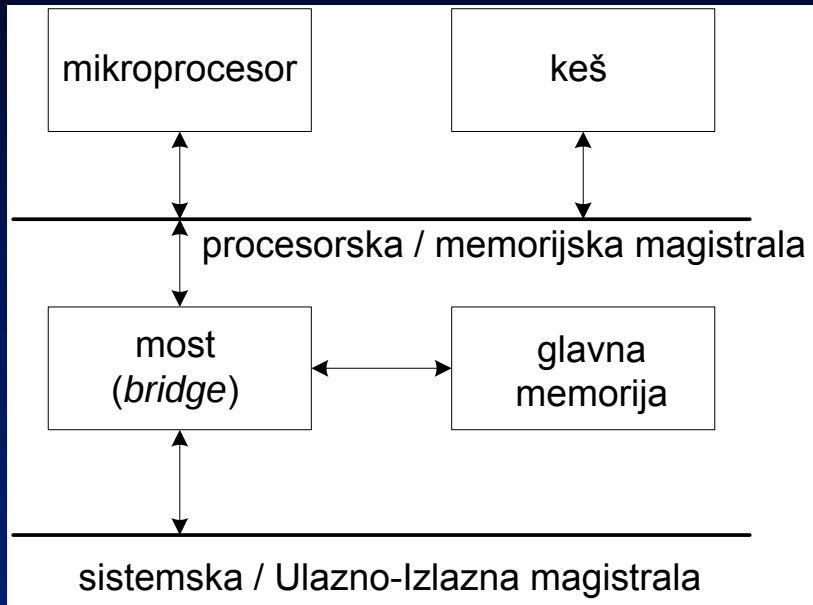
Prednost – jednostavnost izvodjenja

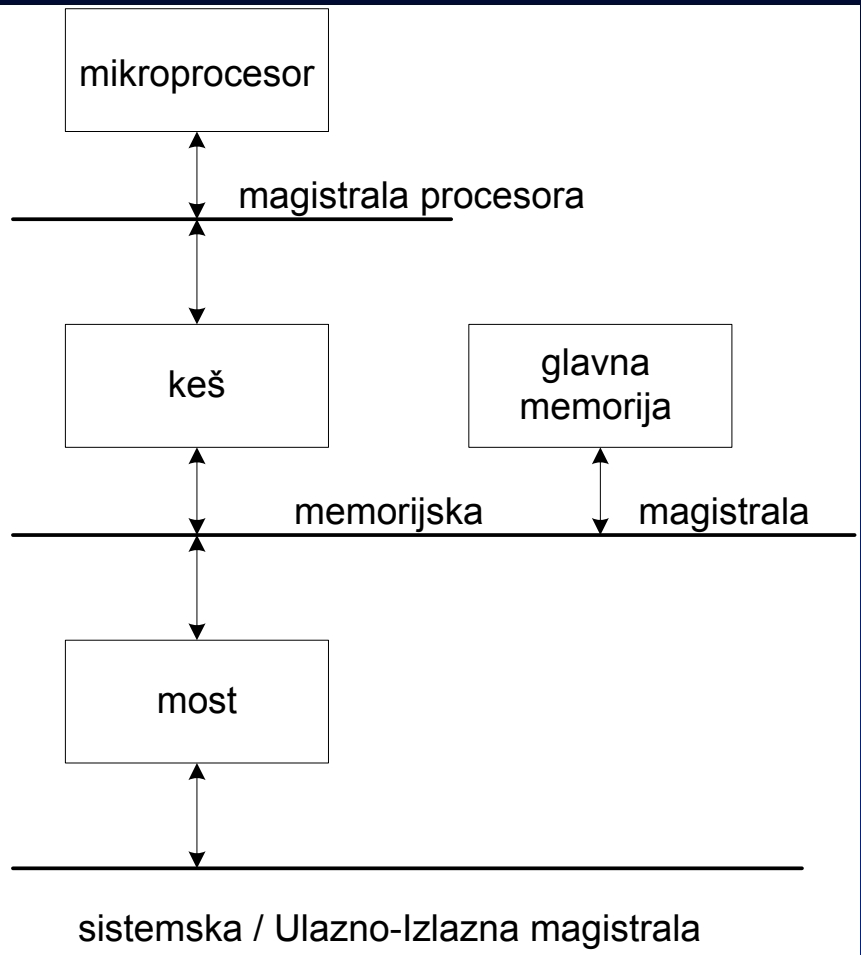
Nedostatak – ne uzima u obzir realne potrebe za prenosom

Tipična rešenja kod magistrala

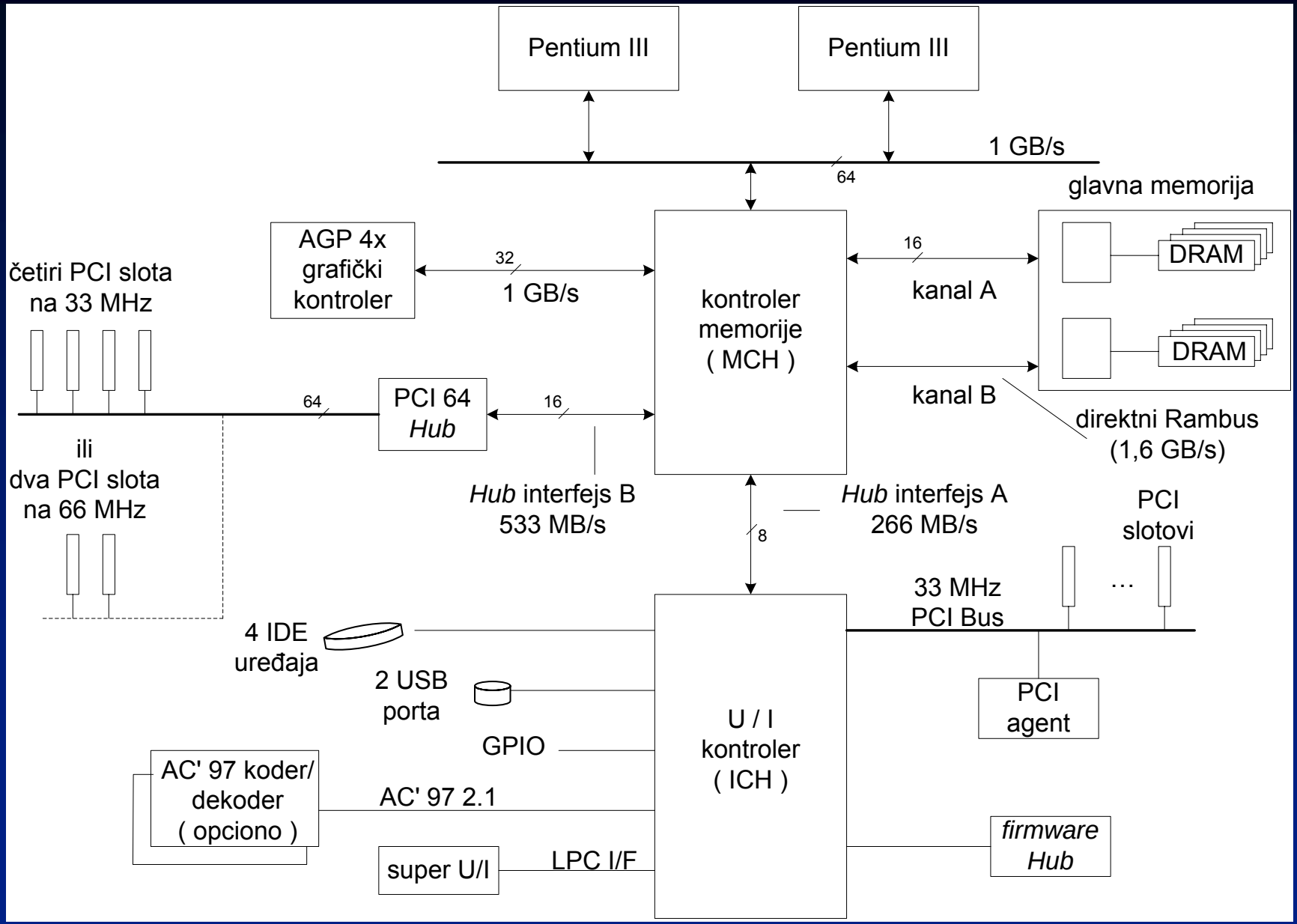


Tipična rešenja kod magistrala sa aspekta povezivanja keša

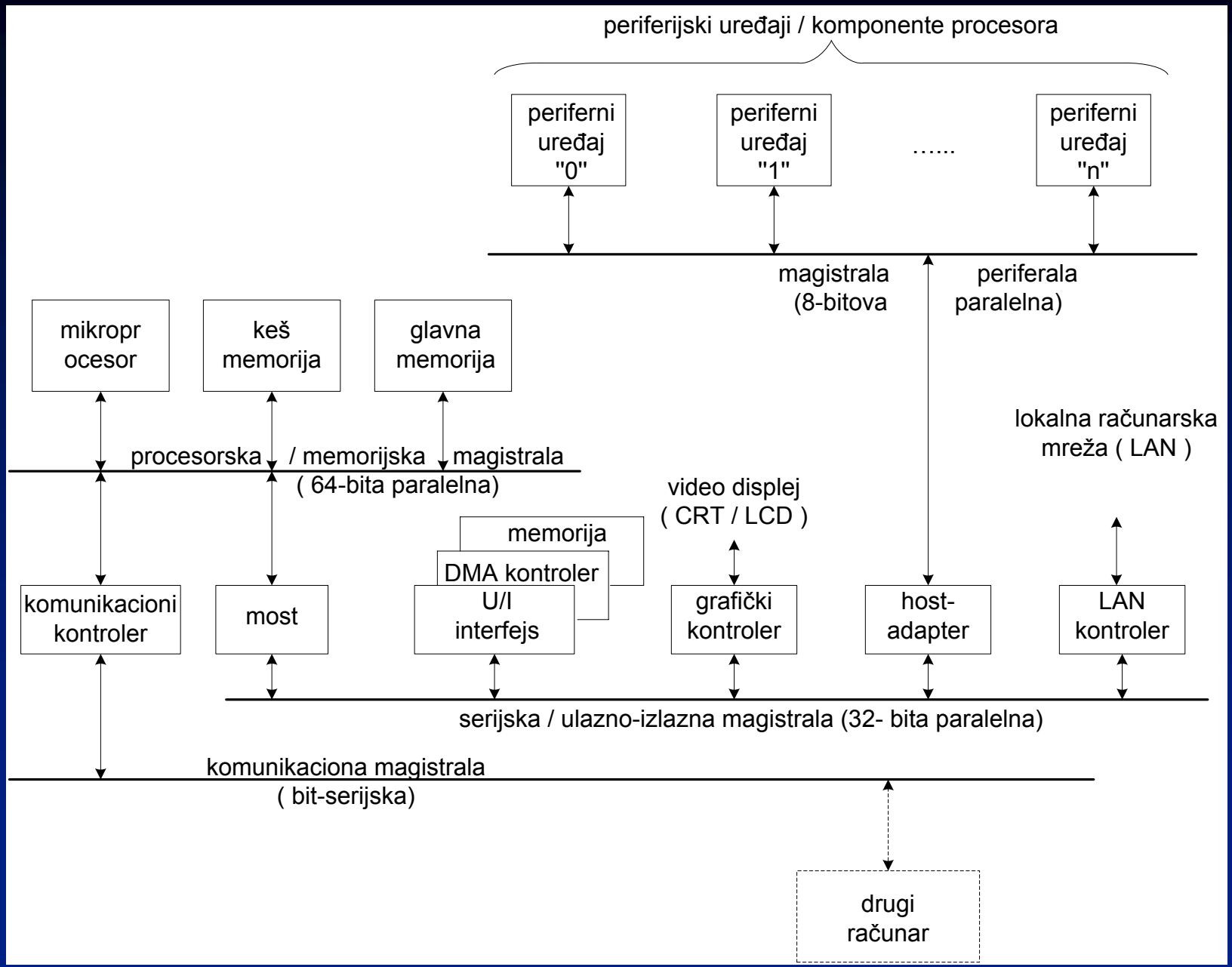




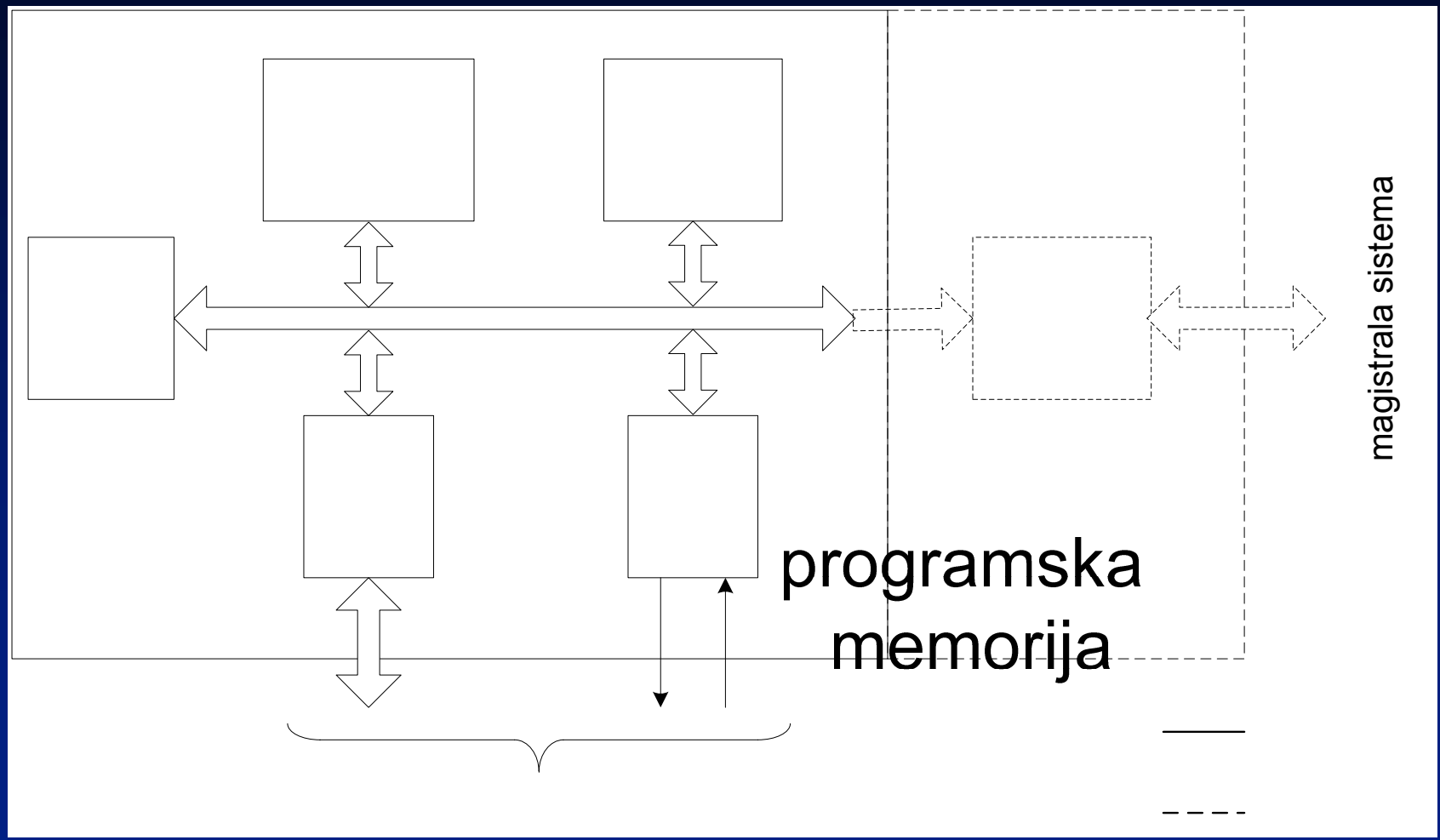
Magistrale kod PC mašina Pentium



Rešenje sa većim brojem magistrala



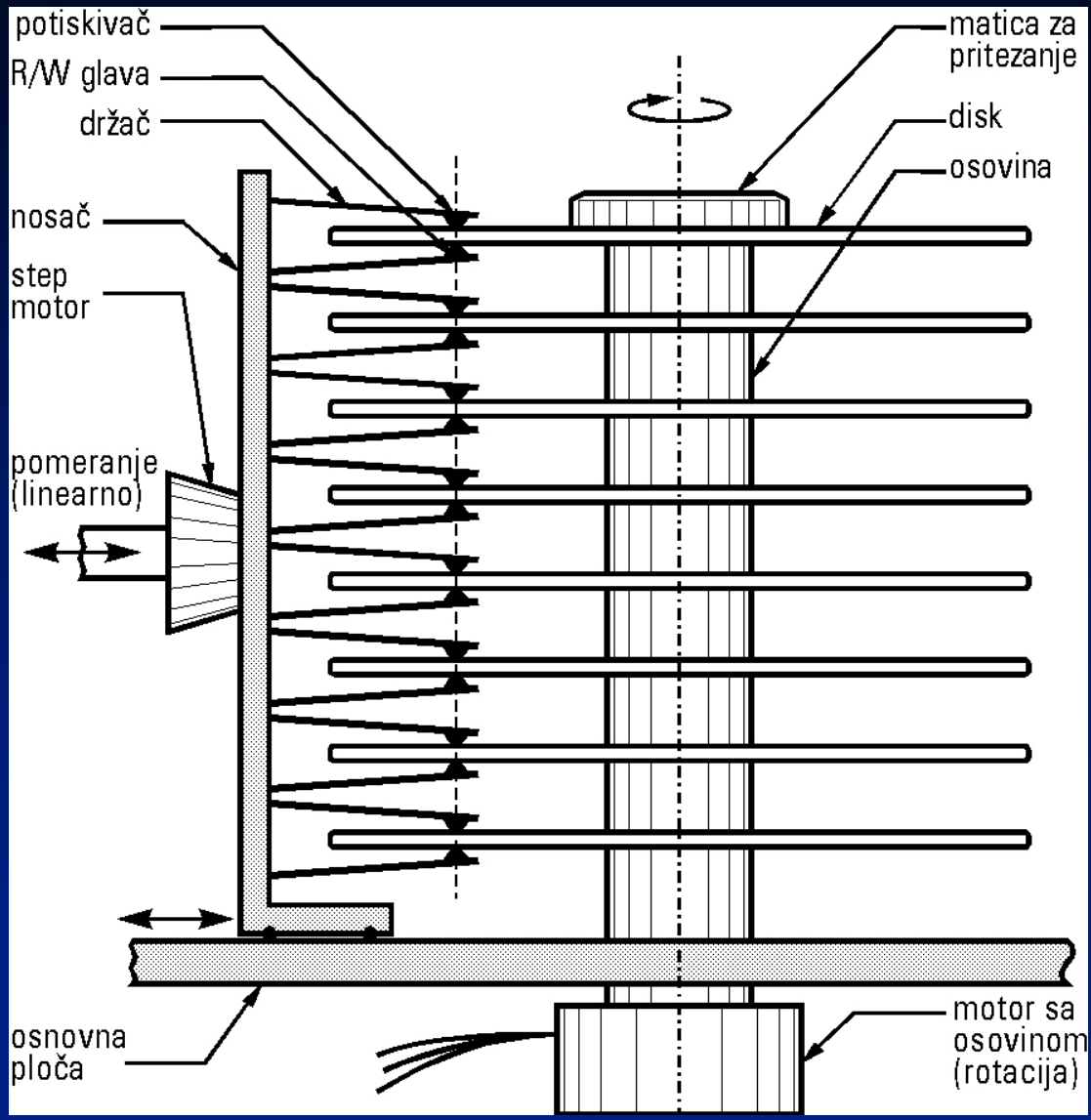
Povezivanje interfejs logike



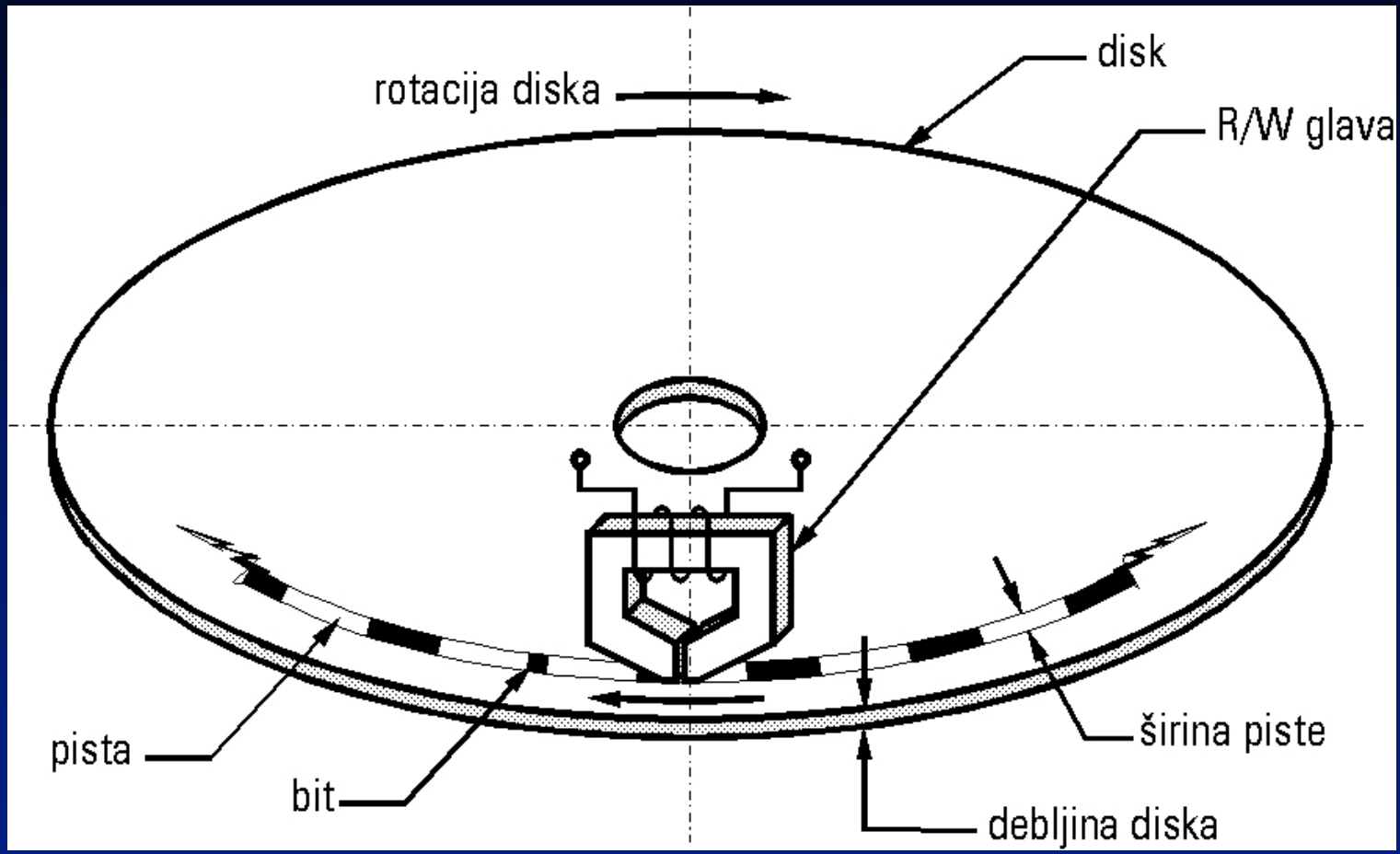
mikropr

SEKUNDARNA MEMORIJA

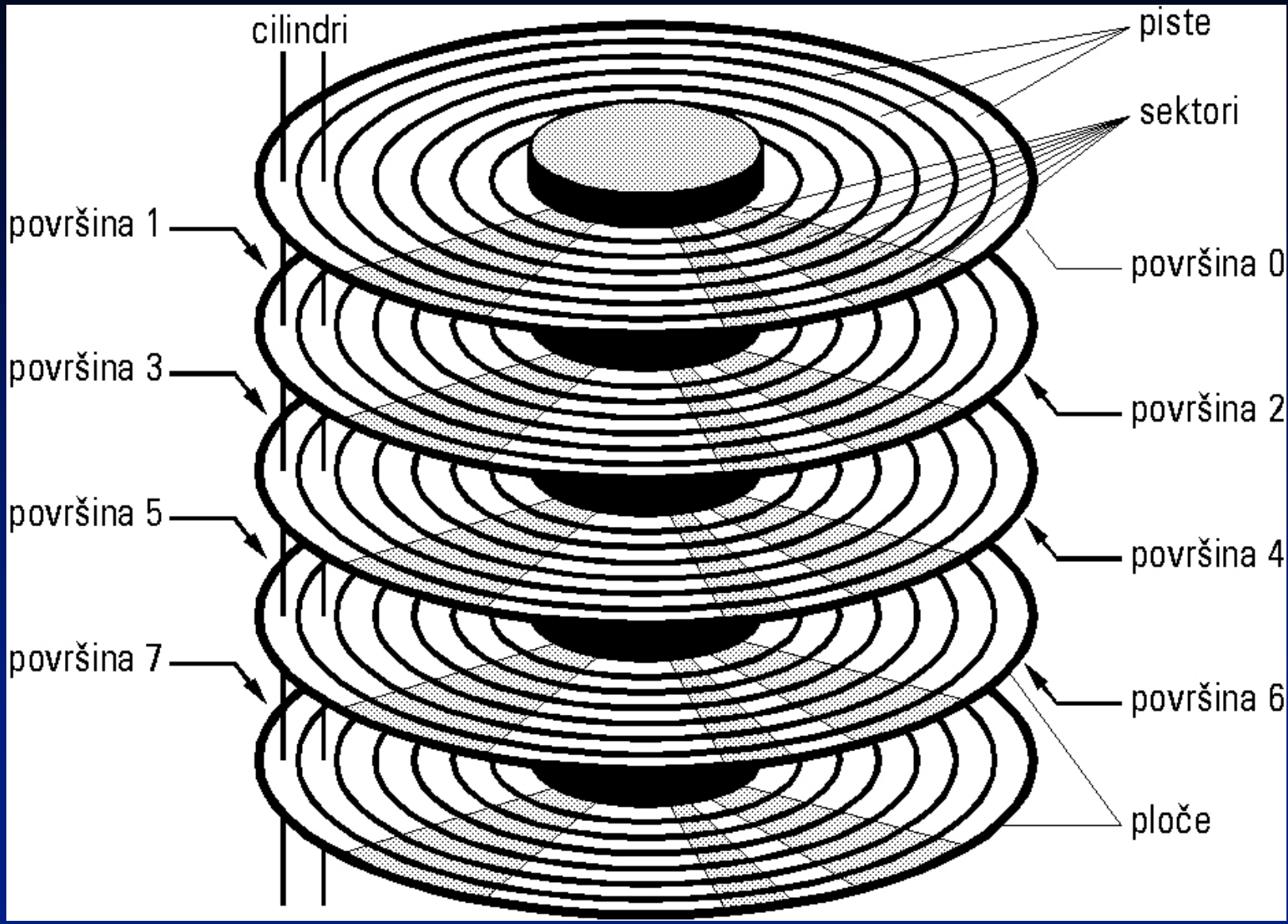
Kruti disk sa većim brojem ploča



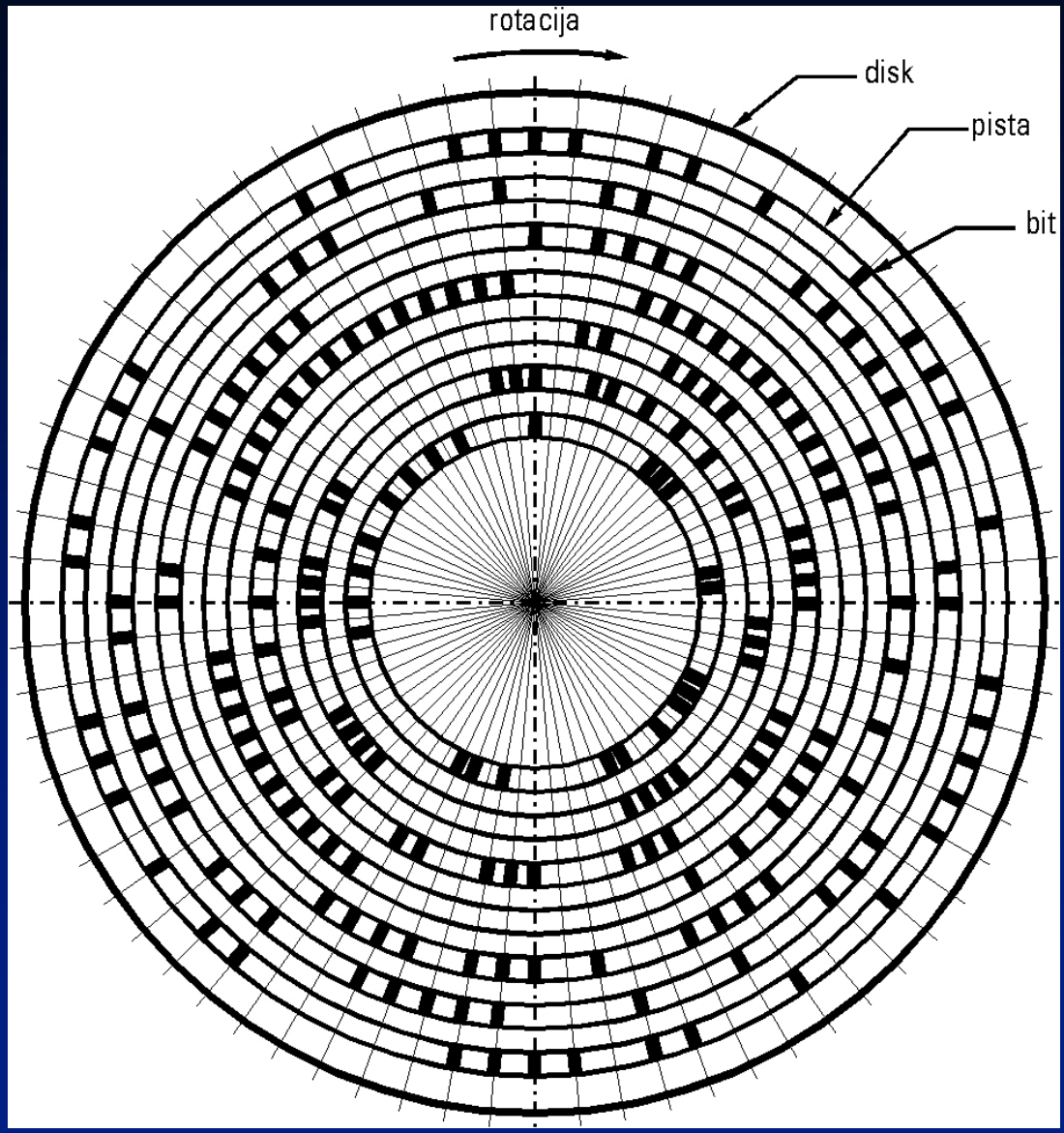
Pogled na jedan disk i jednu glavu



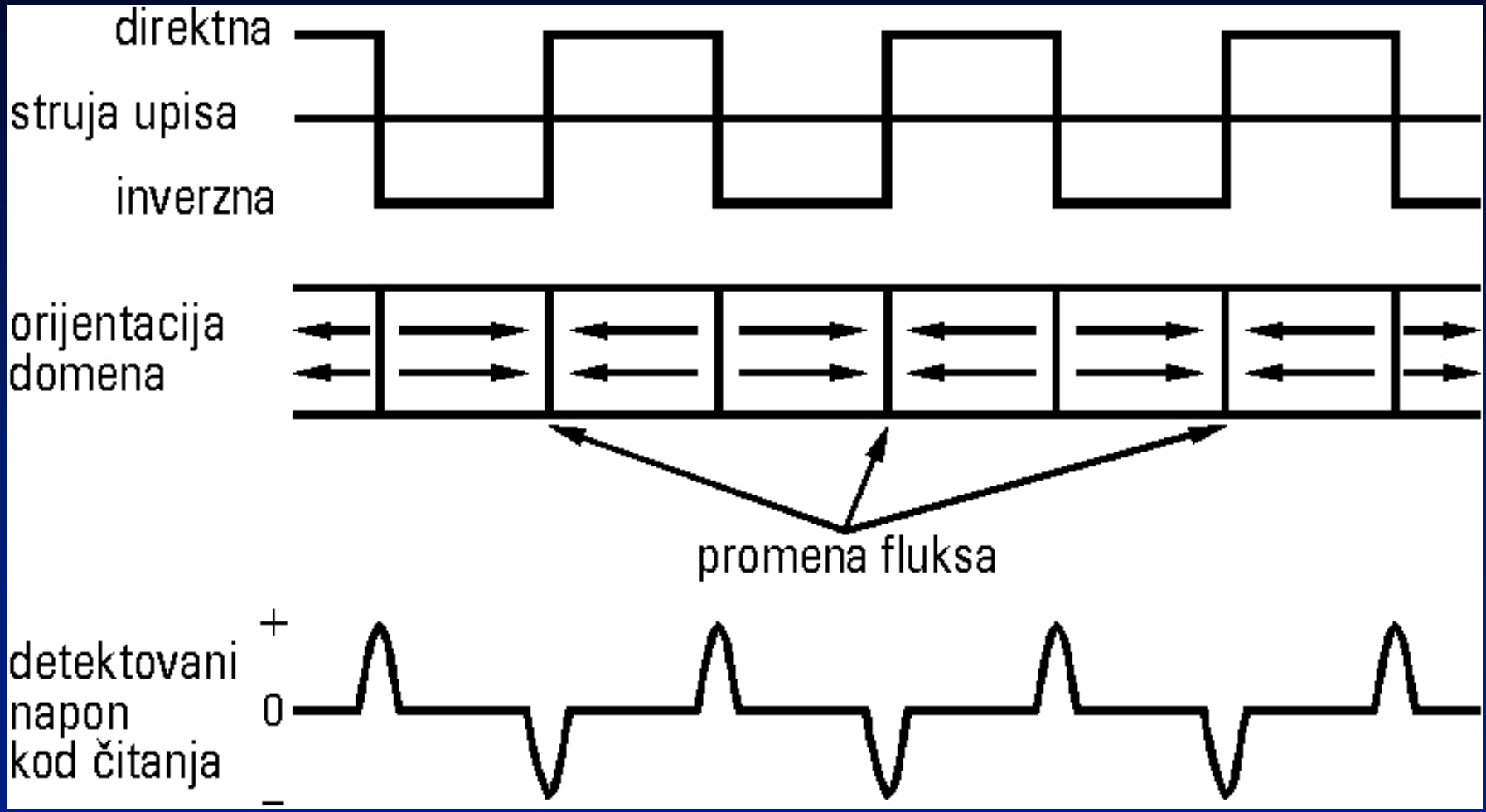
Logičke komponente krutog diska



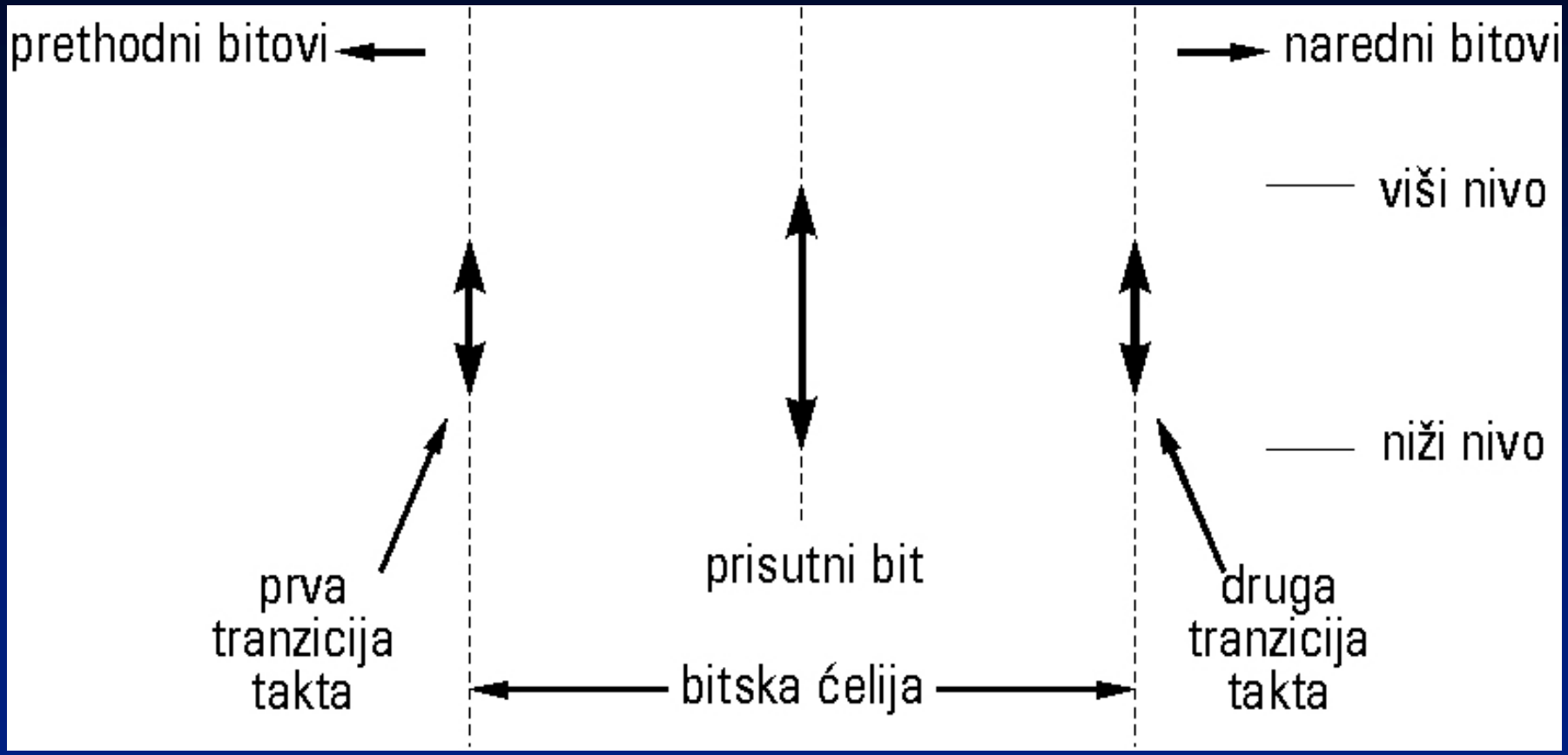
Uredjenost pisti na disku



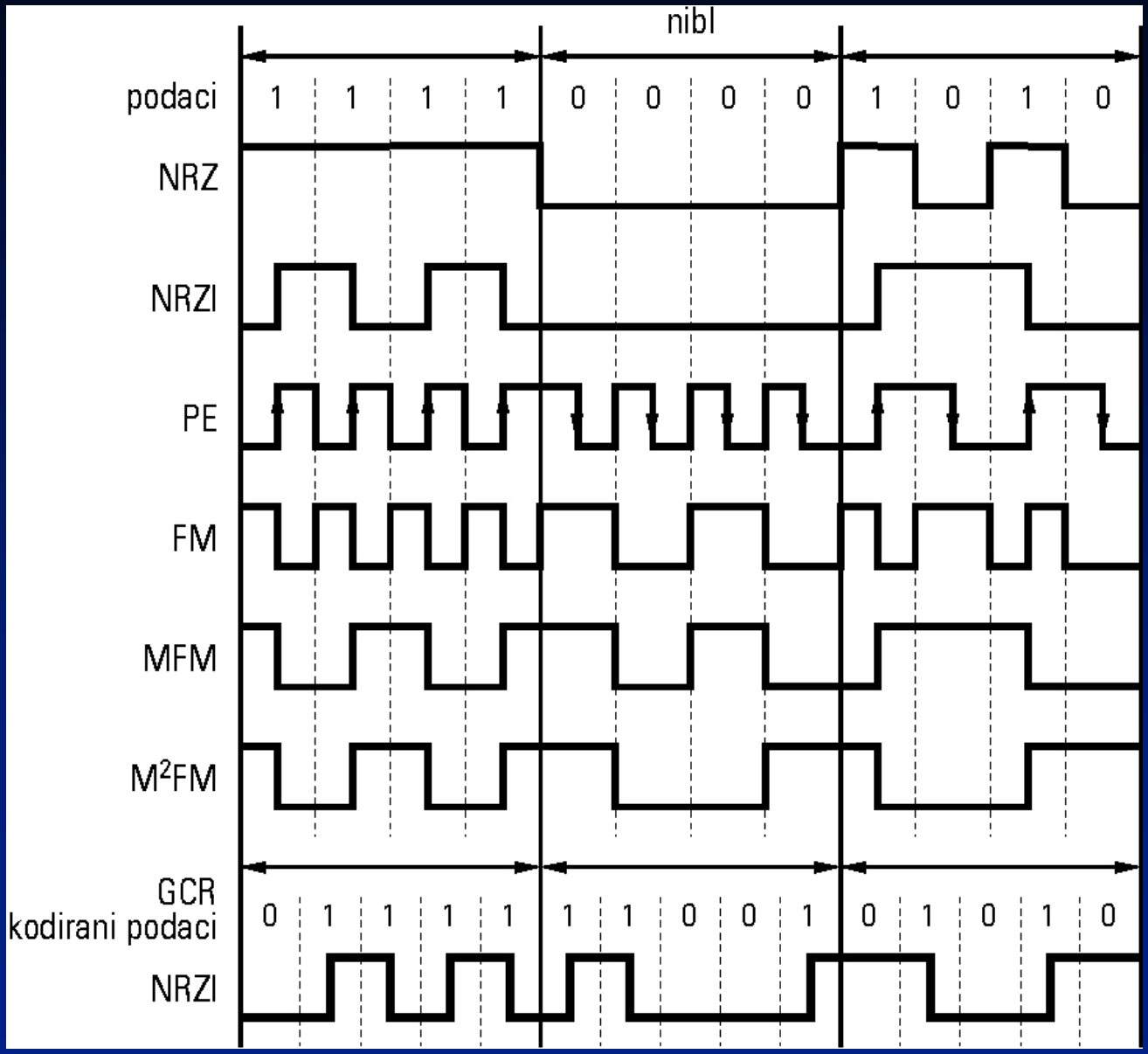
Odnos izmedju struje upisa, promene fluksa i detektovanog signala kod čitanja



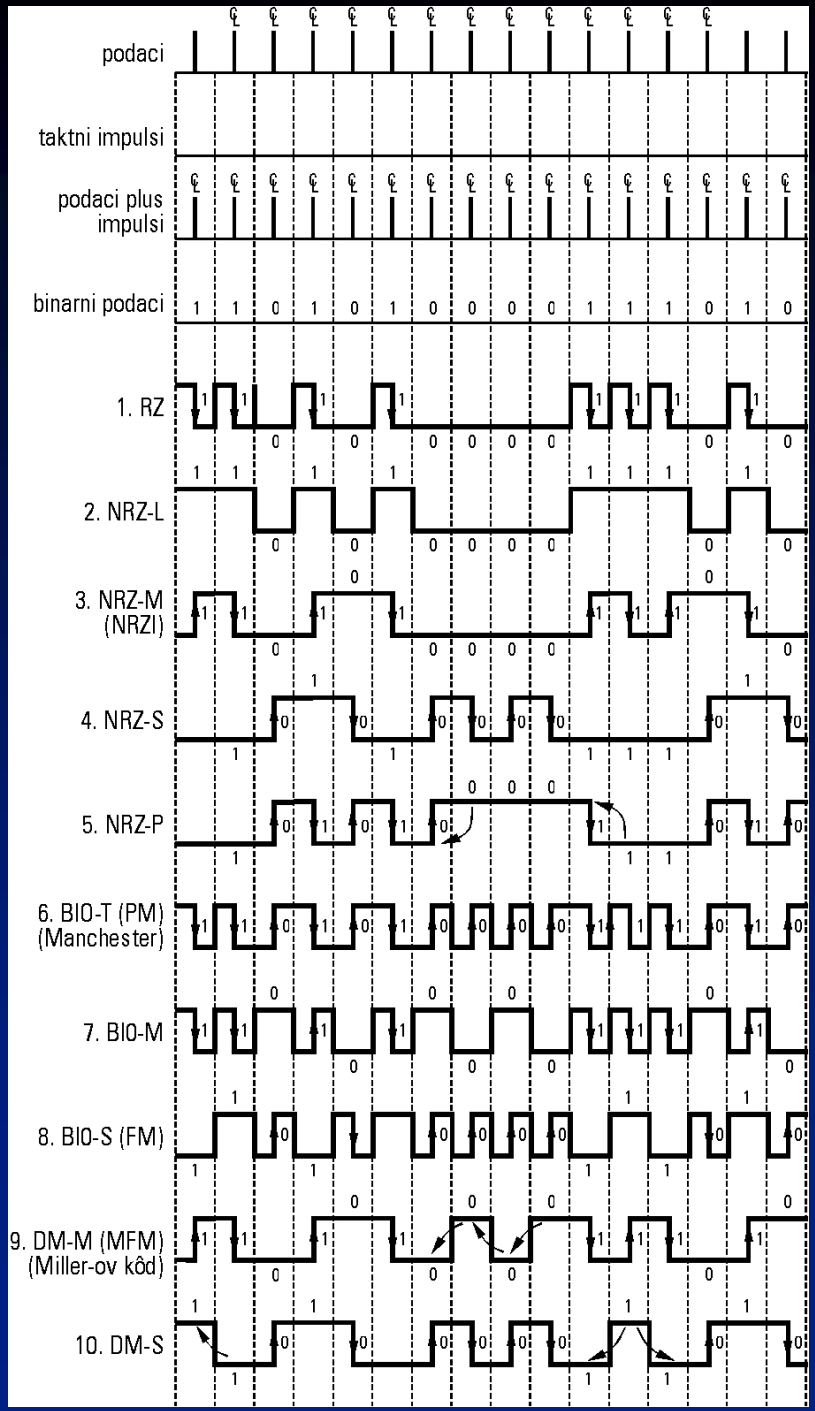
Bitska ćelija



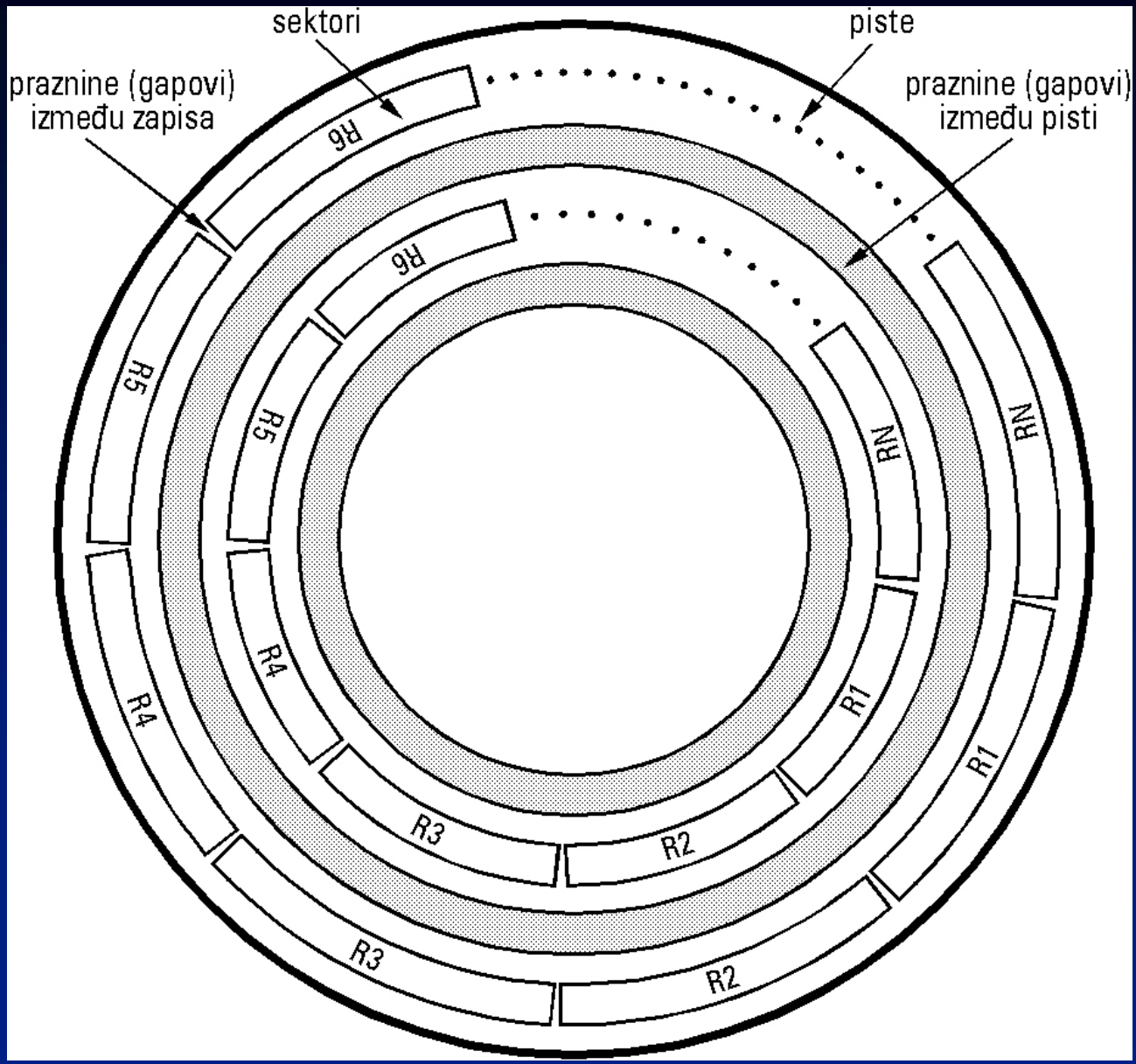
Tipični kodovi kod digitalnog magnetnog zapisa



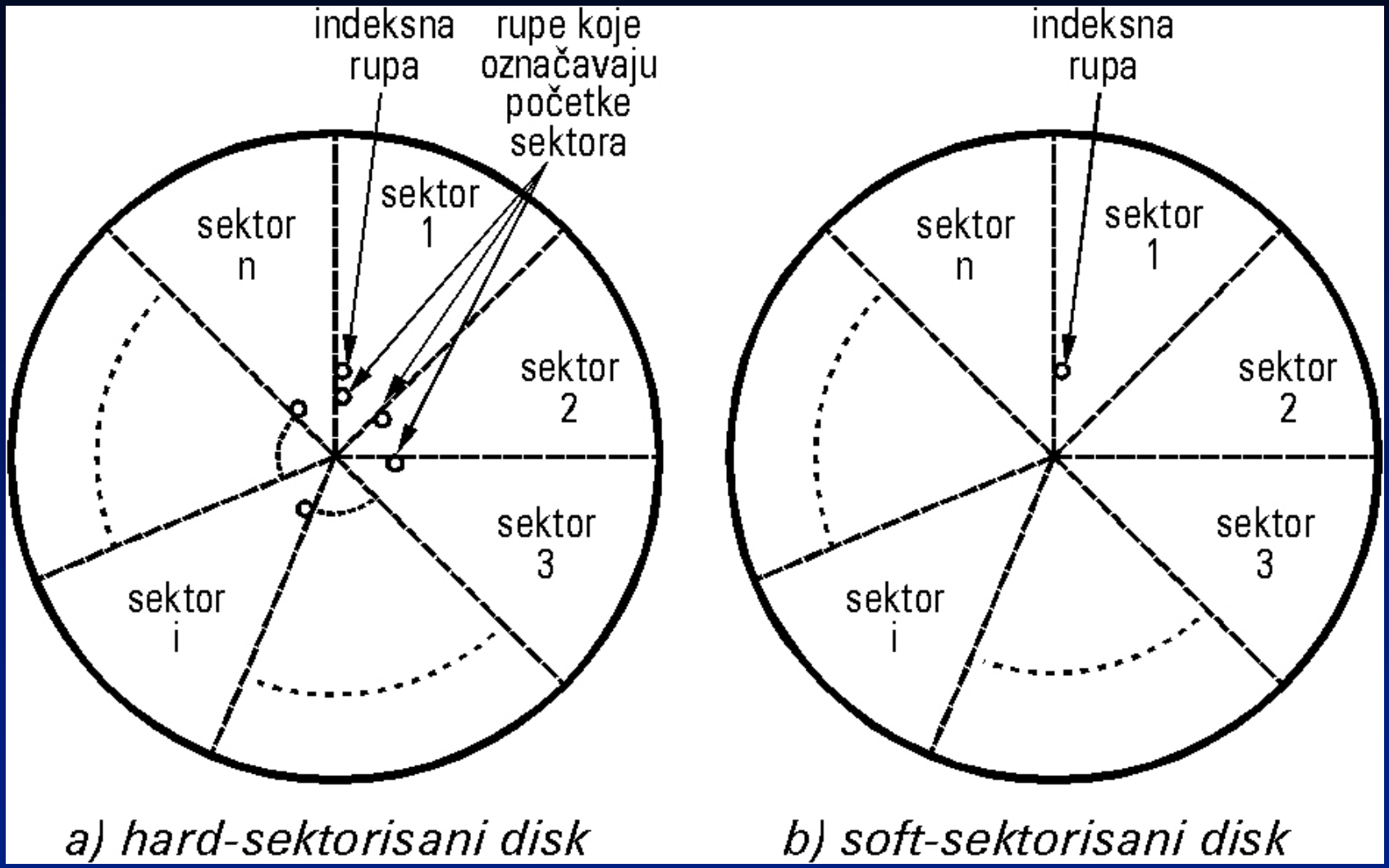
Talasni dijagrami najčešće korišćenih kodova za magnetni zapis



Izgled diska sa aspekta pisti i sektora



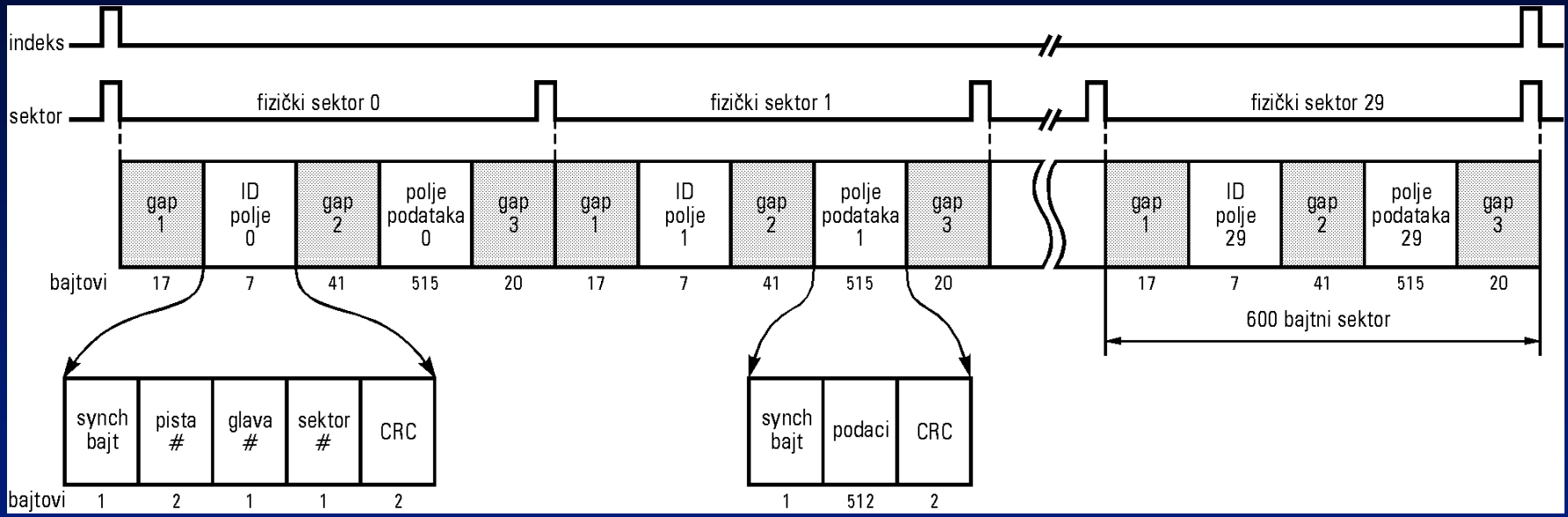
Način sektorisanja kod diskova



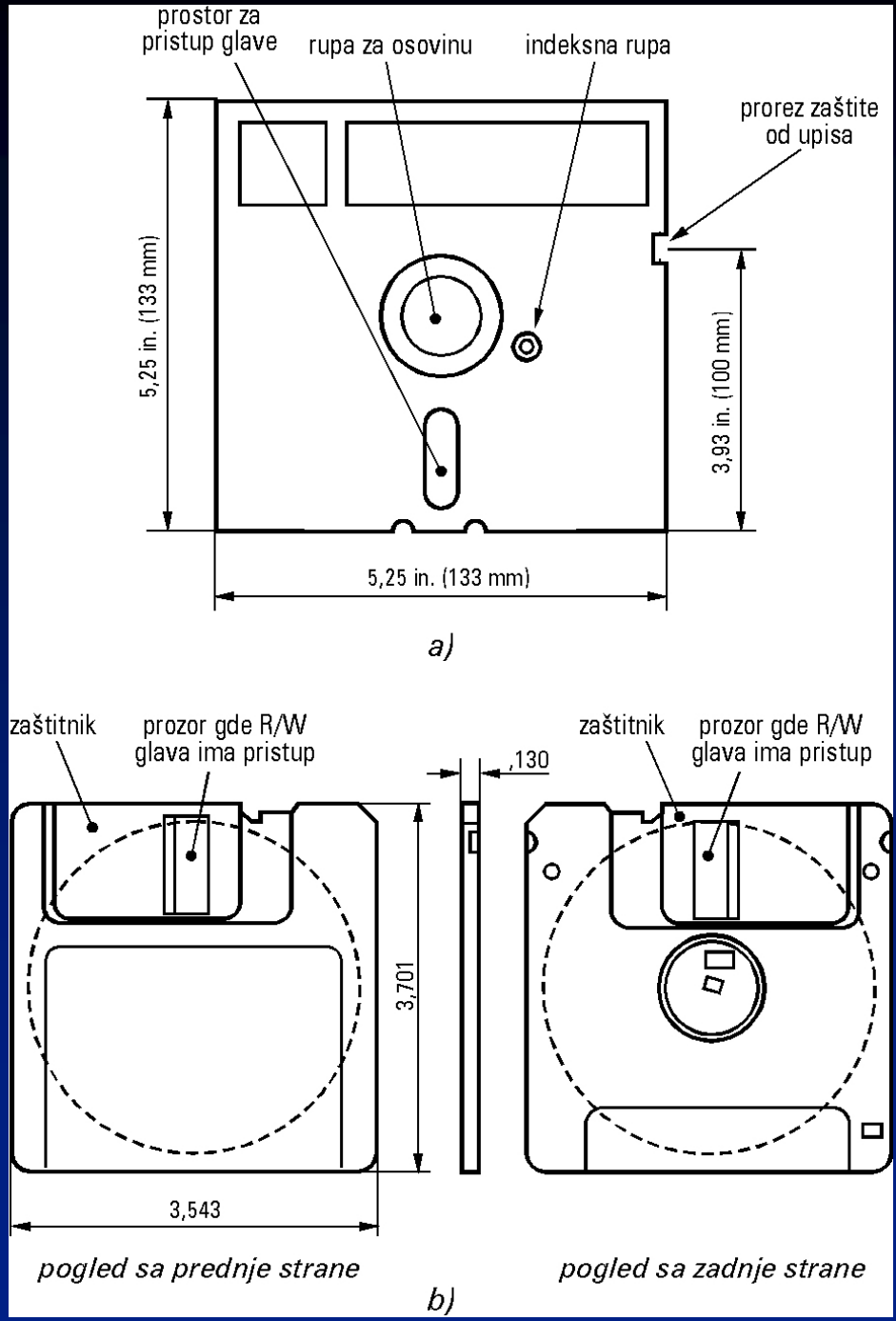
a) hard-sektorisani disk

b) soft-sektorisani disk

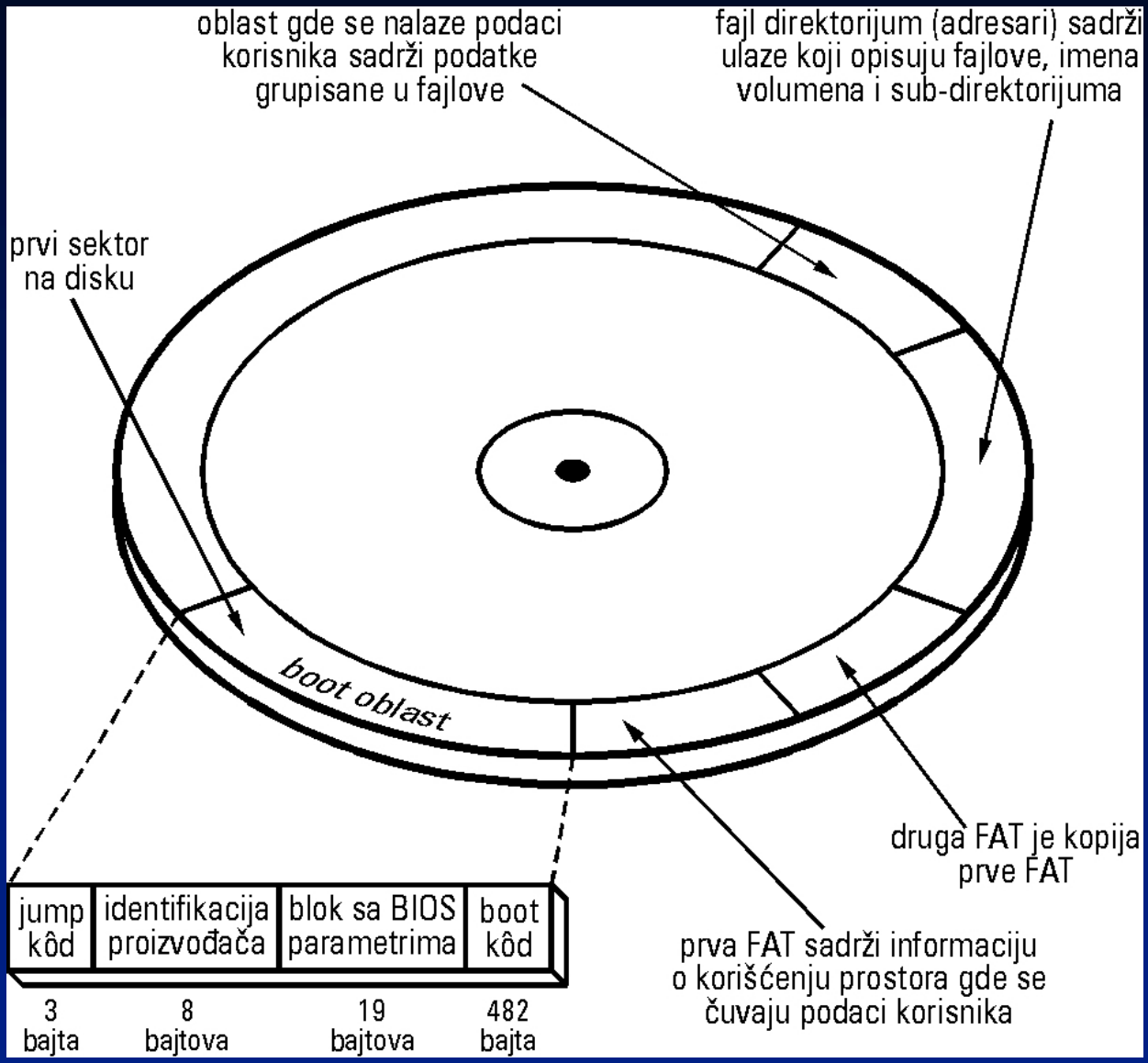
Formatiranje *Winchester* diska



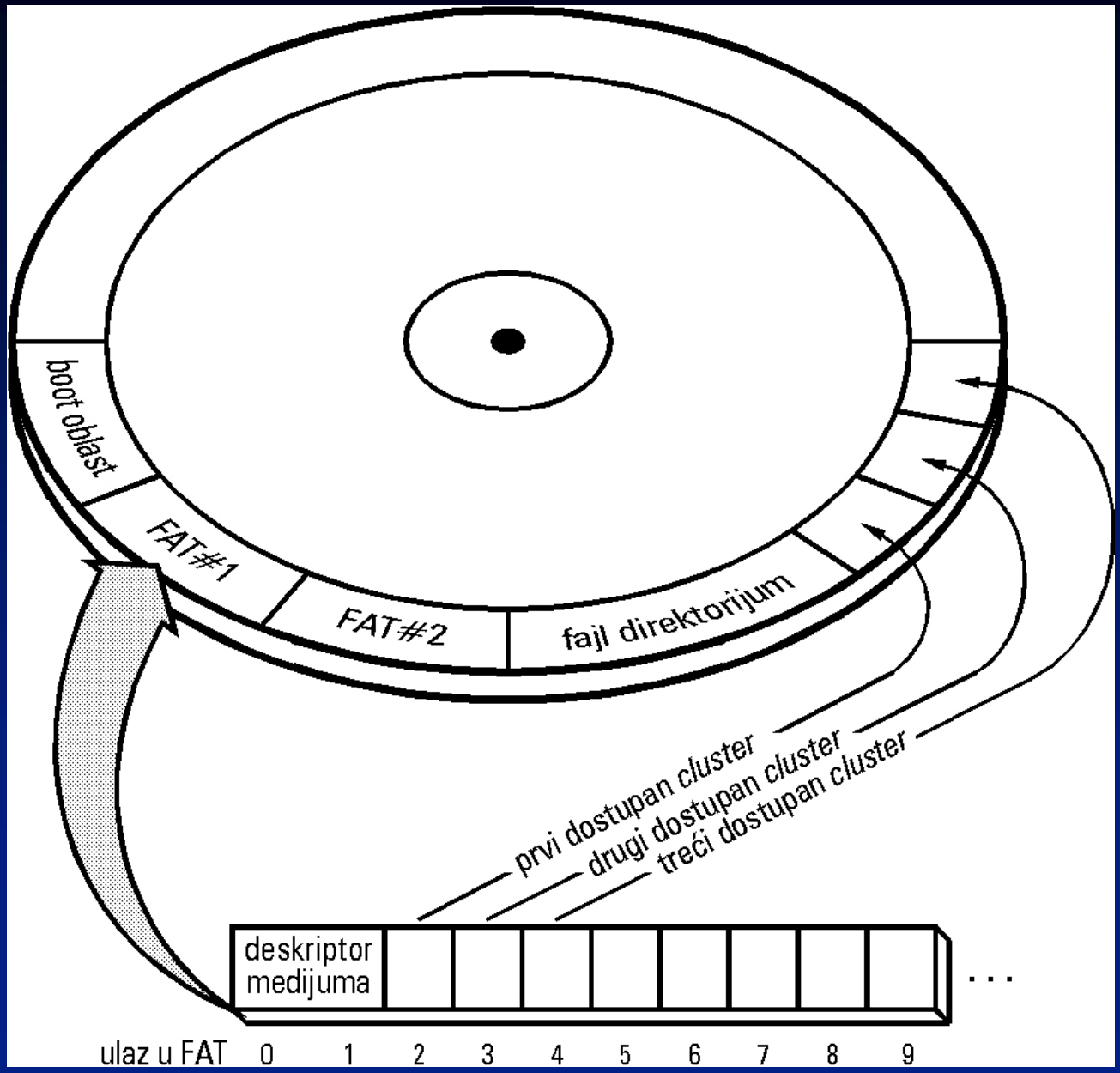
Izgled mini i mikro flopi disketa



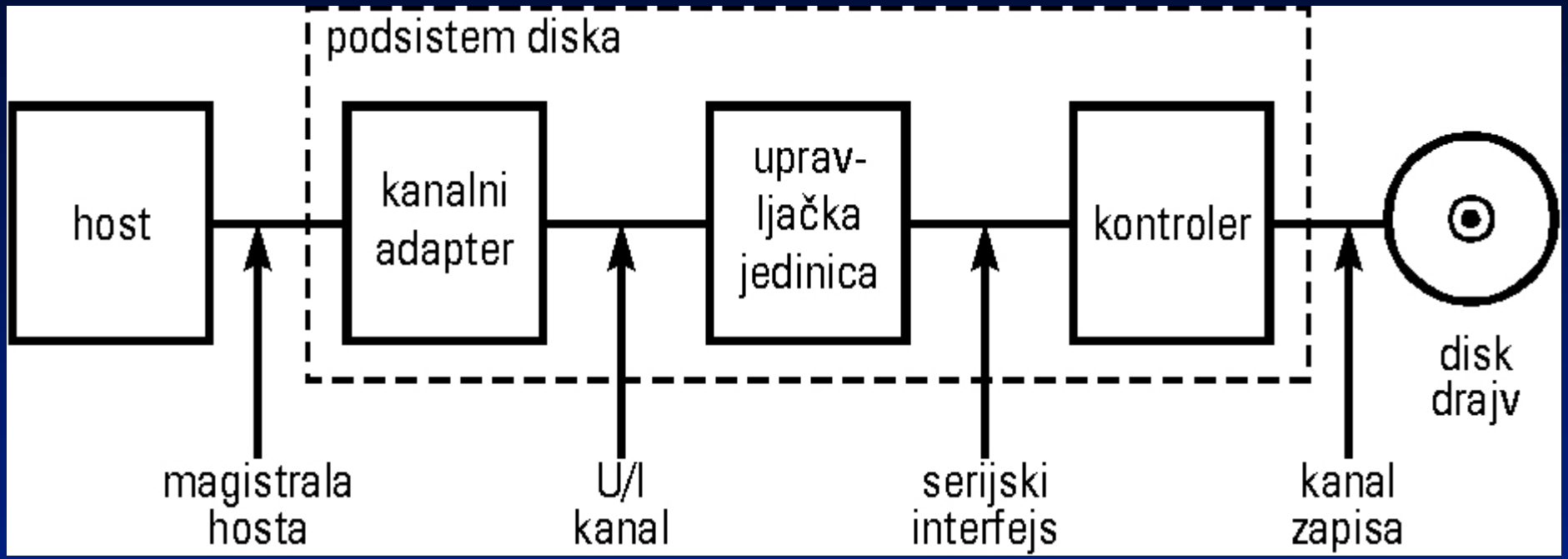
Tipičan prikaz *boot* oblasti



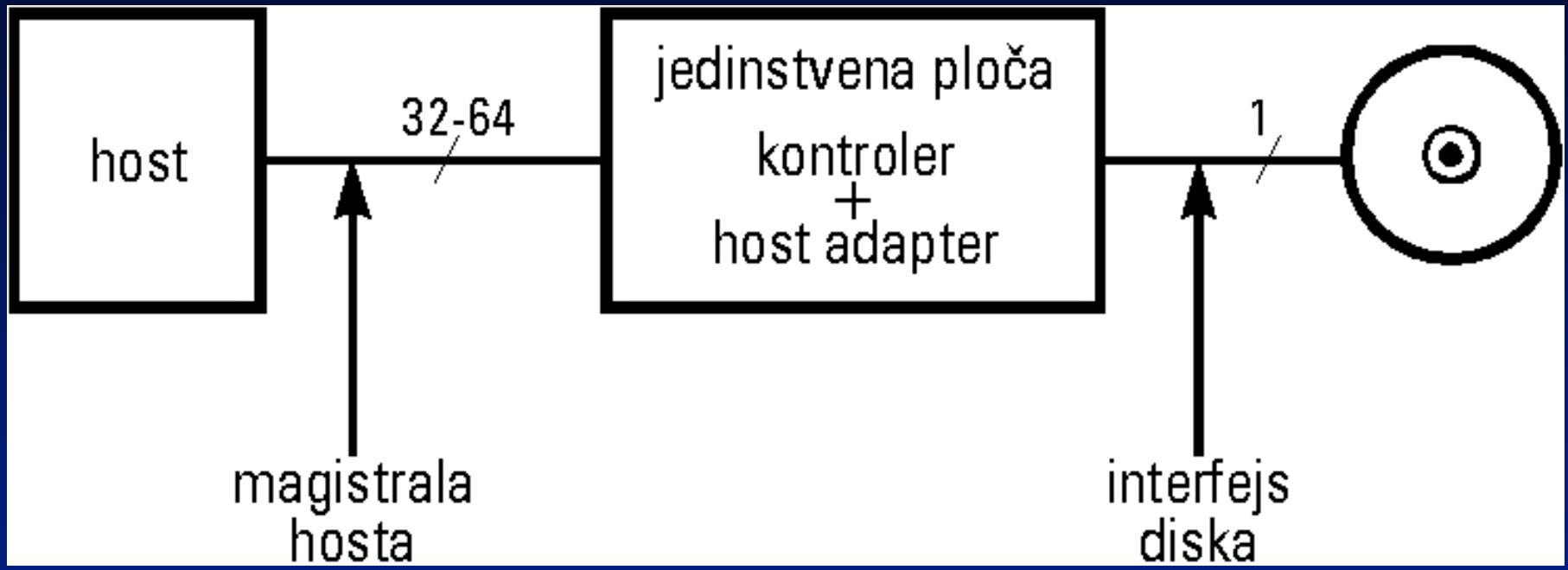
Odnos izmedju FAT ulaza i *cluster*-a



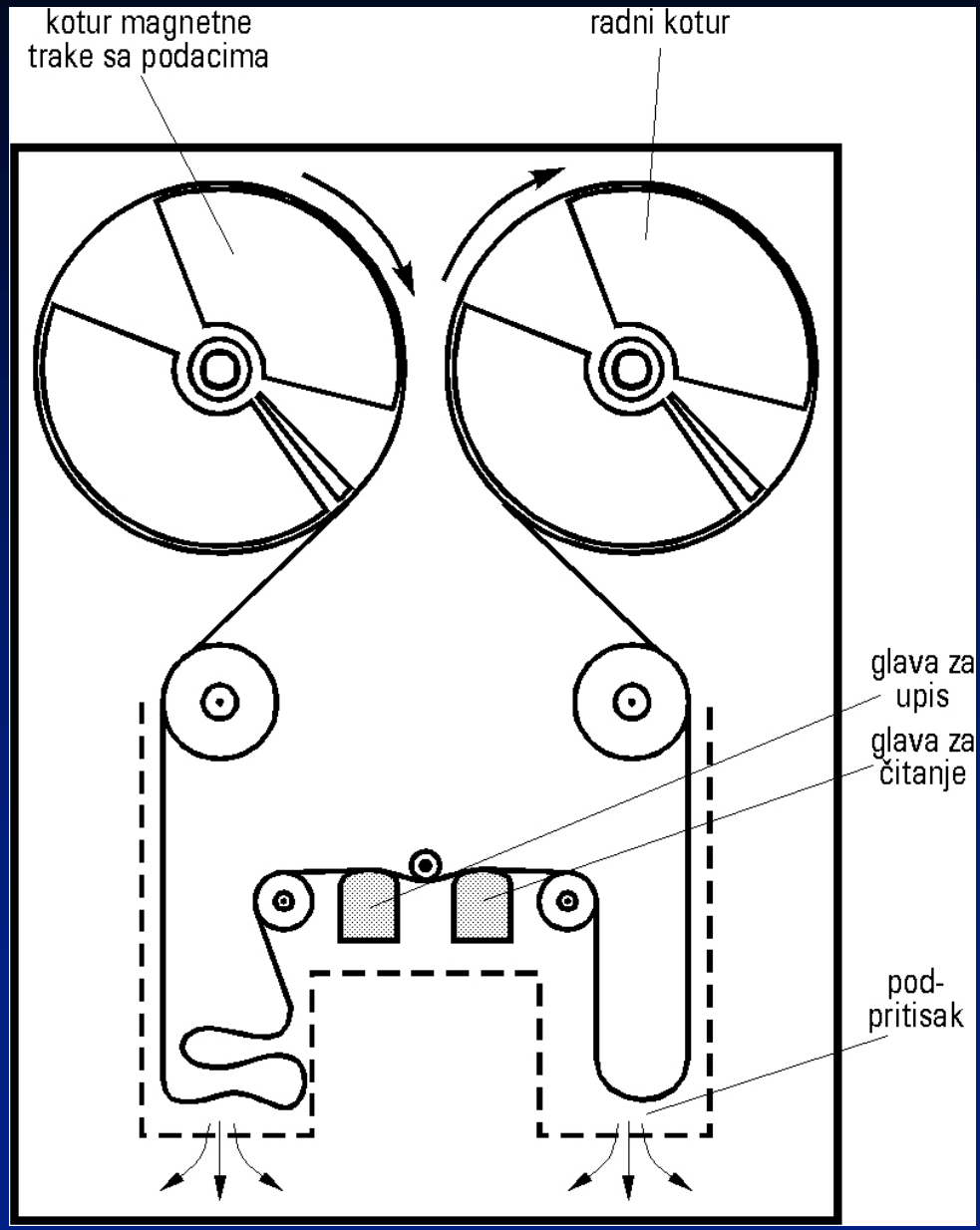
Struktura arhitekture diska



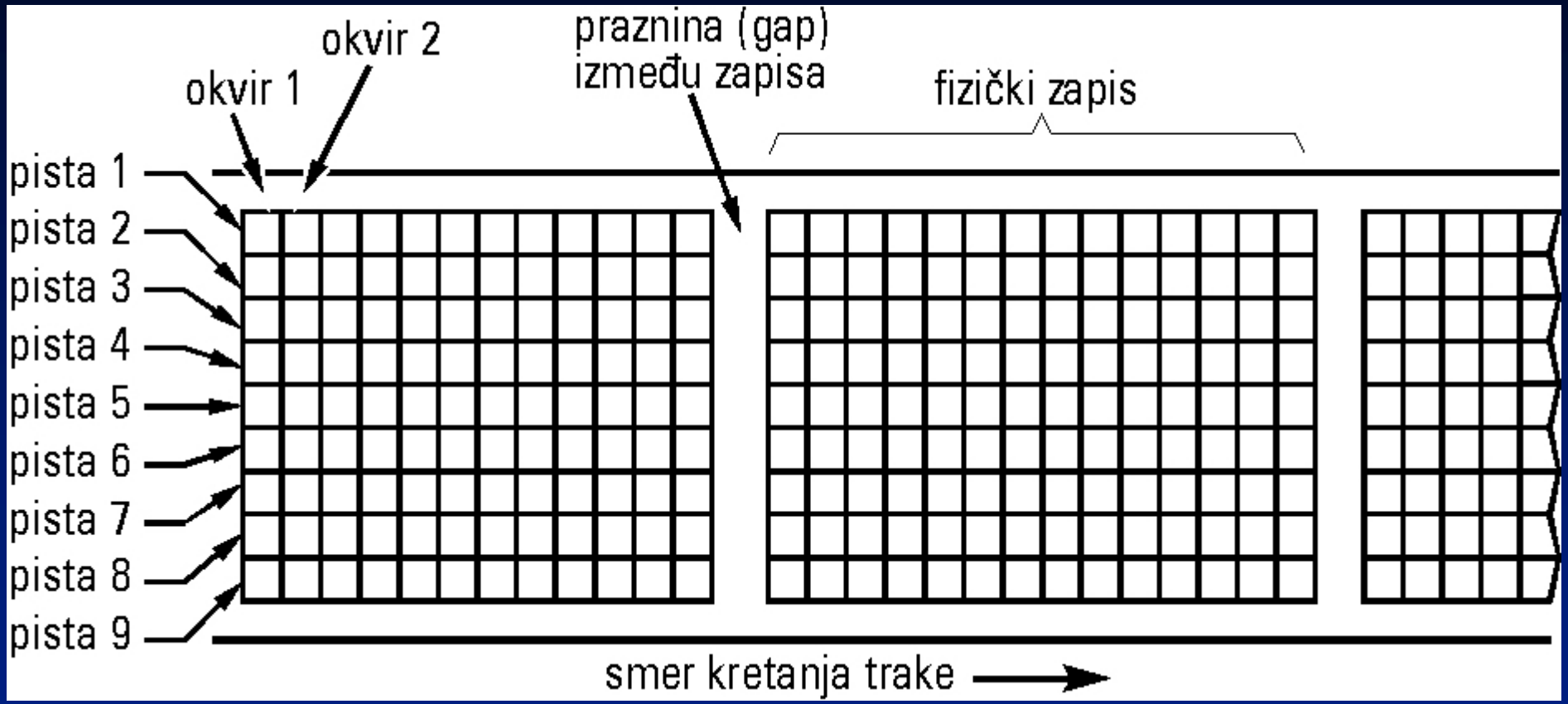
Arhitektura sekundarne magnetne memorije kod IBM PC – AT mašine



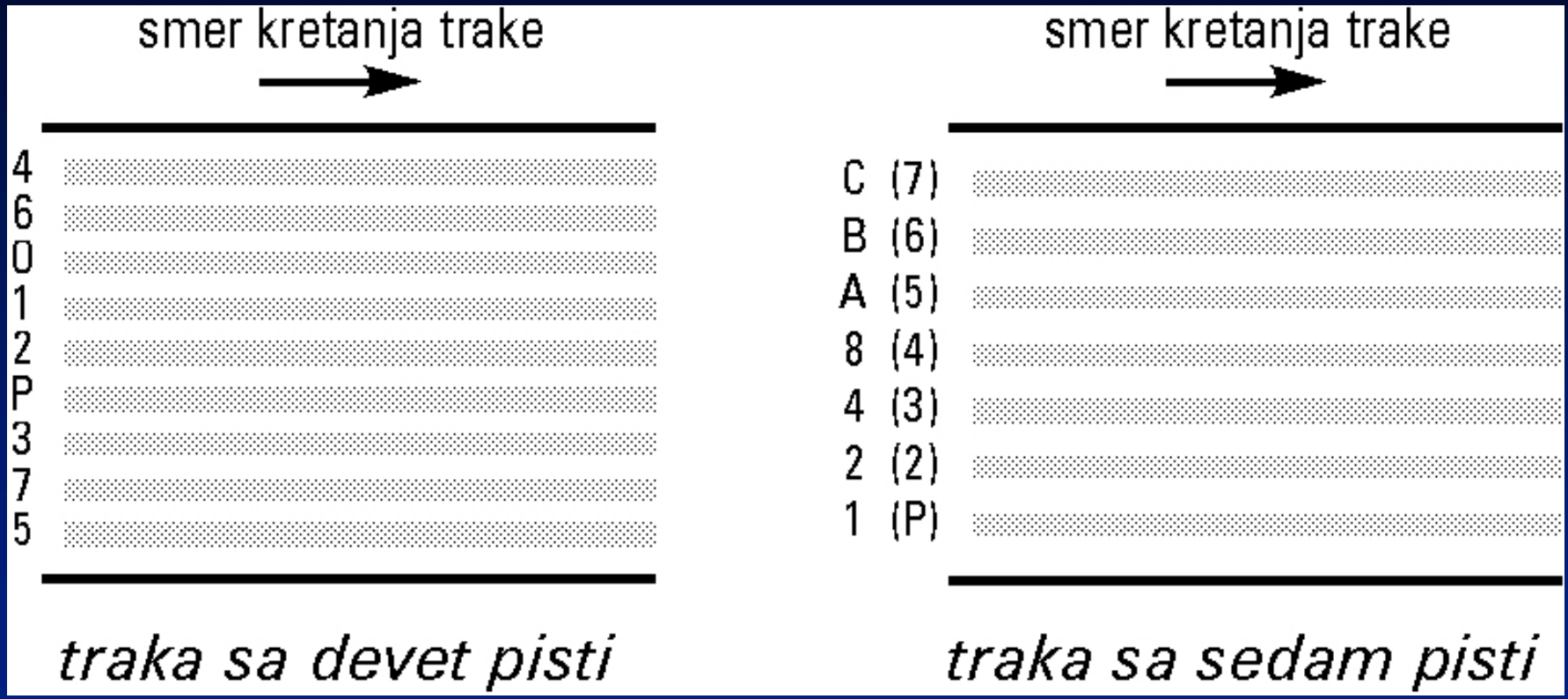
Jedinica magnetne trake



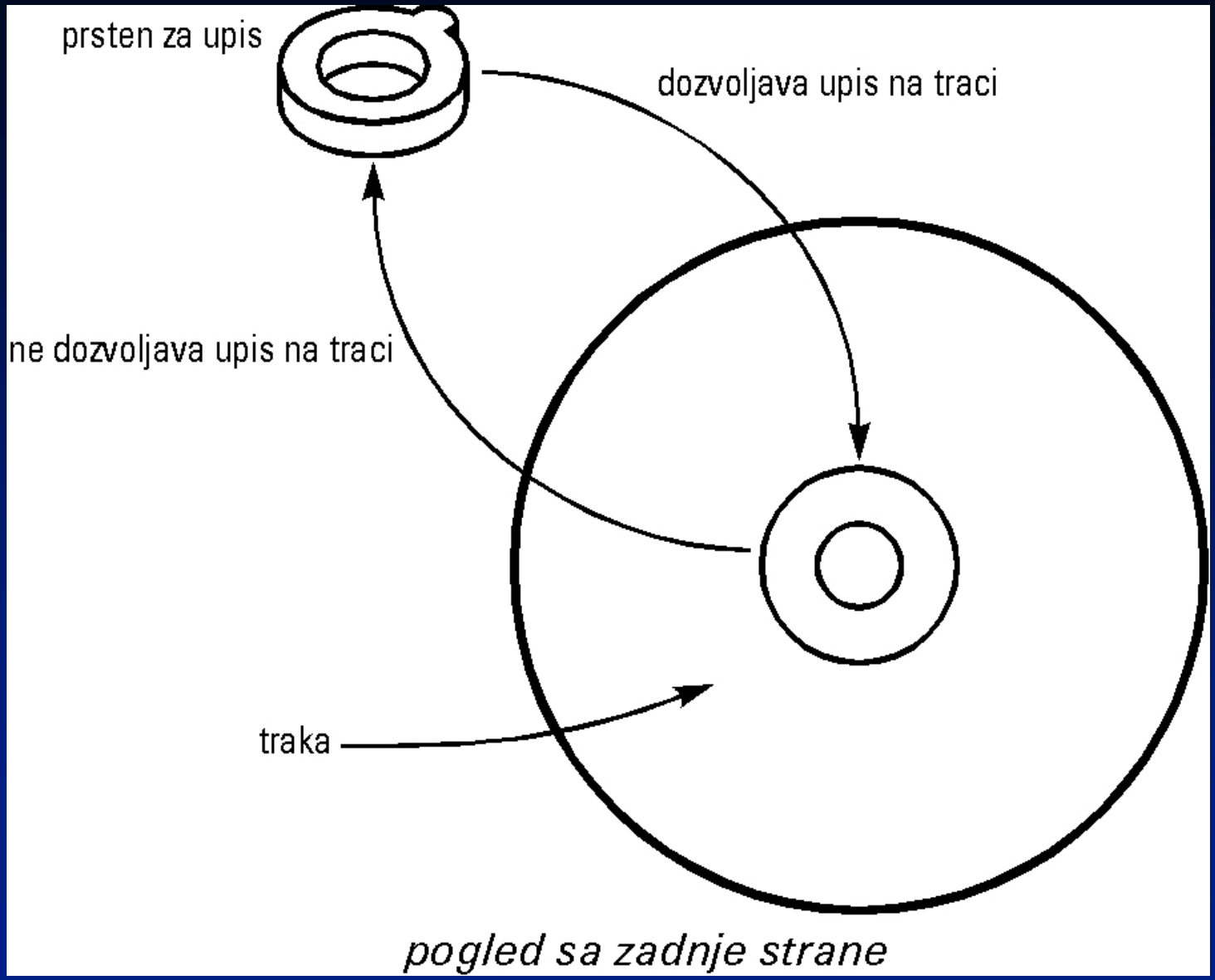
Zapis informacije na magnetnoj traci



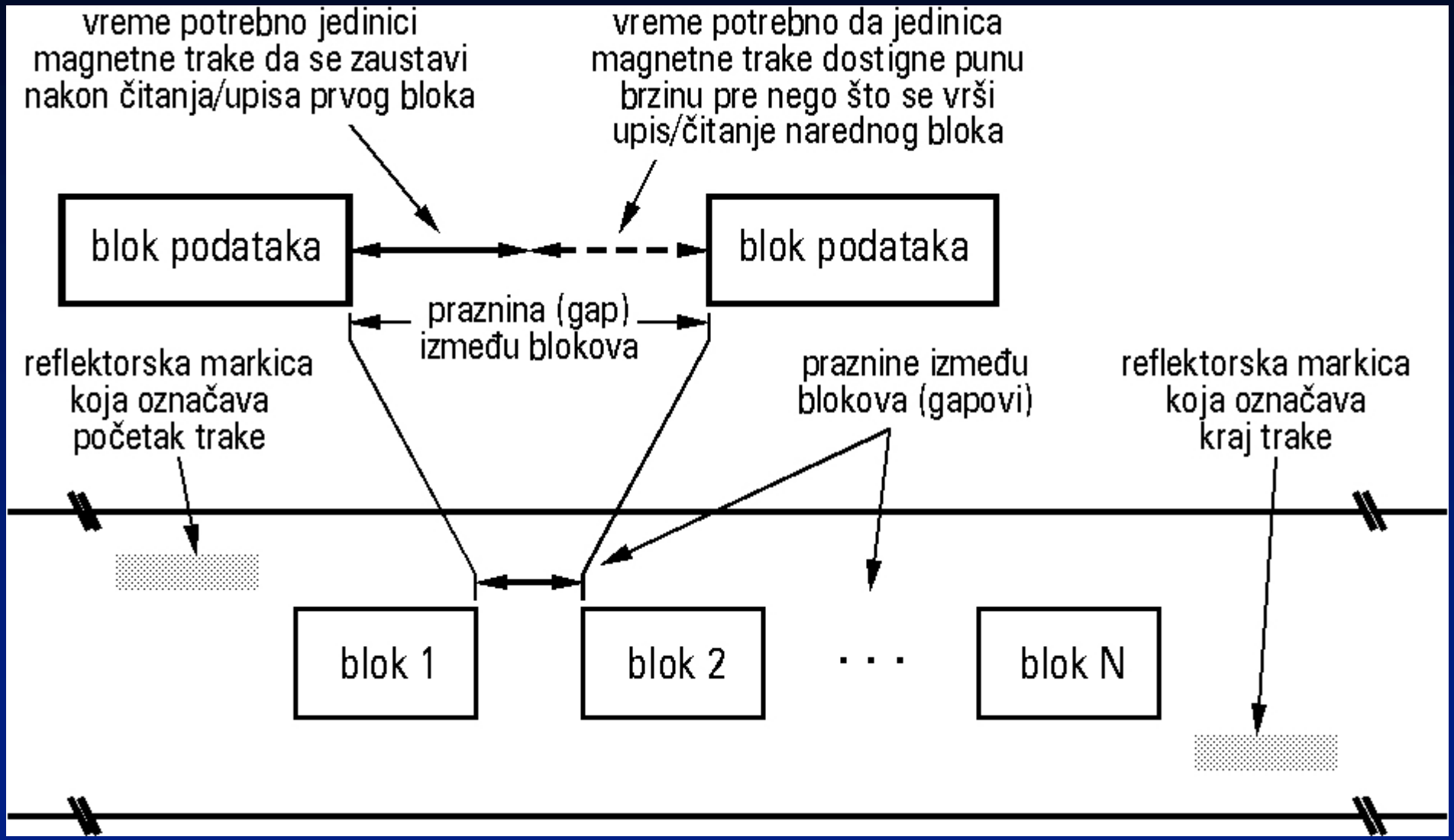
Fizički raspored pisti



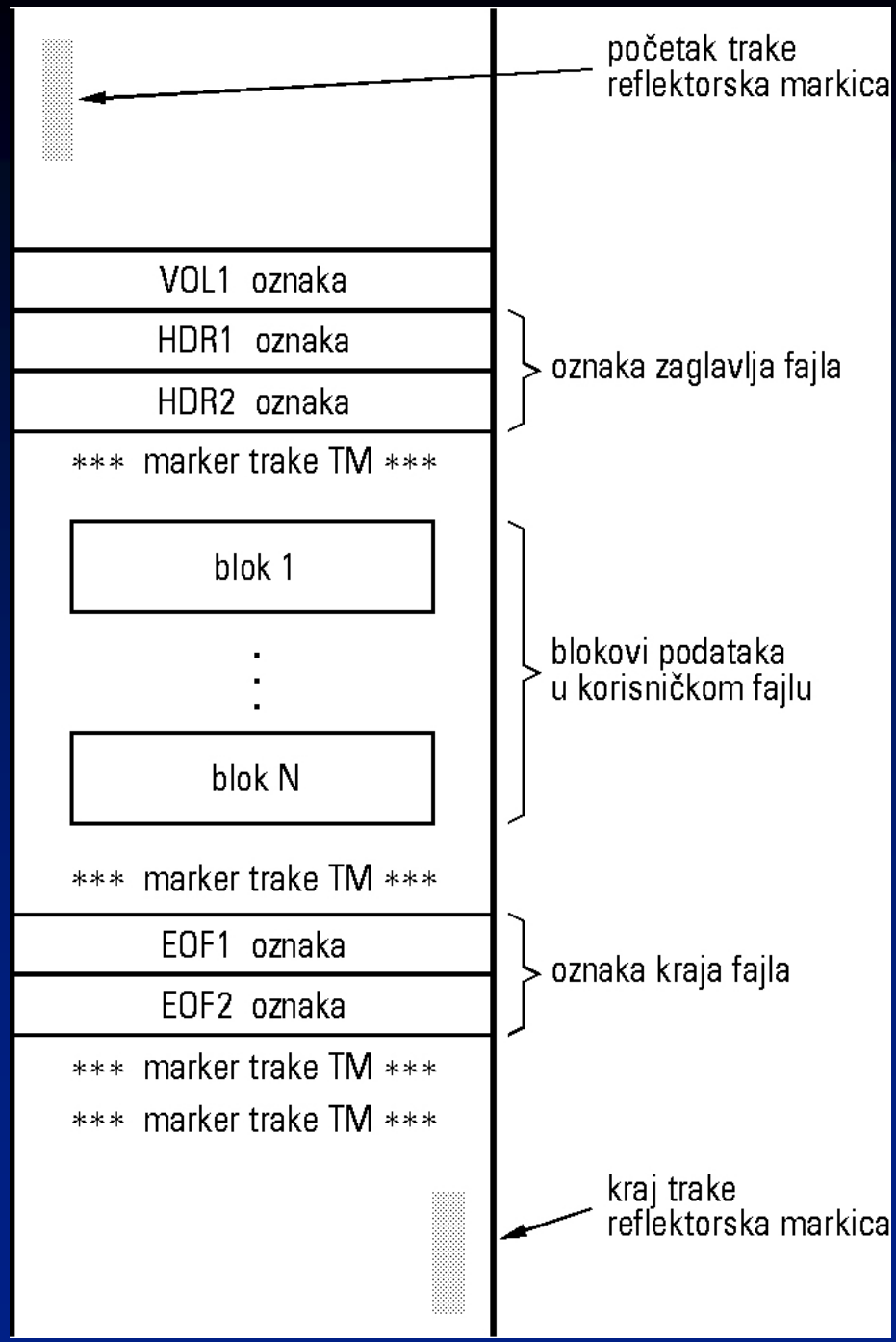
Traka i uloga prstena za upis



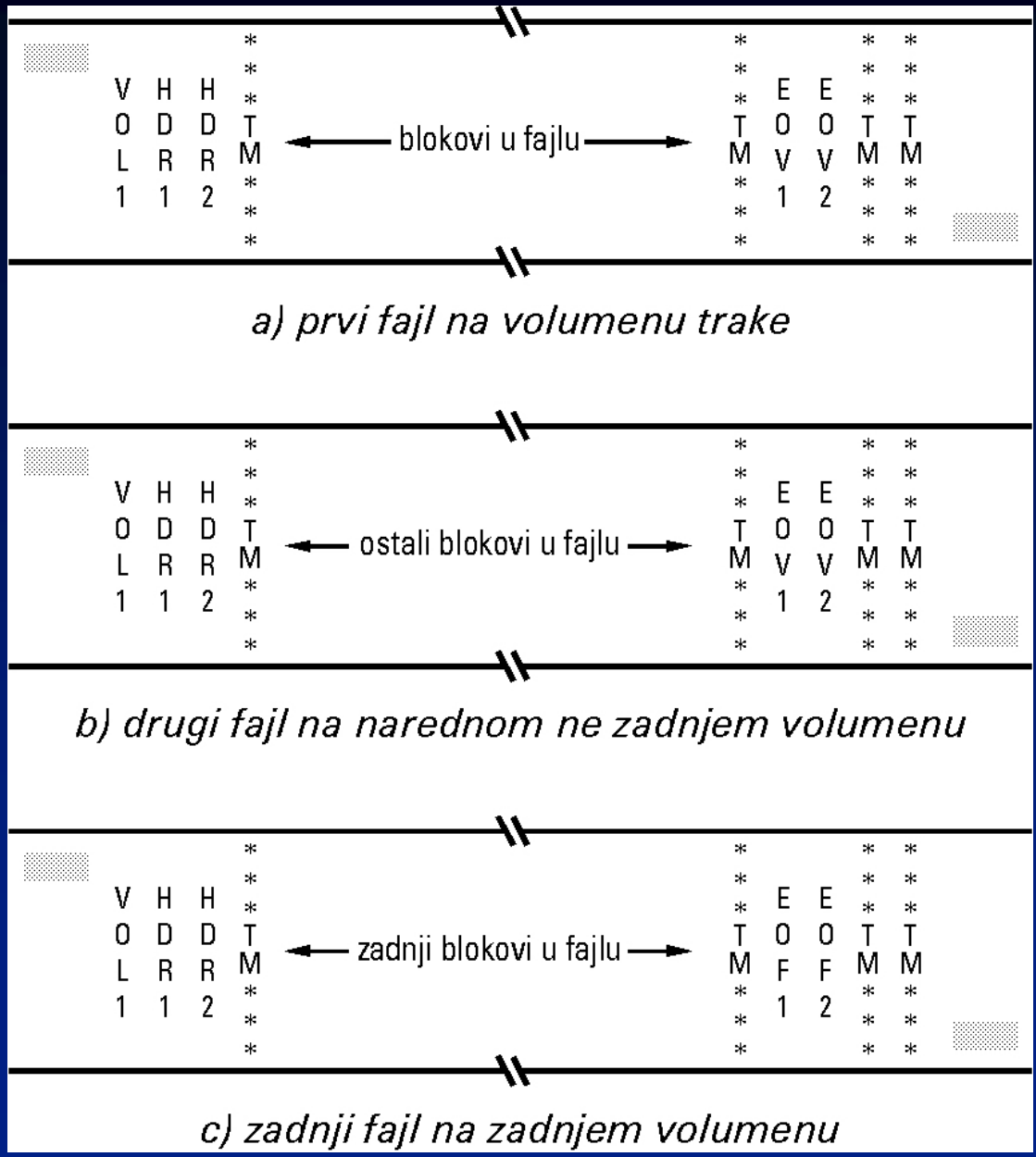
Fizička struktura trake



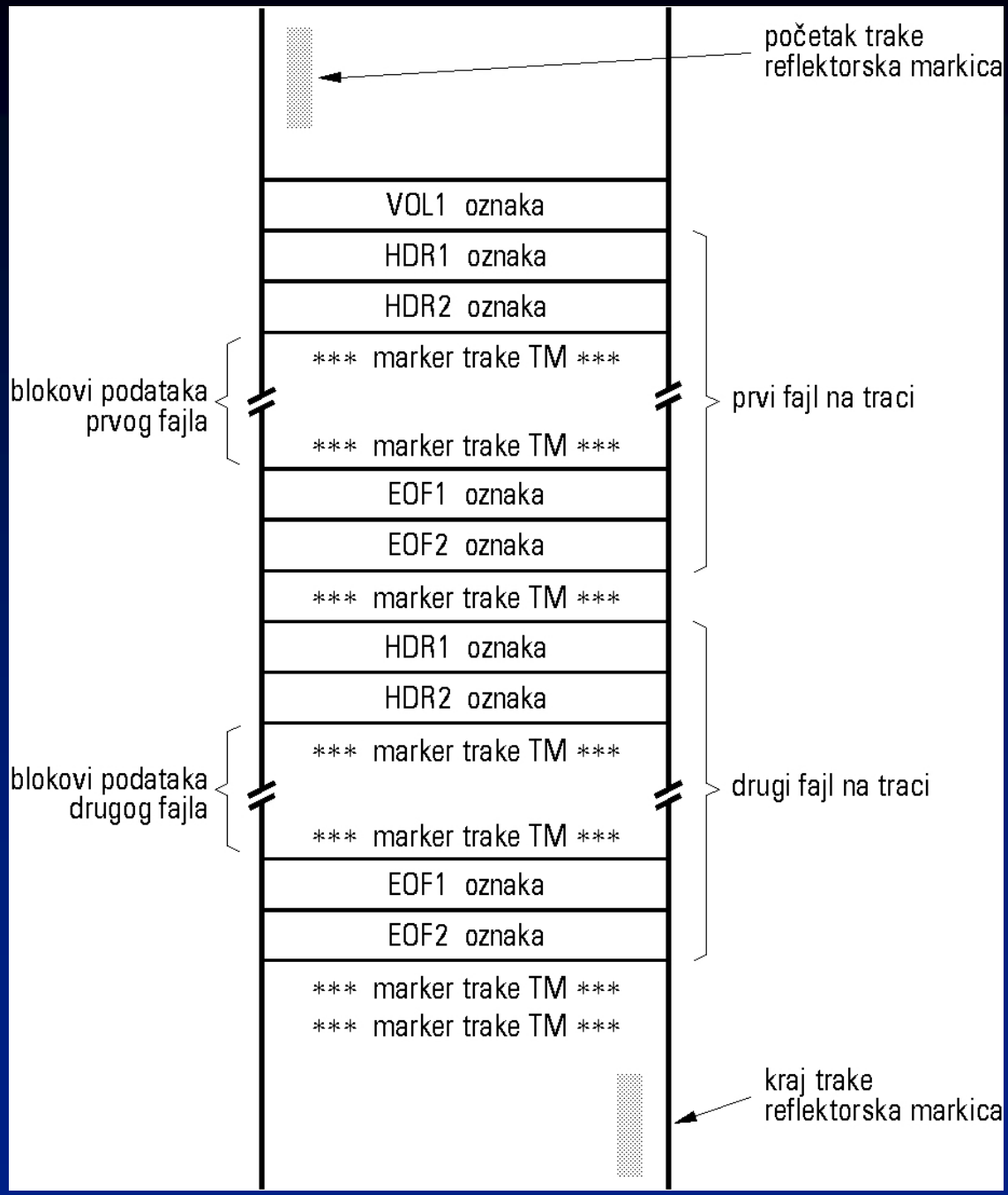
Struktura datoteka na jednom volumenu



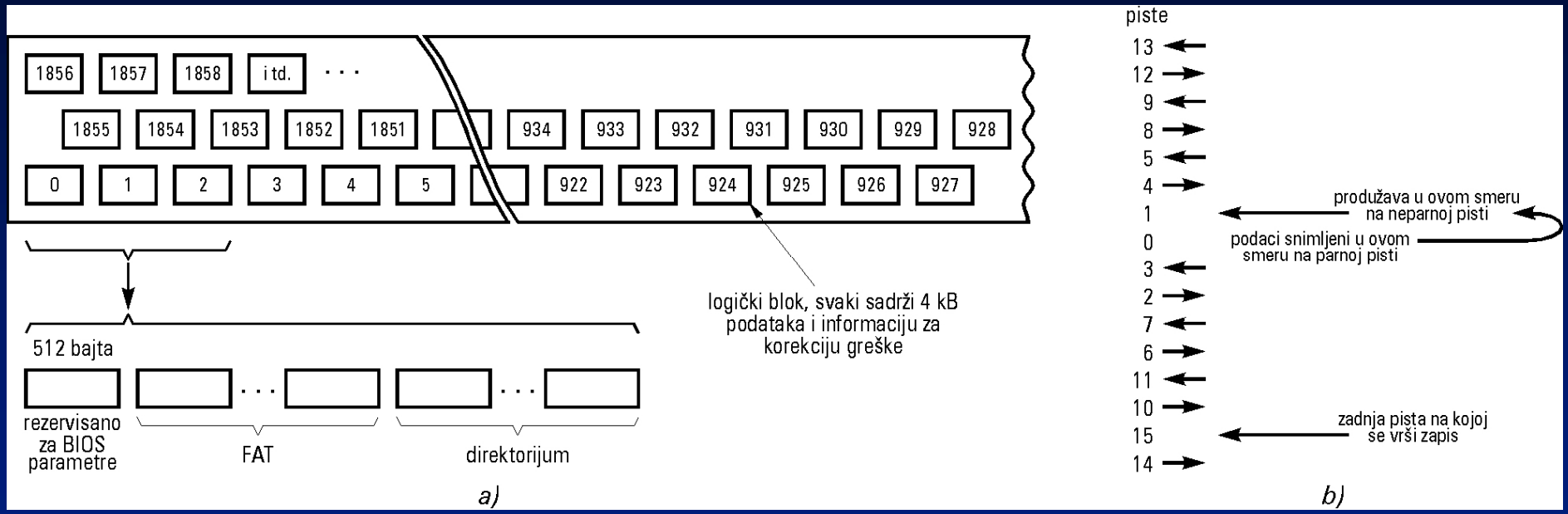
Datoteka na više volumena



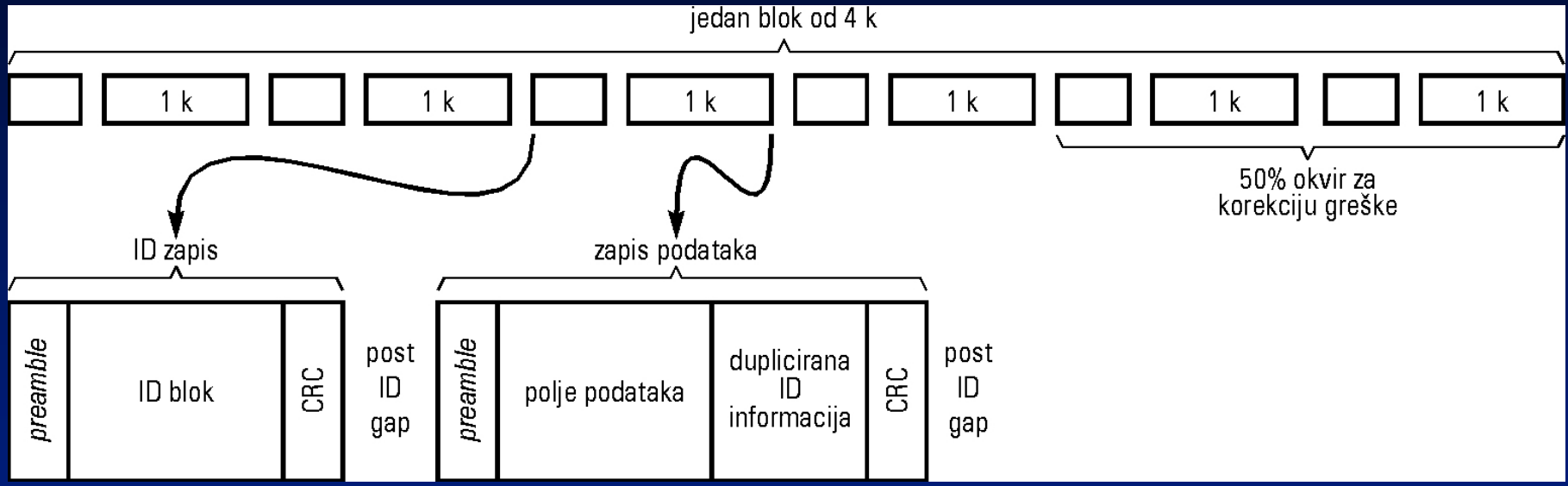
Traka sa više datoteka



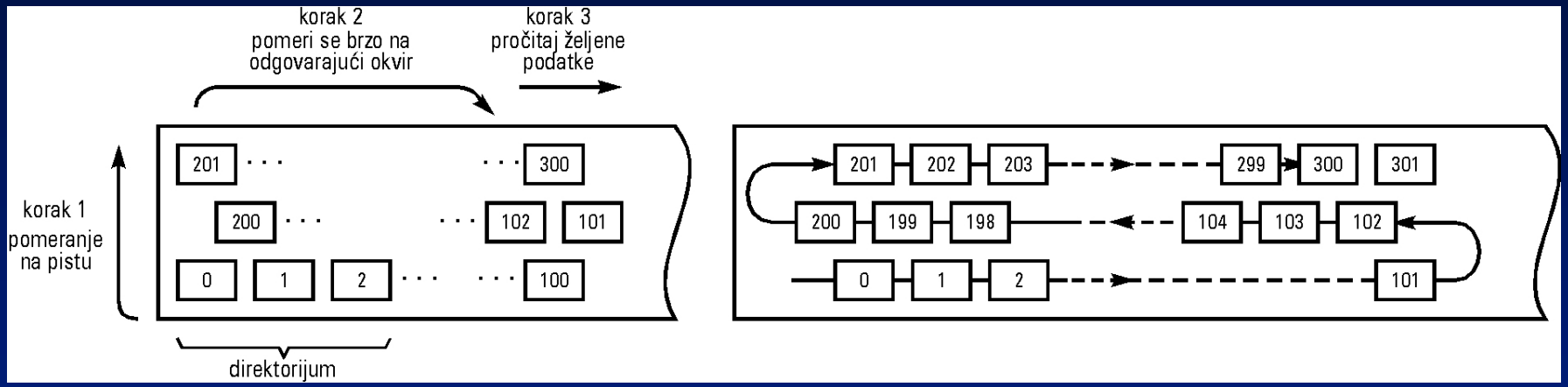
Organizacija zapisa kod *streamer* traka



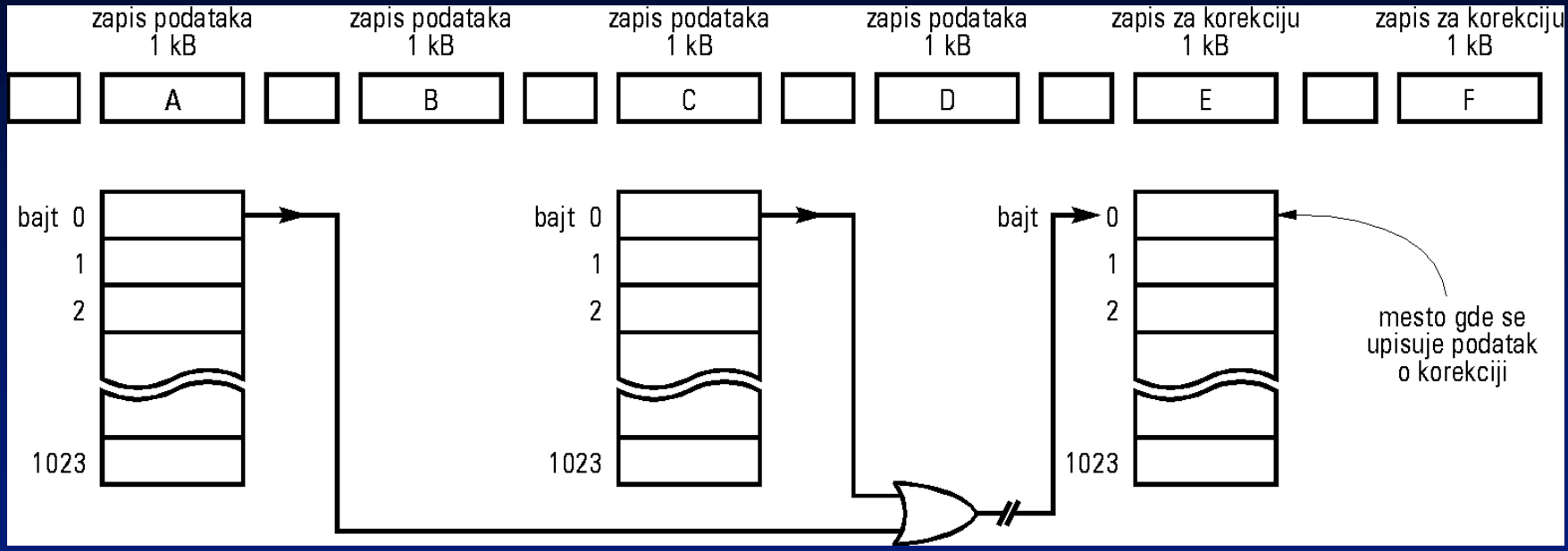
PC/T format zapisa



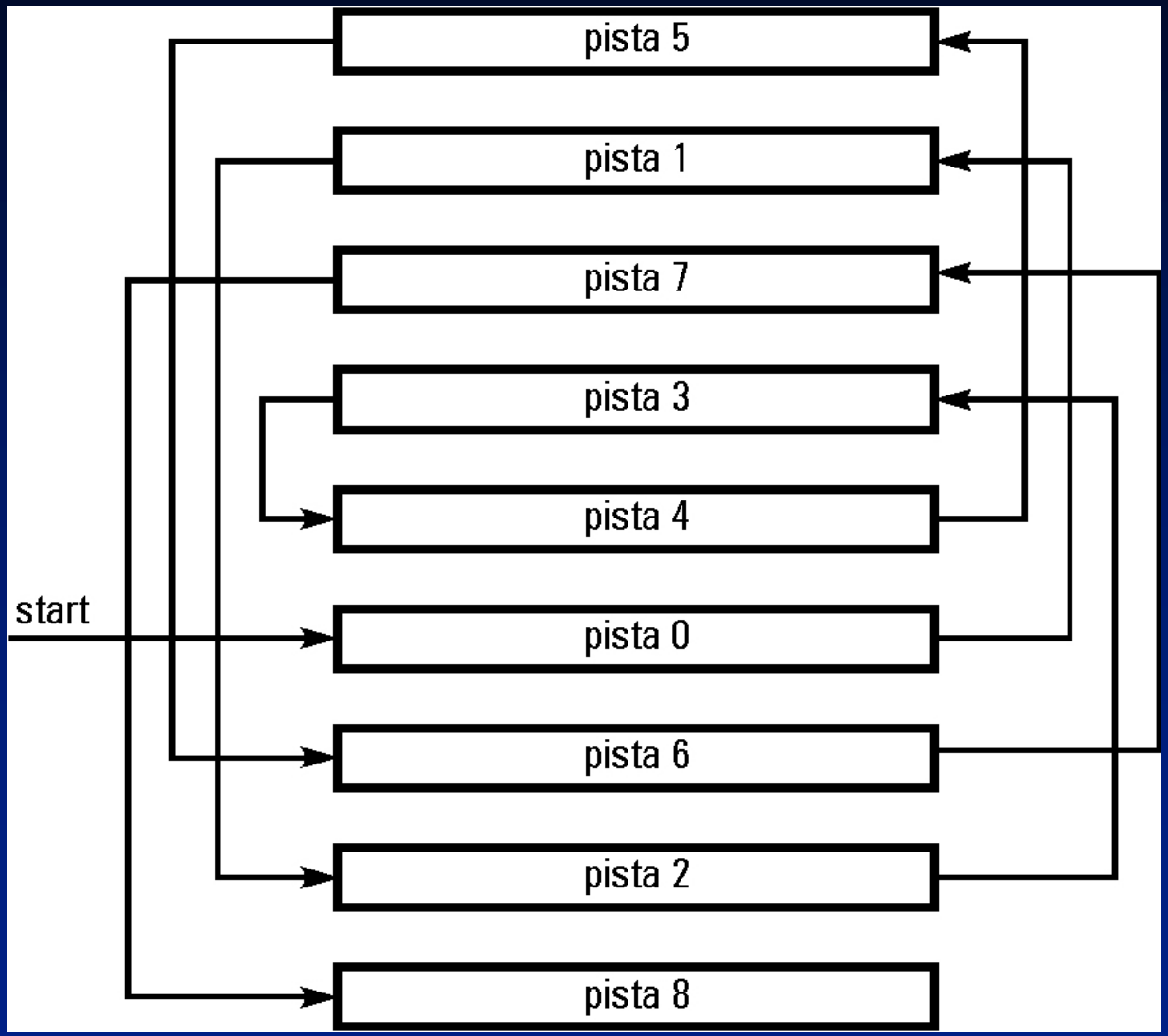
Lociranje bloka kod PC/T formata



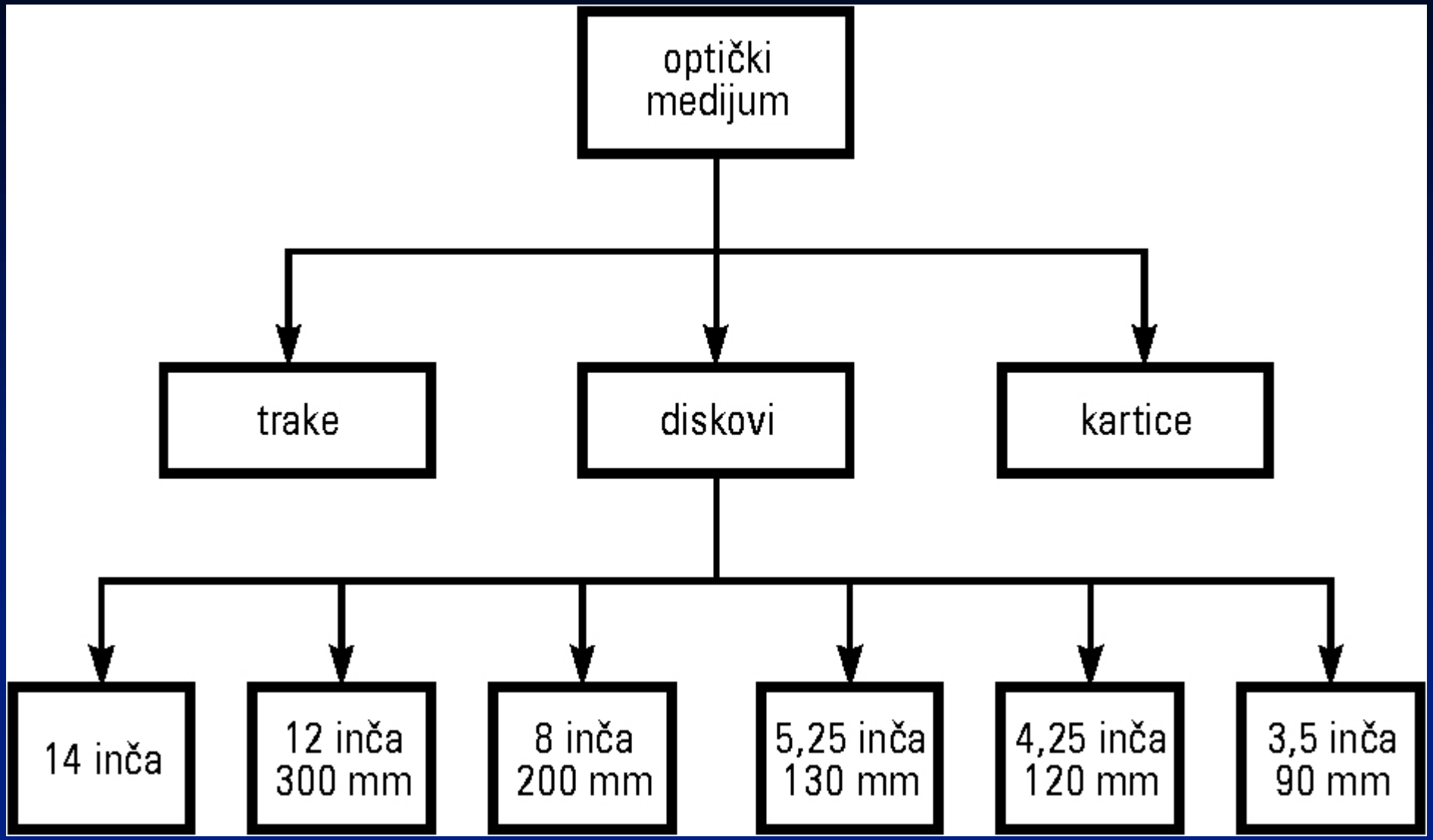
Korekcija grešaka kod PC/T formata



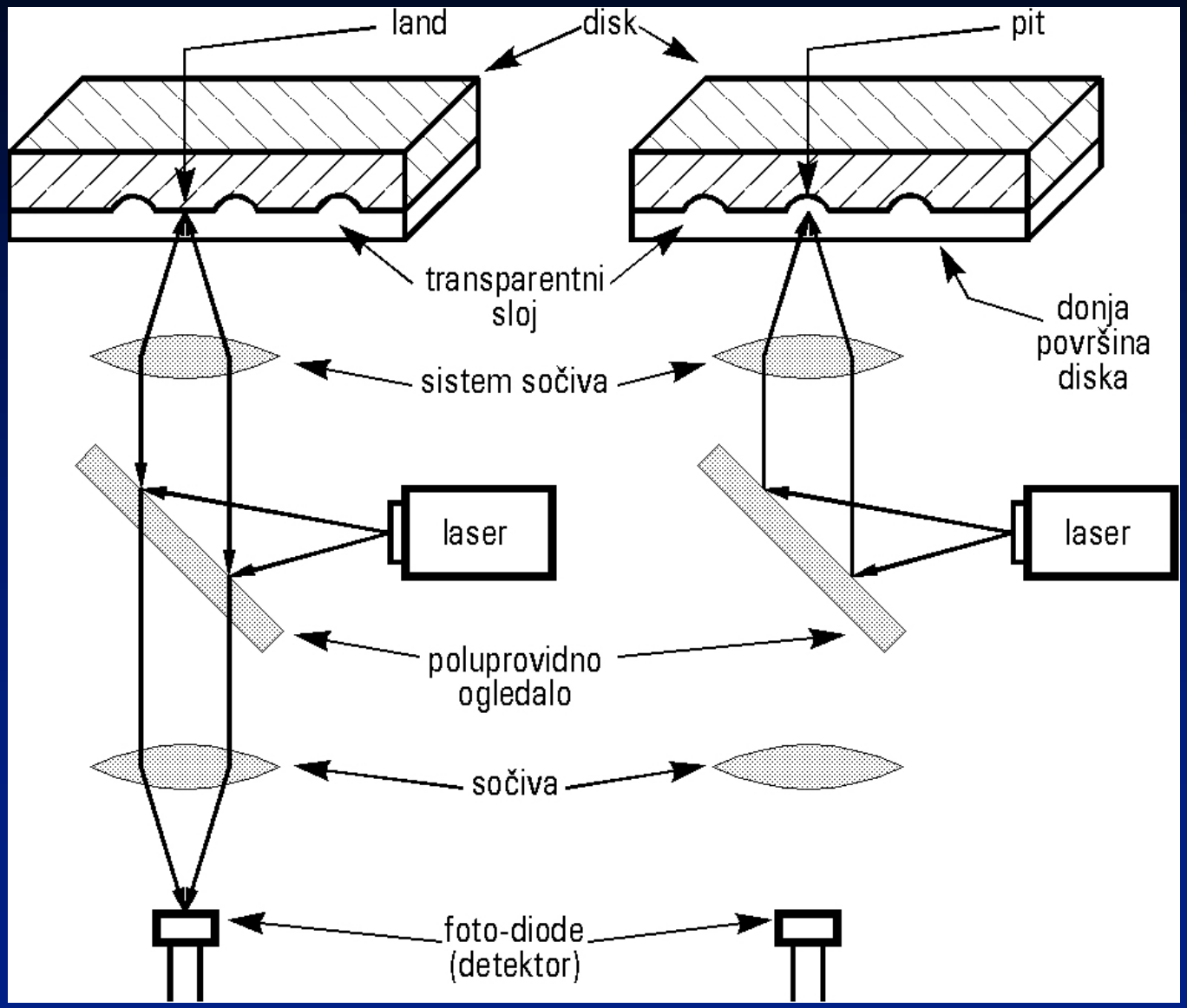
QIC-24 format sa devet pisti



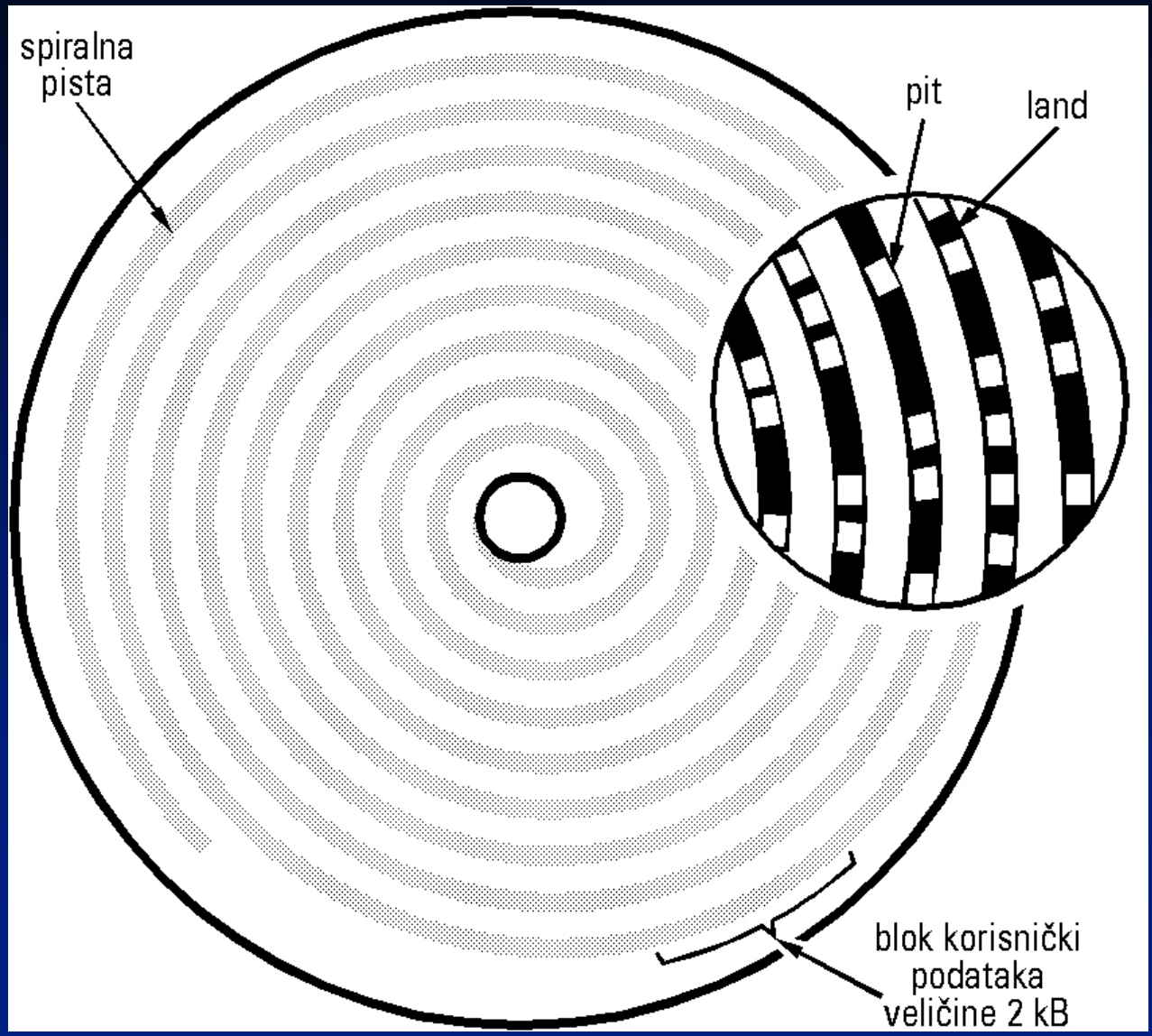
Formati medijuma za optički zapis



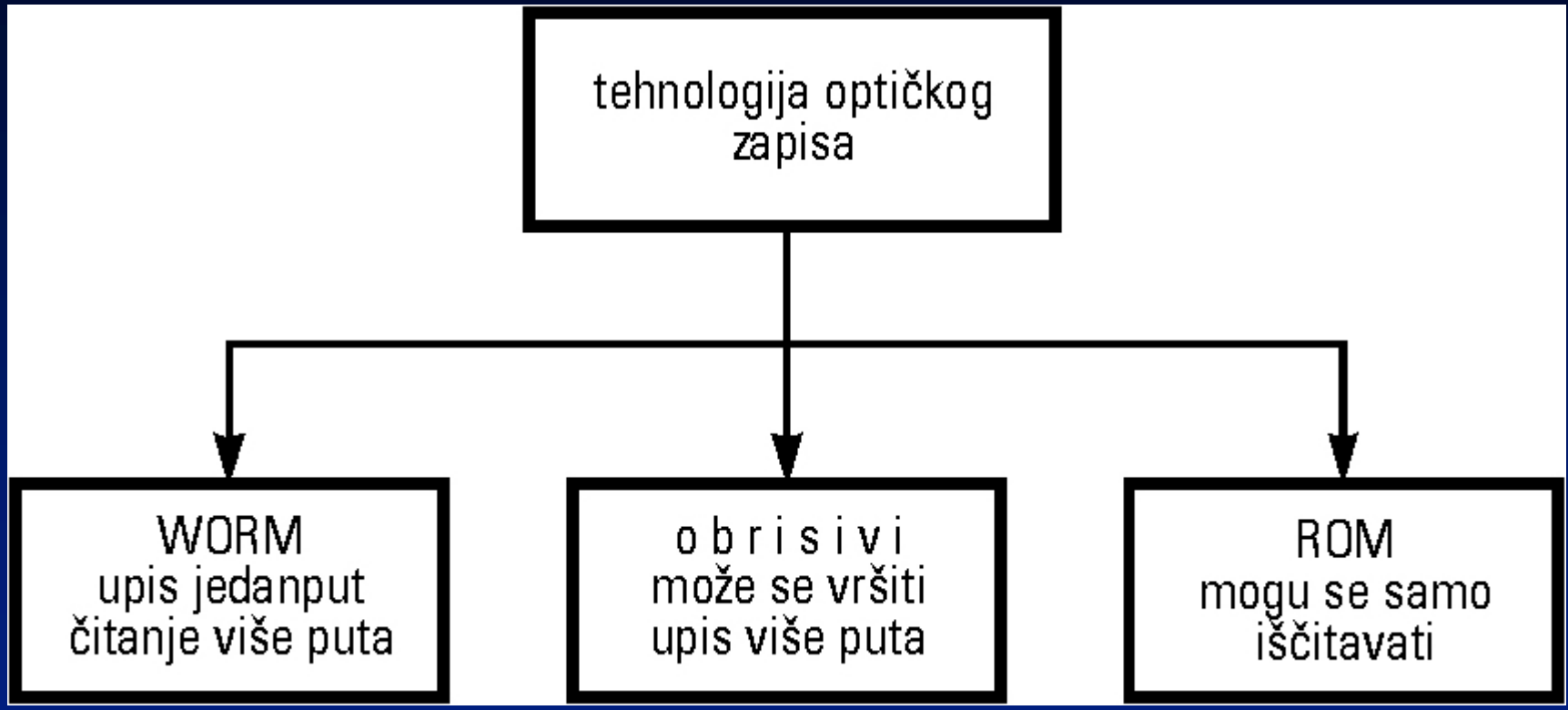
Princip rada CD-ROM-a



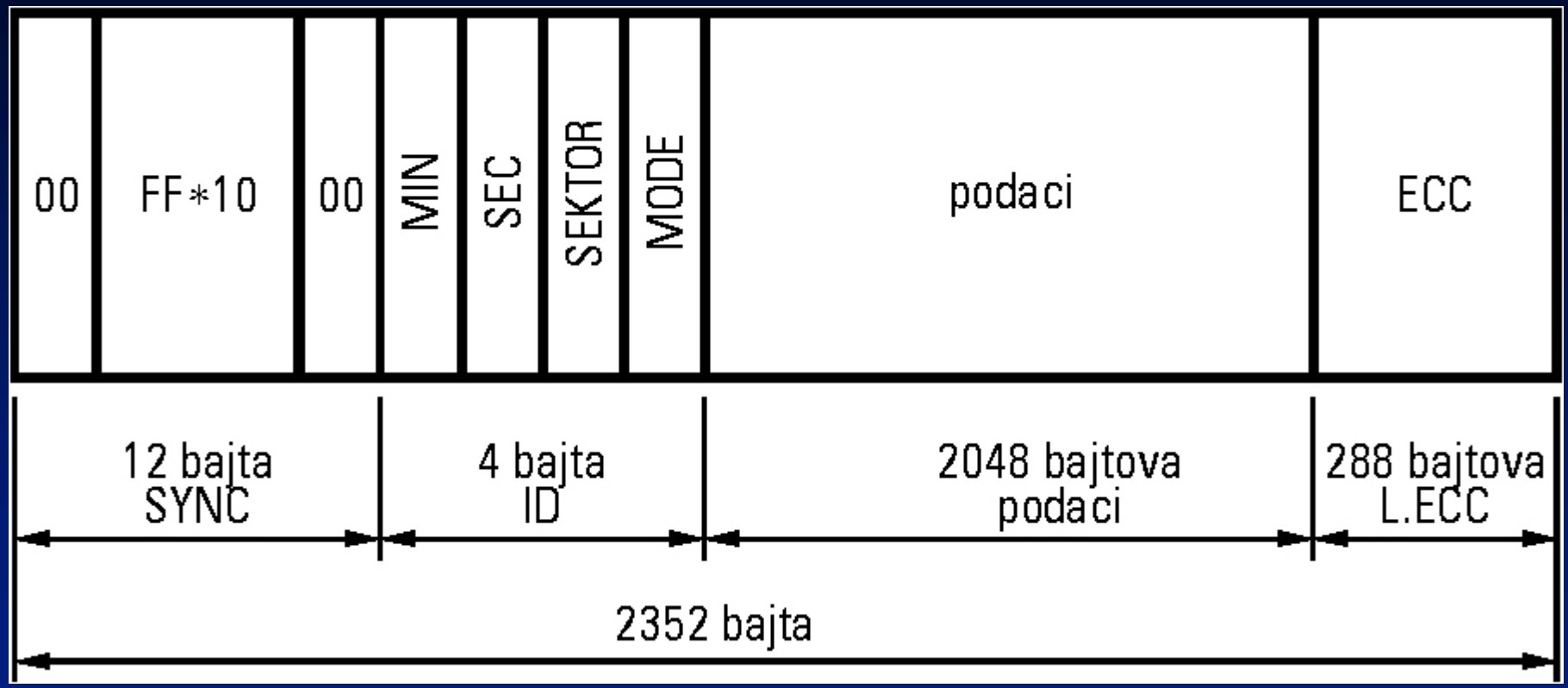
Format *CD-ROM*-a



Tehnologija zapisa



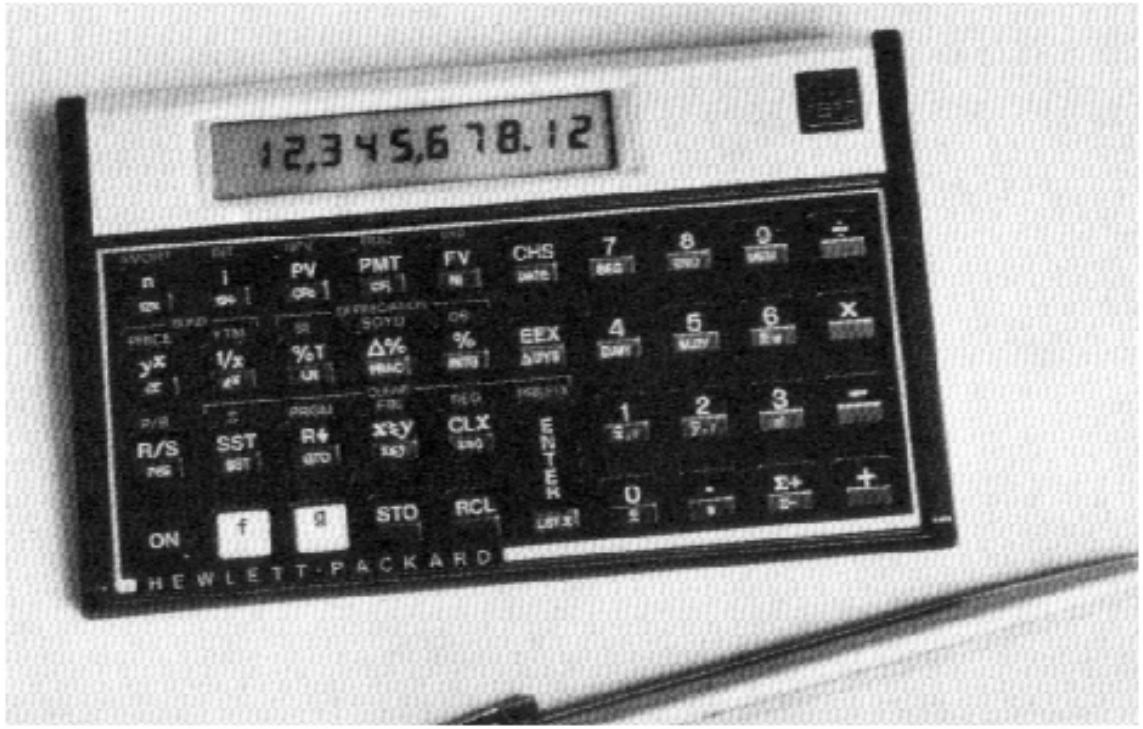
Format bloka podataka kod *CD-ROM*-a



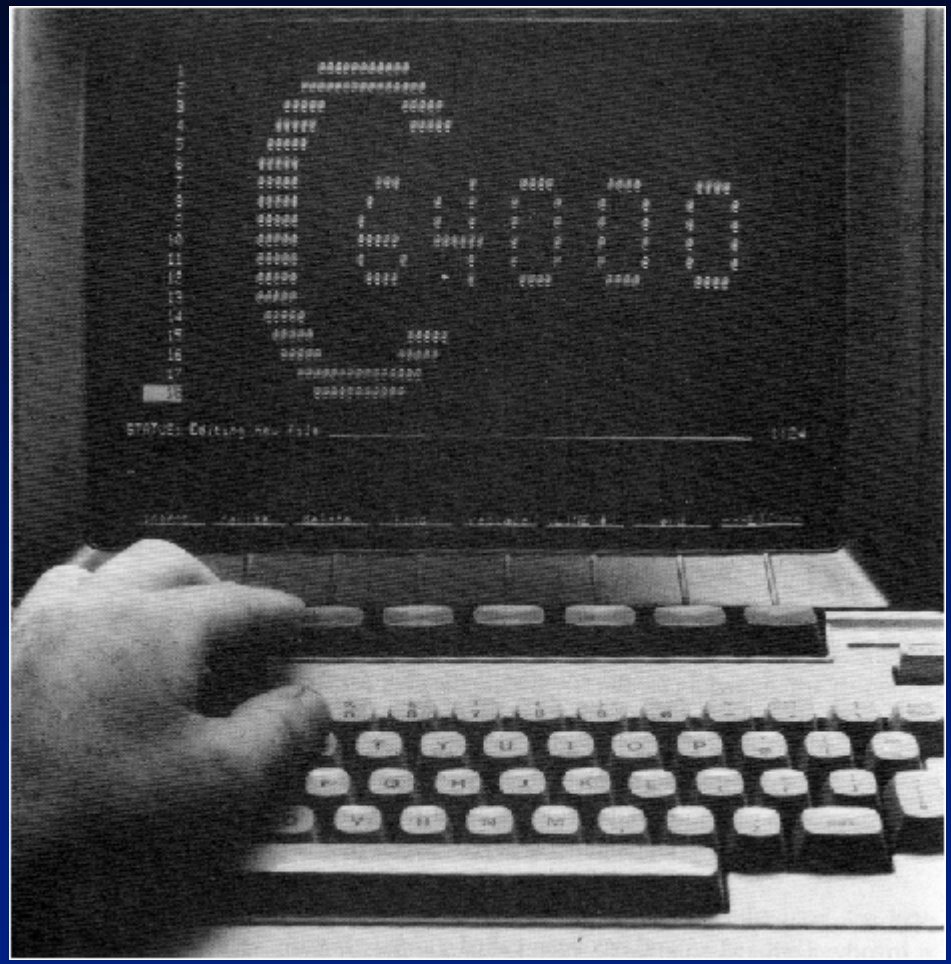
ULAZNO-IZLAZNI UREDJAJI

Uredjaji kod kojih tasteri na tastaturi imaju jednu funkciju

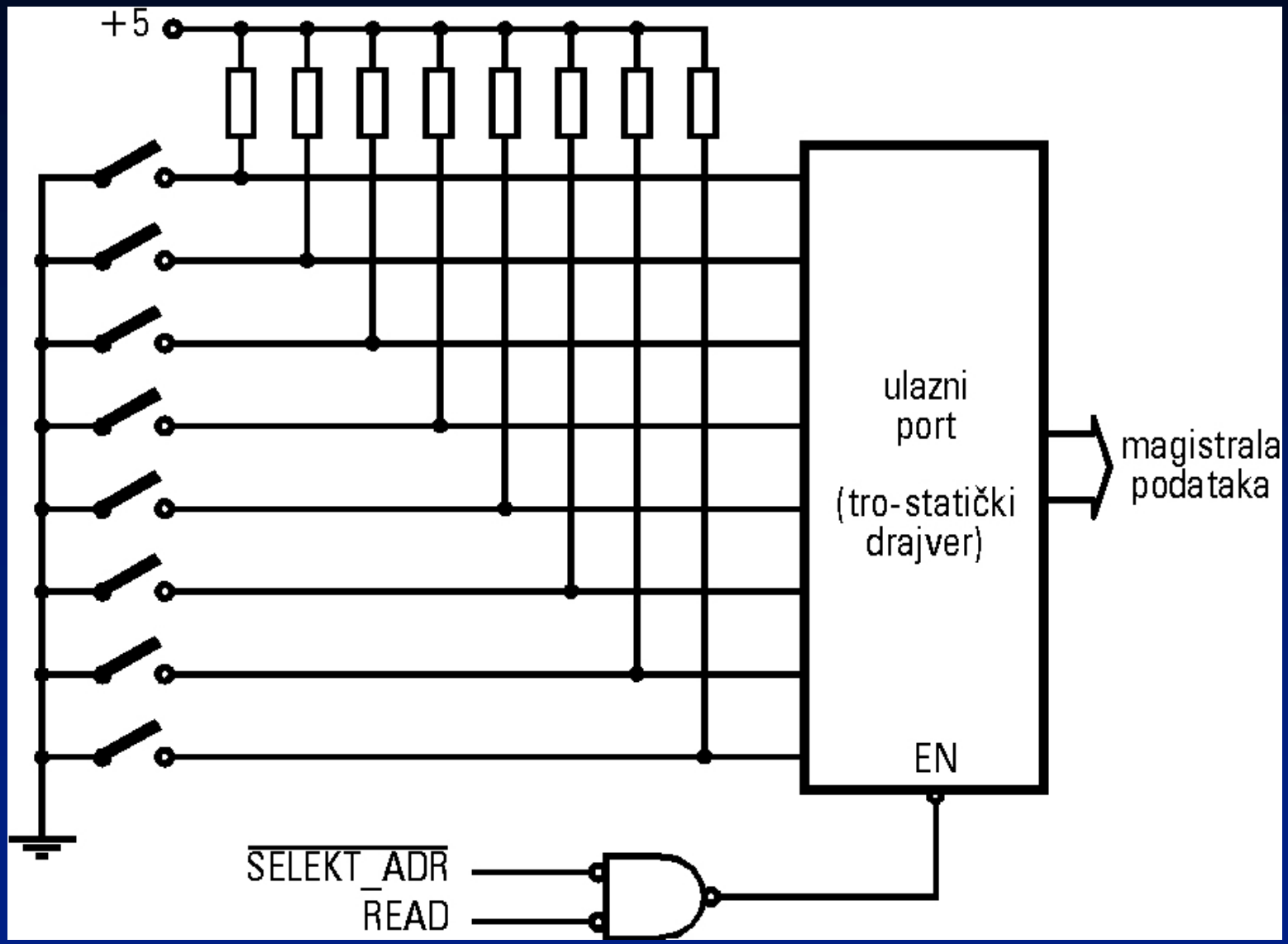
Tastatura koja koristi dva načina rada za promenu funkcije dirki



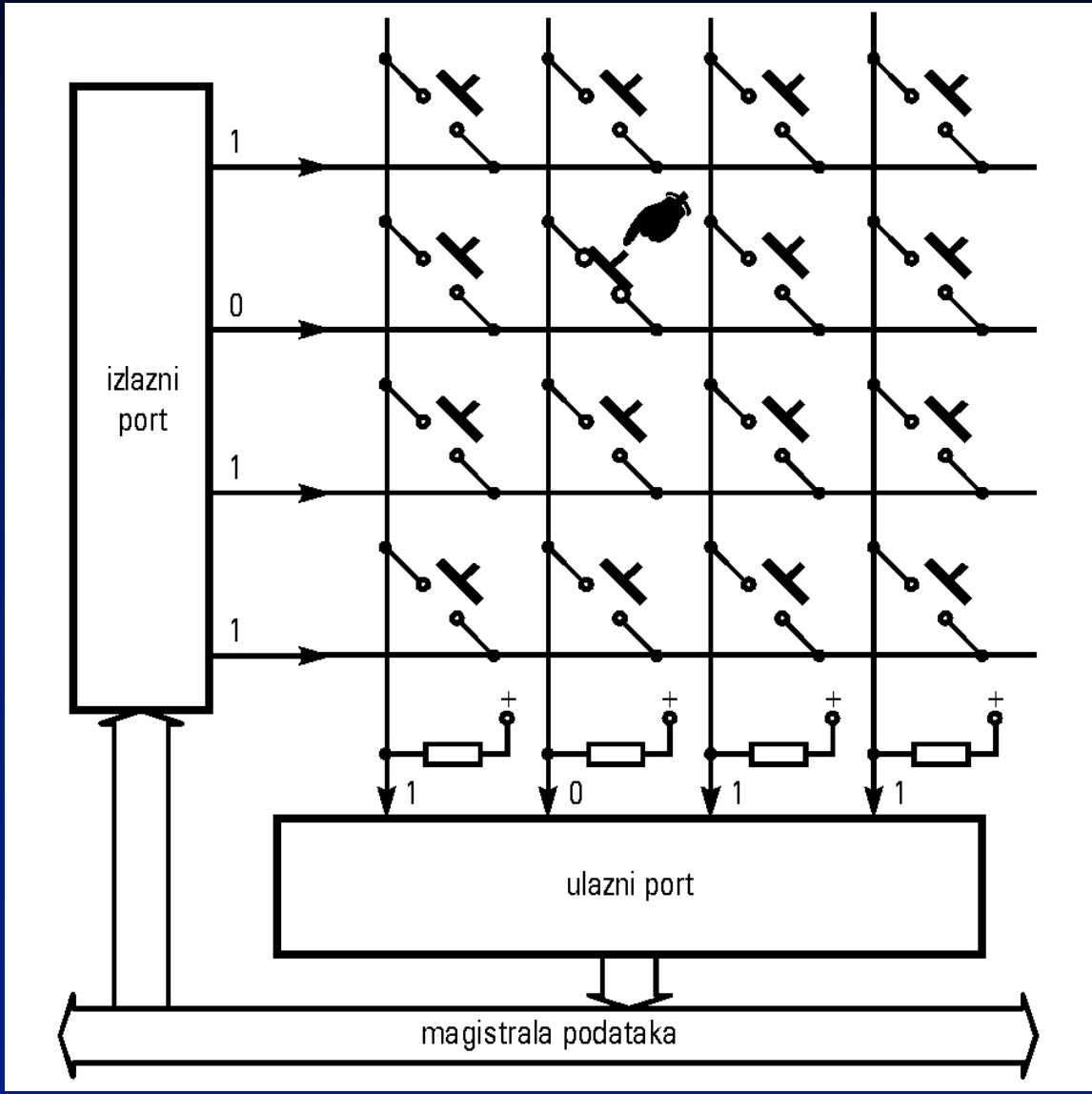
Tastatura koja koristi *soft* dirke



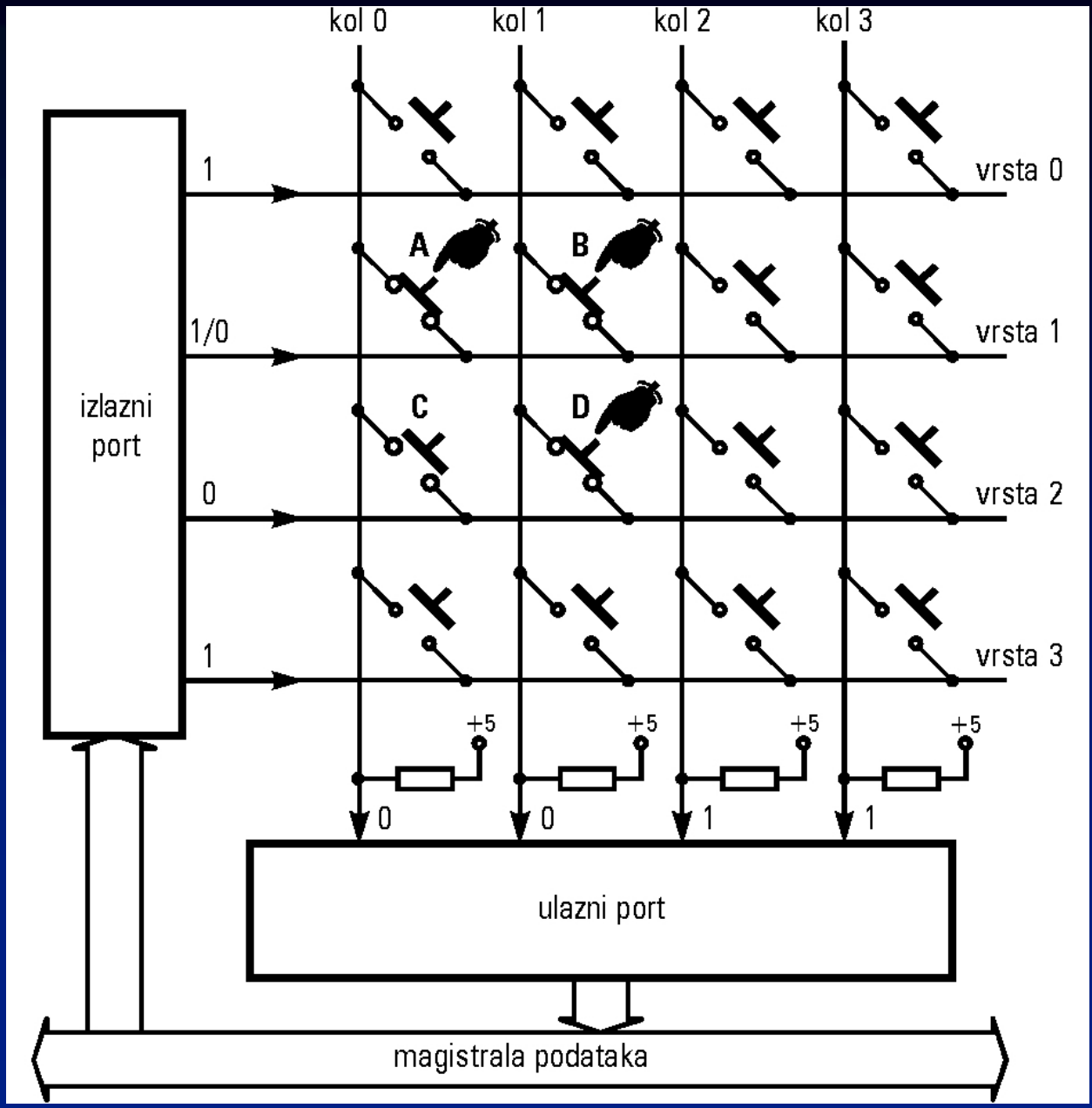
Ne-multipleksirani interfejs



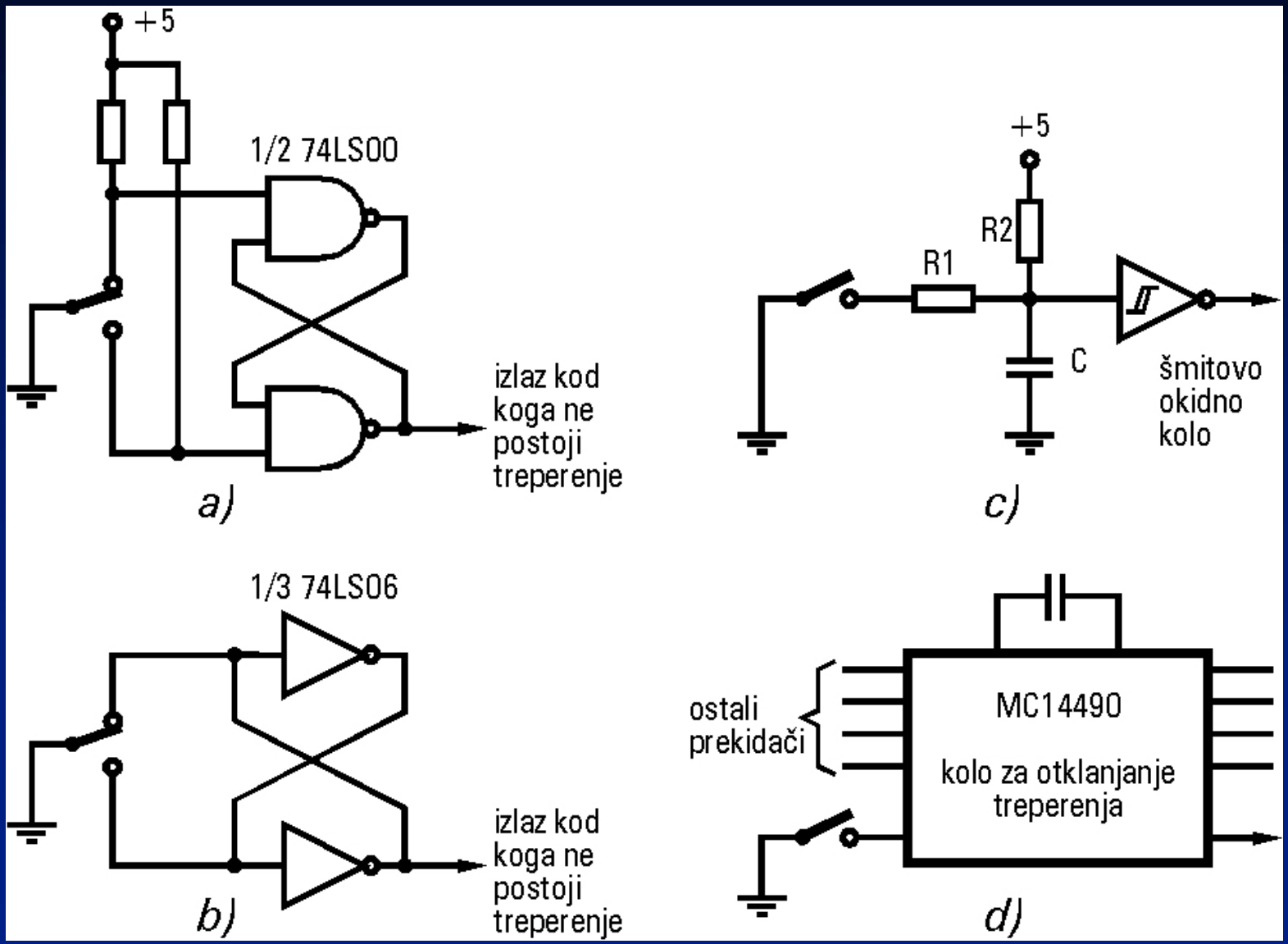
Prihvatanje podataka sa tastature preko multipleksiranog interfejsa



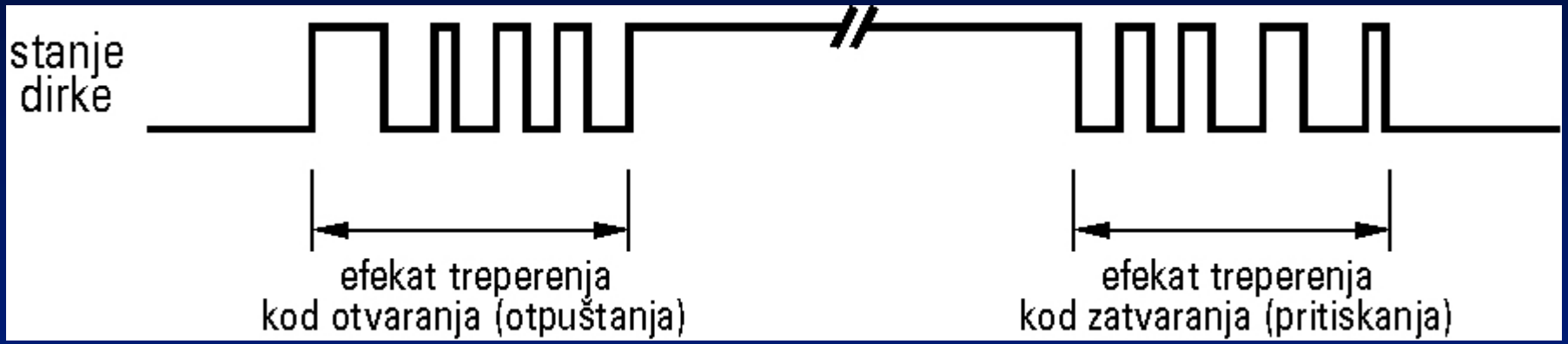
Istovremeno pritiskanje tri dirki



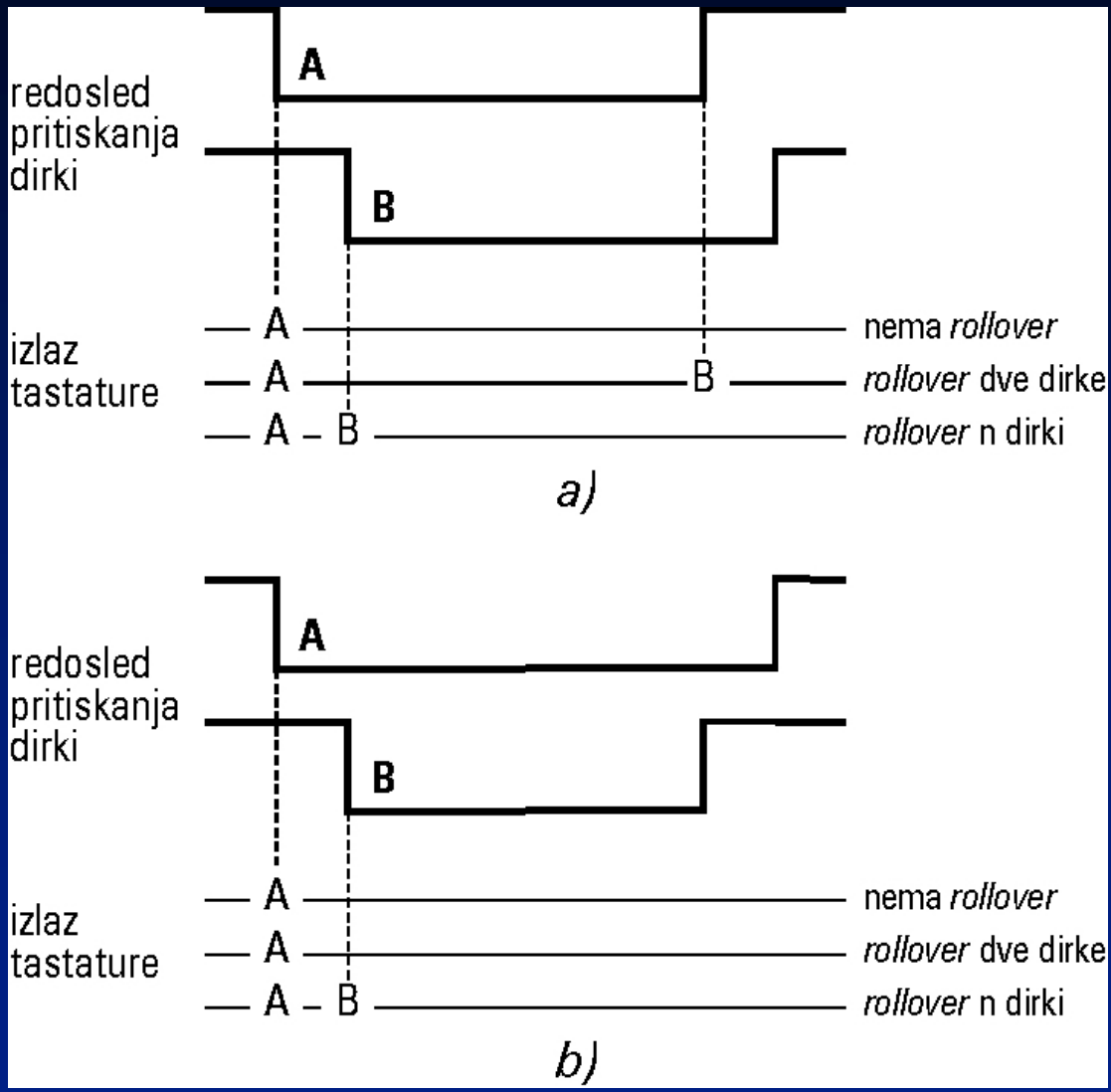
Hardverski metodi za eliminisanje uticaja efekta treperenja dirki



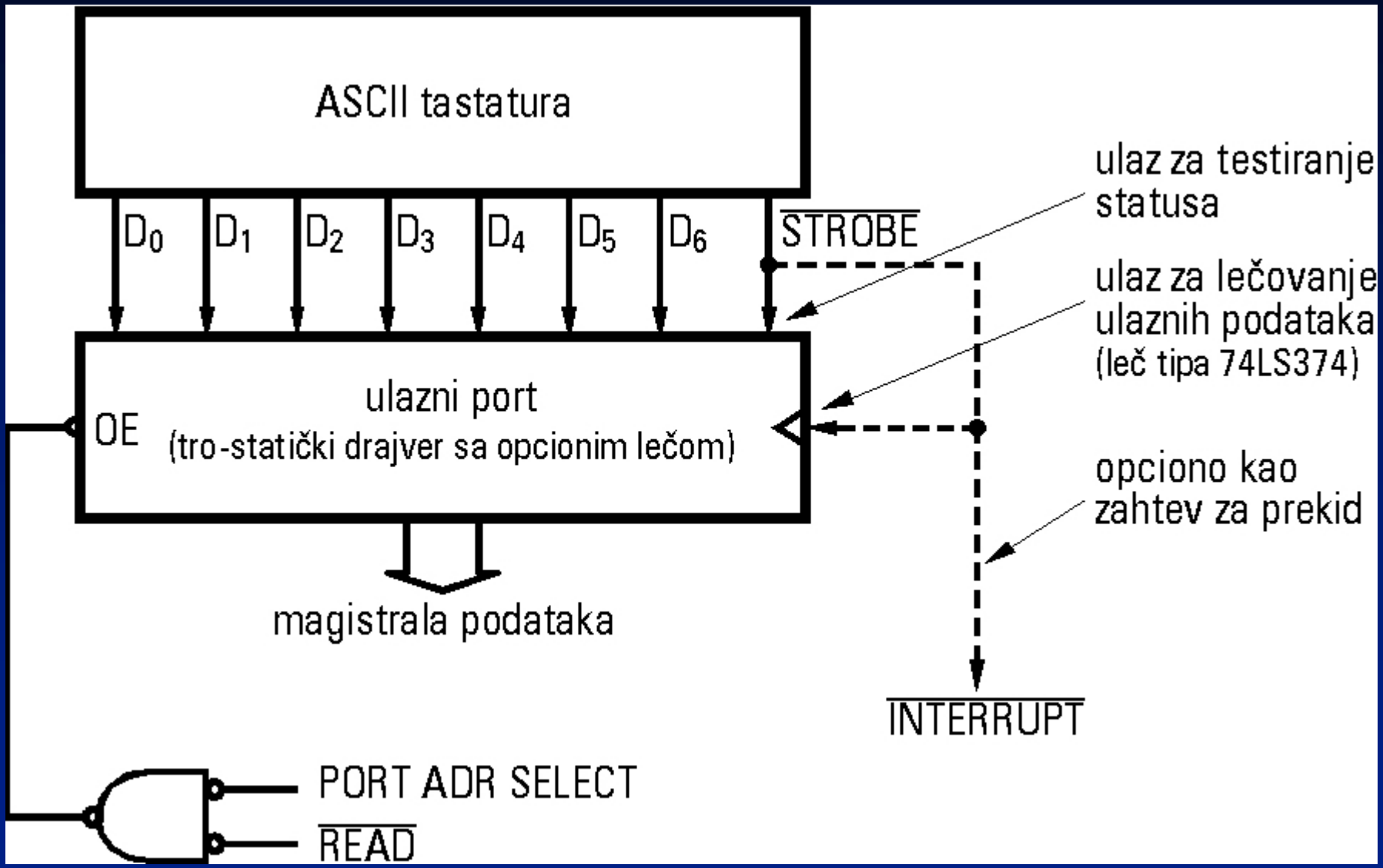
Efekat treperenja dirki



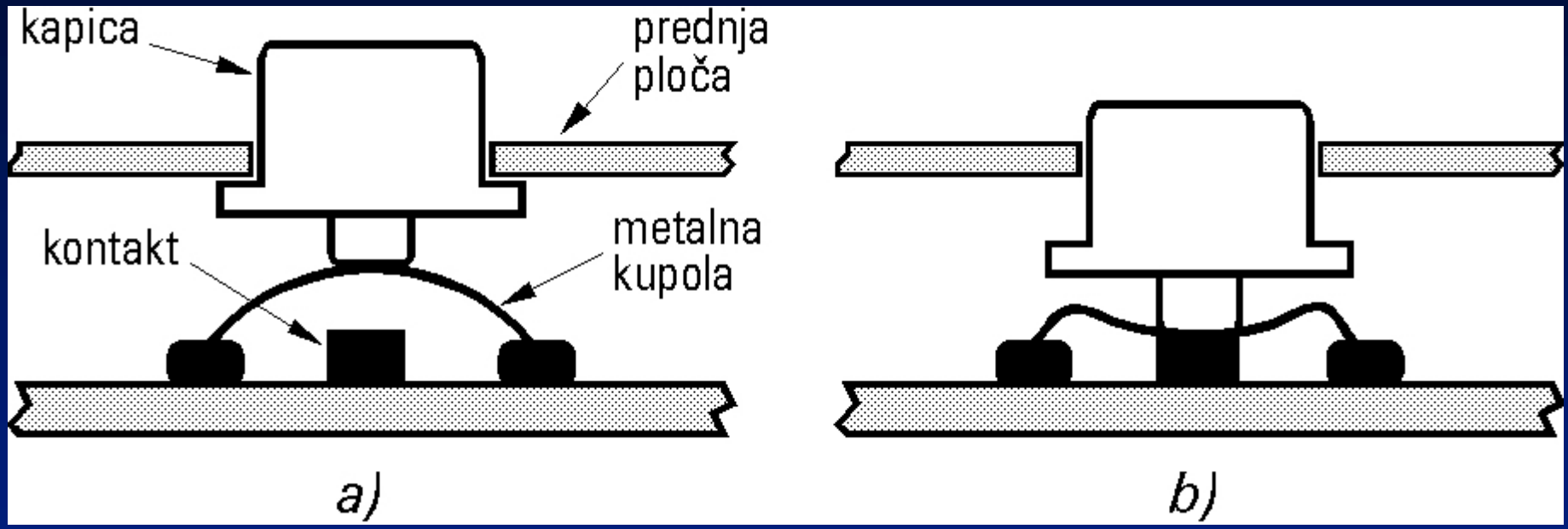
Vremenski dijagrami tipični za *rollover*



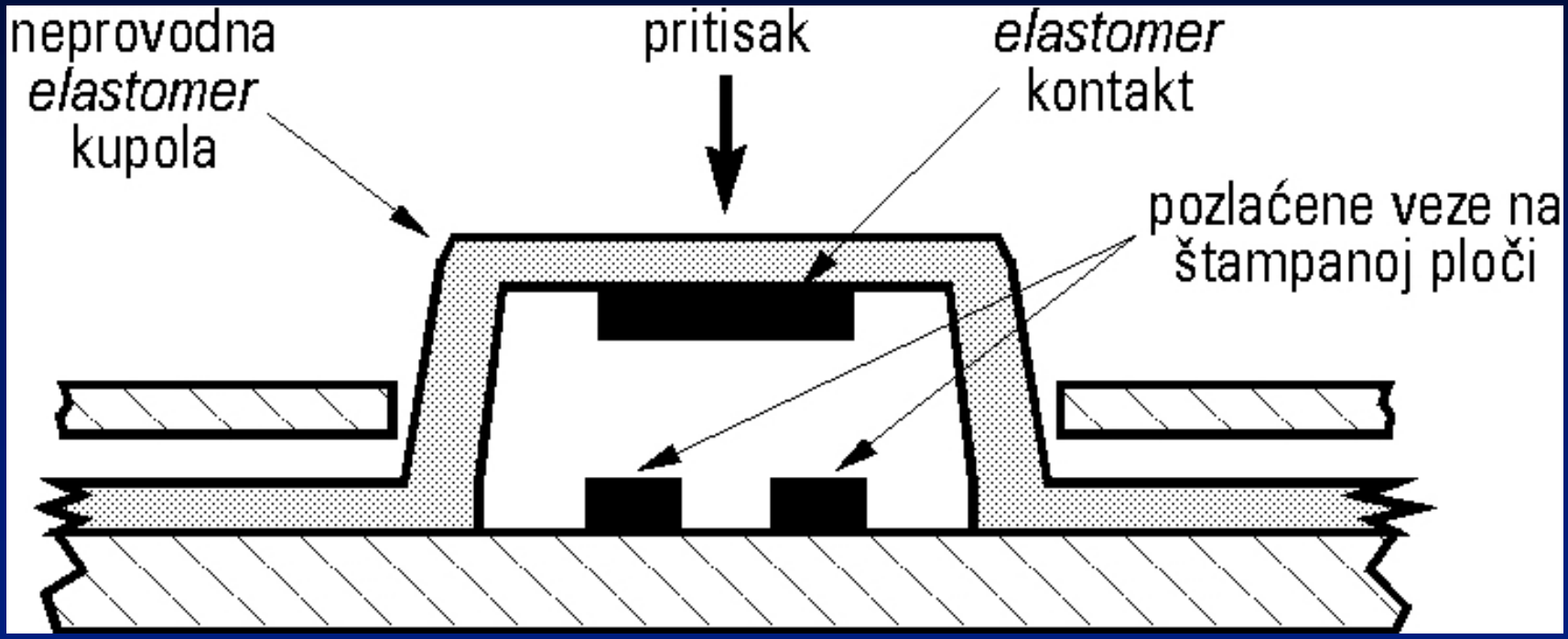
Sprezanje tastature i računara preko ulaznog porta



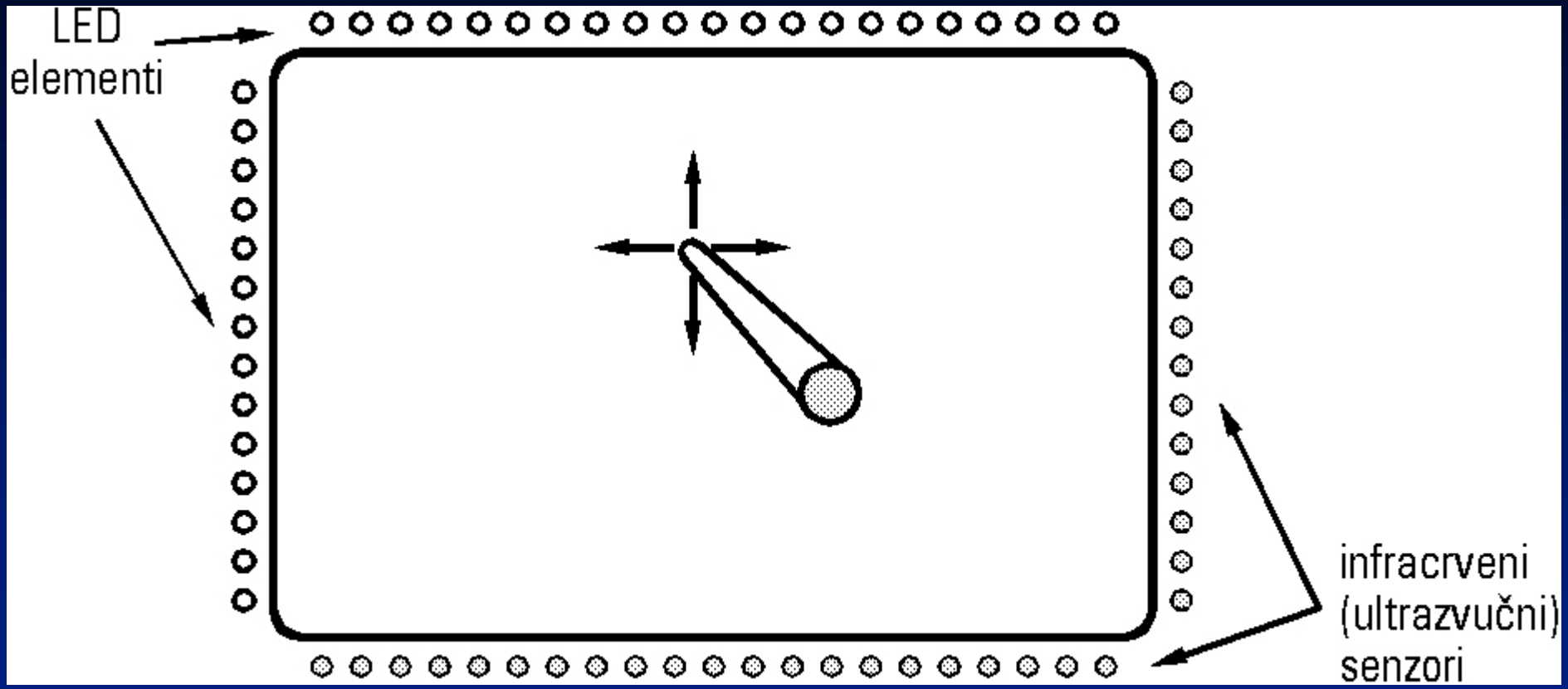
Dirka sa metalnom kupolom



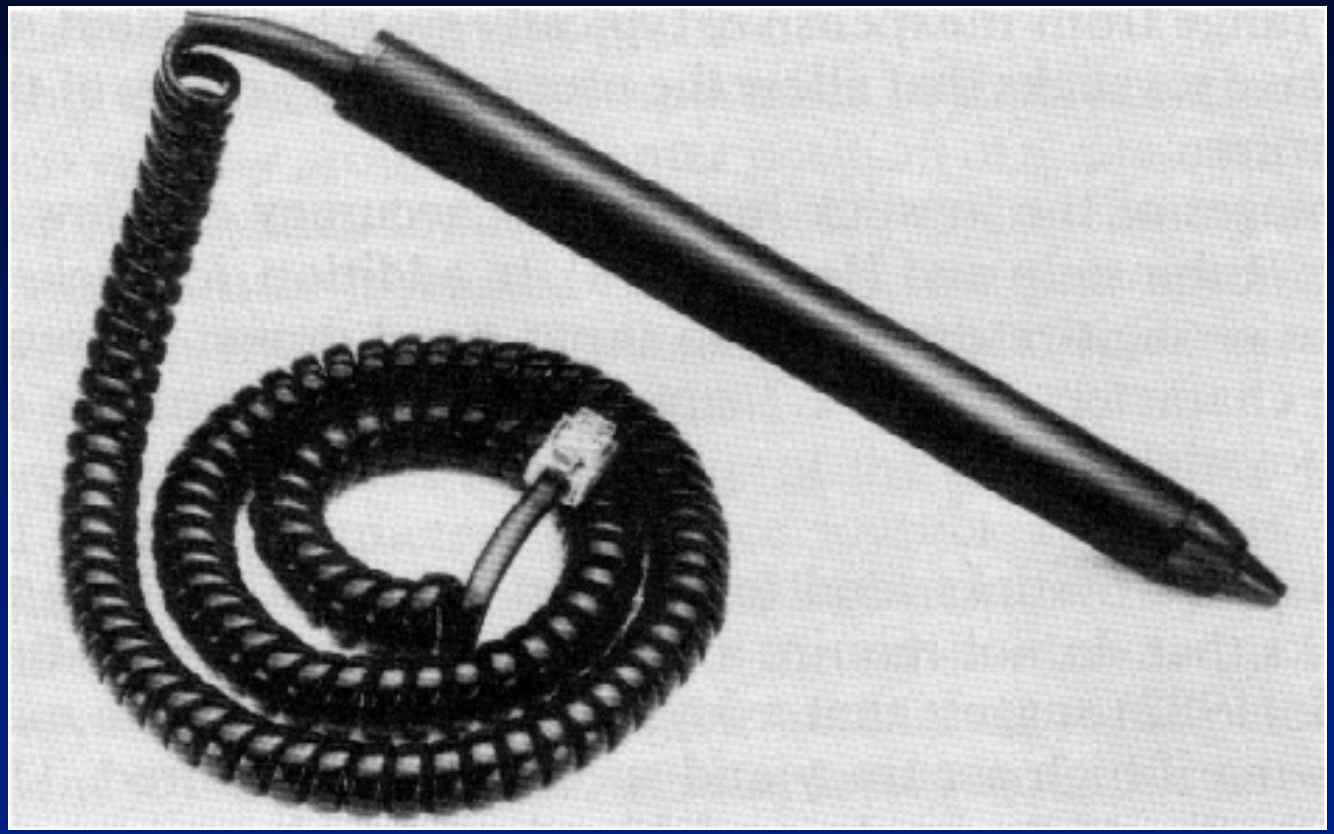
Presek dirke realizovane od provodne gume



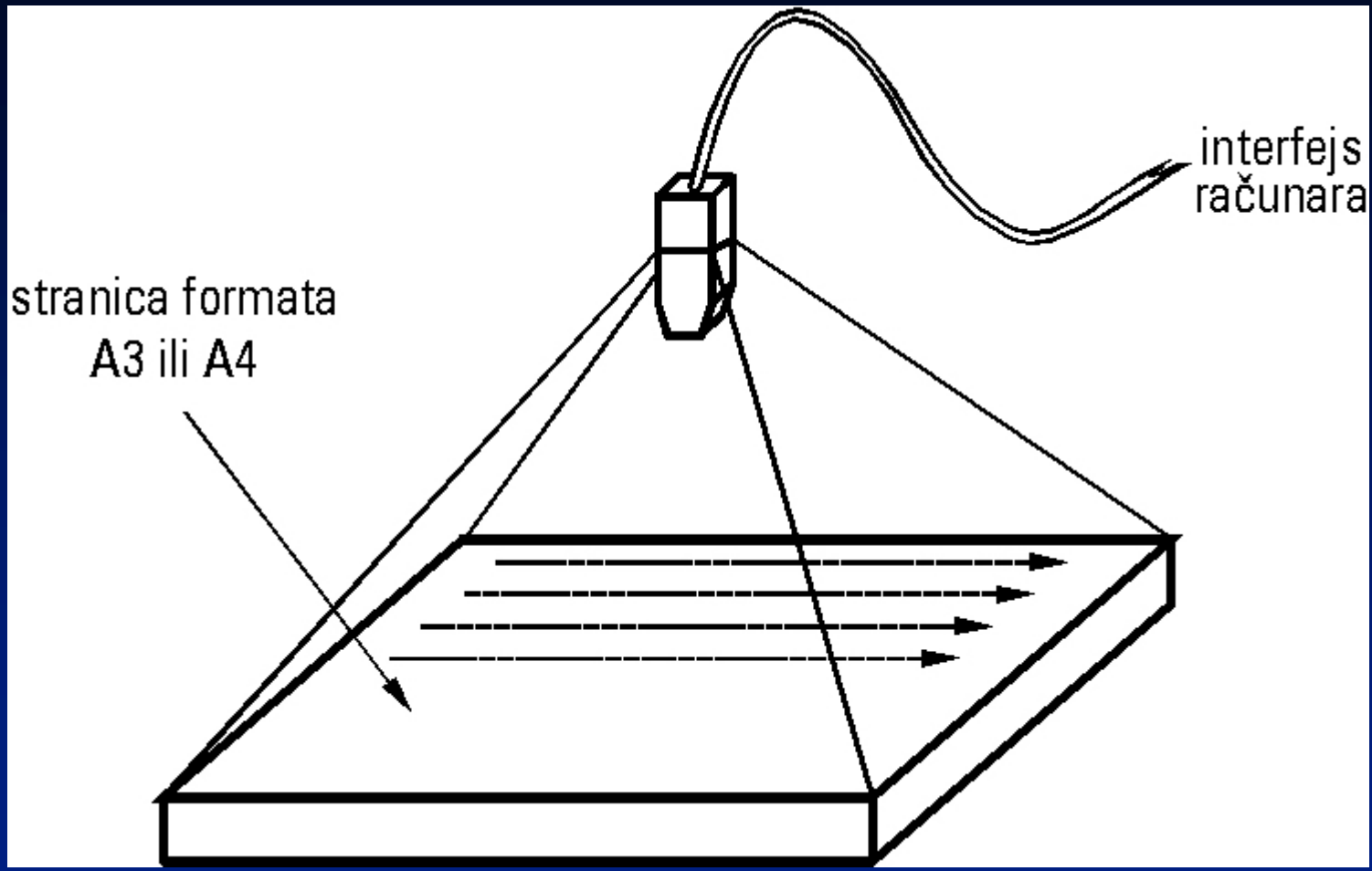
Displej osetljiv na dodir sa LED elementima



Izgled svetlećeg pera

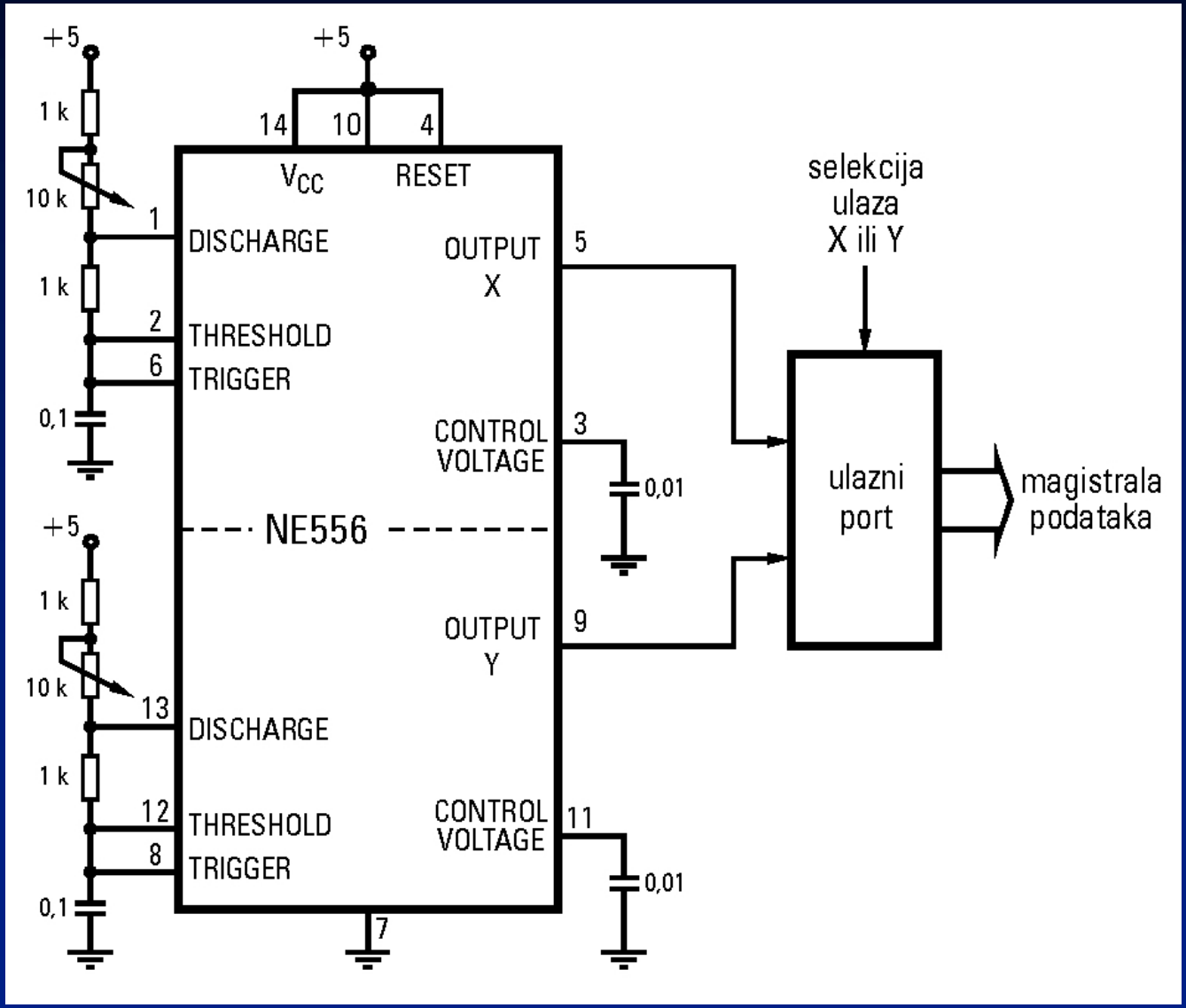


Fizički izgled jednog X-Y skenera



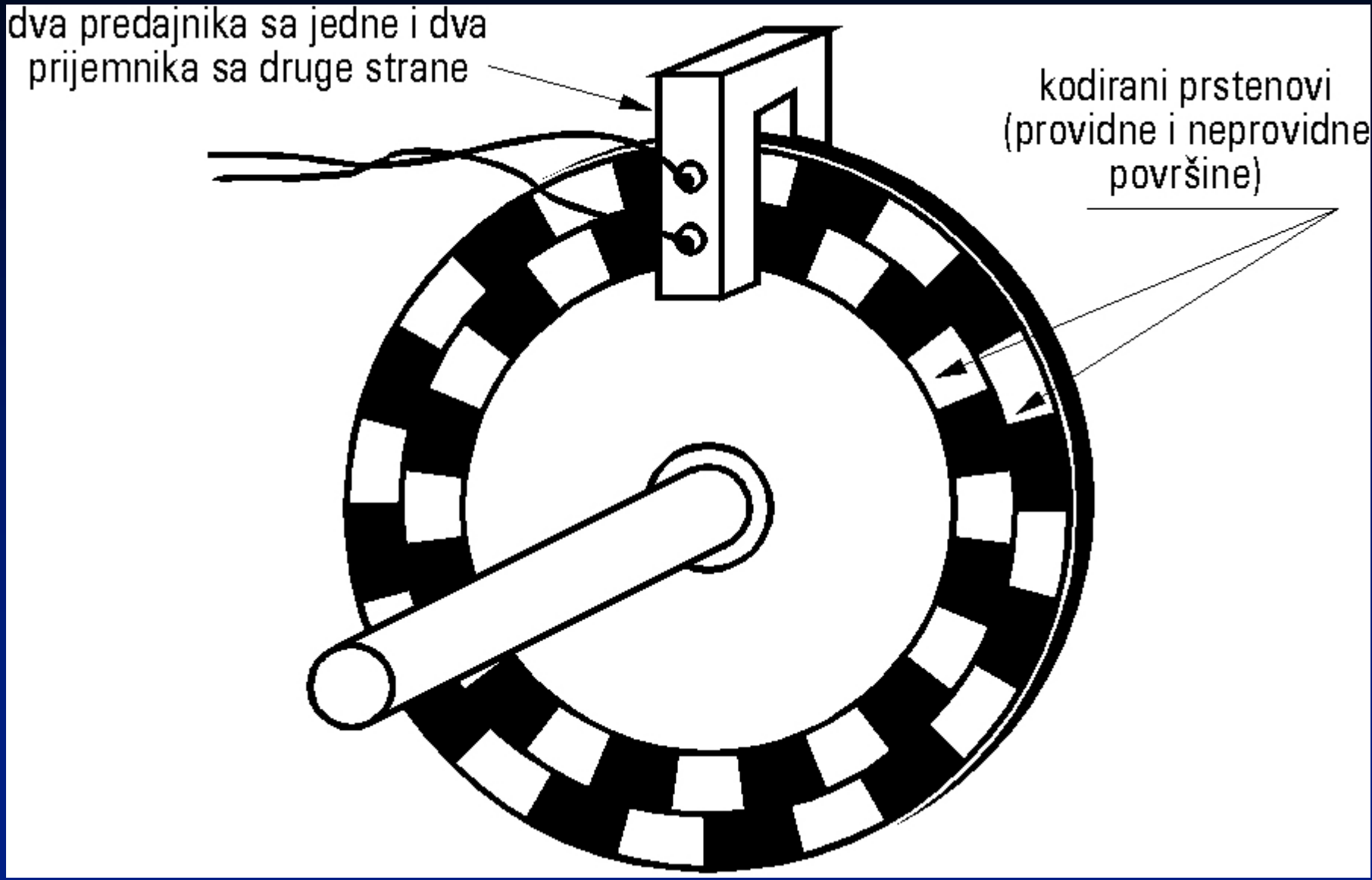
Fizički izgled džoistika

Sprezanje džoistika na računar preko dvostrukog tajmera

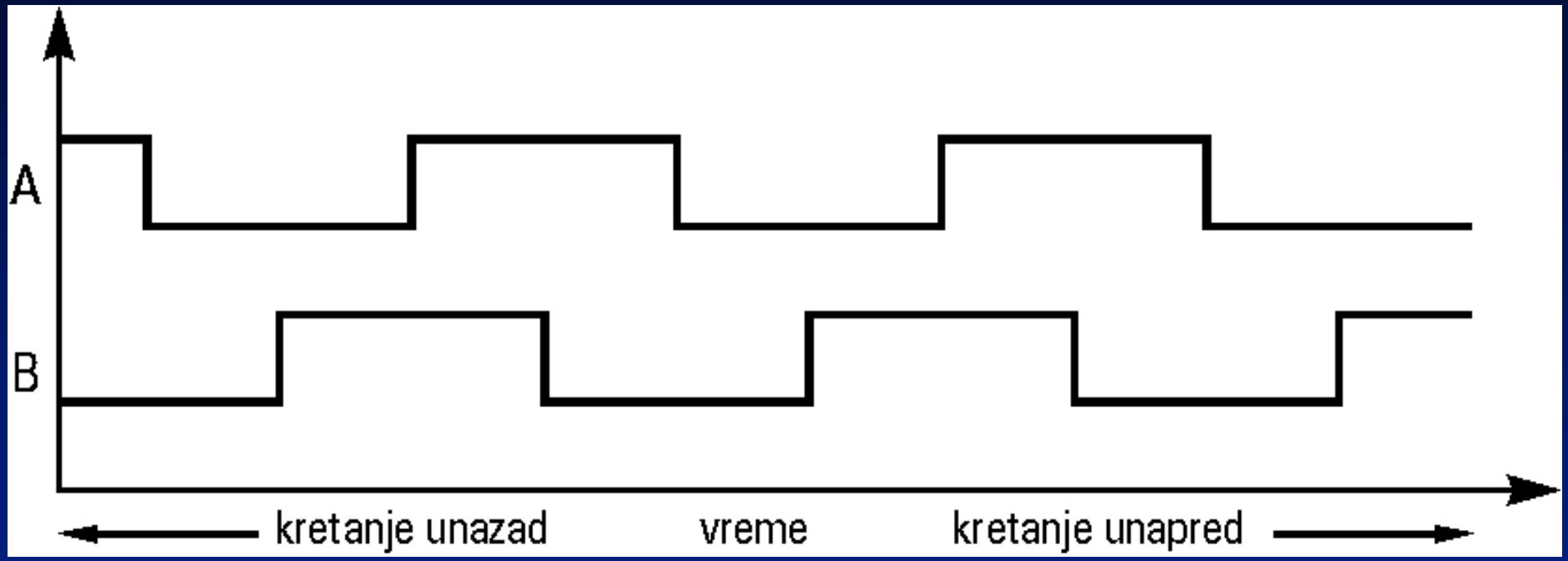


Fizički izgled trekbola

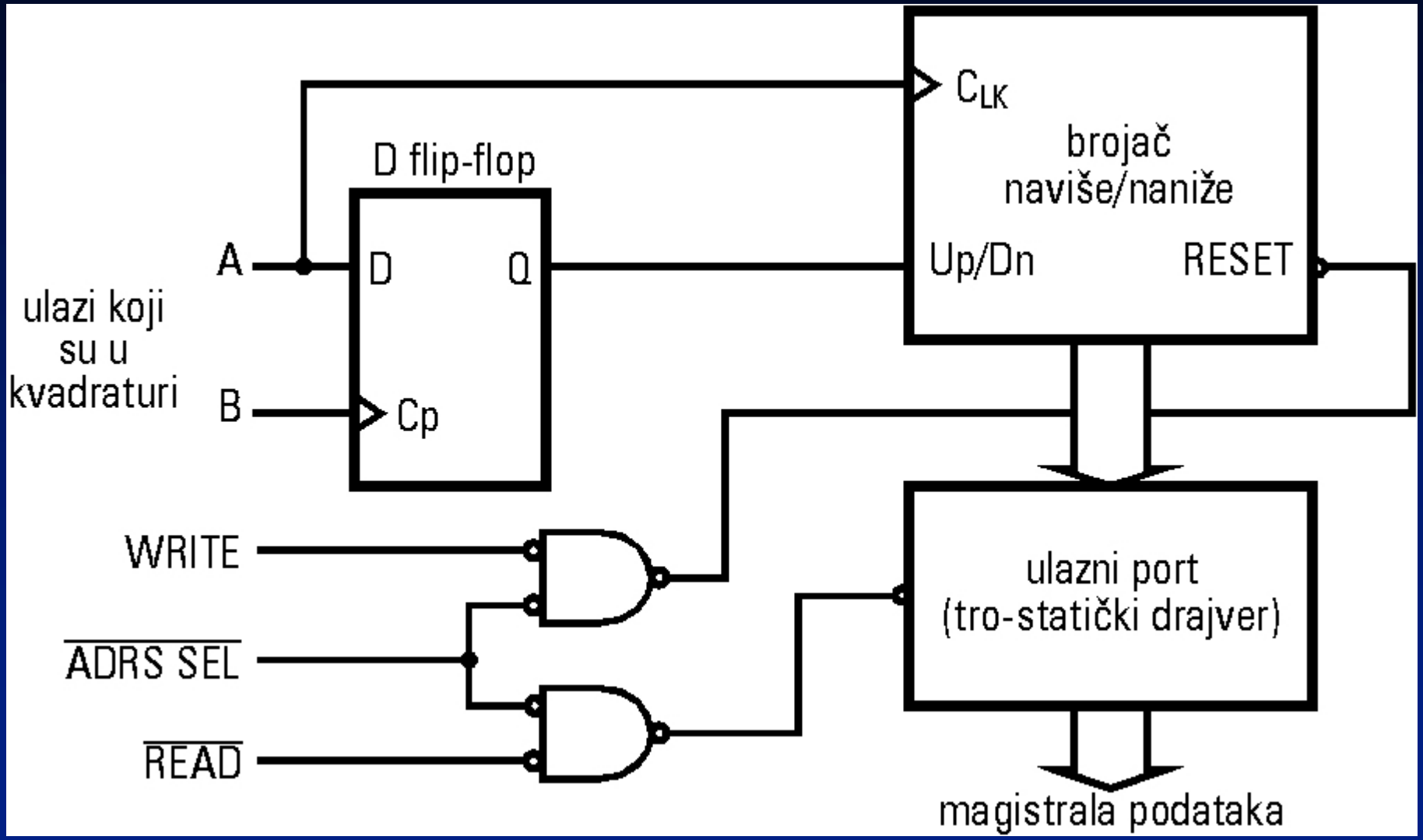
Optički disk od trekbola



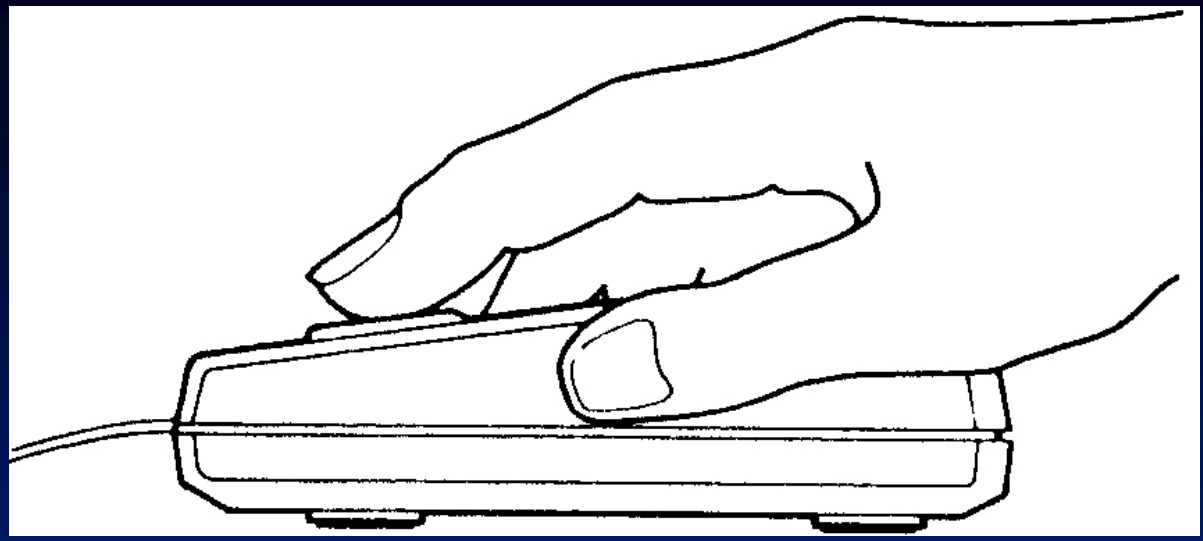
Signali na izlazu trekbola

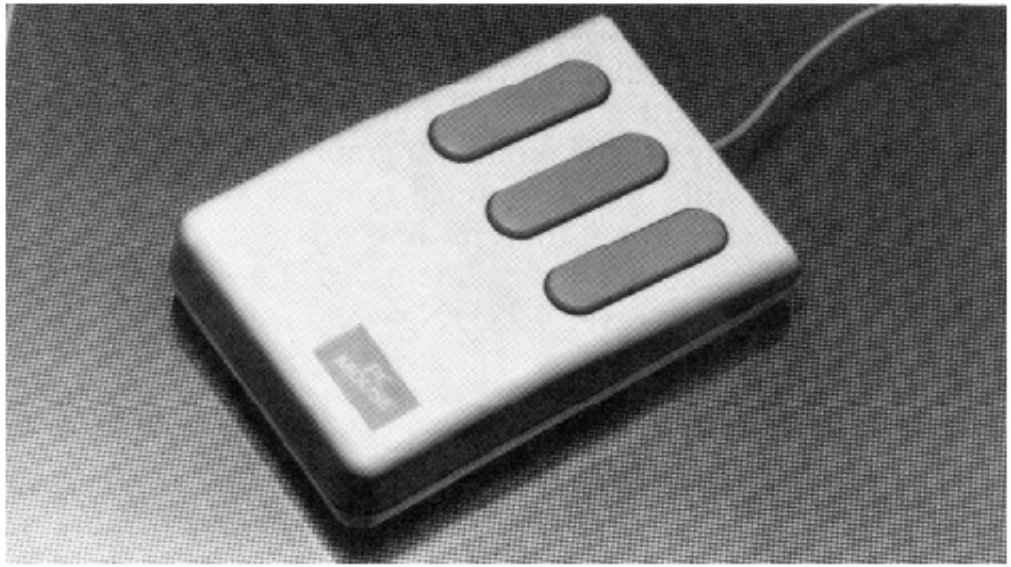


Hardverski kvadraturni dekoder

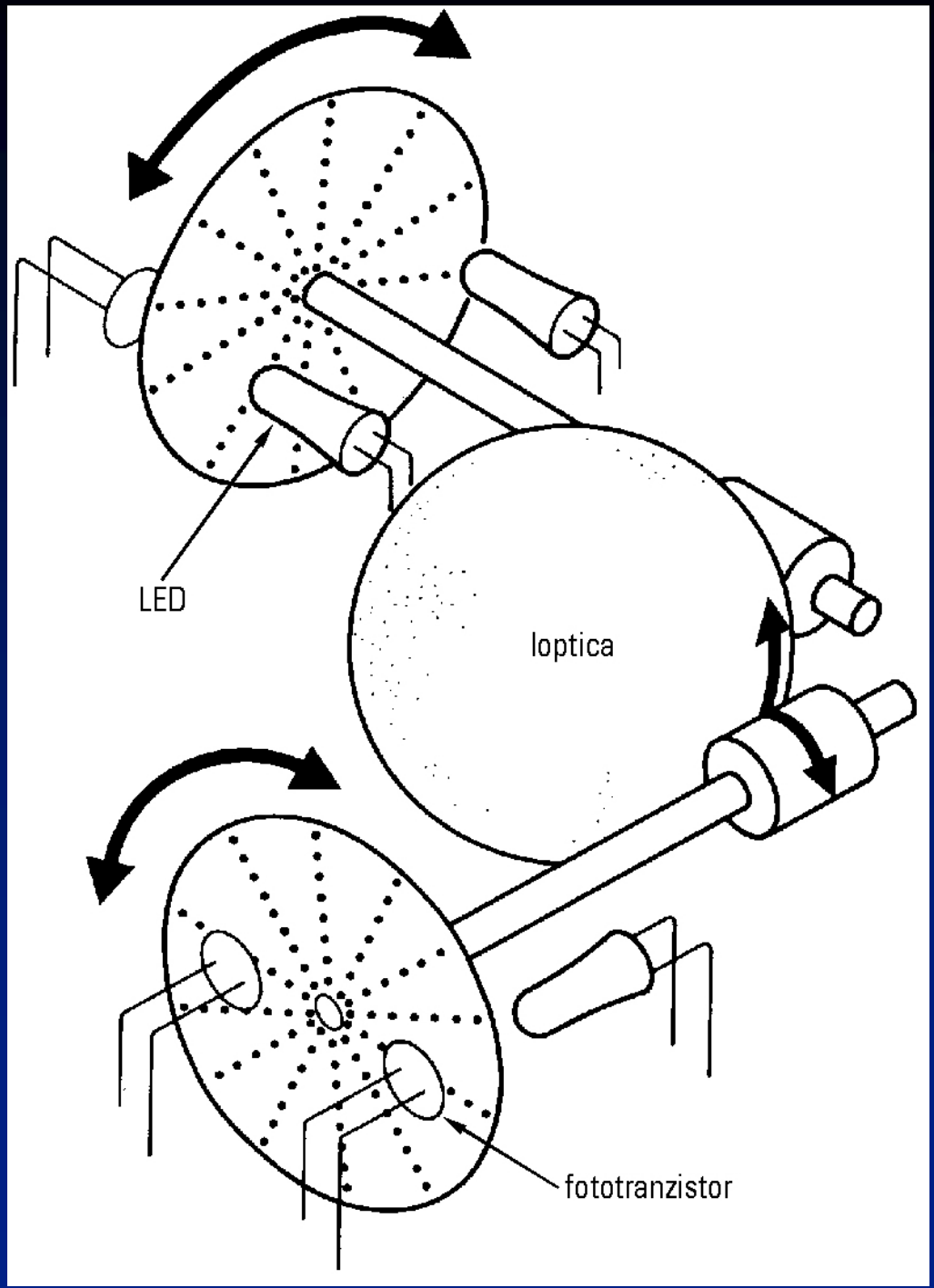


Miš

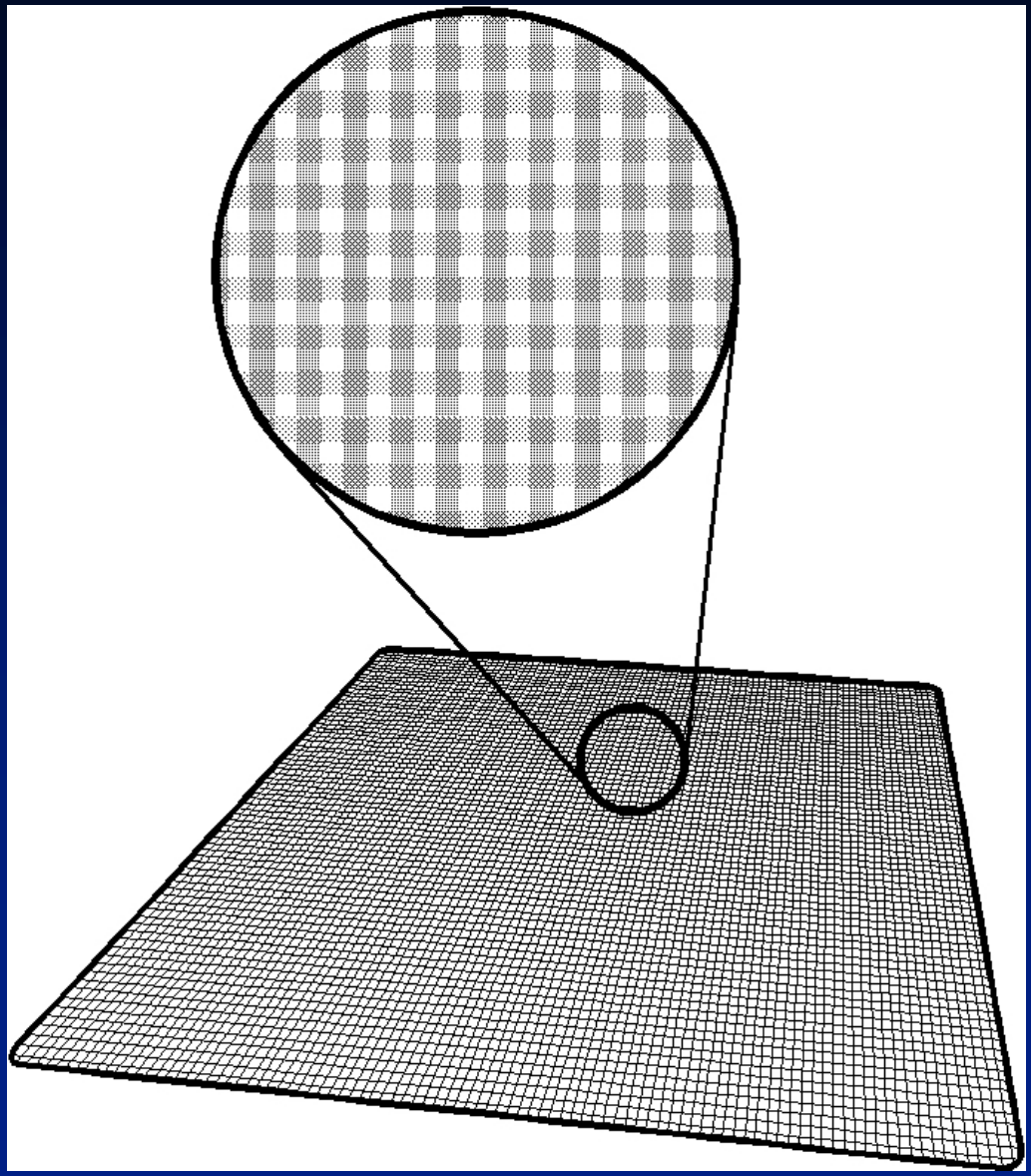




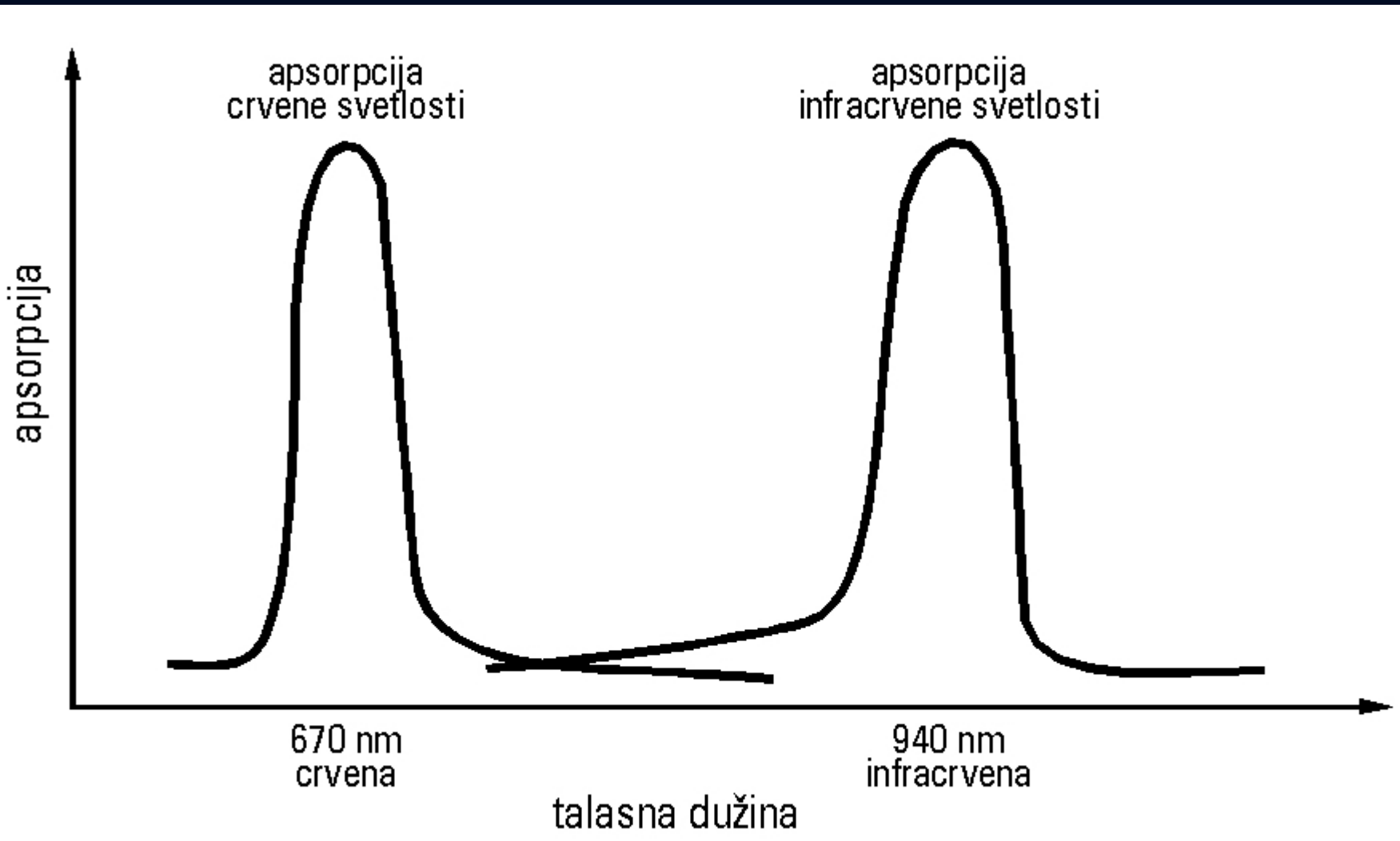
Elektromehanički miš



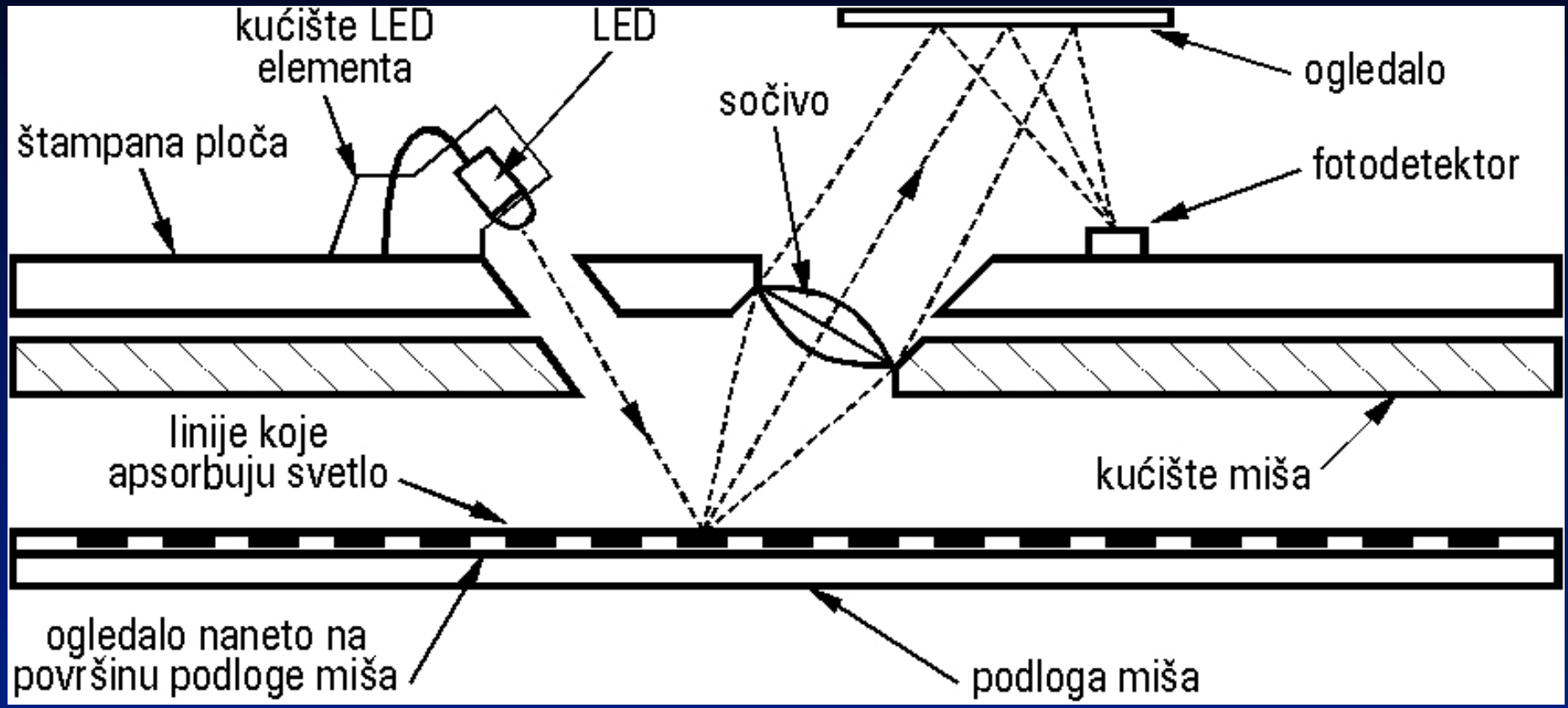
Podloga za optički miš



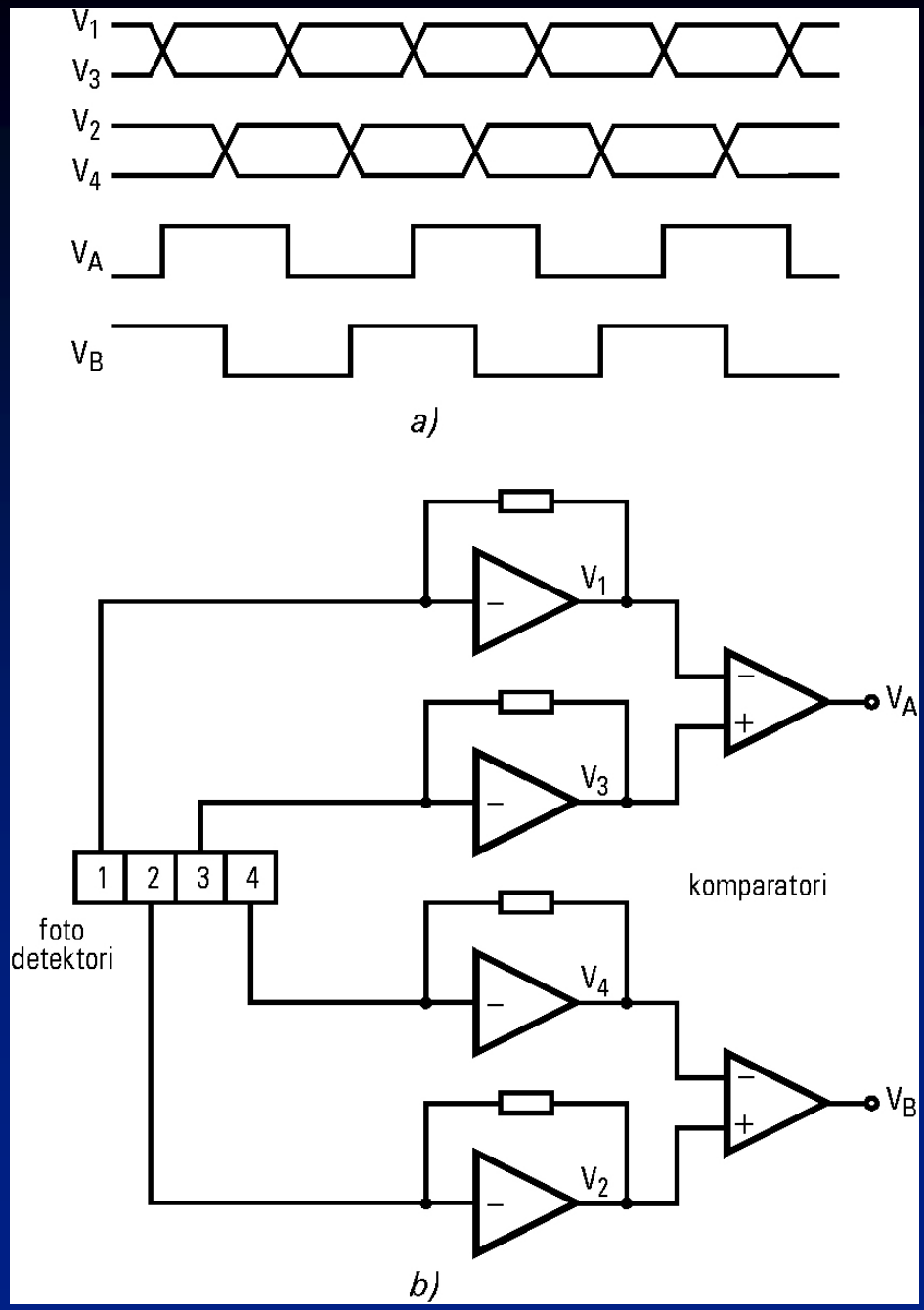
Apsorbcija svetlosti kod optičkog miša



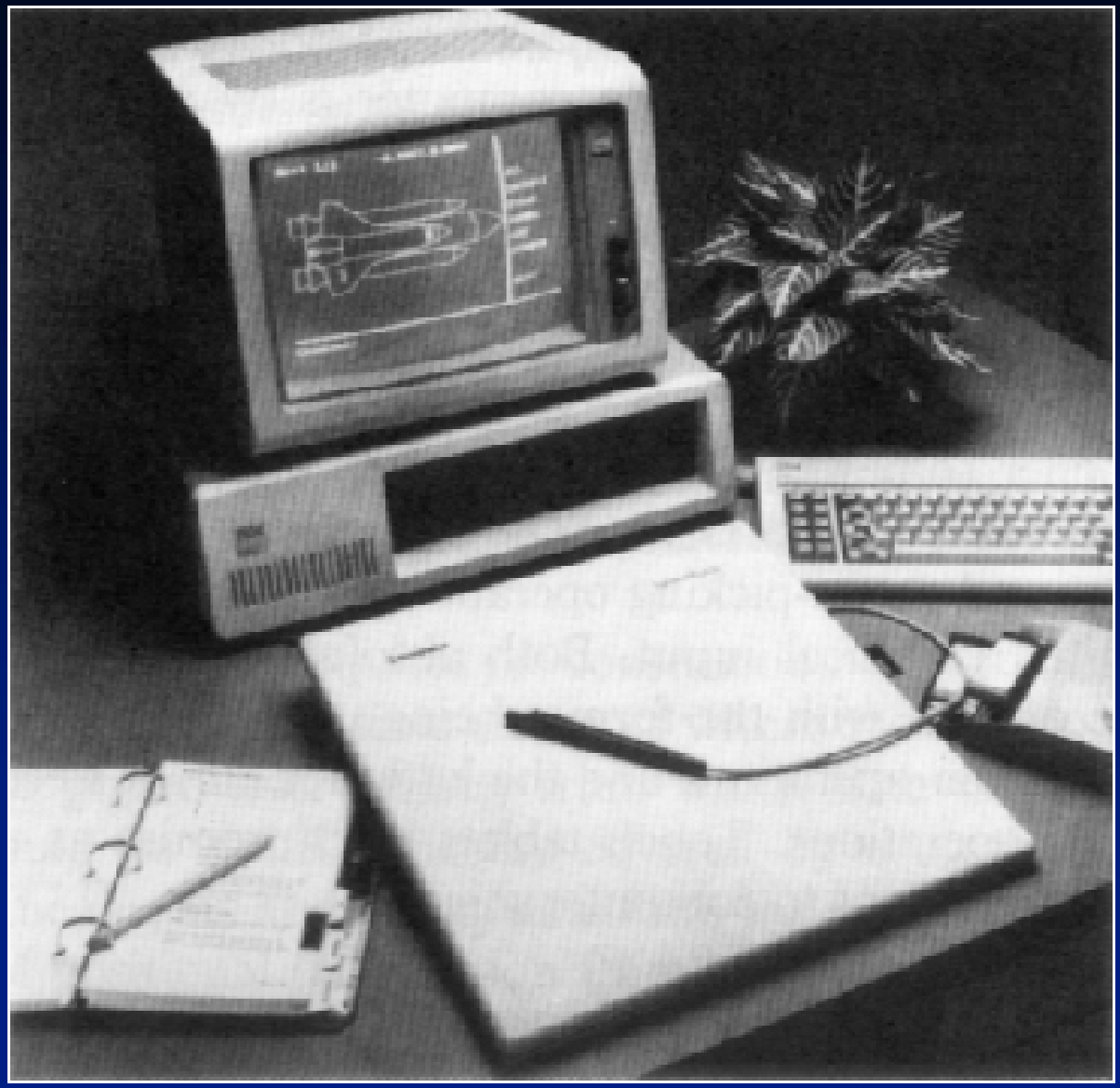
Konstruktivni detalji optičkog miša



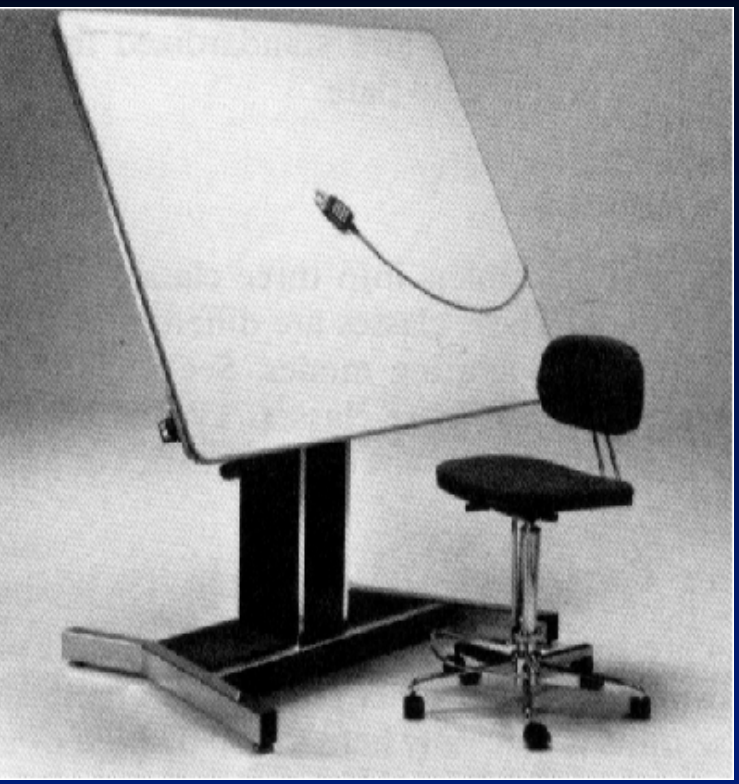
Interfejs elektronika optičkog miša

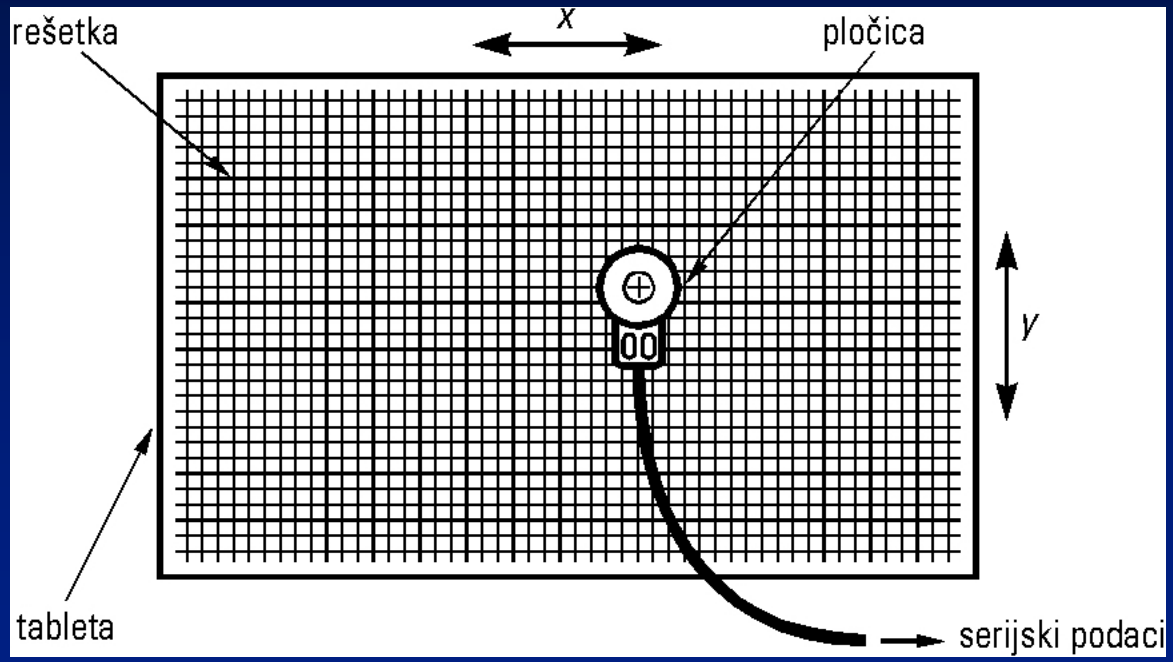


Digitajzer u obliku olovke

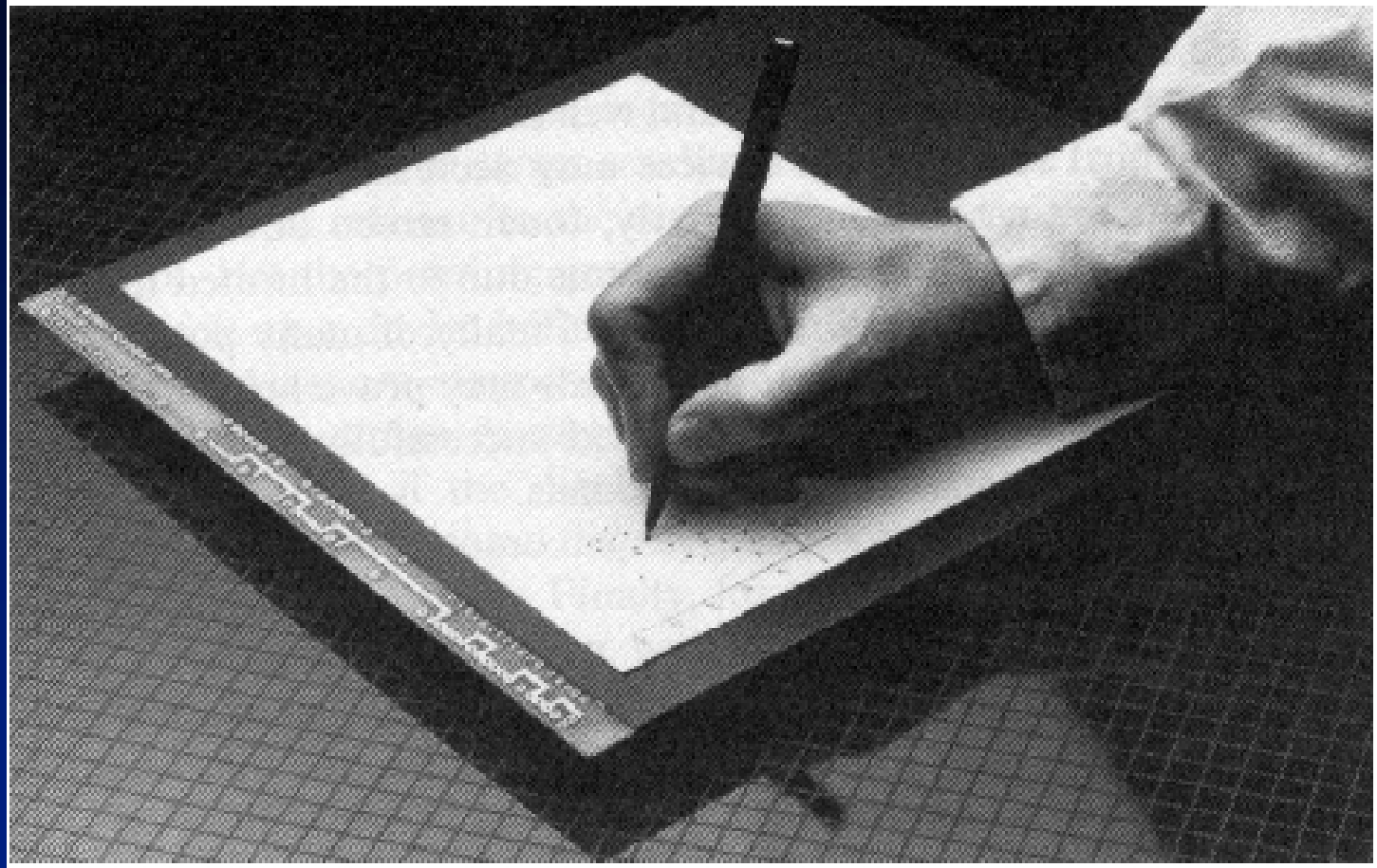


Tableta

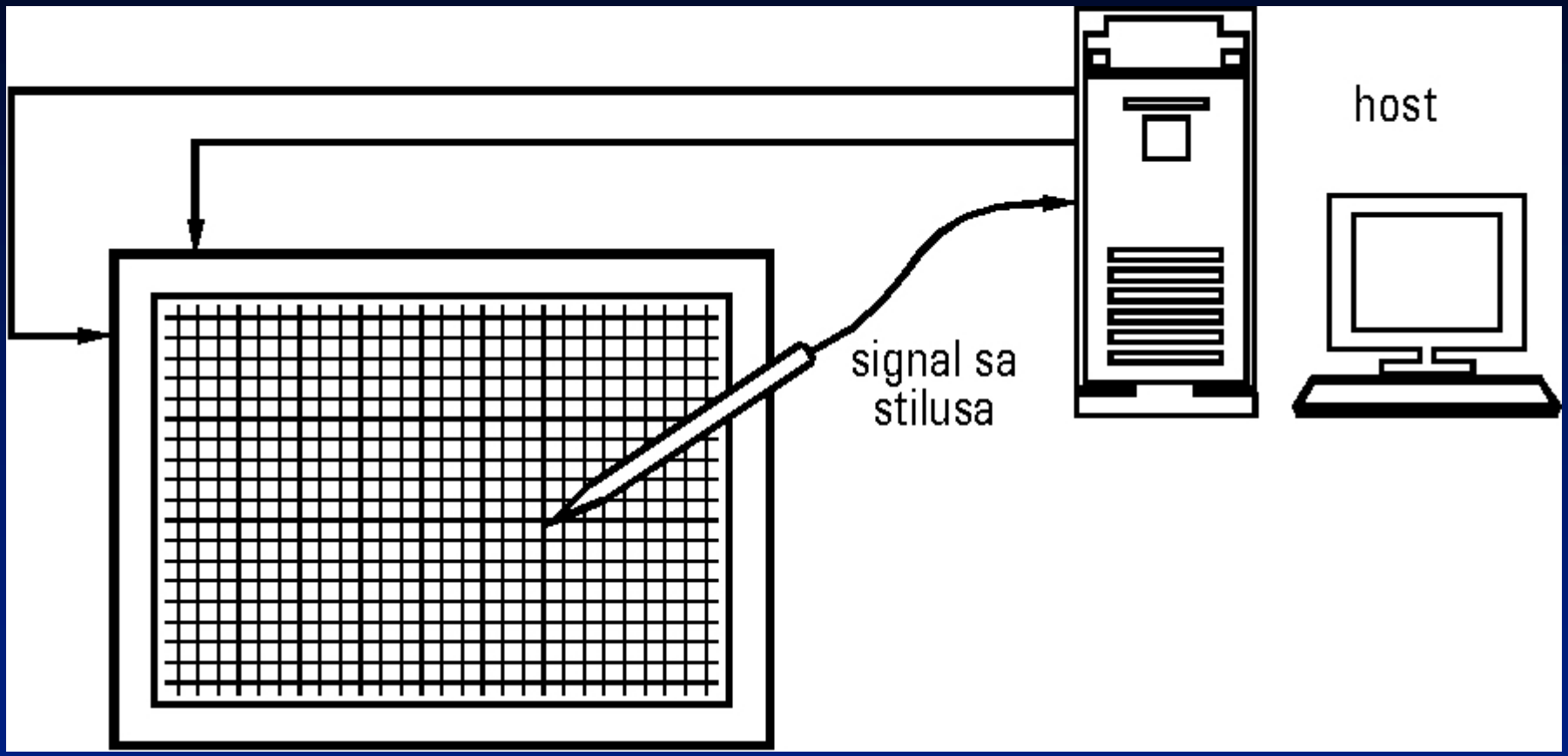




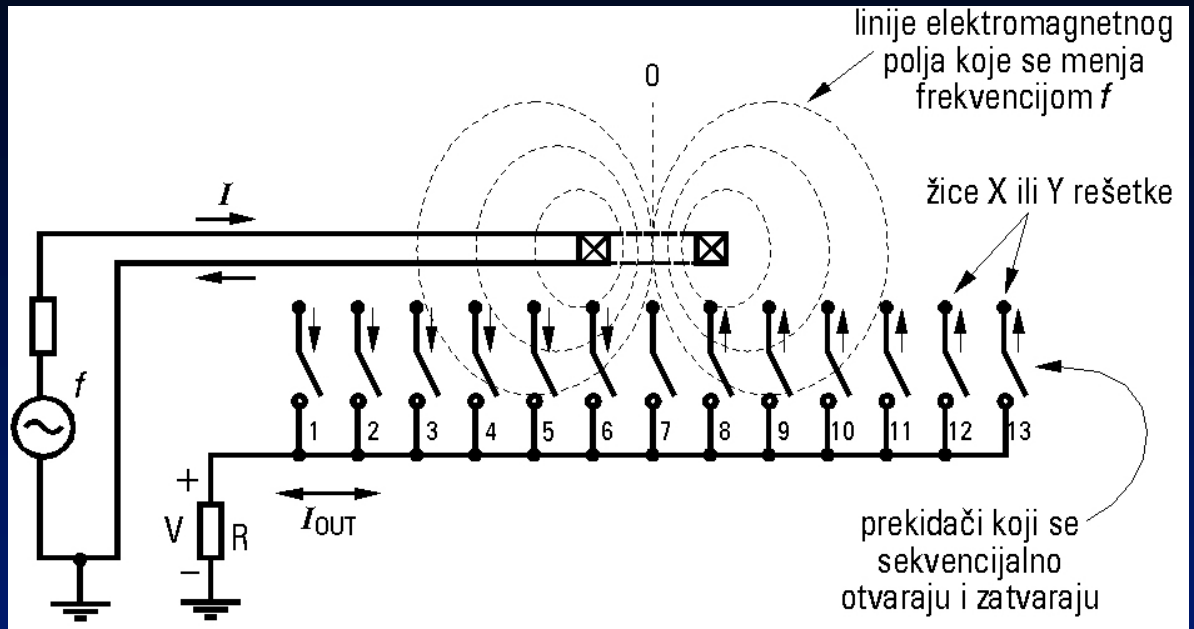
Touch tableta



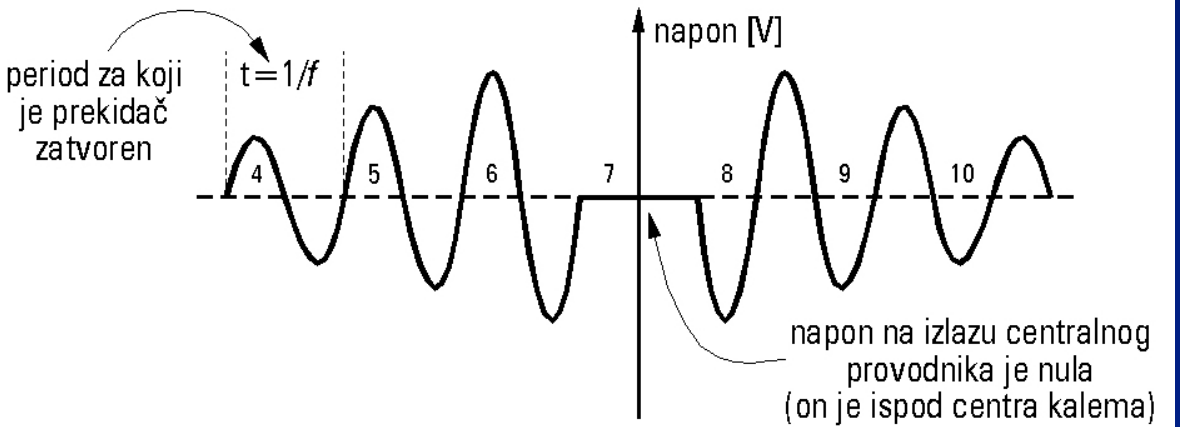
Matrični način kodiranja tablete



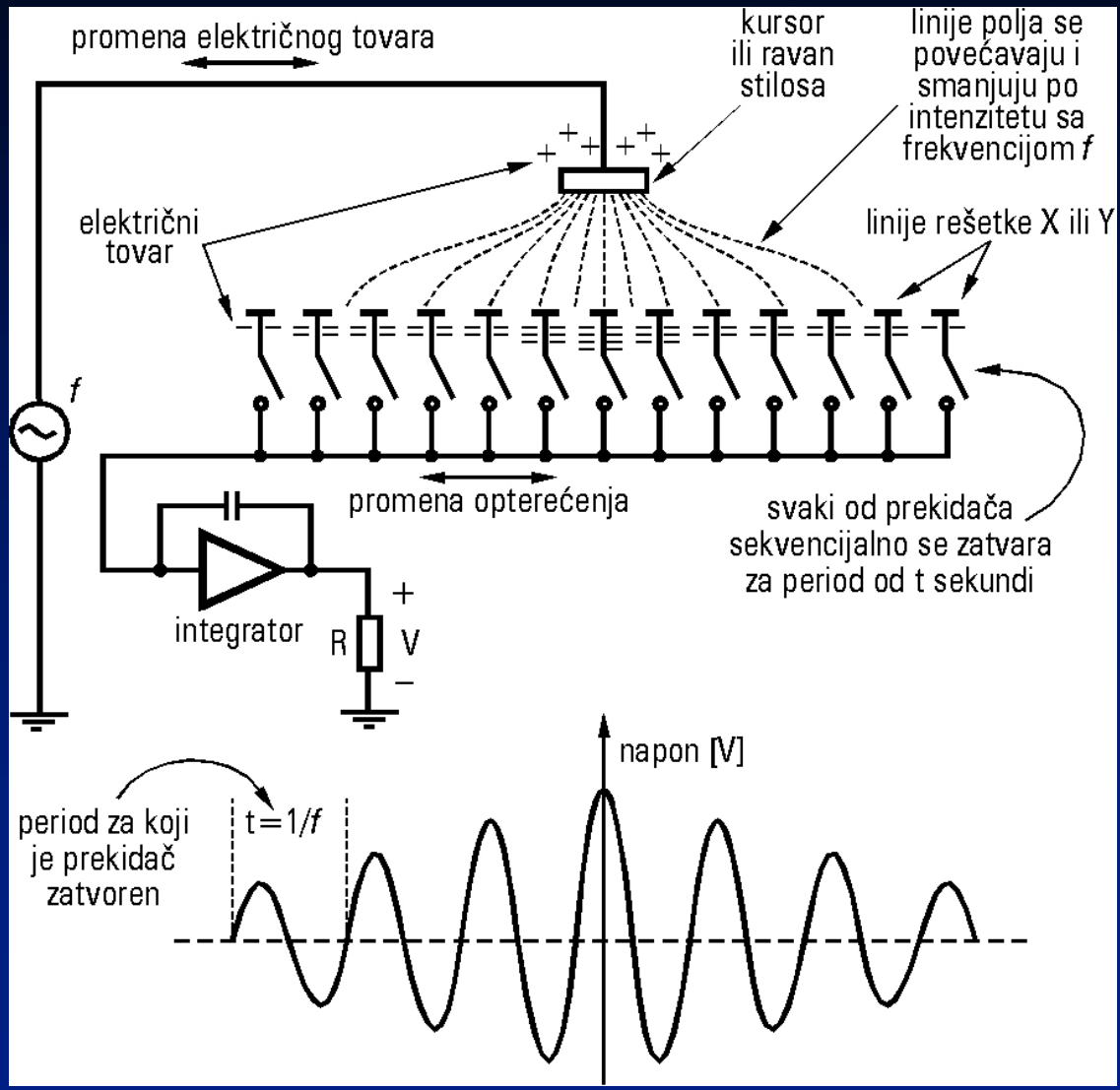
Princip rada elektromagnetne tablete



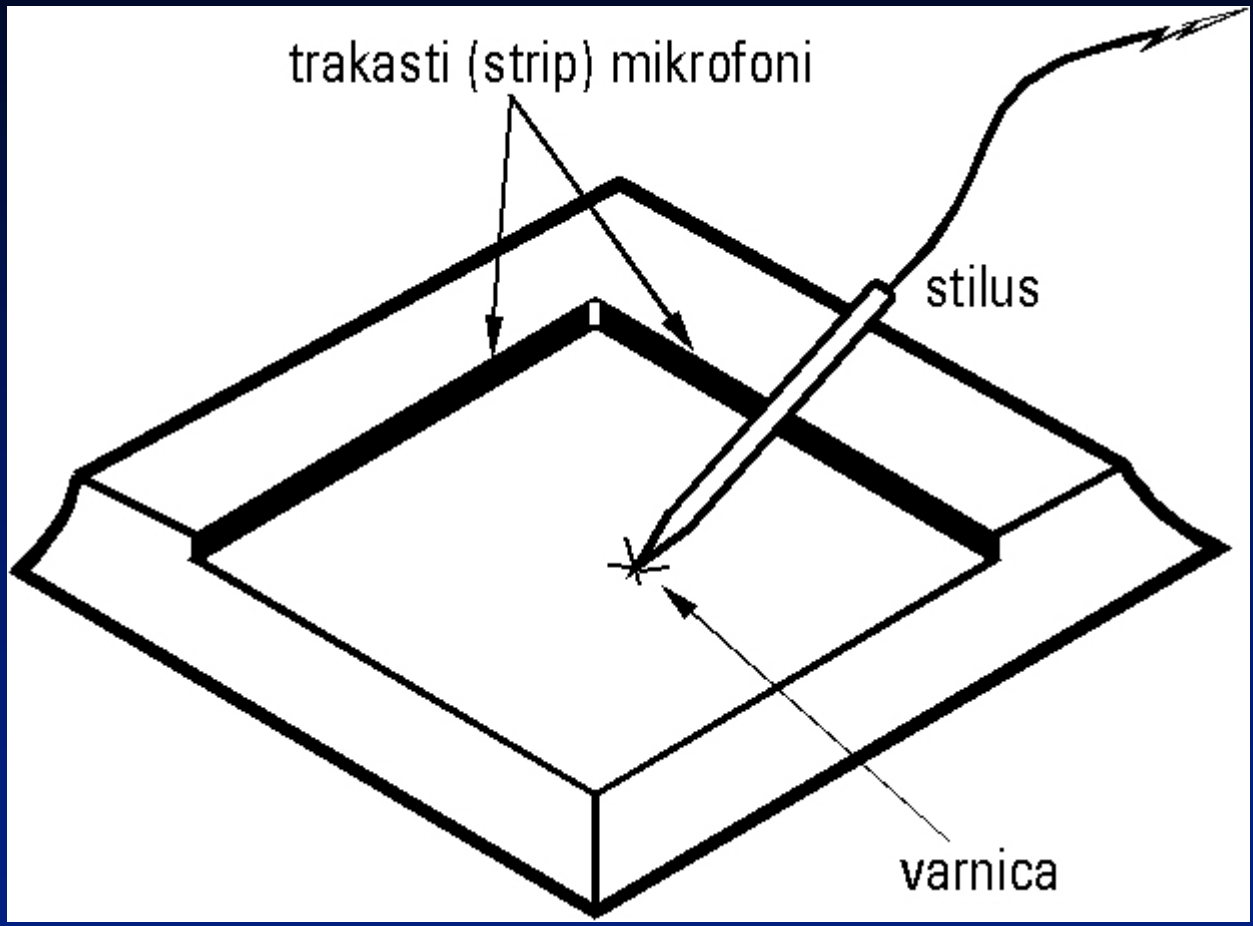
Napomena: Smer strelice označava fazu, a veličina amplitudu struje kada se prekidač zatvori.



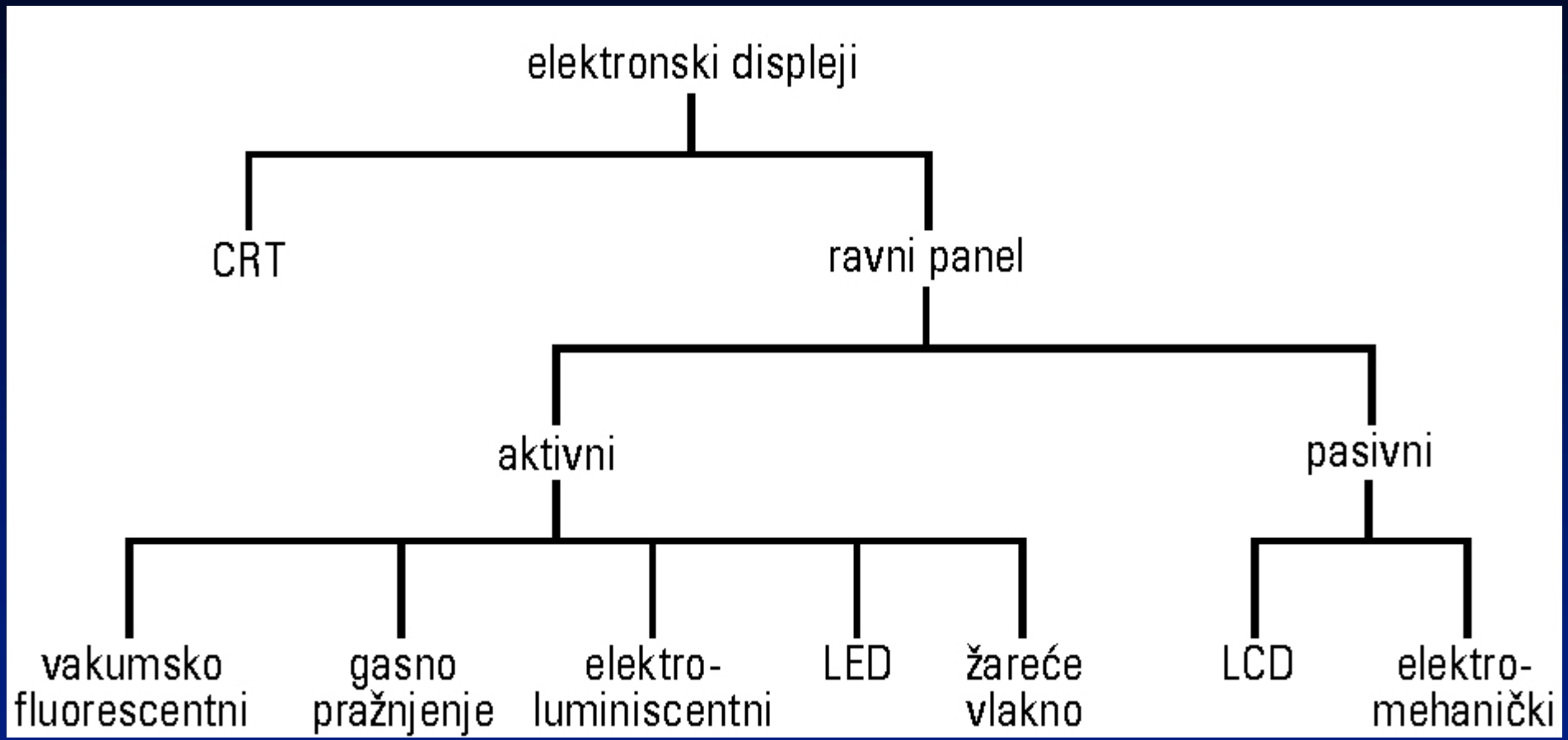
Princip rada elektrostatičke tablete



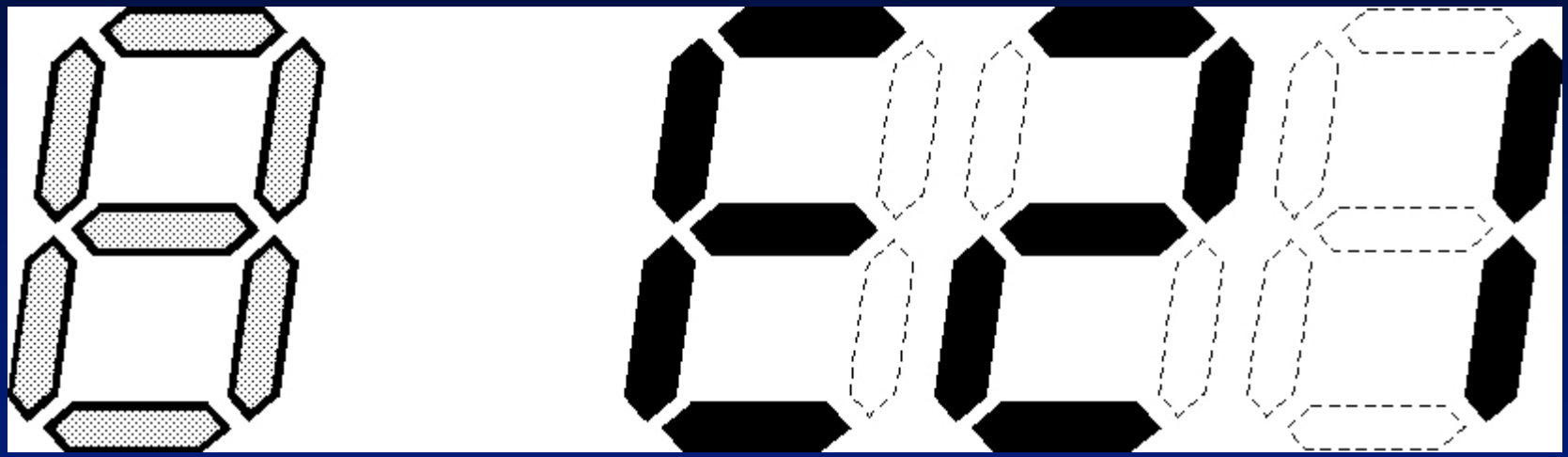
Akustička tableta



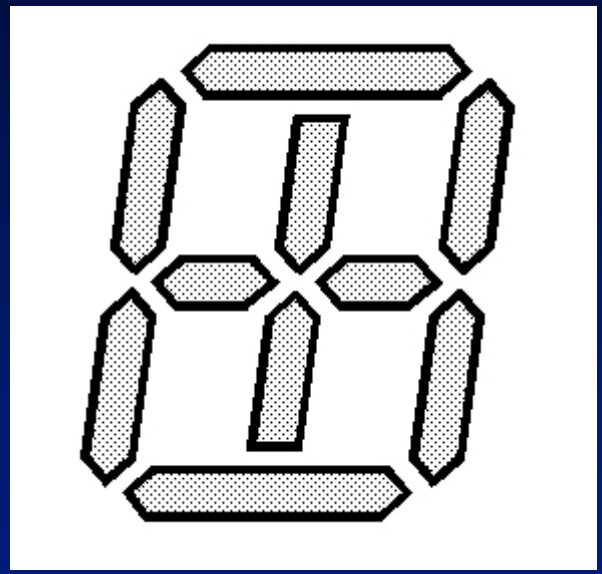
Podela displeja



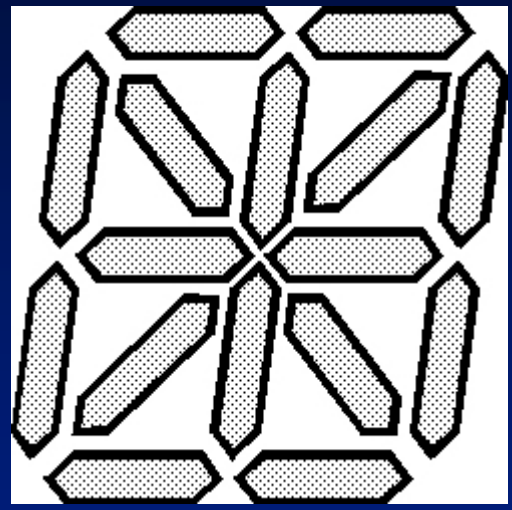
Sedmosegmentni prikaz



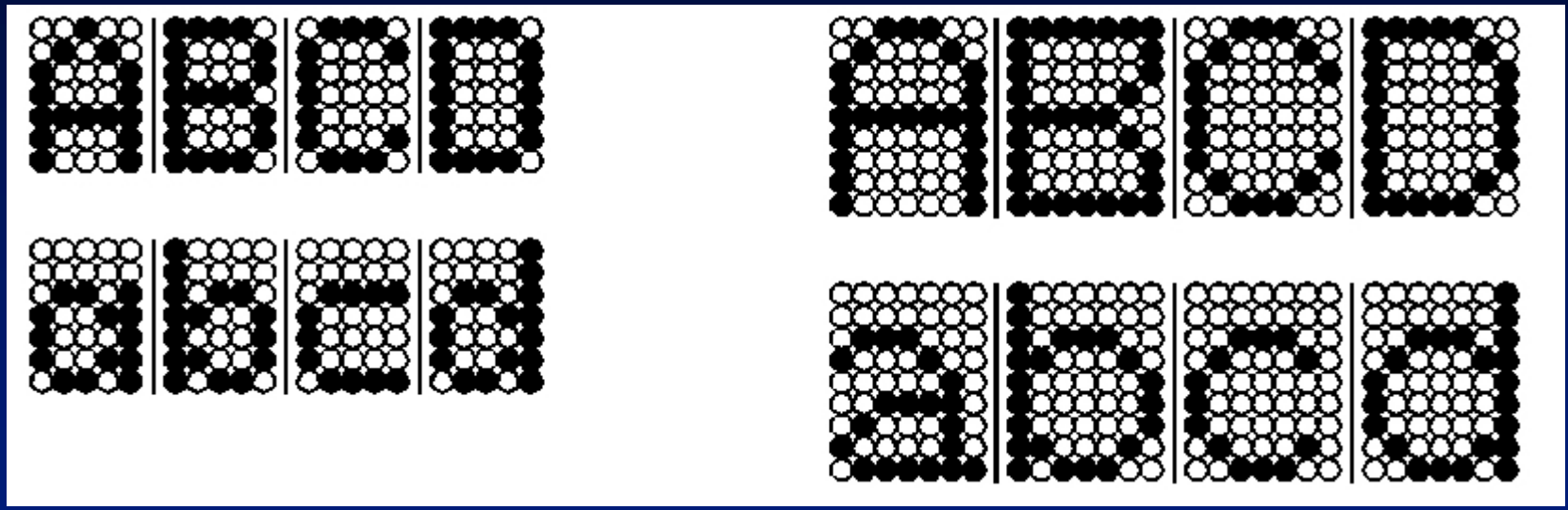
Desetosegmentni prikaz



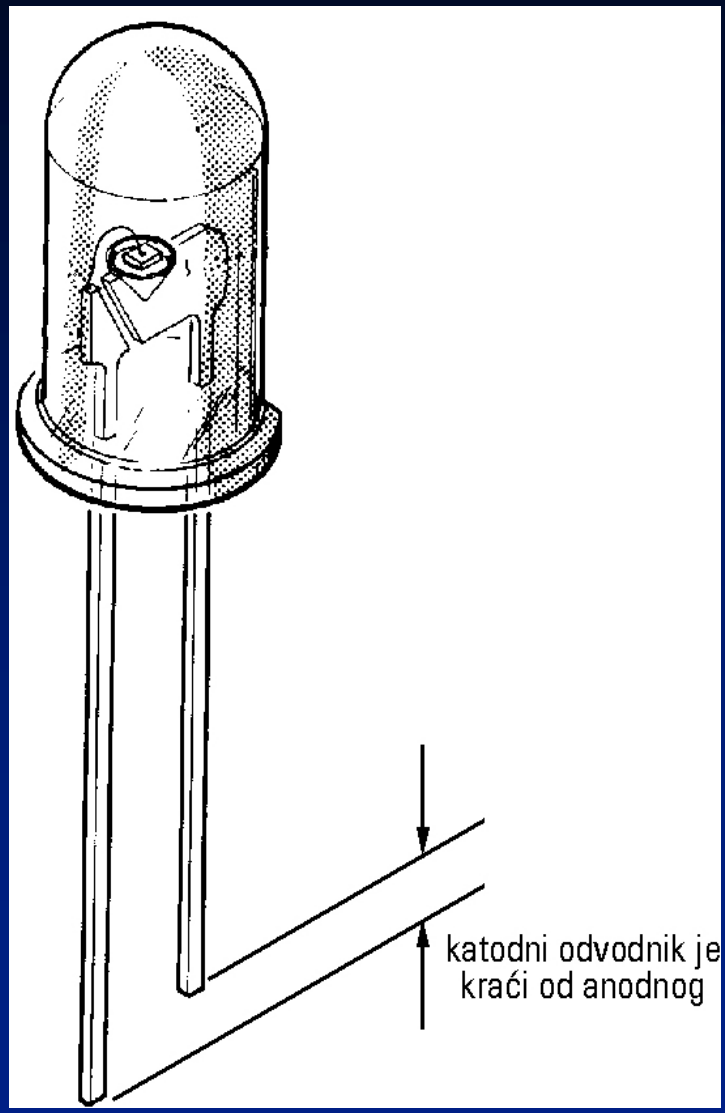
Šesnaestosegmentni prikaz



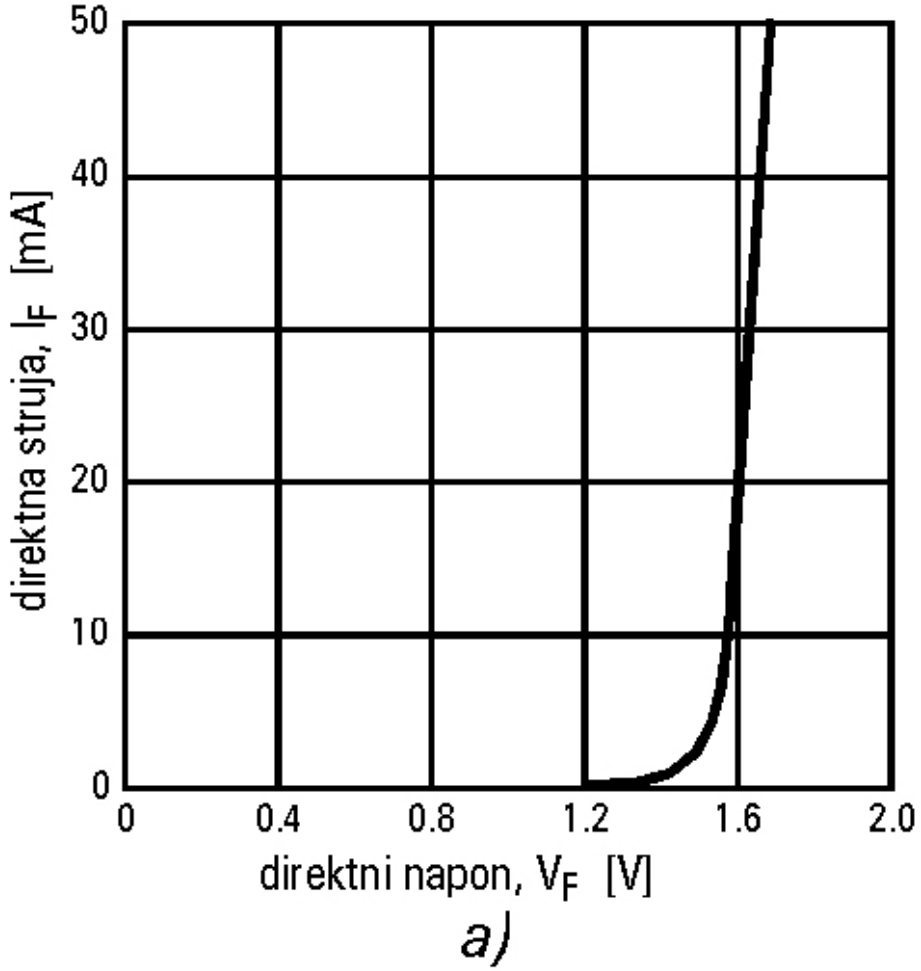
Matrični prikaz

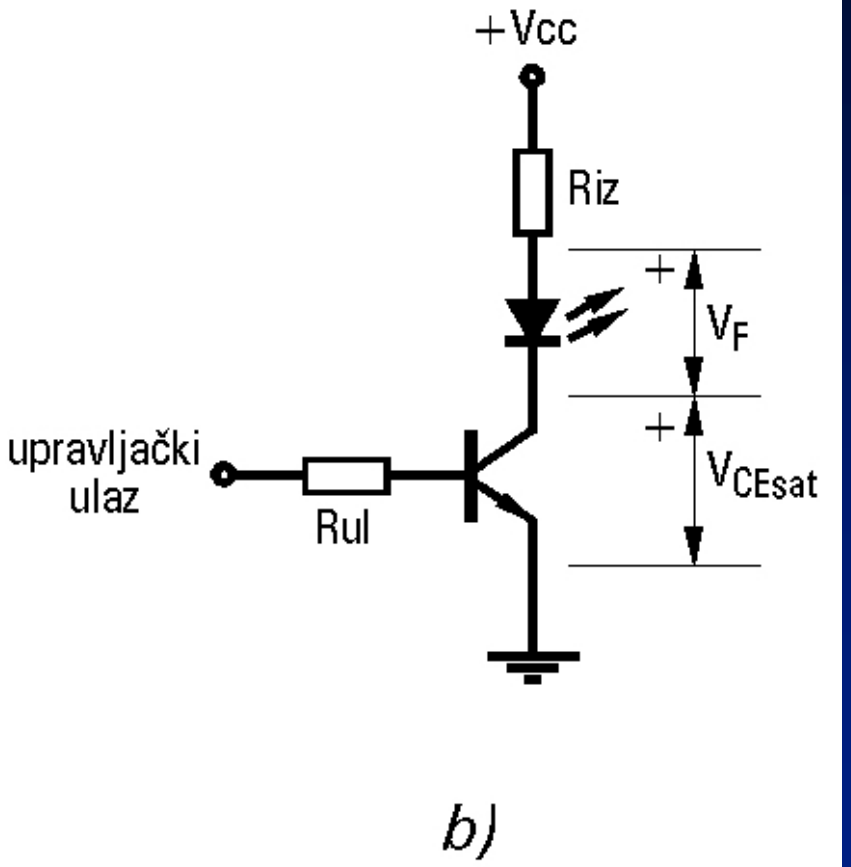


LED elemenat

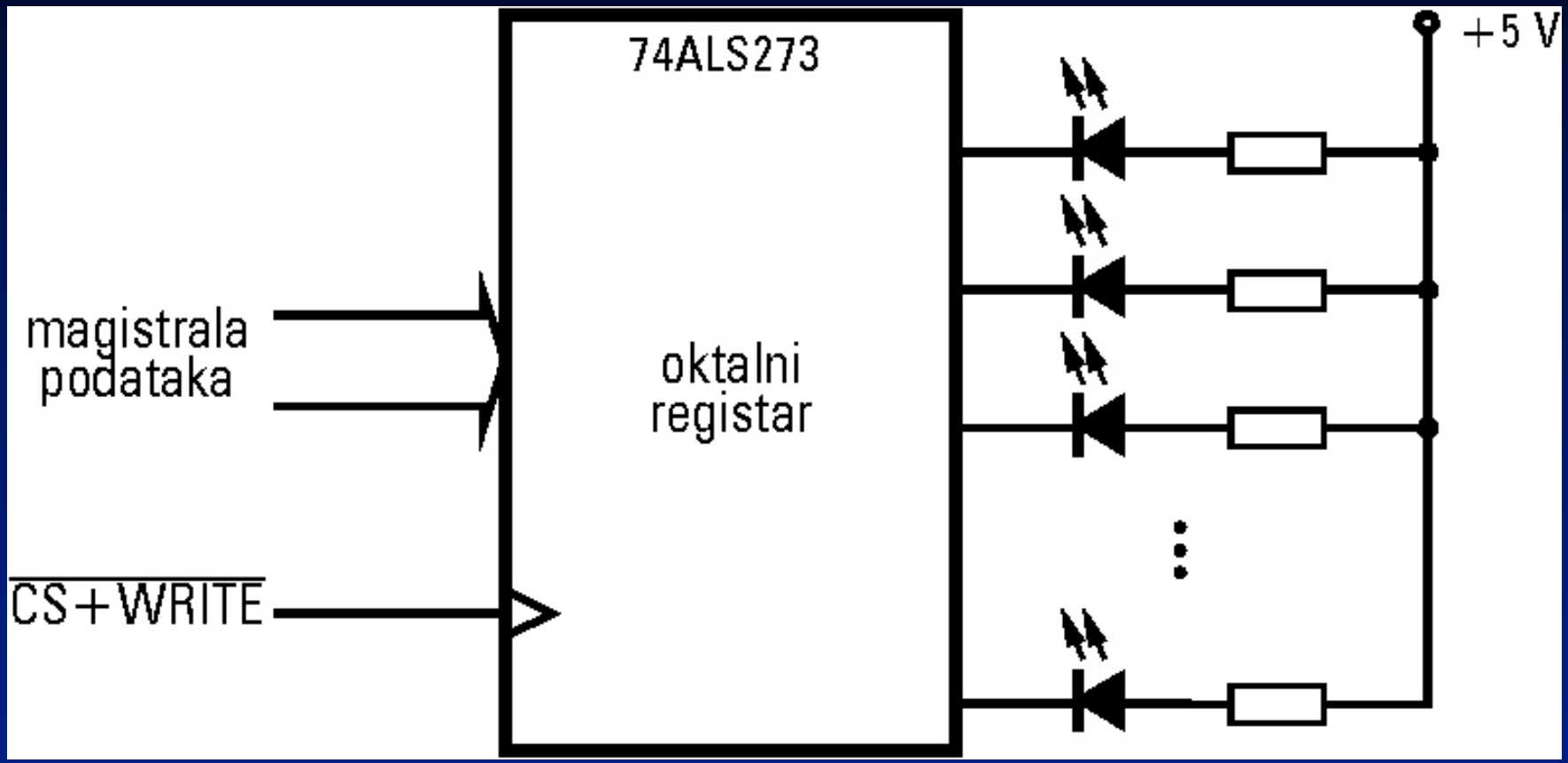


Kolo pobude LED-a

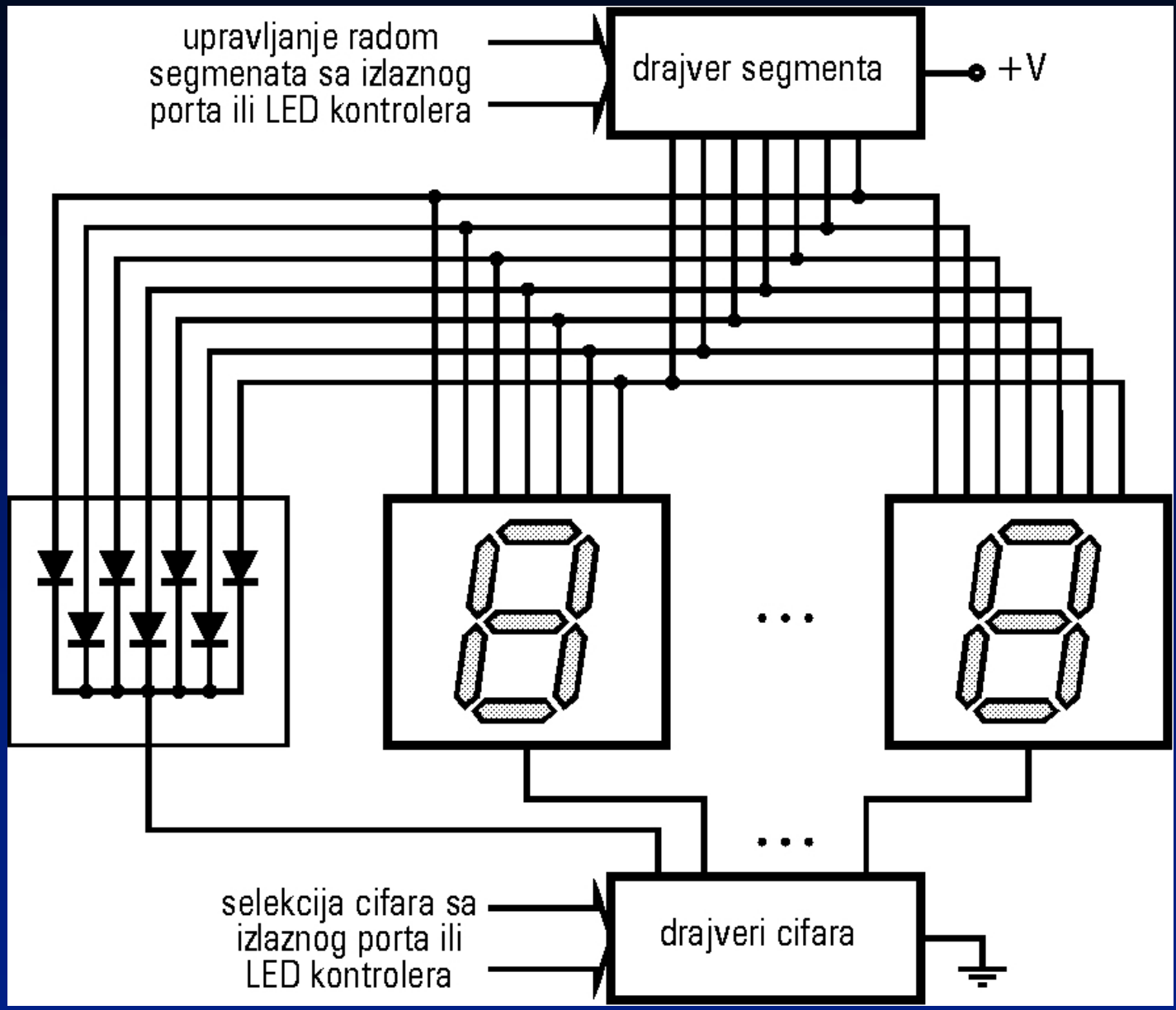




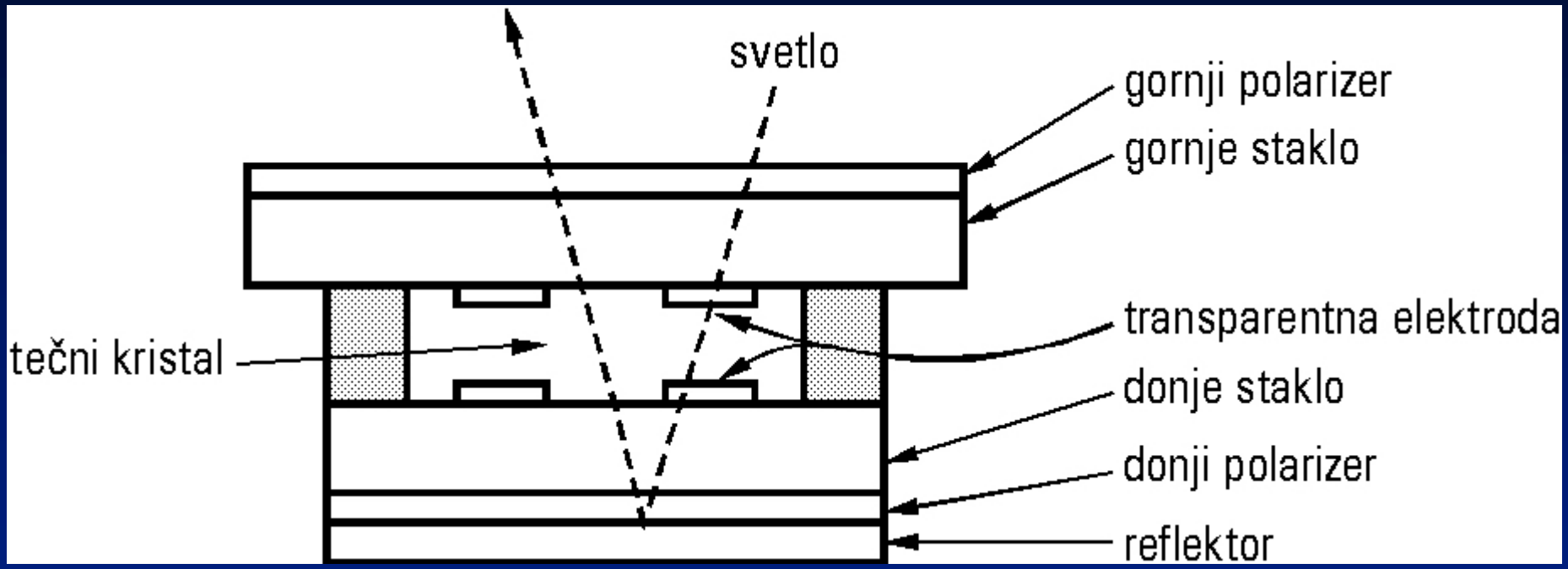
Grupno pobudjivanje LED elemenata



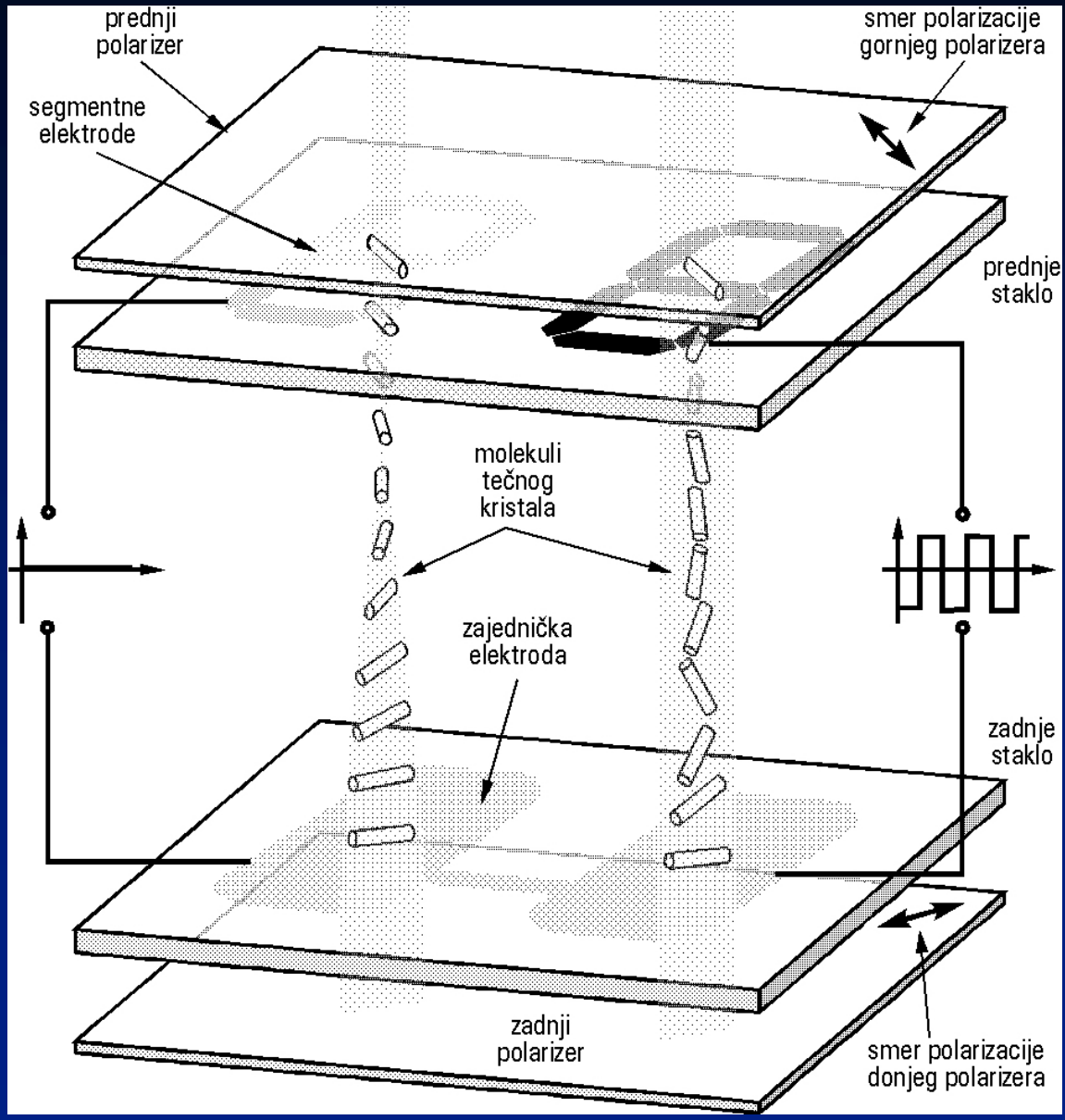
Višecifarski LED displej



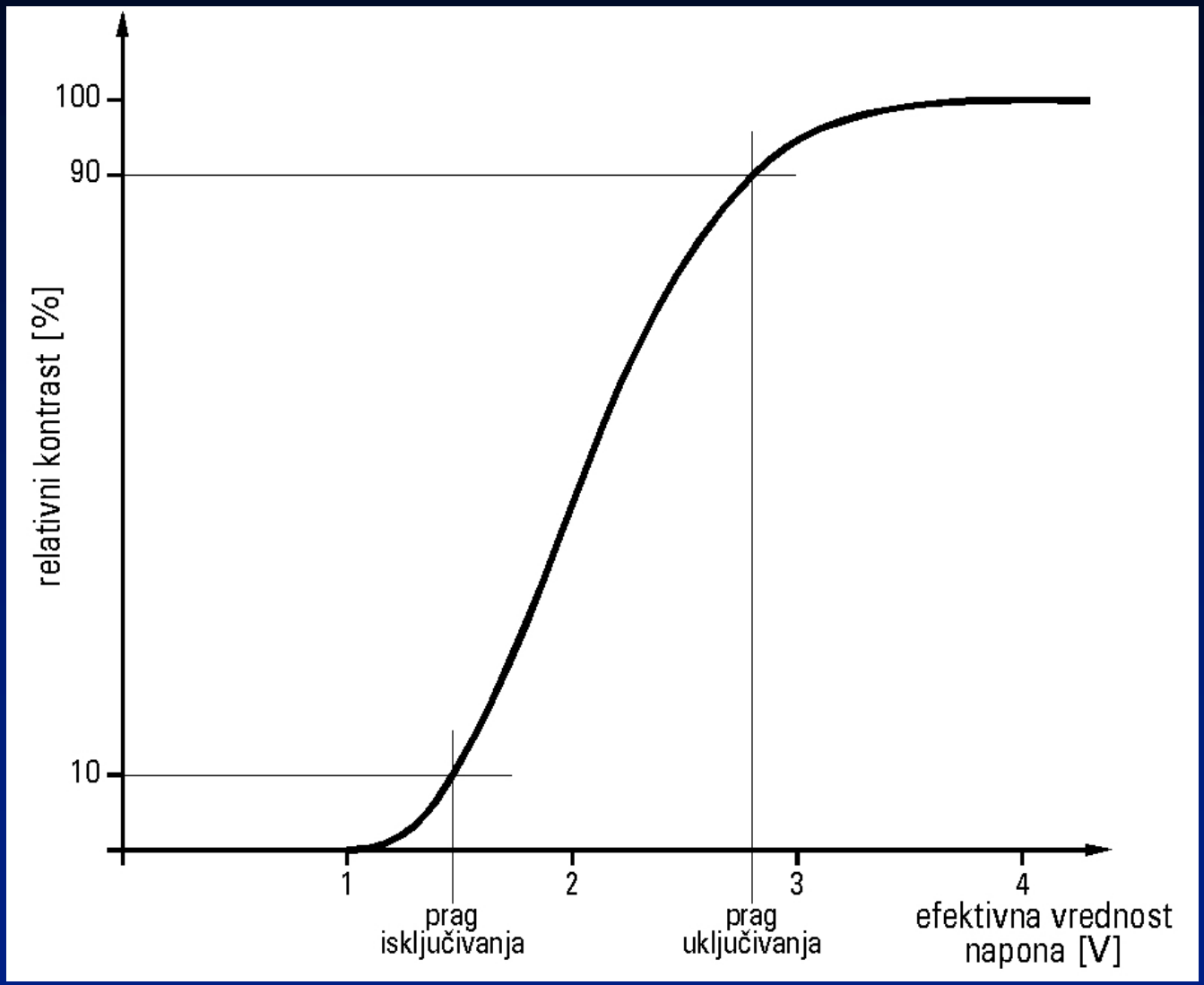
Konstrukcija tipičnog LCD-a



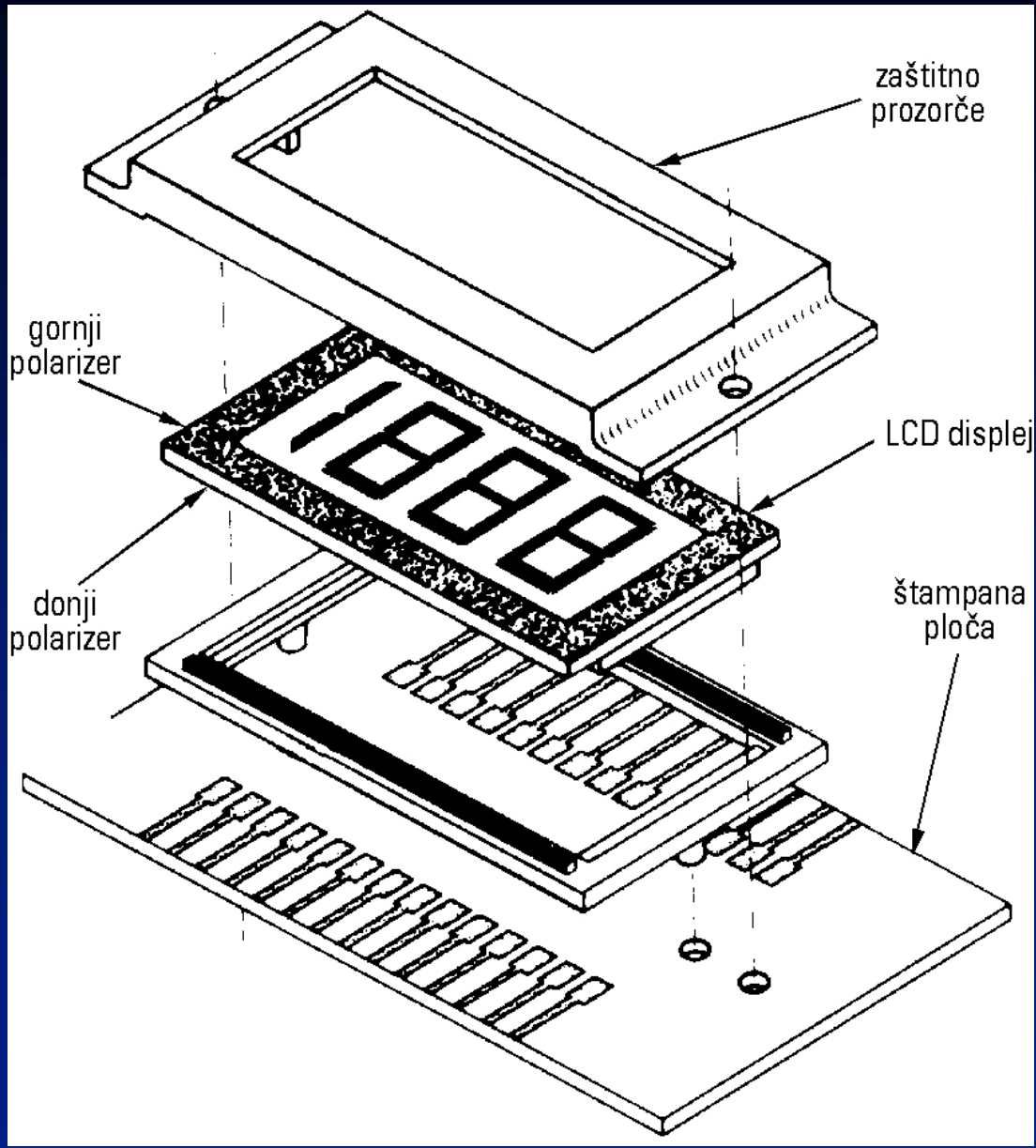
LCD sa diskretnom pobudom



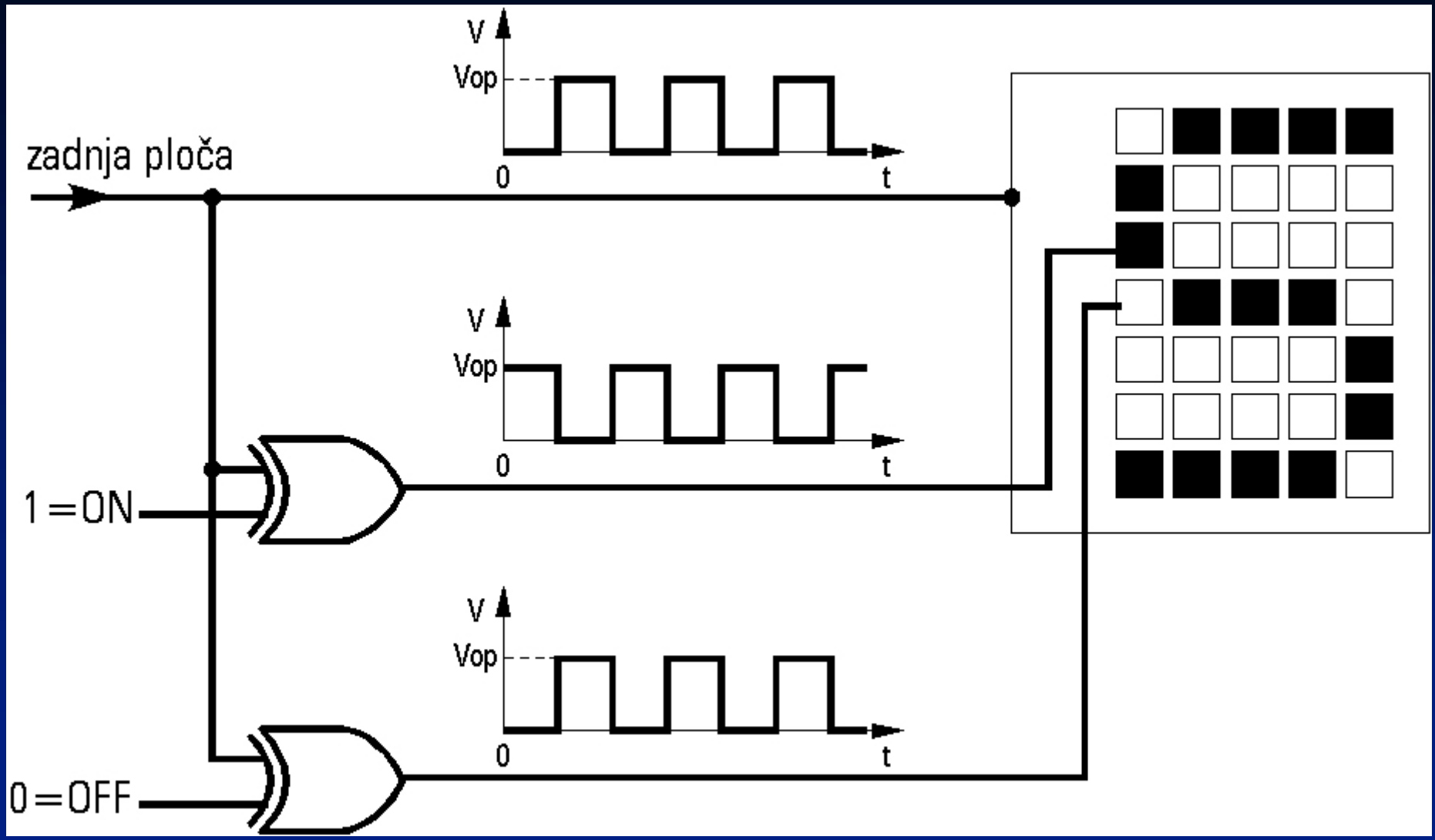
Zavisnost kontrasta



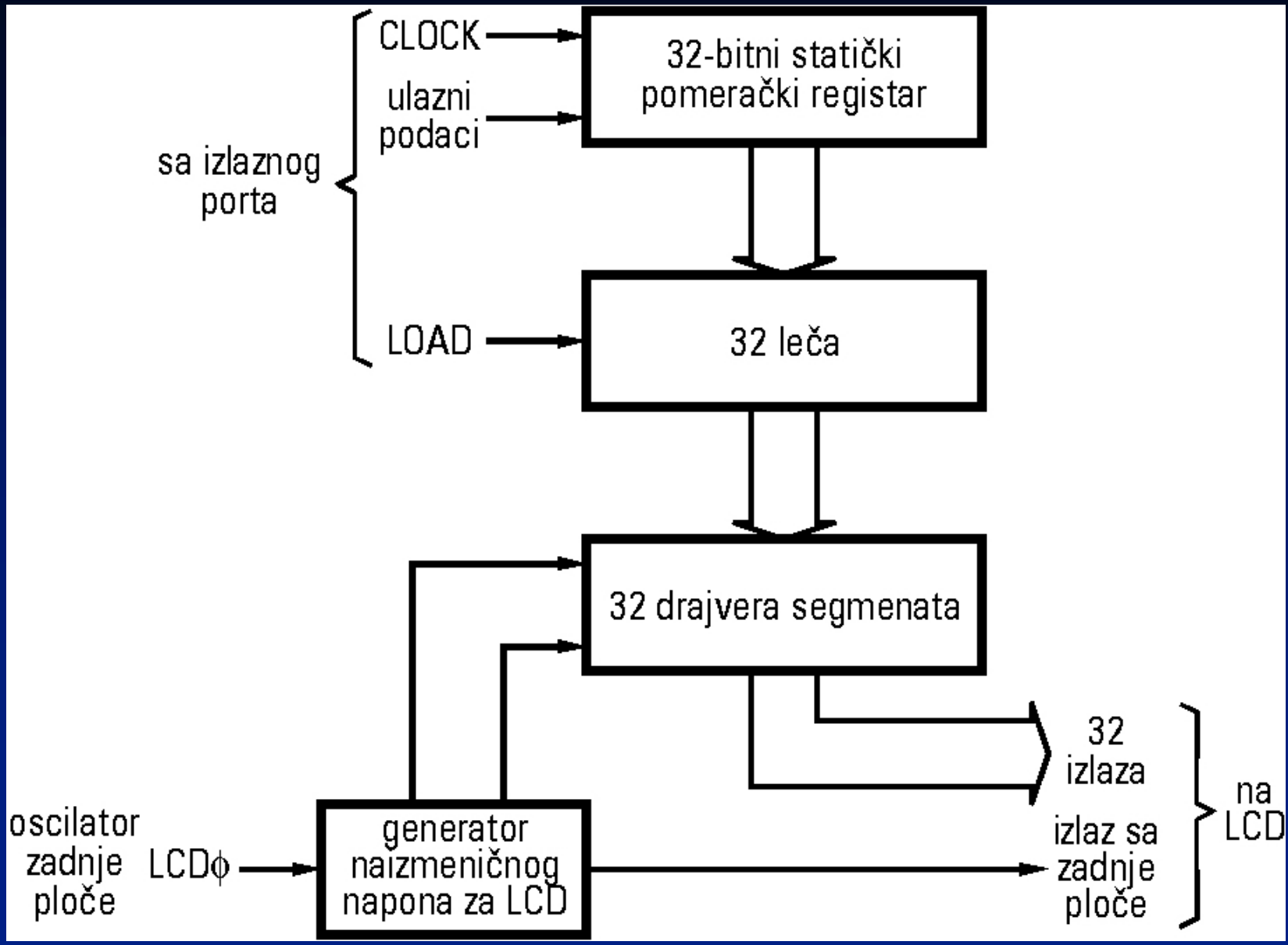
Konstruktivni delovi *LCD*-a



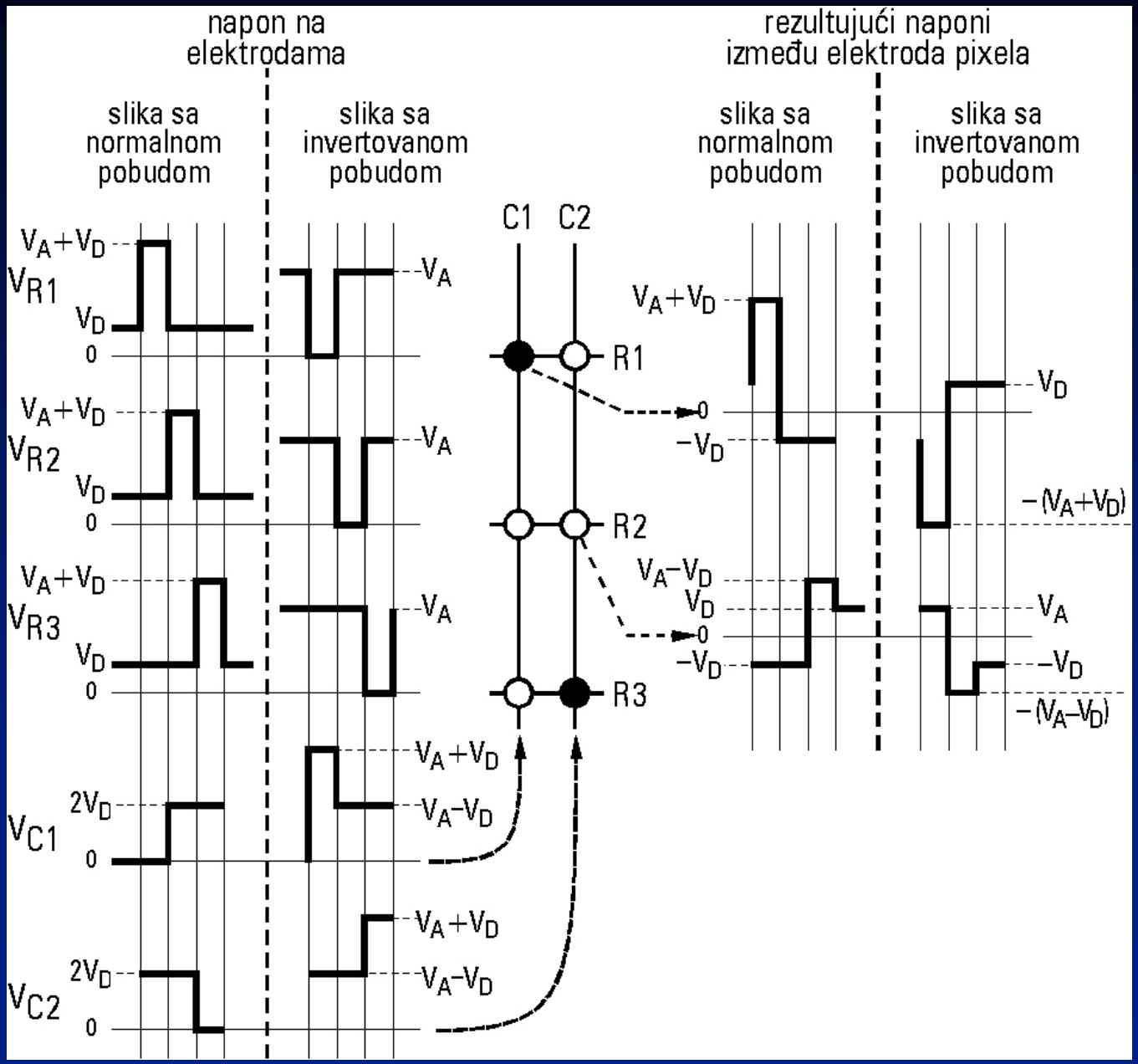
Kolo za pobudu *LCD*-a



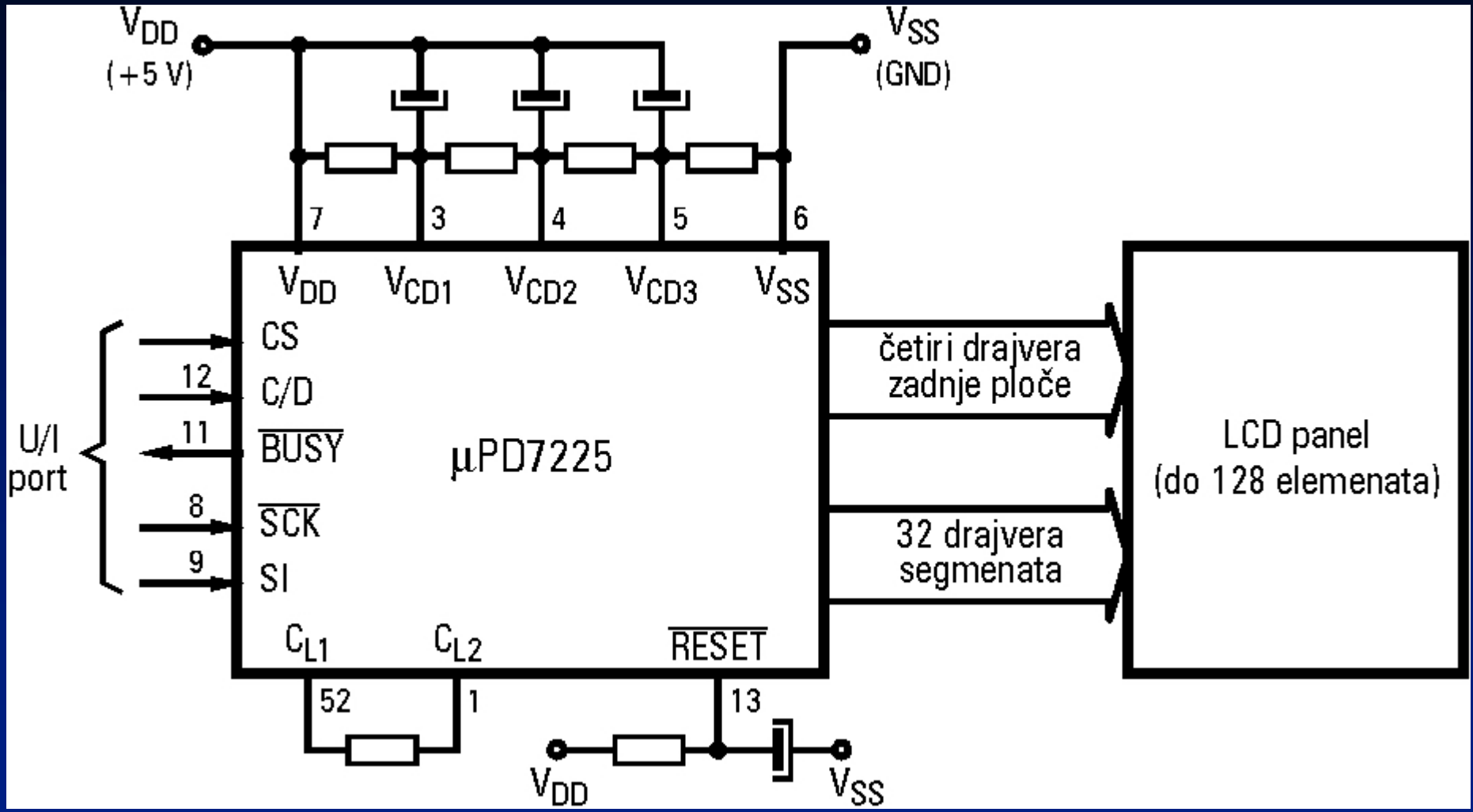
Interna struktura *LCD* pobudnog kola



Multipleksiranje pobuda kod *LCD*-a

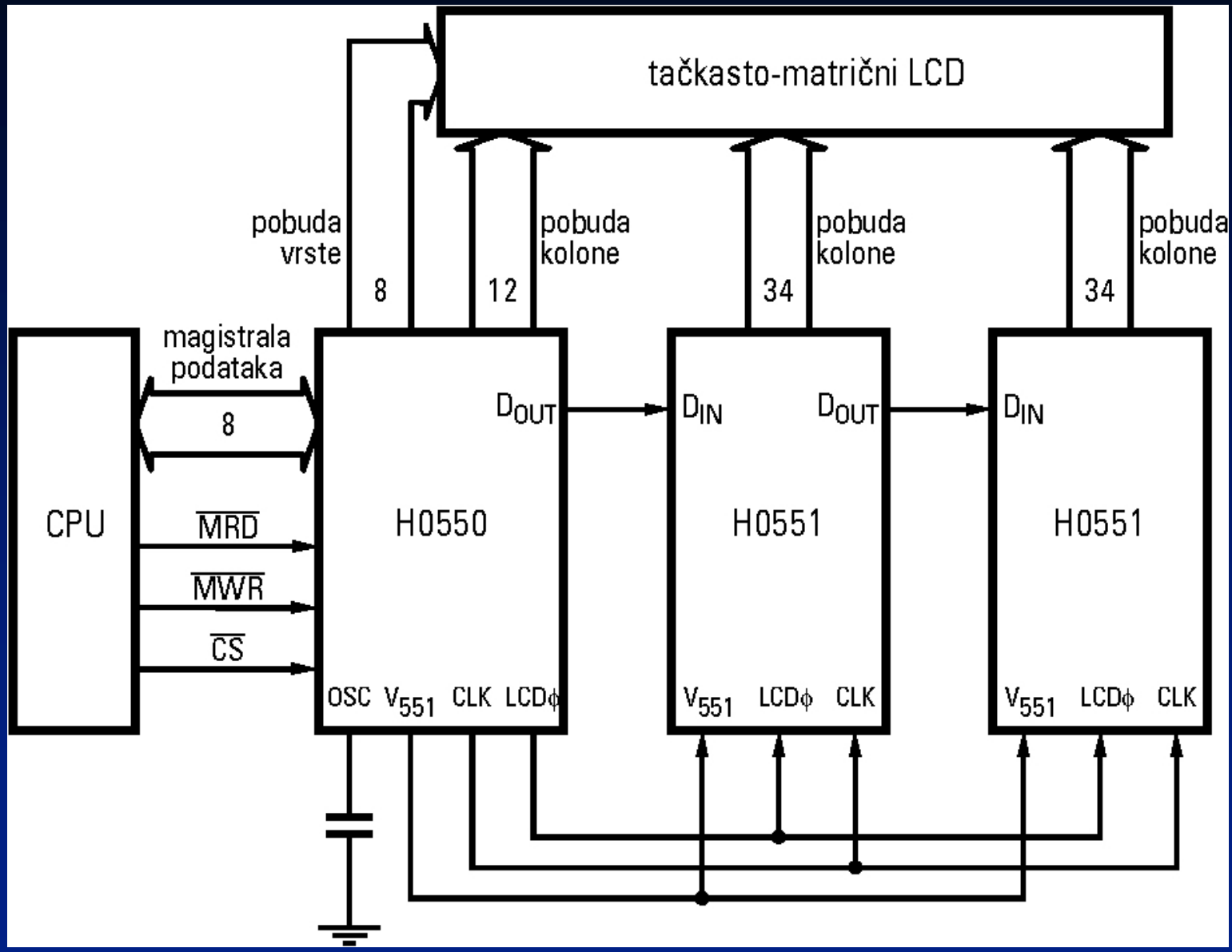


Tipično povezivanje *LCD*-a

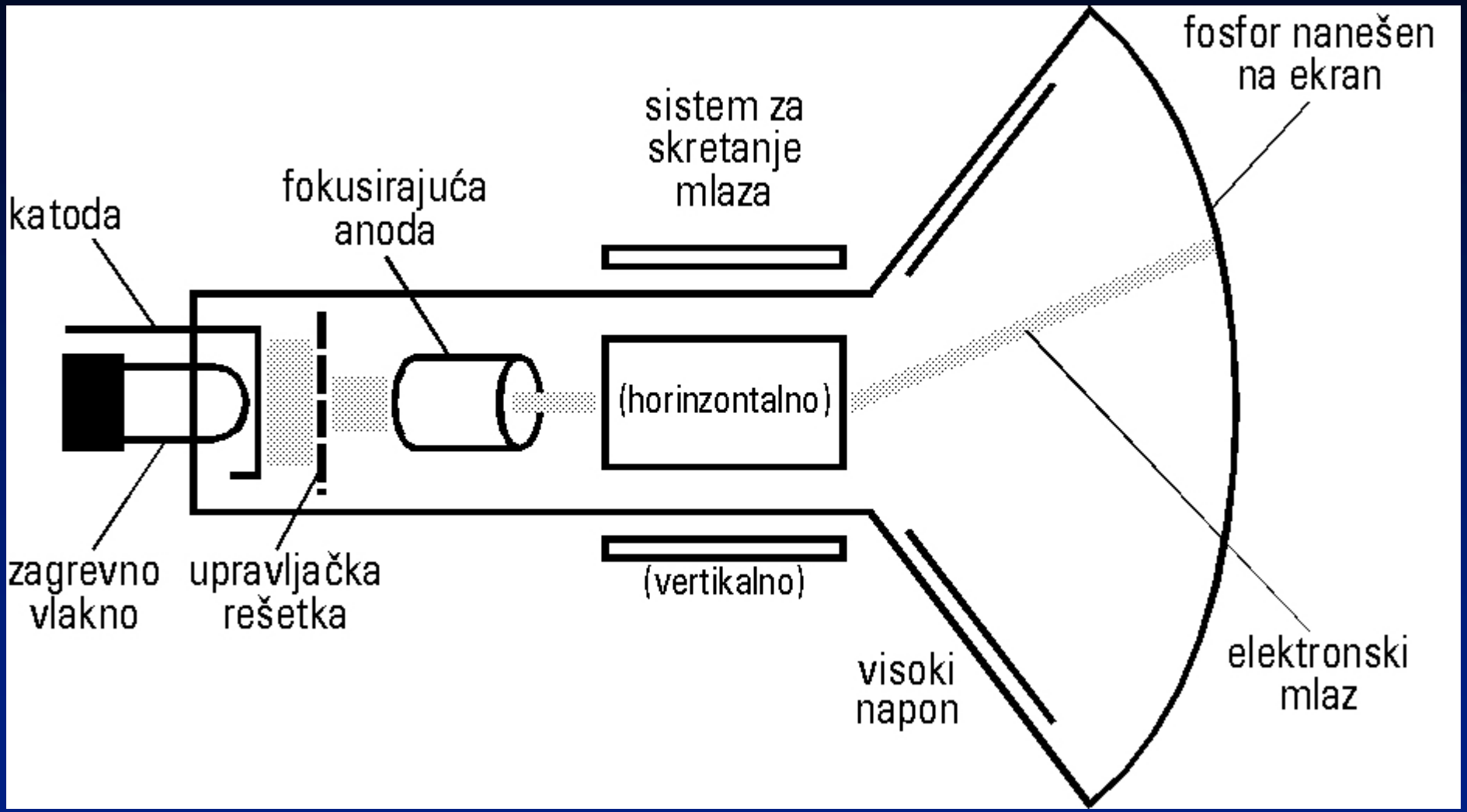


Tačkasto-matrični displej modul

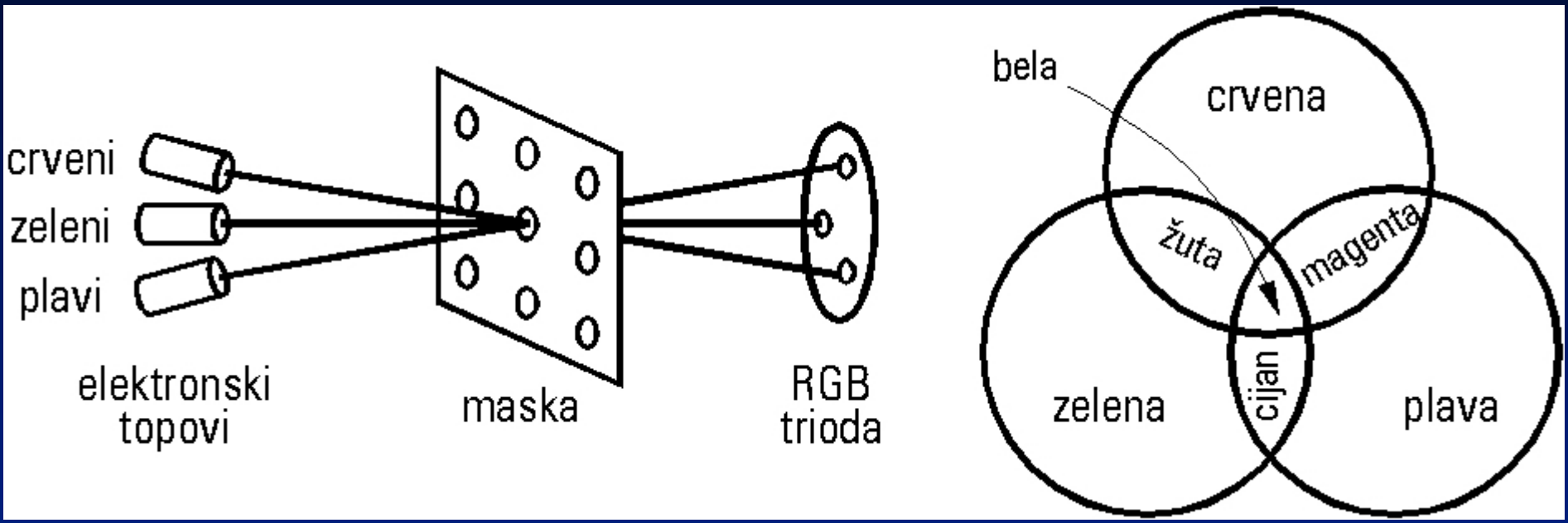
Pobuda tačkasto-matričnog *LCD* displeja



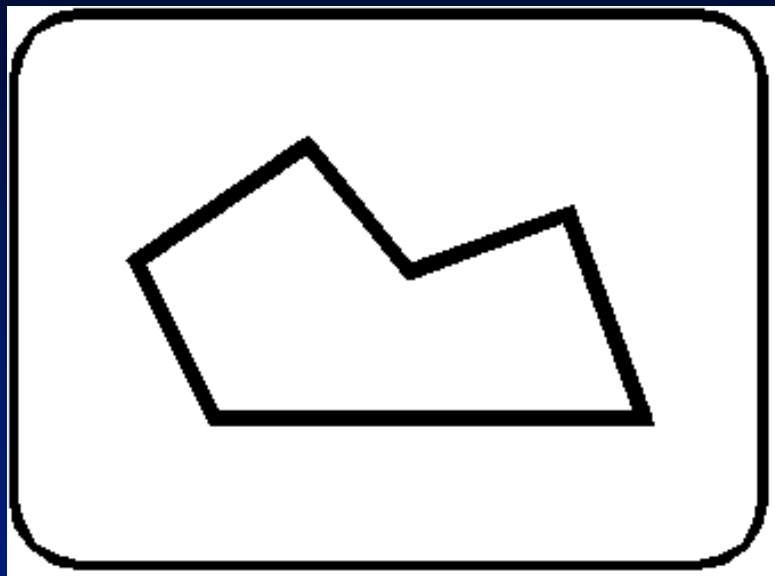
Katodna cev - CRT



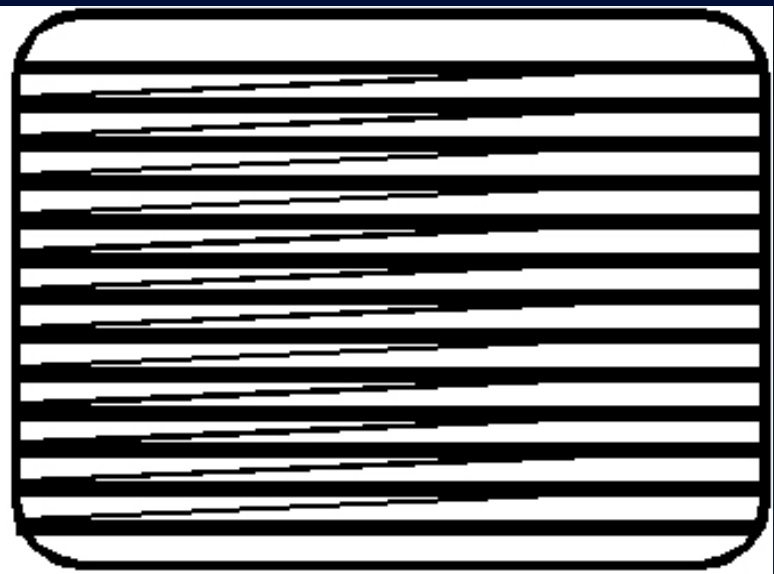
Princip rada CRT-a u boji



Dva načina skretanja mlaza kod CRT prikaza

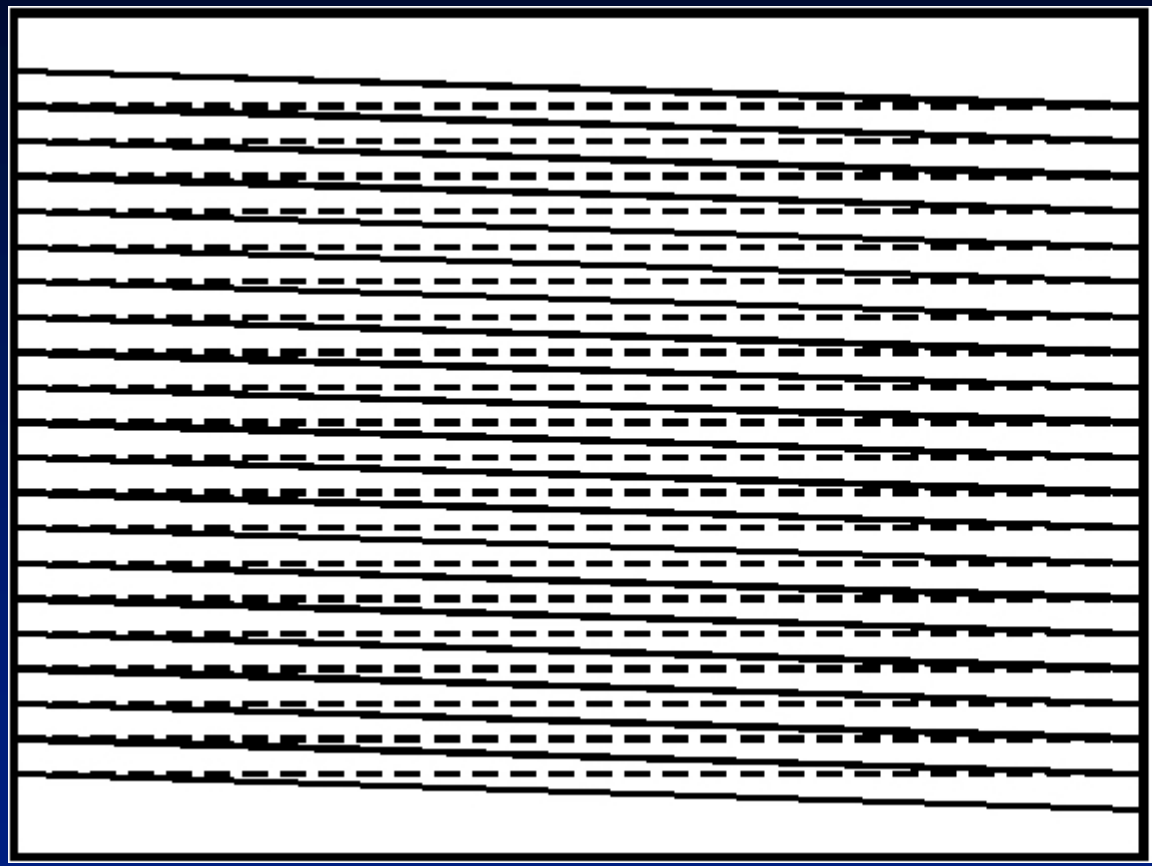


proizvoljno skretanje



raster skretanje

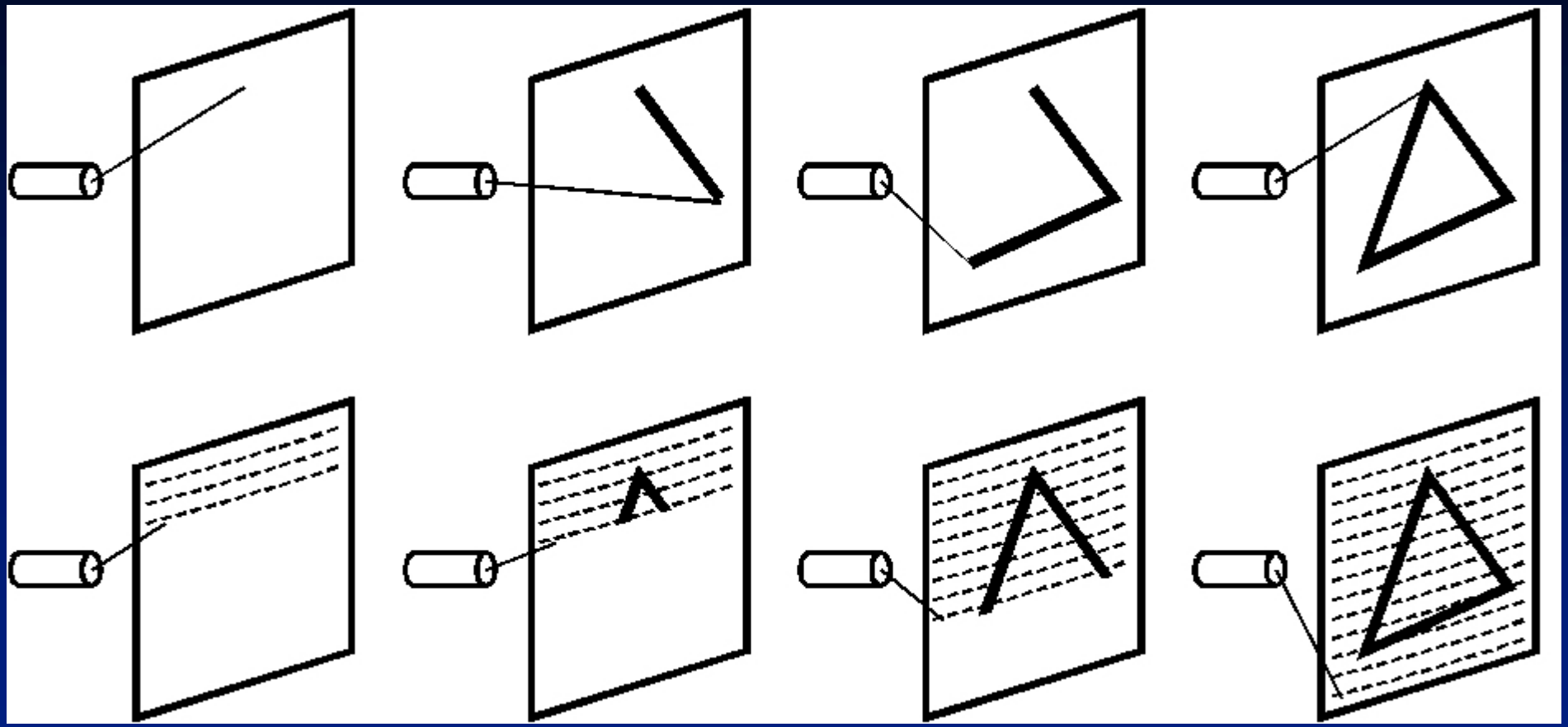
Raster analiza



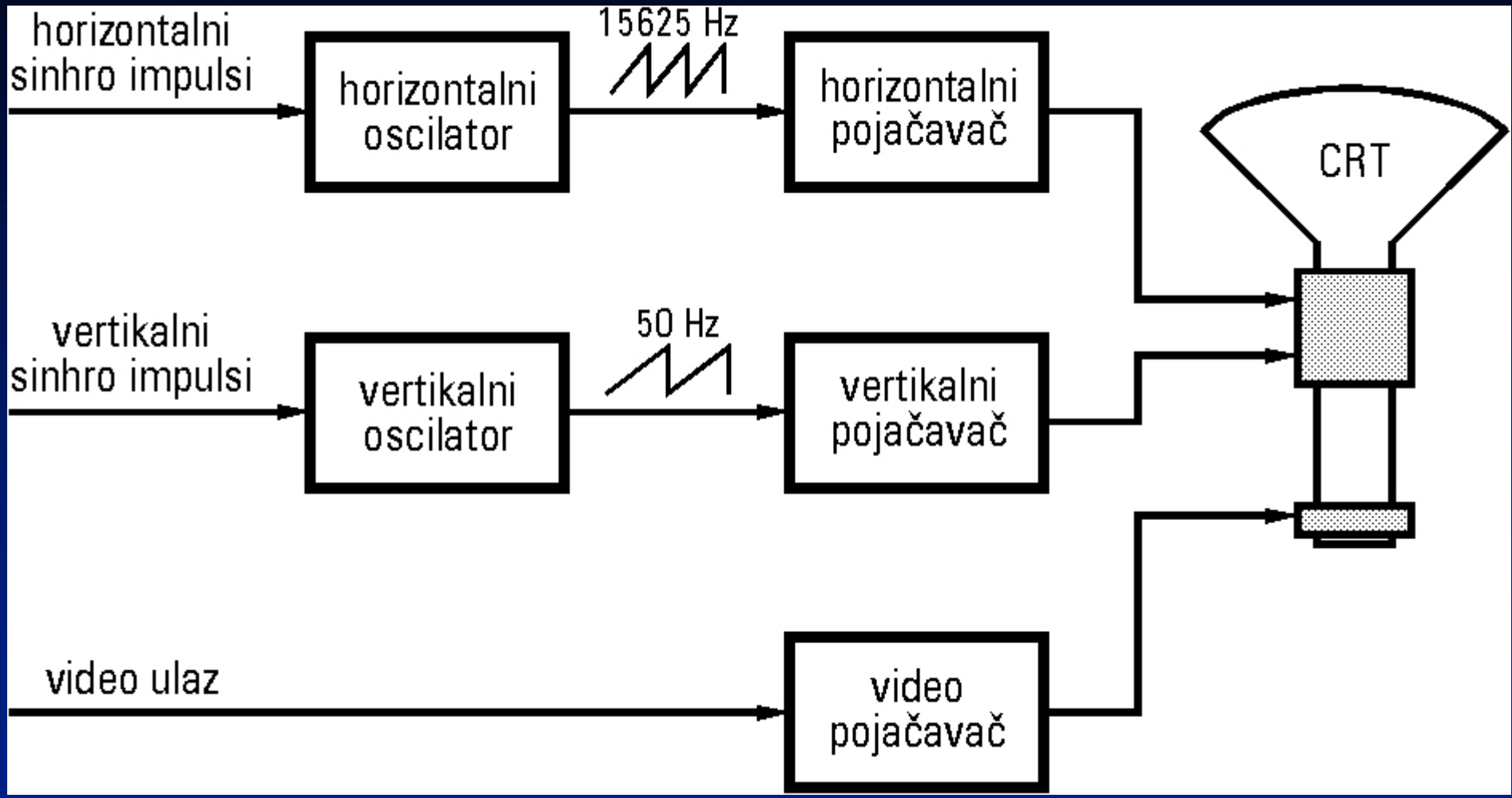
analizatorska linija (vidljiva)

povratna linija (nevidljiva)

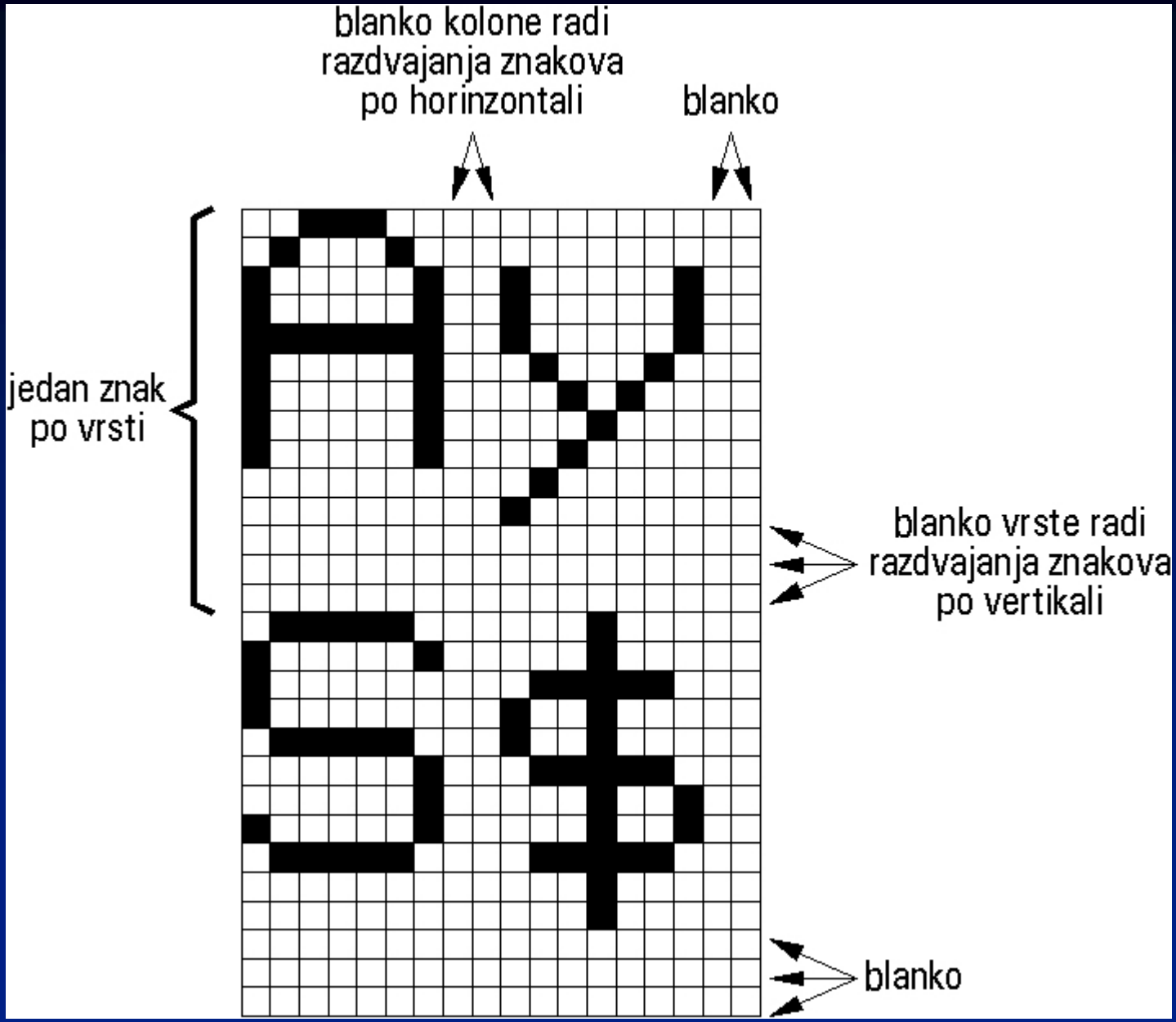
Vektorska i raster analiza



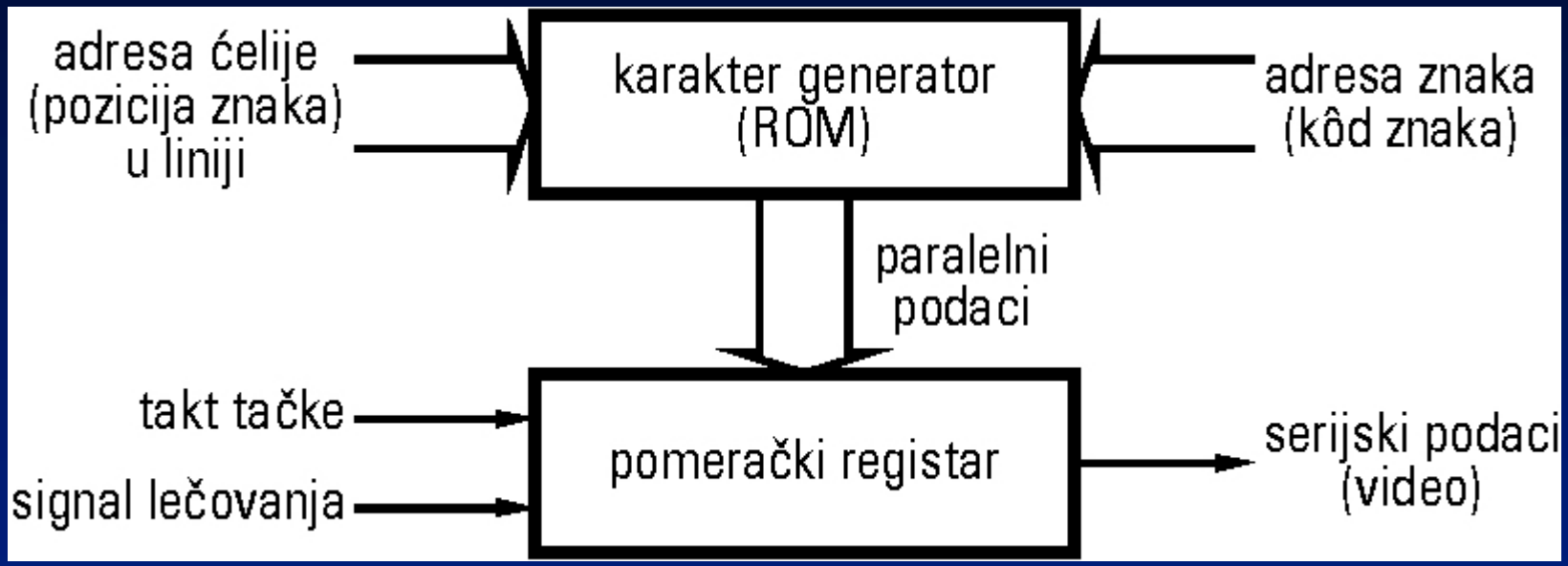
Gradivni blokovi CRT monitora



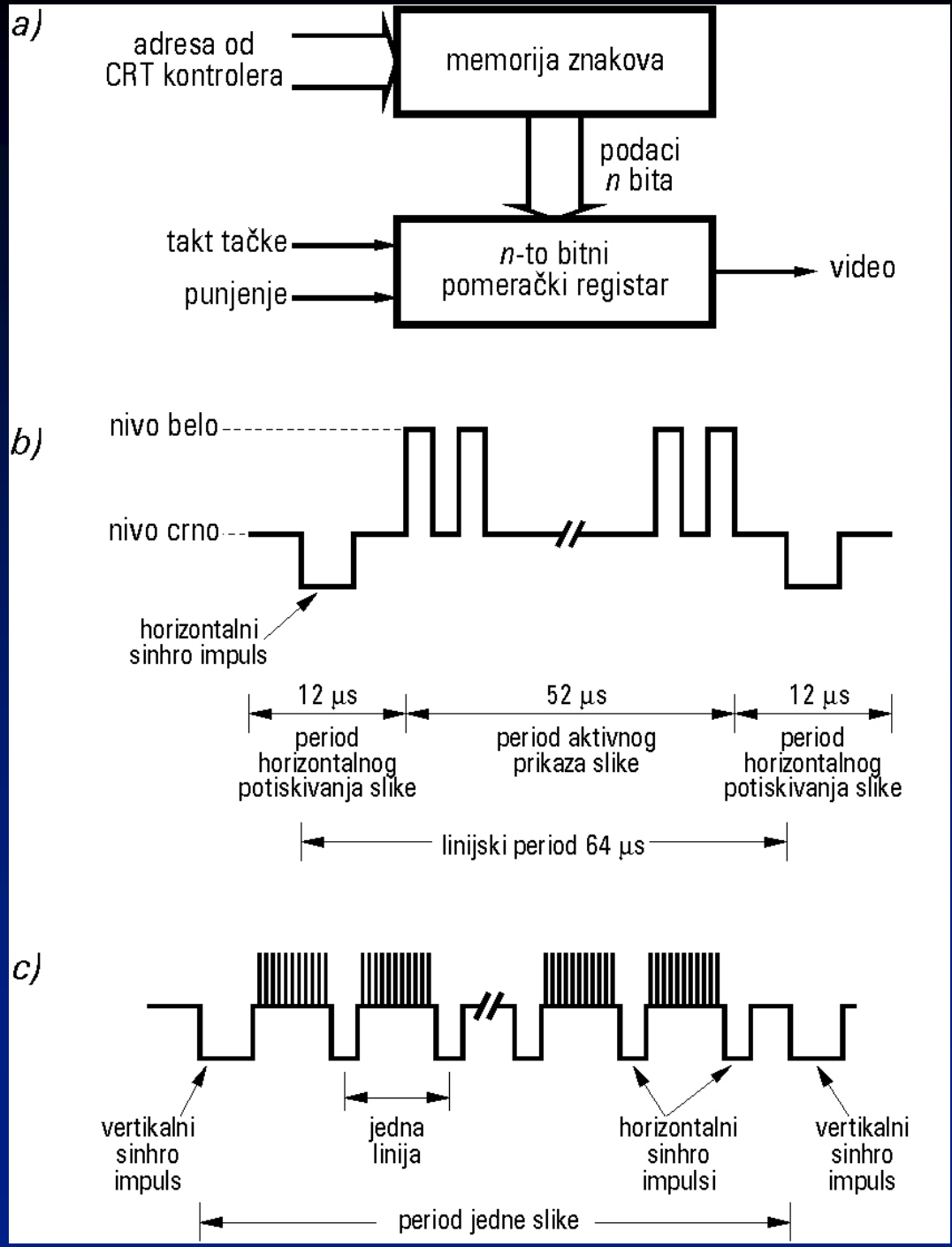
Matrica prikaza znaka



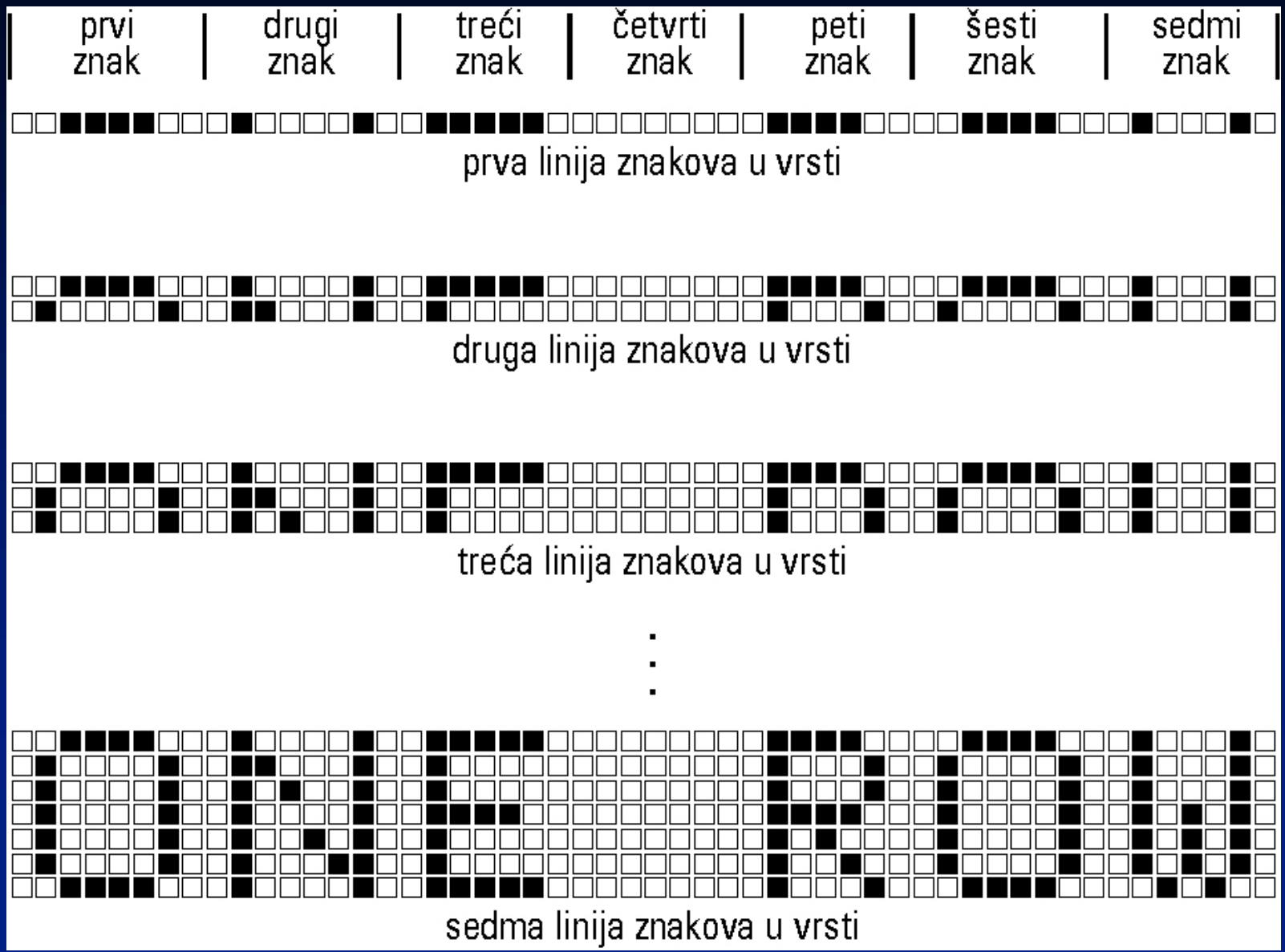
Ulazi i izlazi karakter generatora



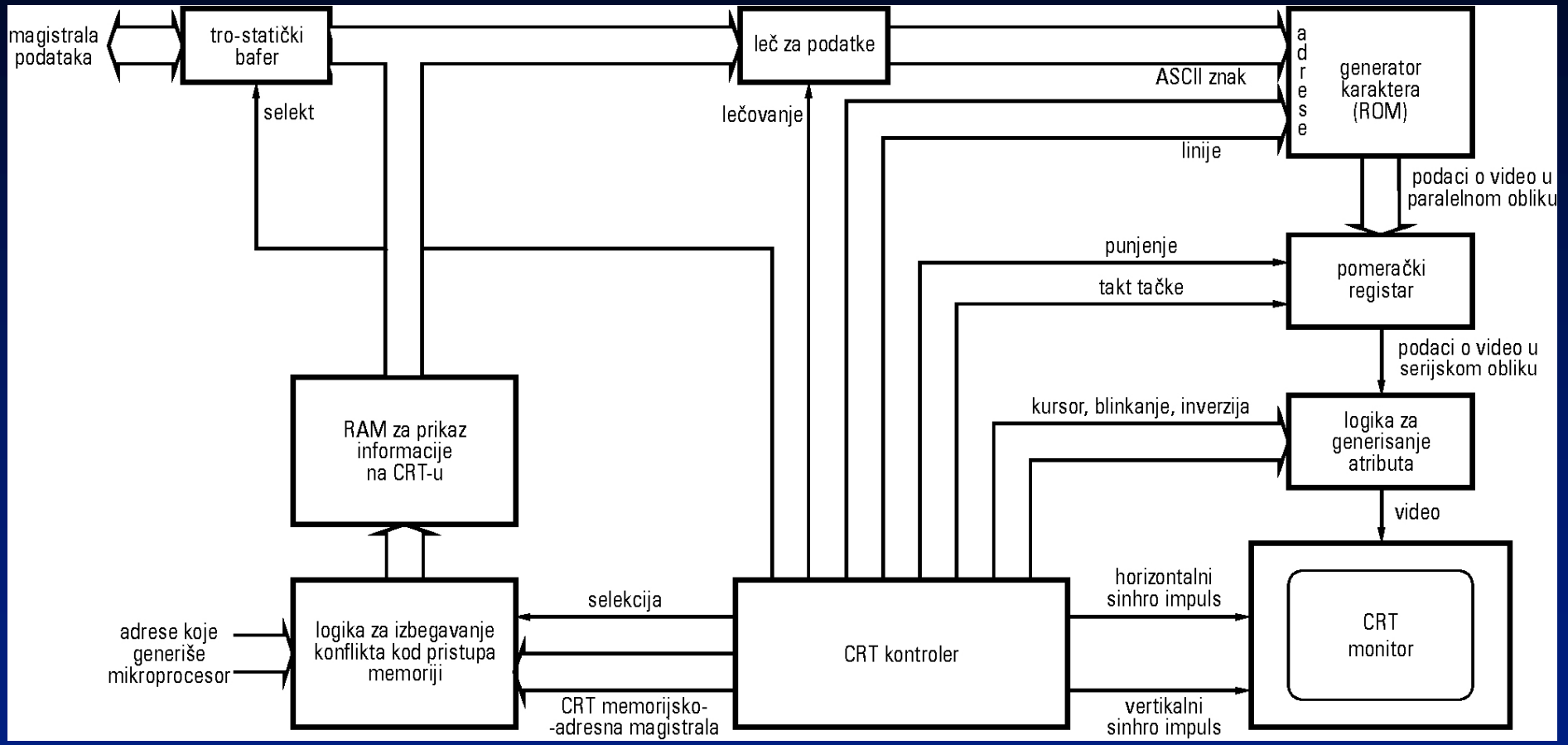
Načini generisanja video informacije



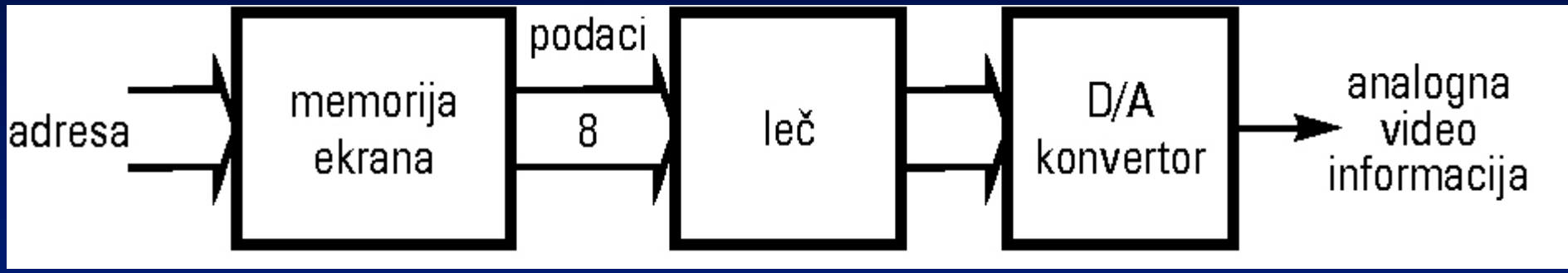
Sekvenca prikaza



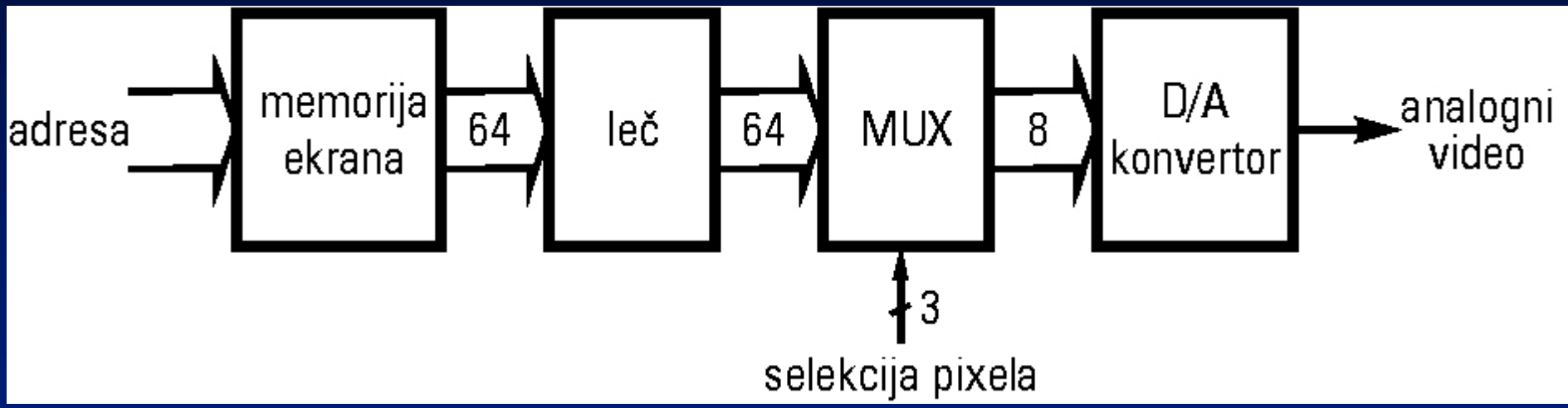
Kompletna elektronika CRT prikaza



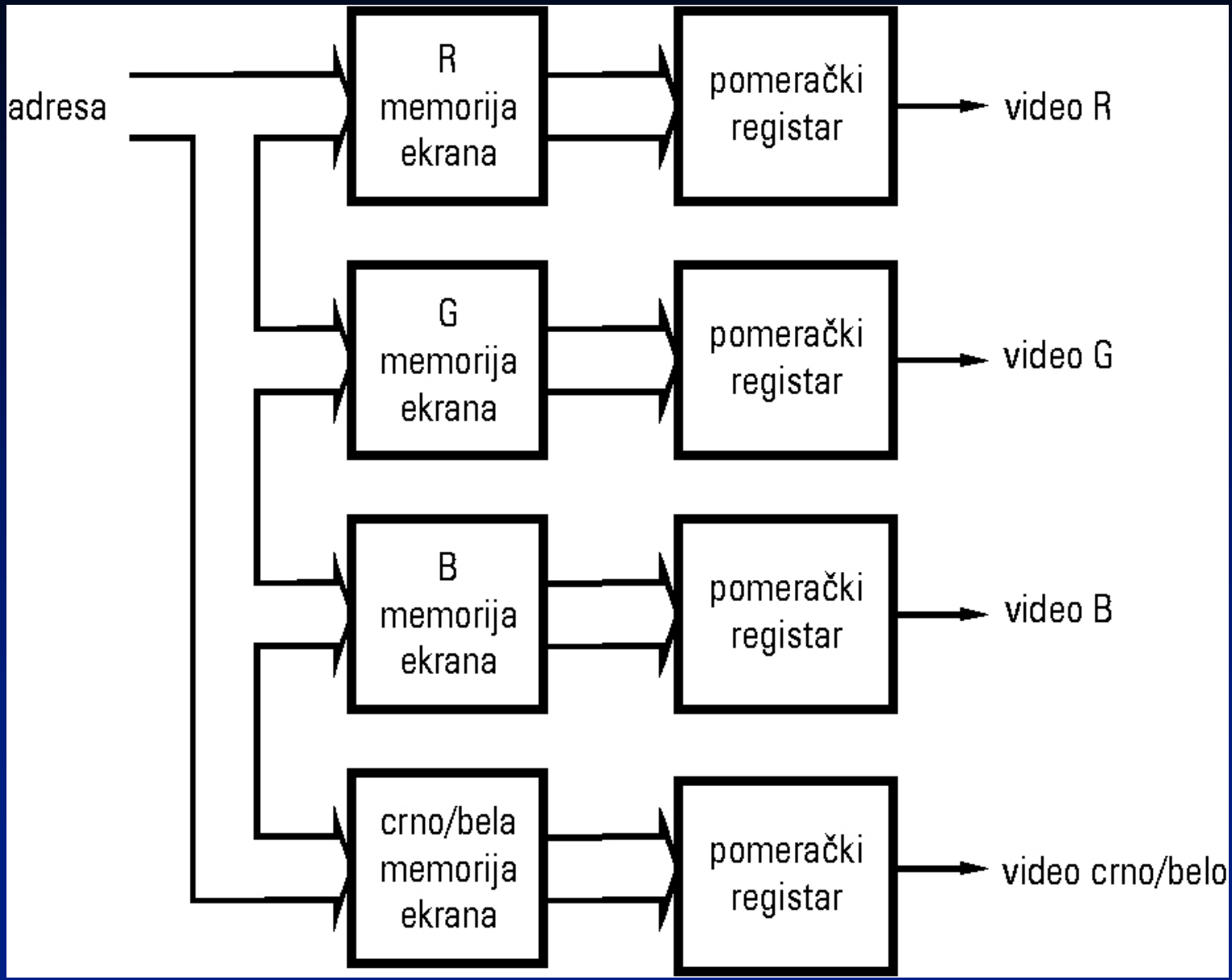
Monohromatski video grafički prikaz



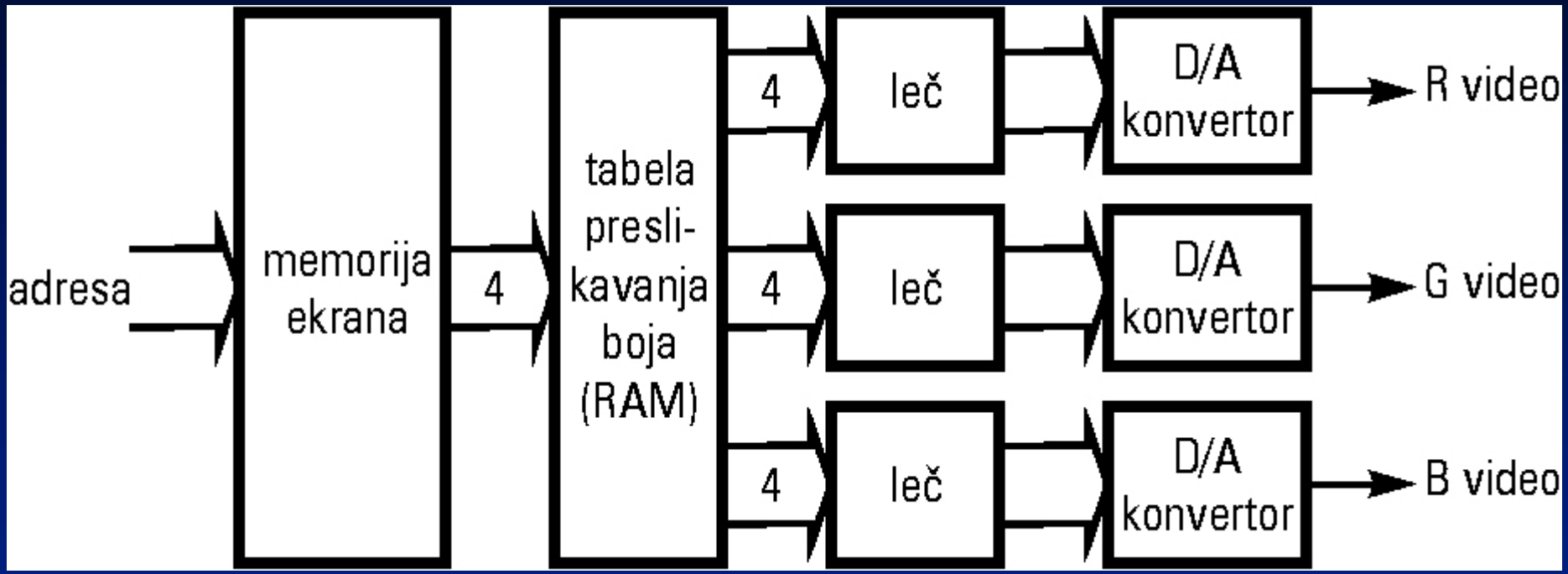
Grafički prikaz sa multipleksiranim izlazom



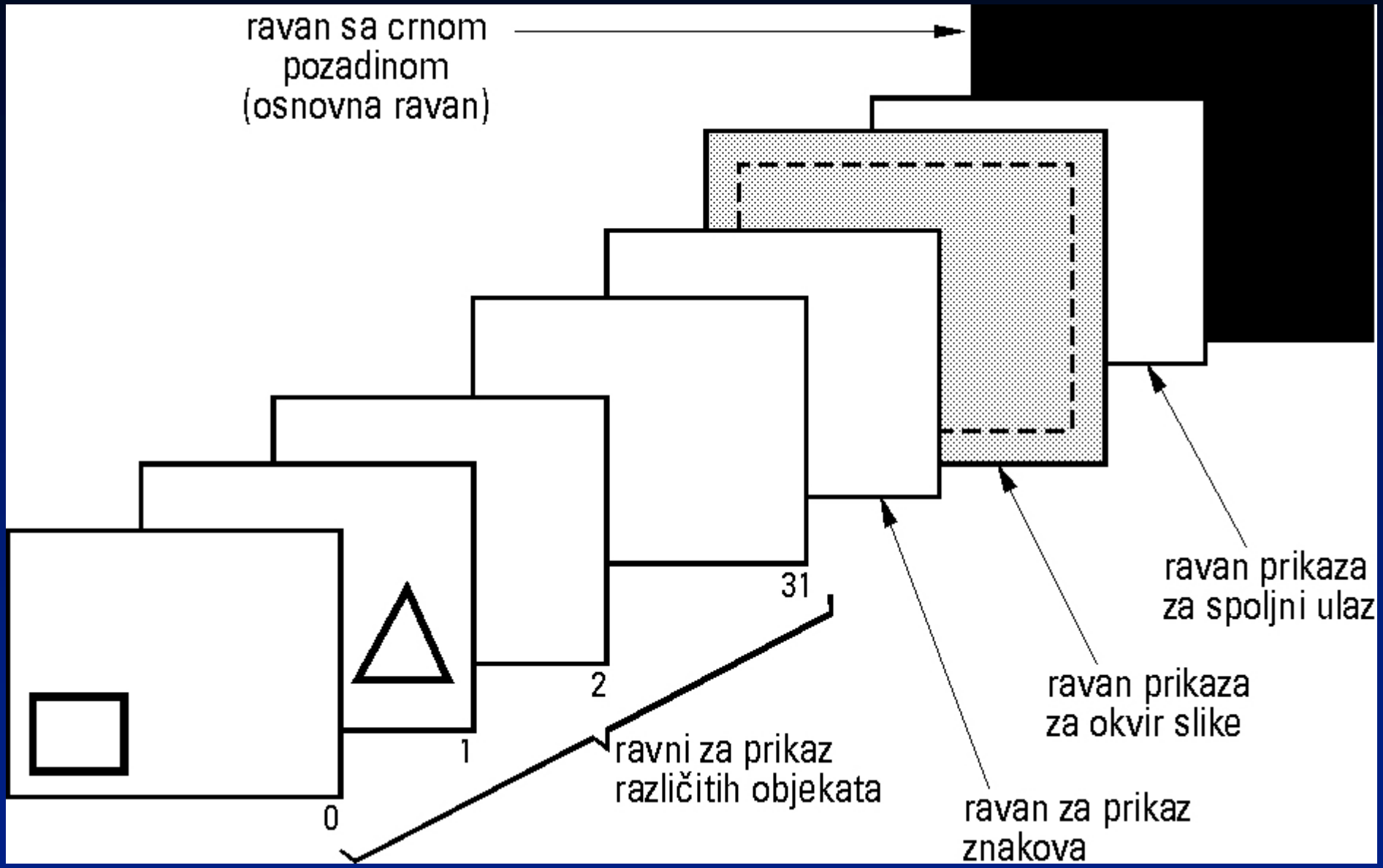
Video generator CRT kontrolera u boji



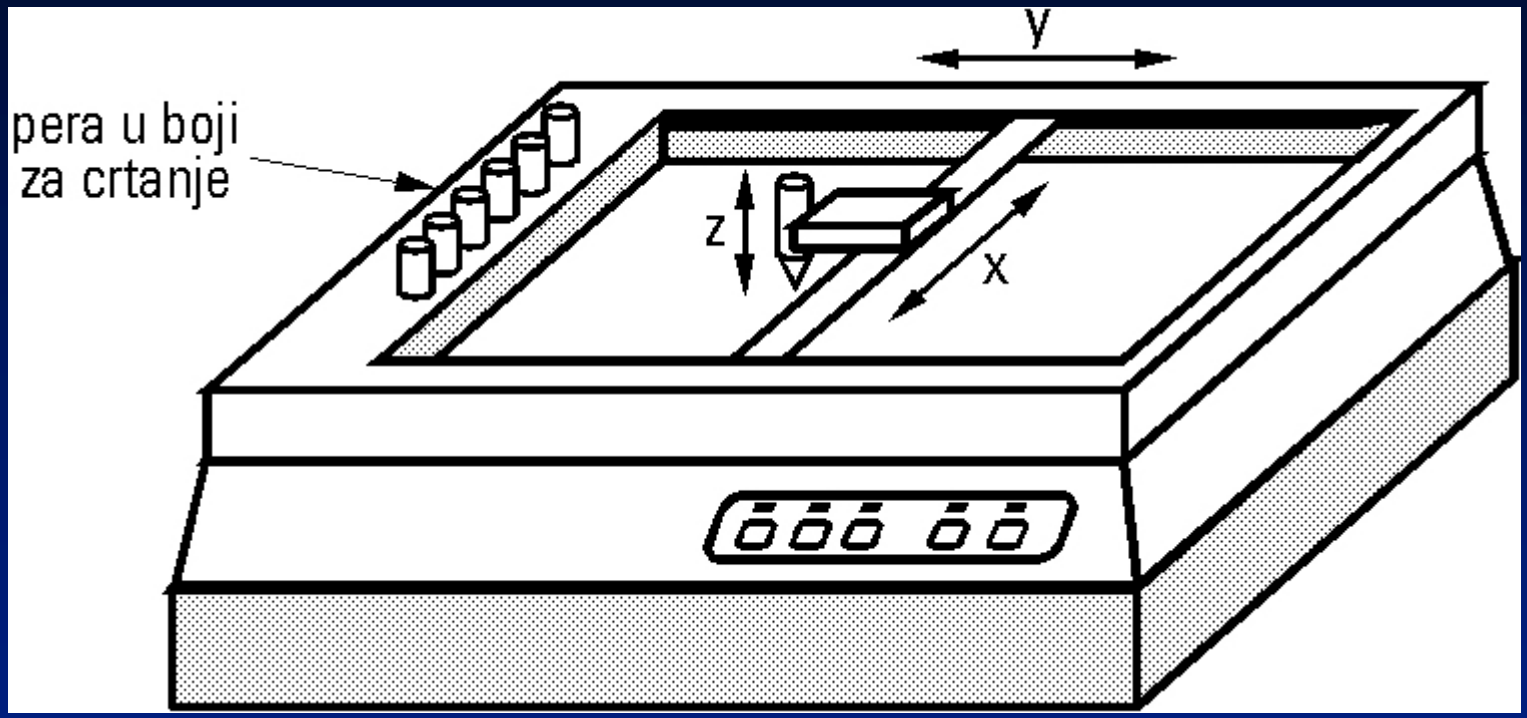
Princip korišćenja tabela preslikavanja



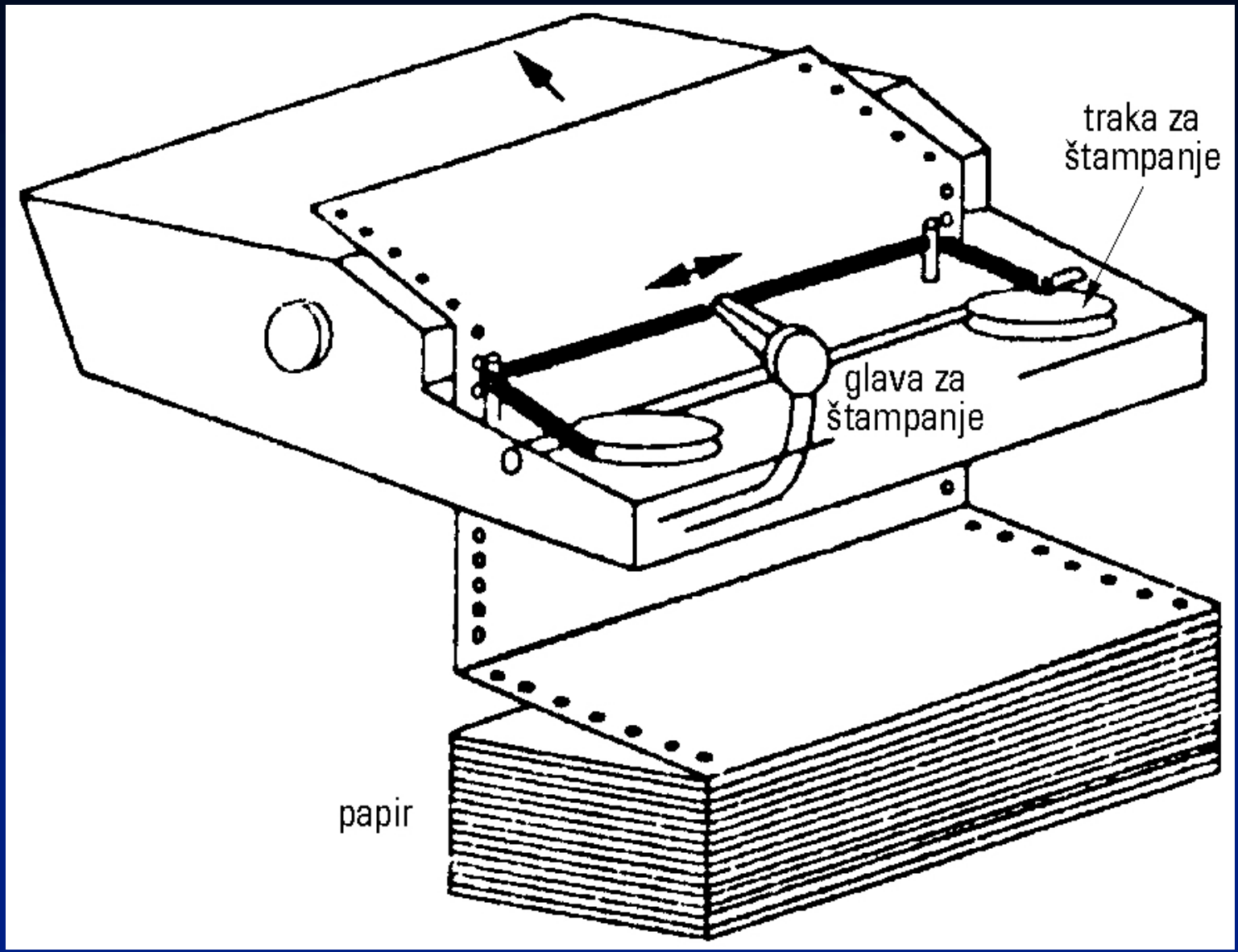
Princip prikaza kod video displej generatora



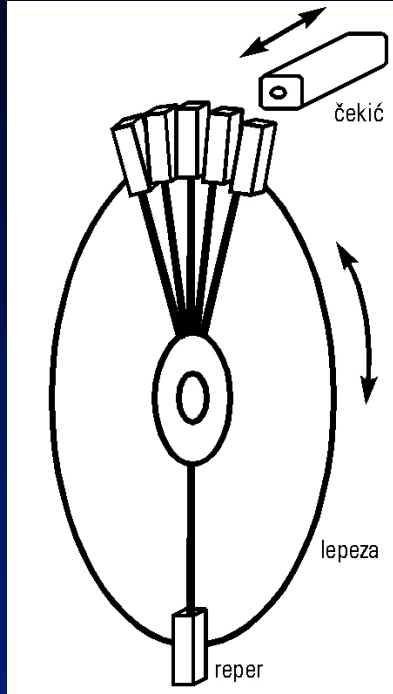
X-Y ploter



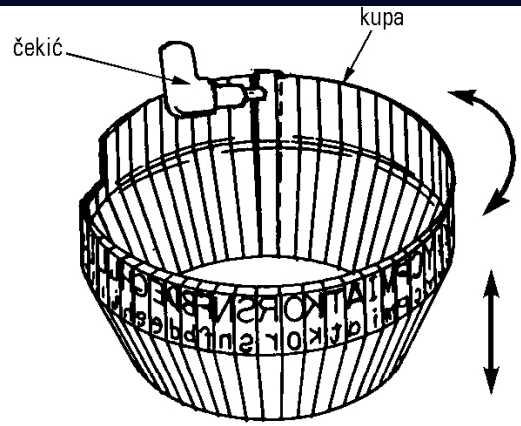
Opšti izgled serijskog štampača



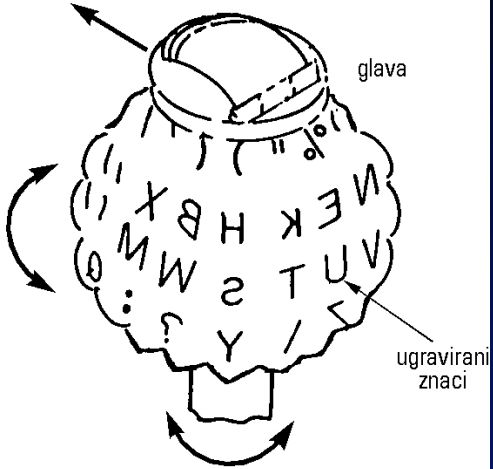
Tipovi serijskog štampača



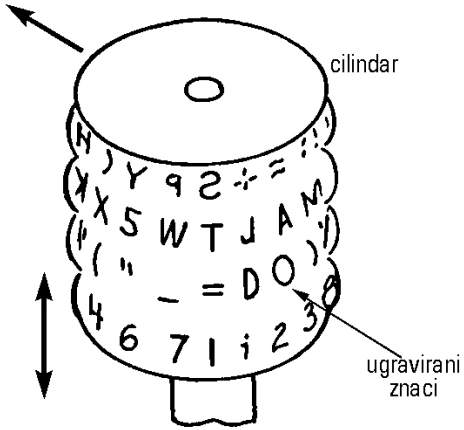
a) serijski štampač sa lepezom



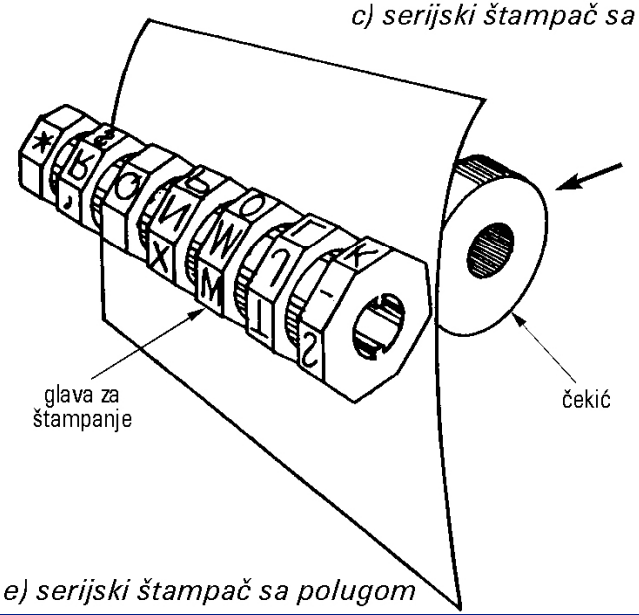
b) serijski štampač sa kupom



c) serijski štampač sa loptom

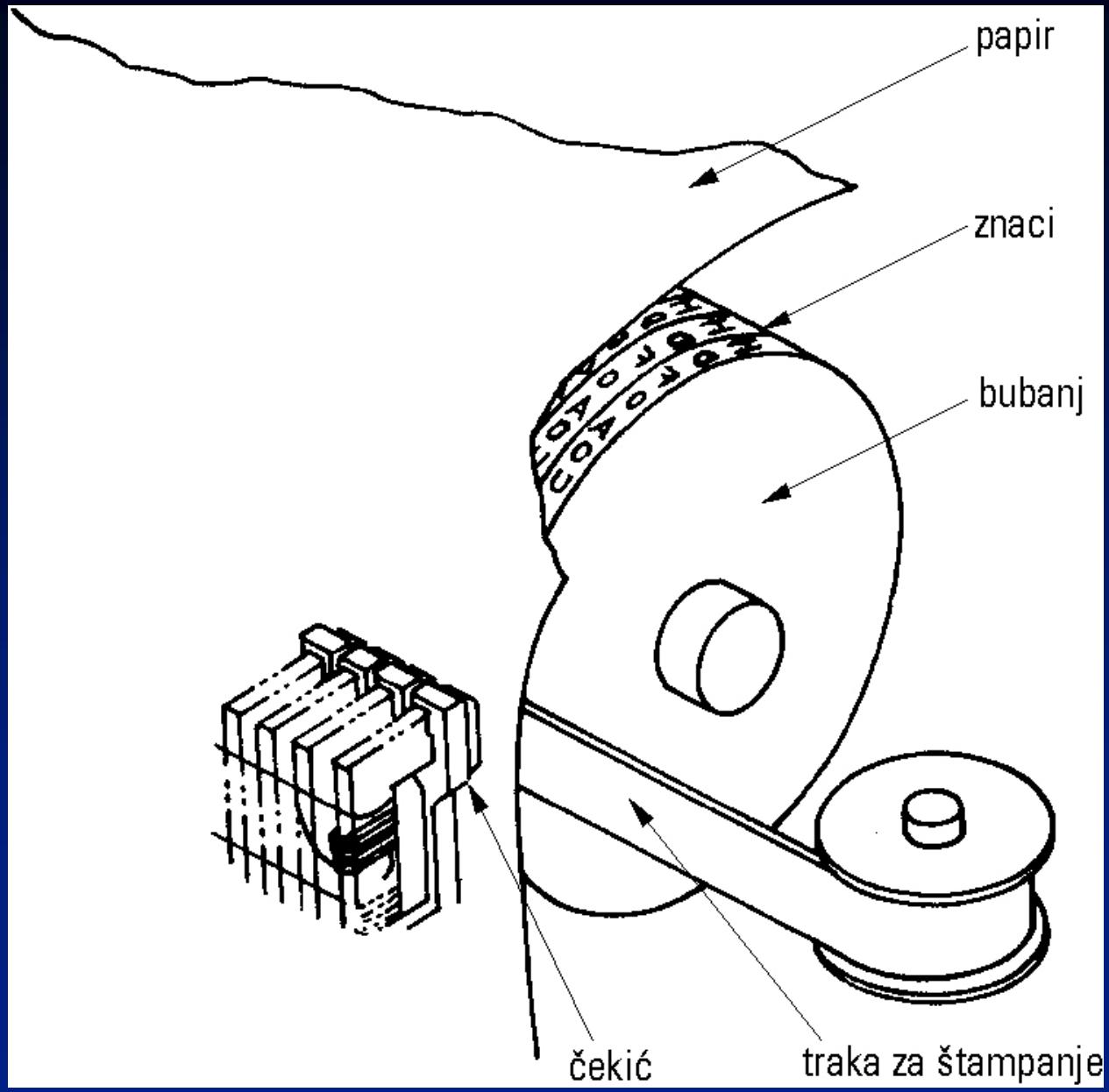


d) serijski štampač sa cilindrom

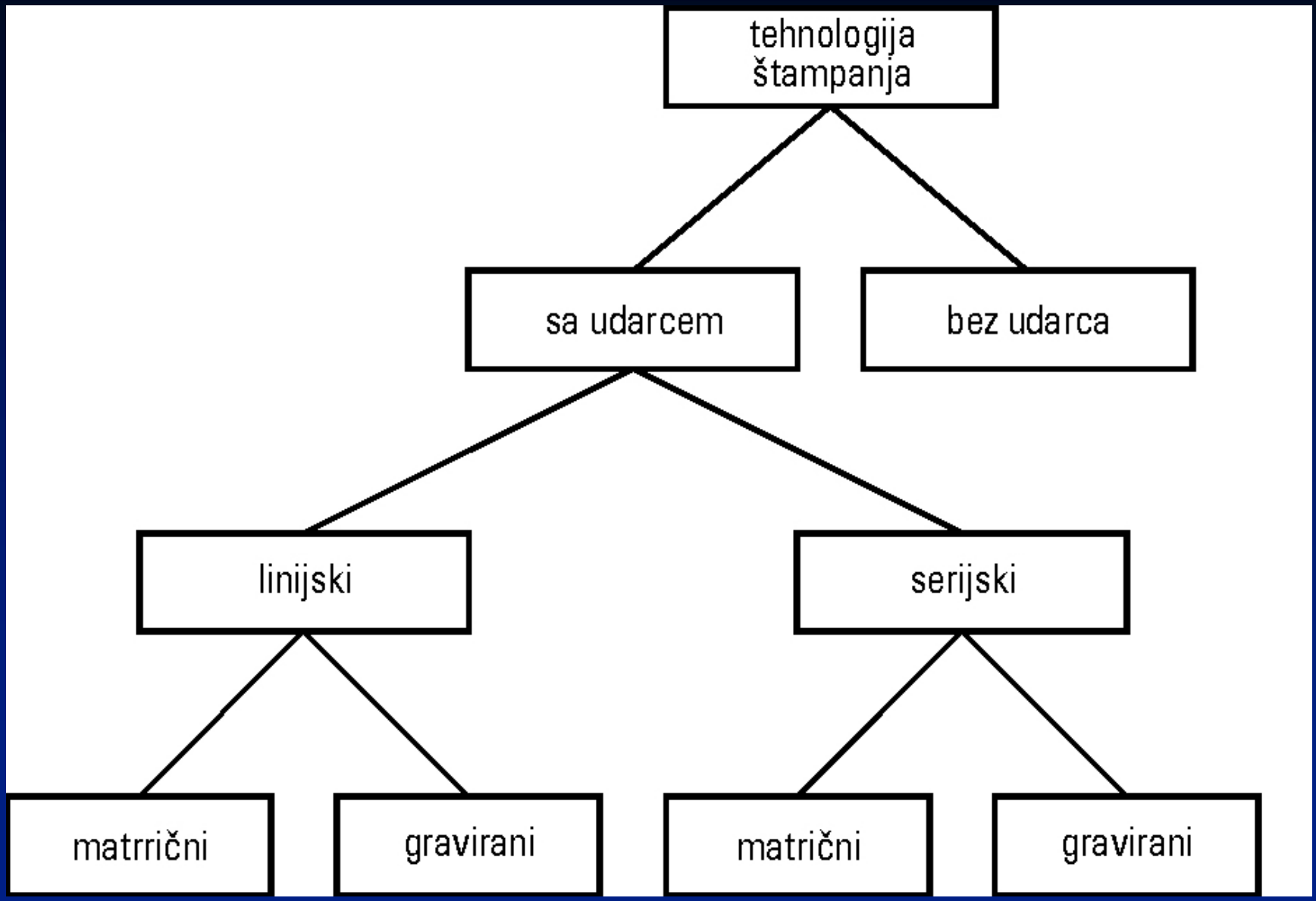


e) serijski štampač sa polugom

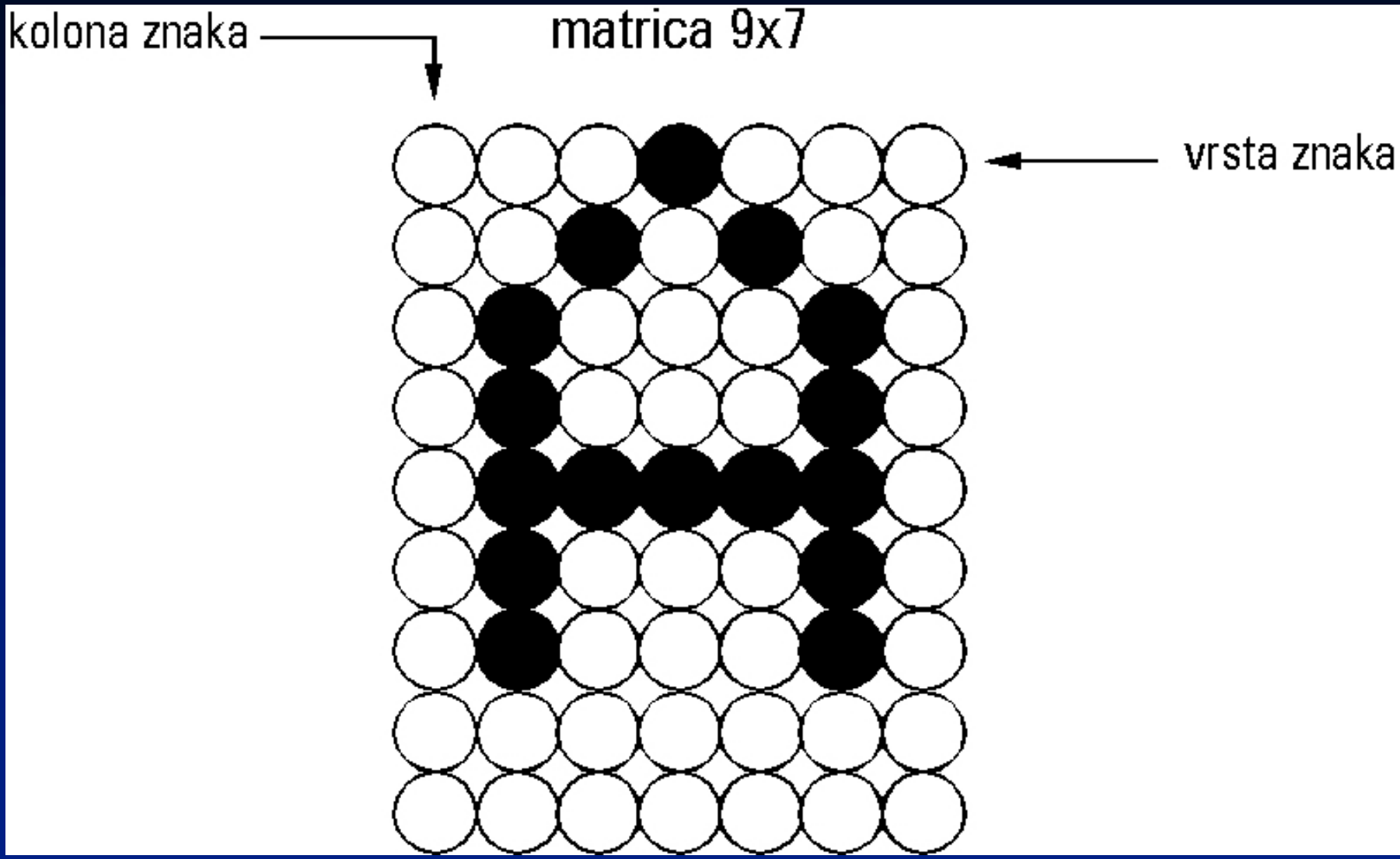
Linijski štampač



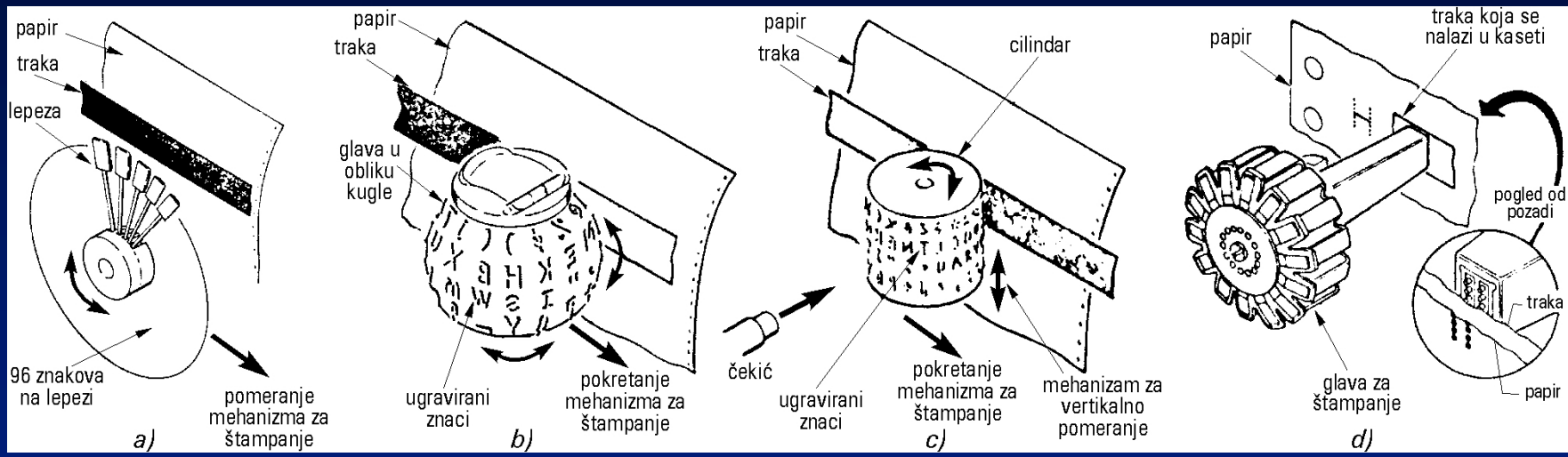
Tehnologije štampanja



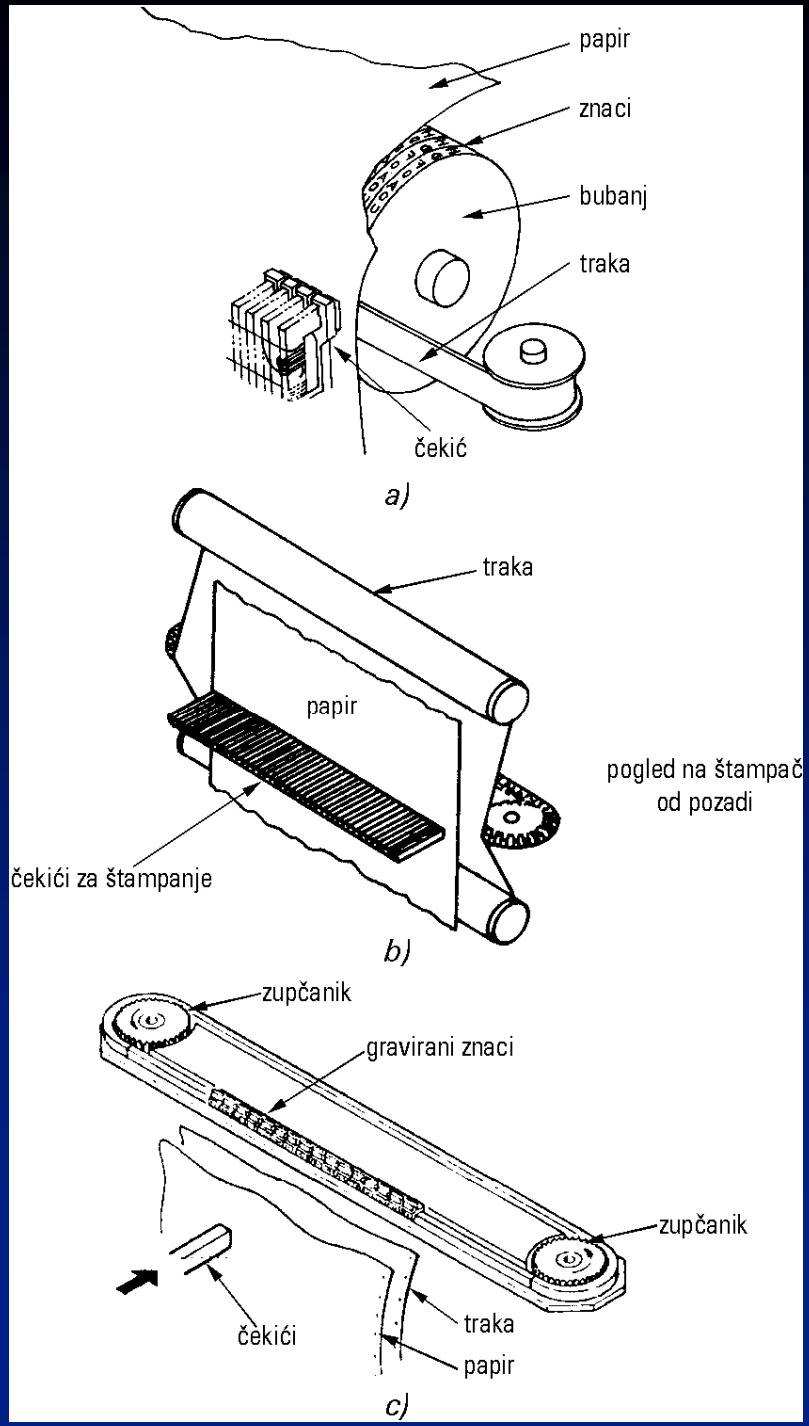
Tačkasto štampanje



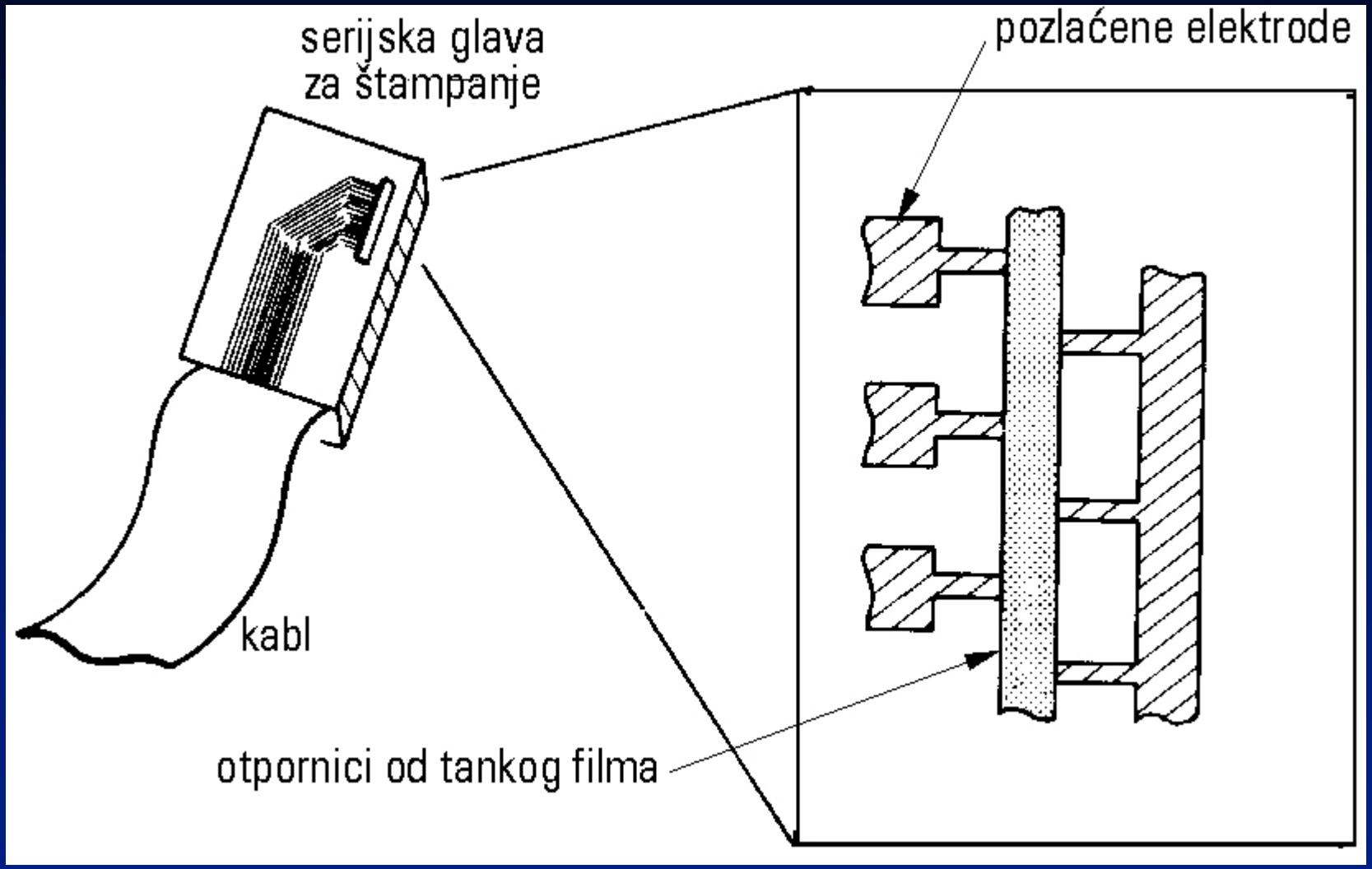
Impact štampači koji štampaju sa prednje strane



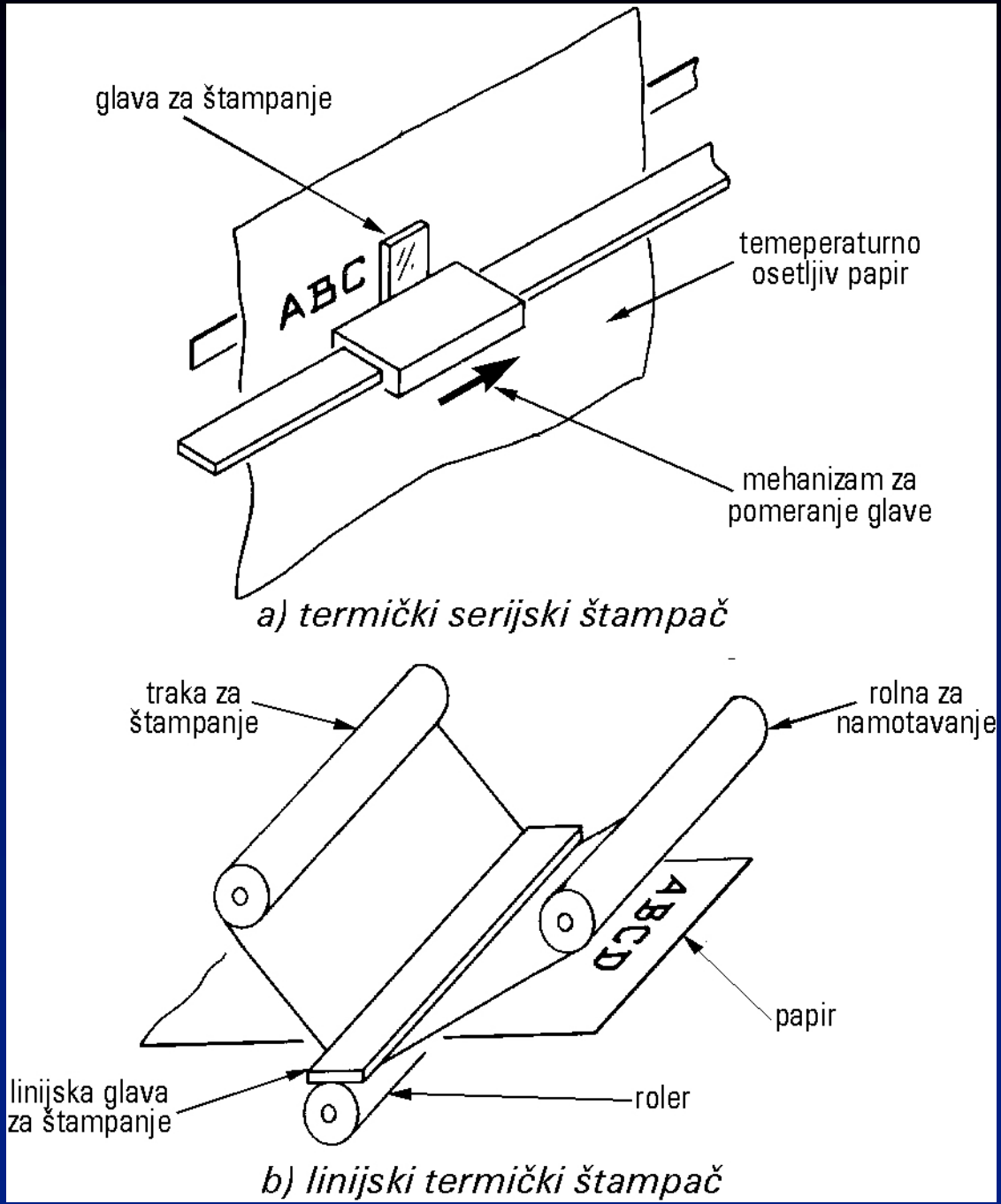
Rear-striking štampači



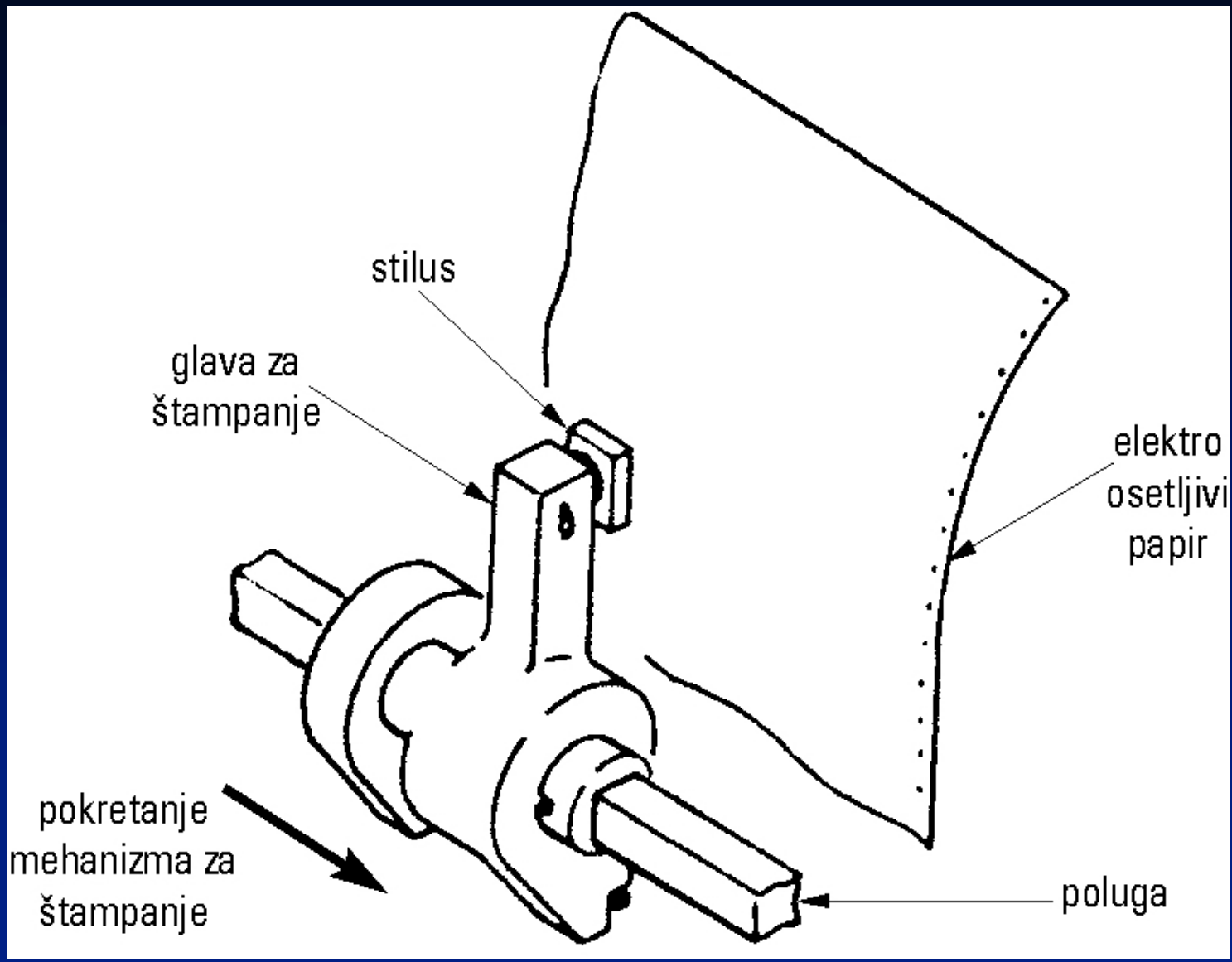
Termičko štampanje



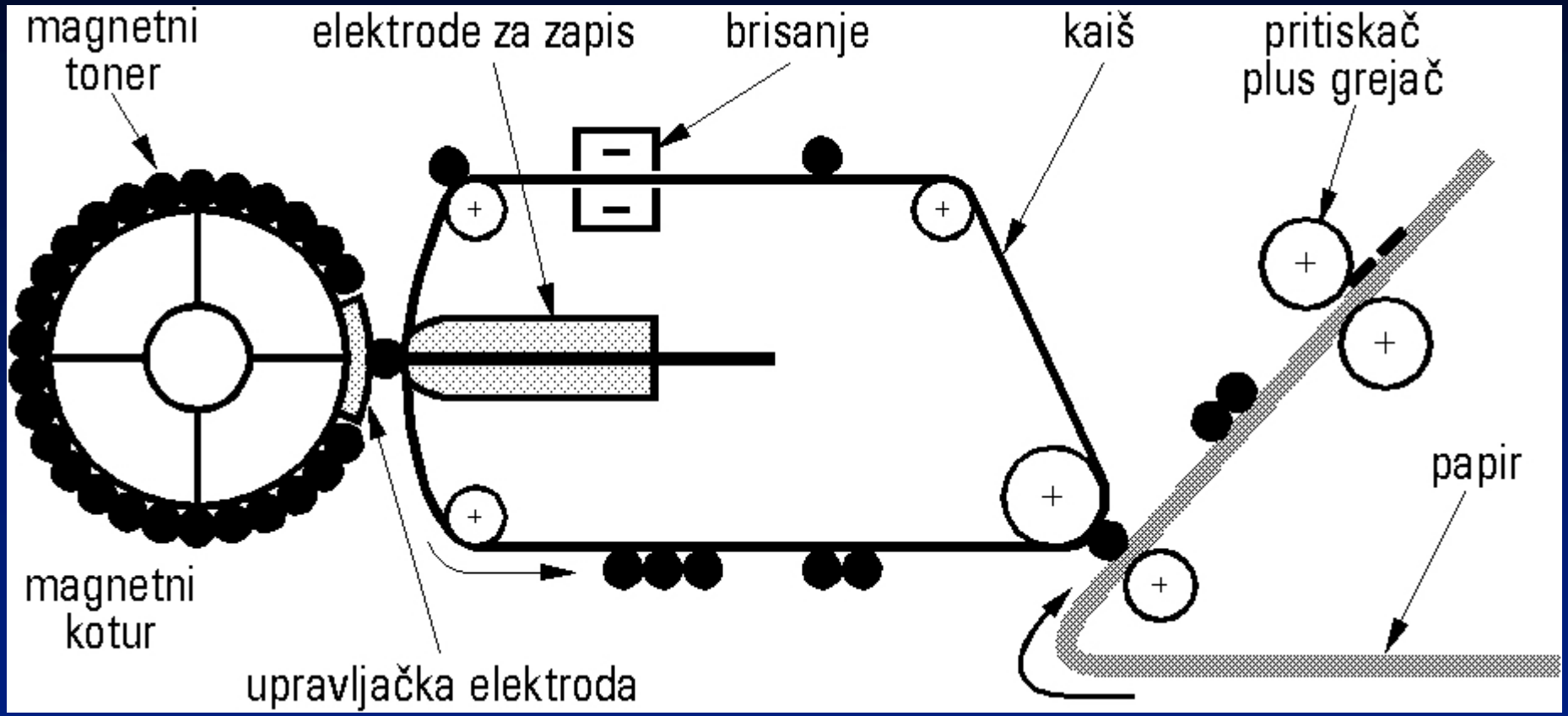
Termički štampači



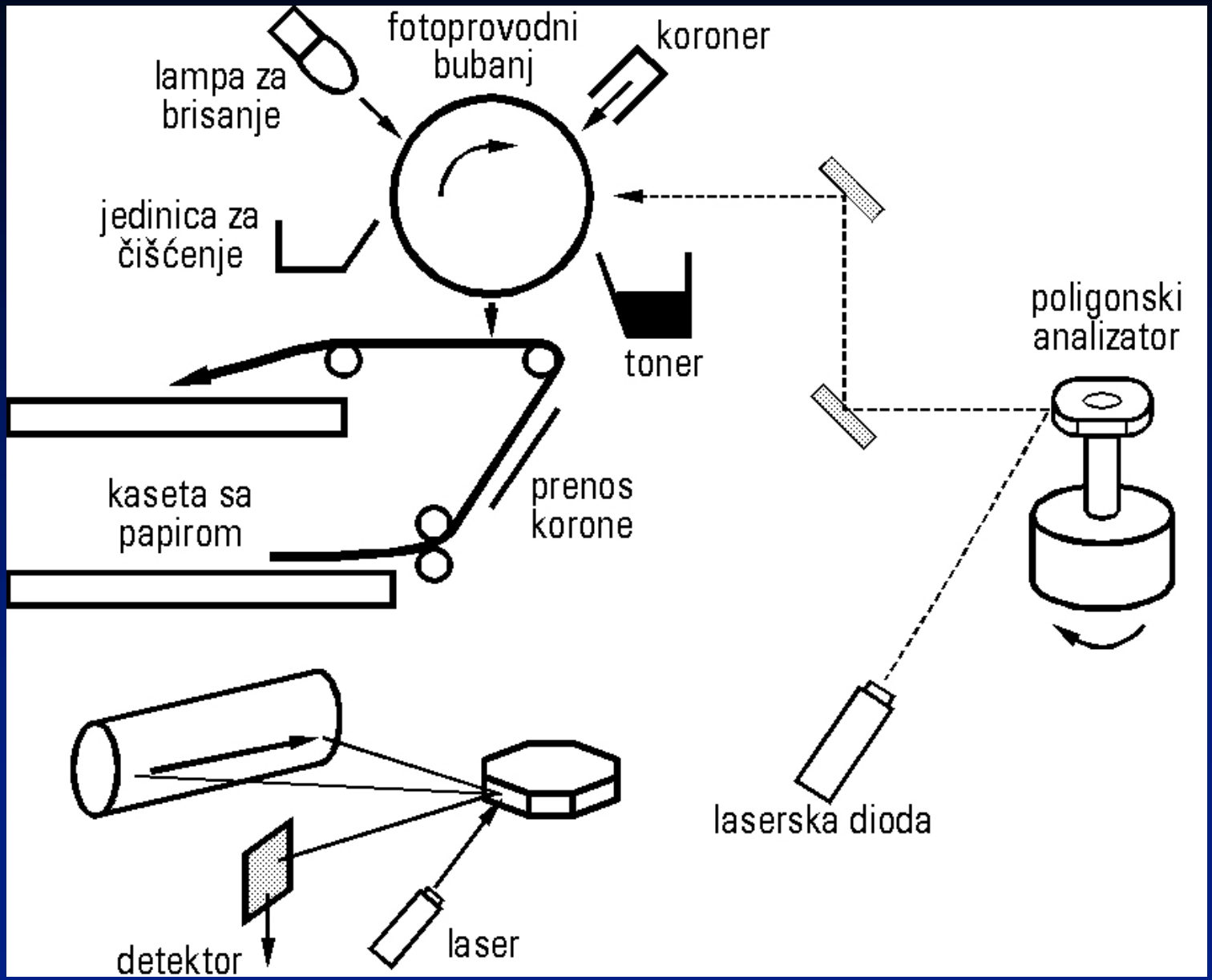
Štampač koji koristi elektro-osetljivi papir



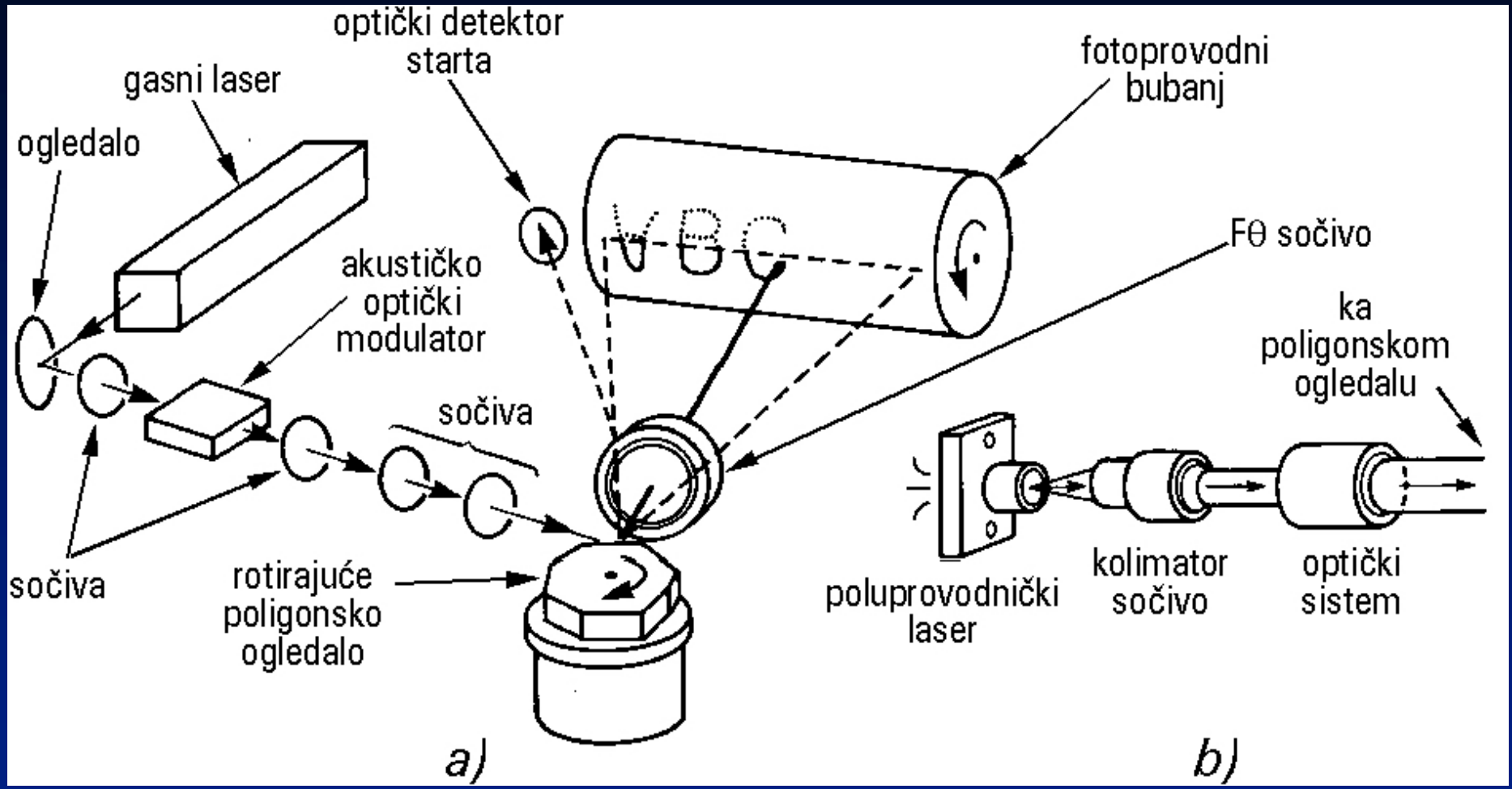
Elektro-statičko štampanje



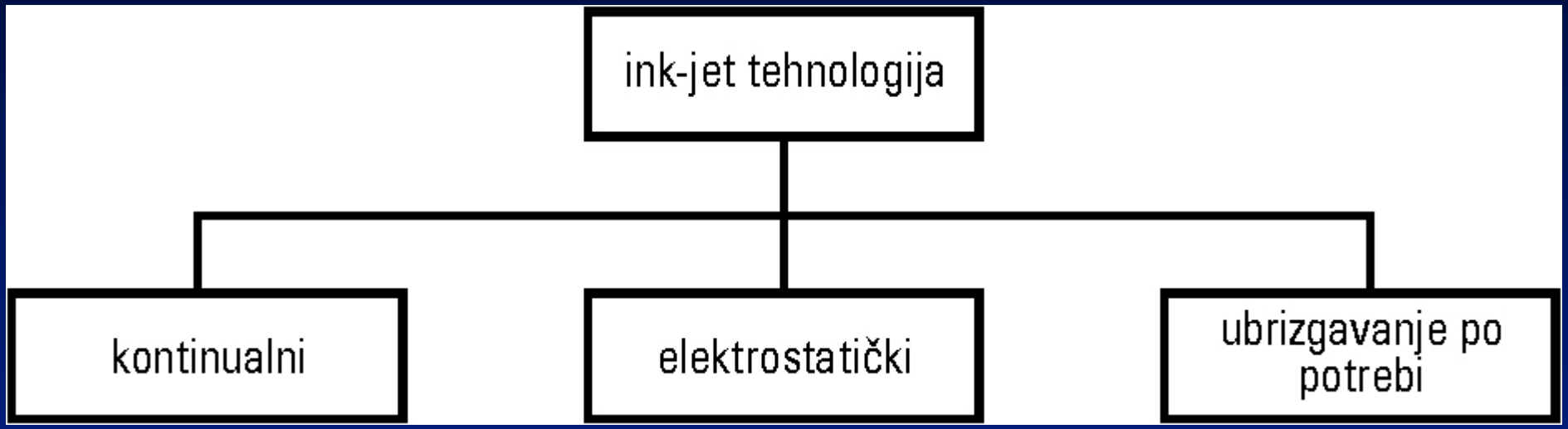
Konstrukcija tipičnog laserskog štampača



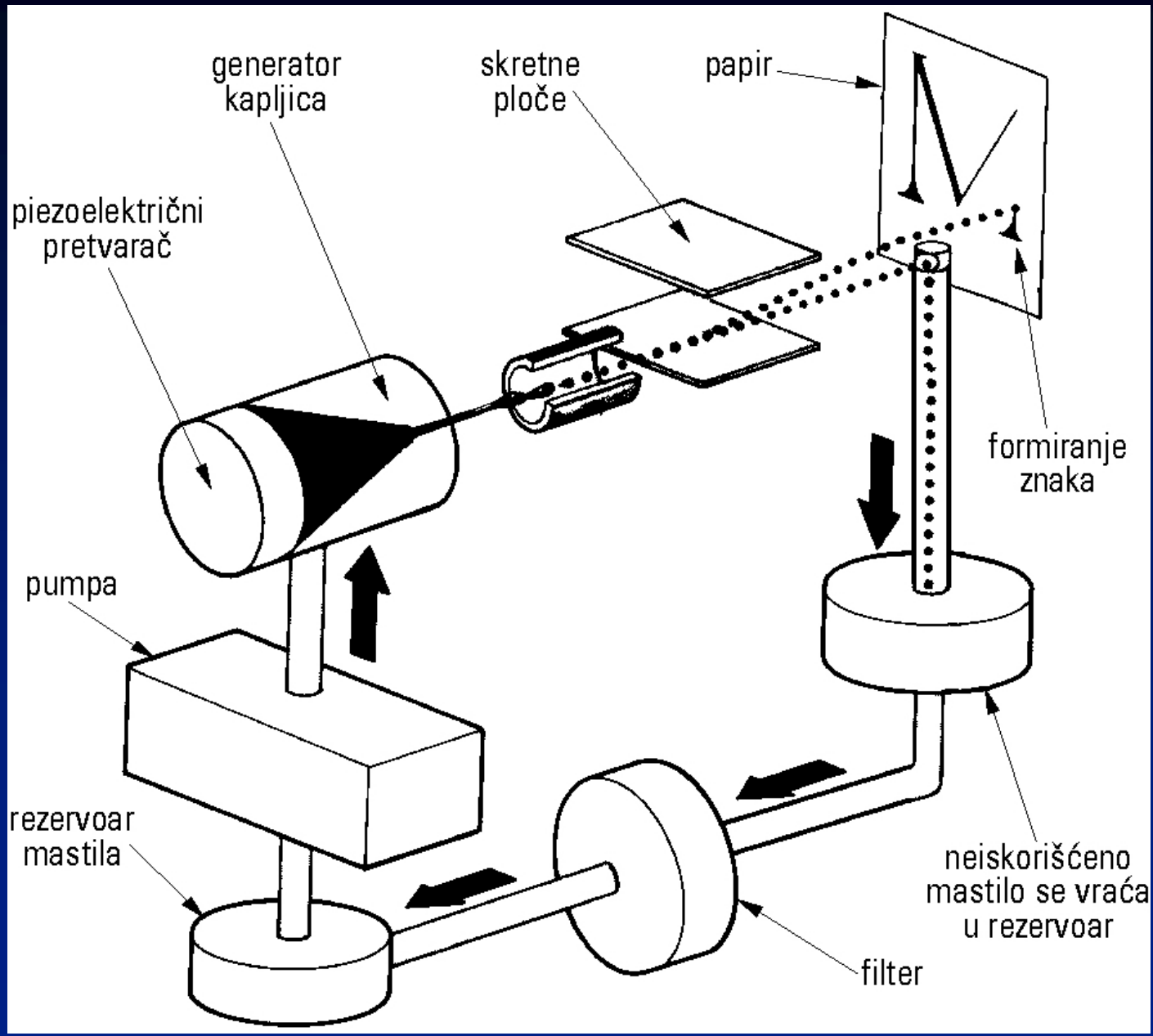
Verzije laserske glave



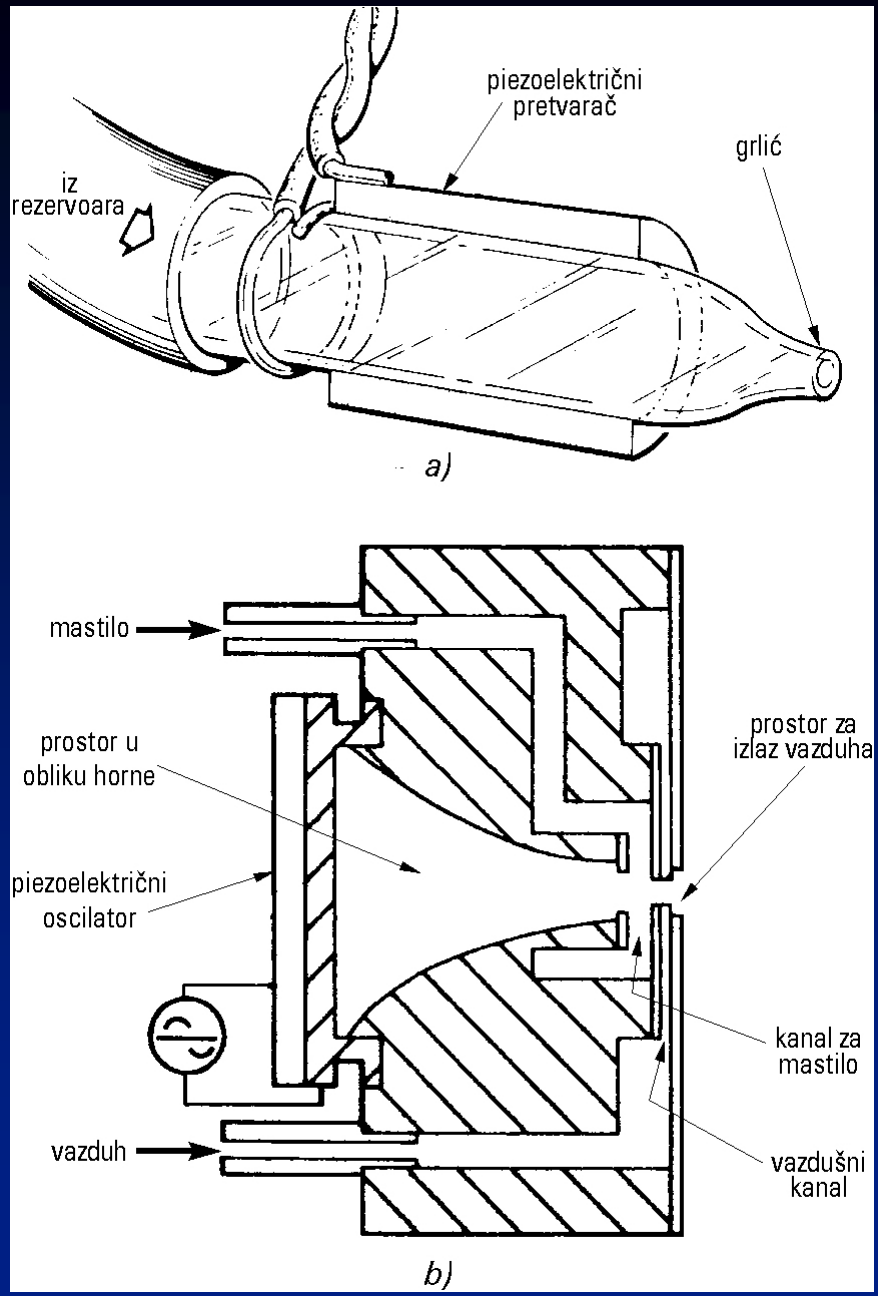
Ink-Jet tehnologija



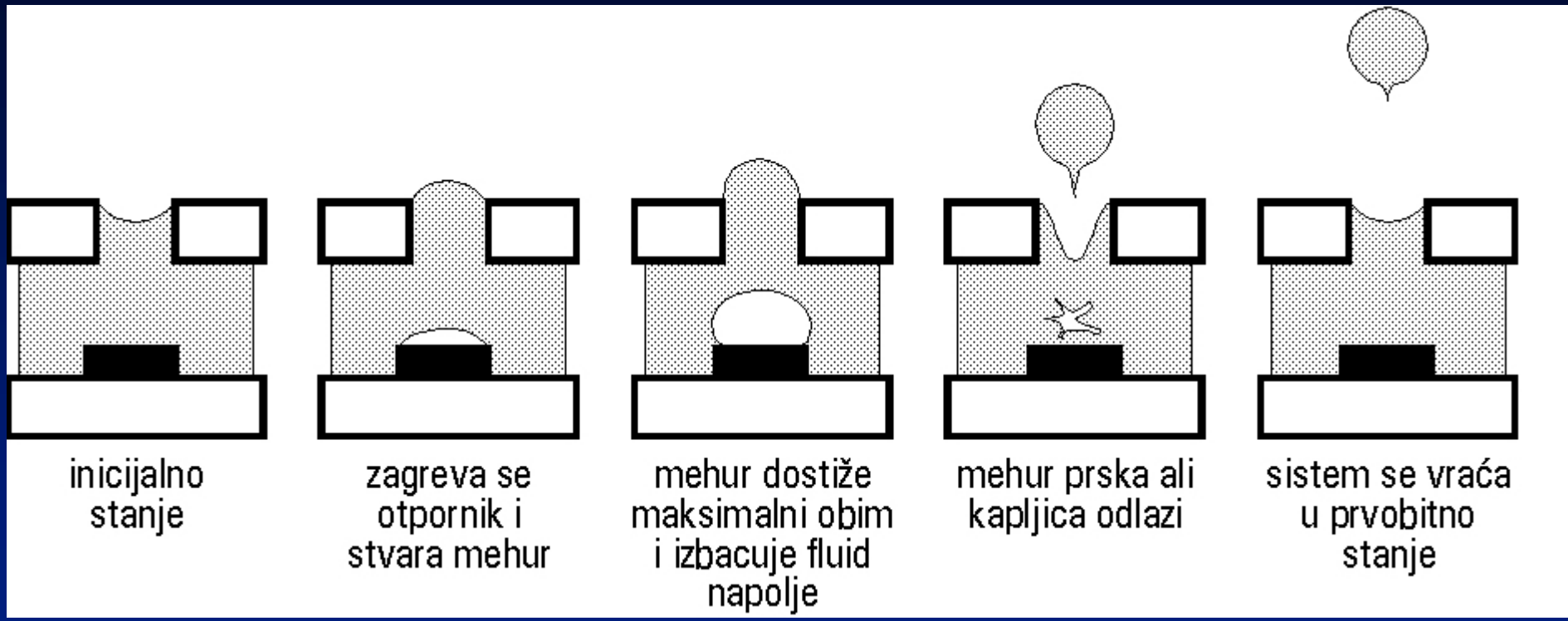
Kontinualni *ink-jet* štampači



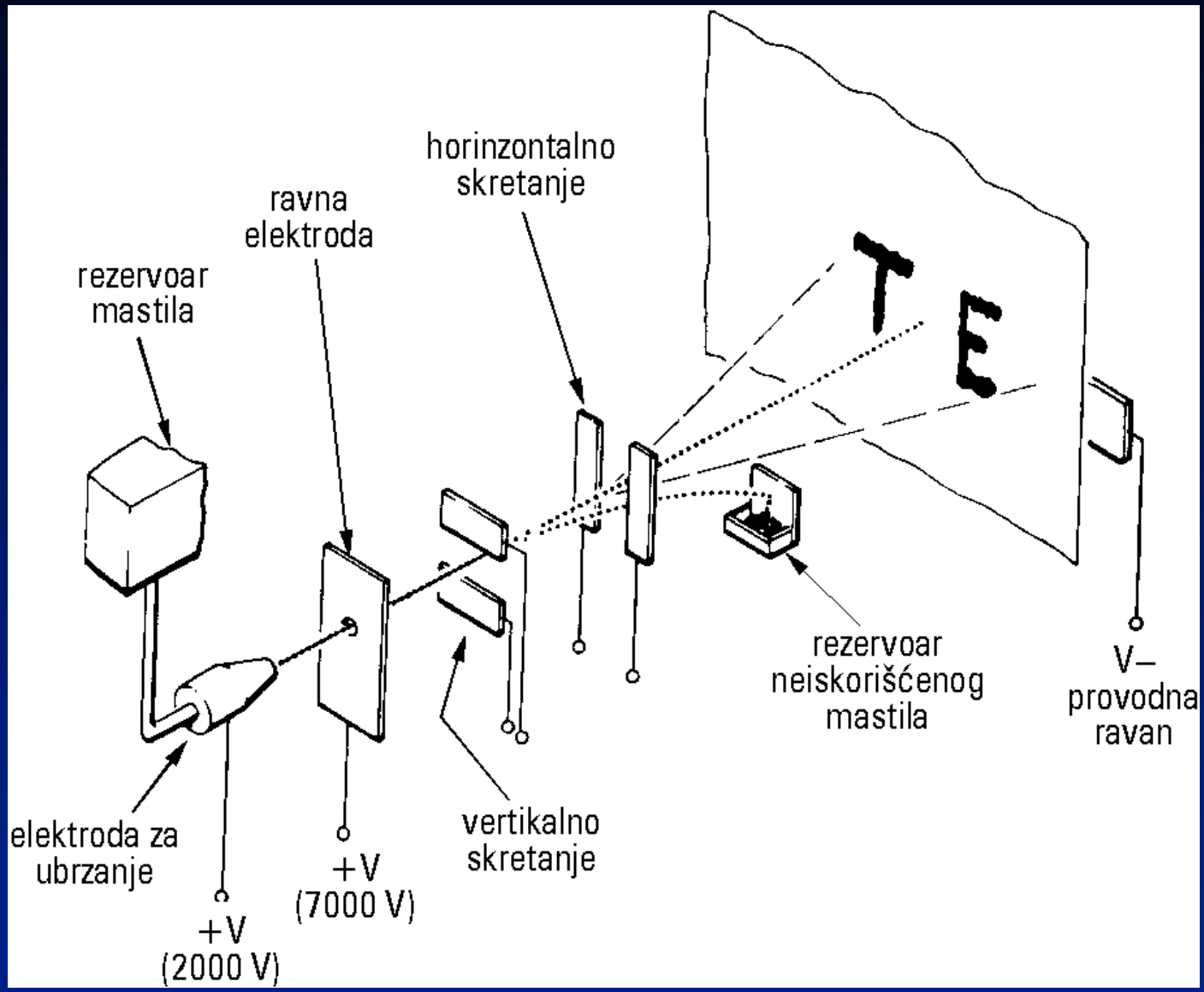
Ink-Jet štampanje sa piezo-električnom pobudom



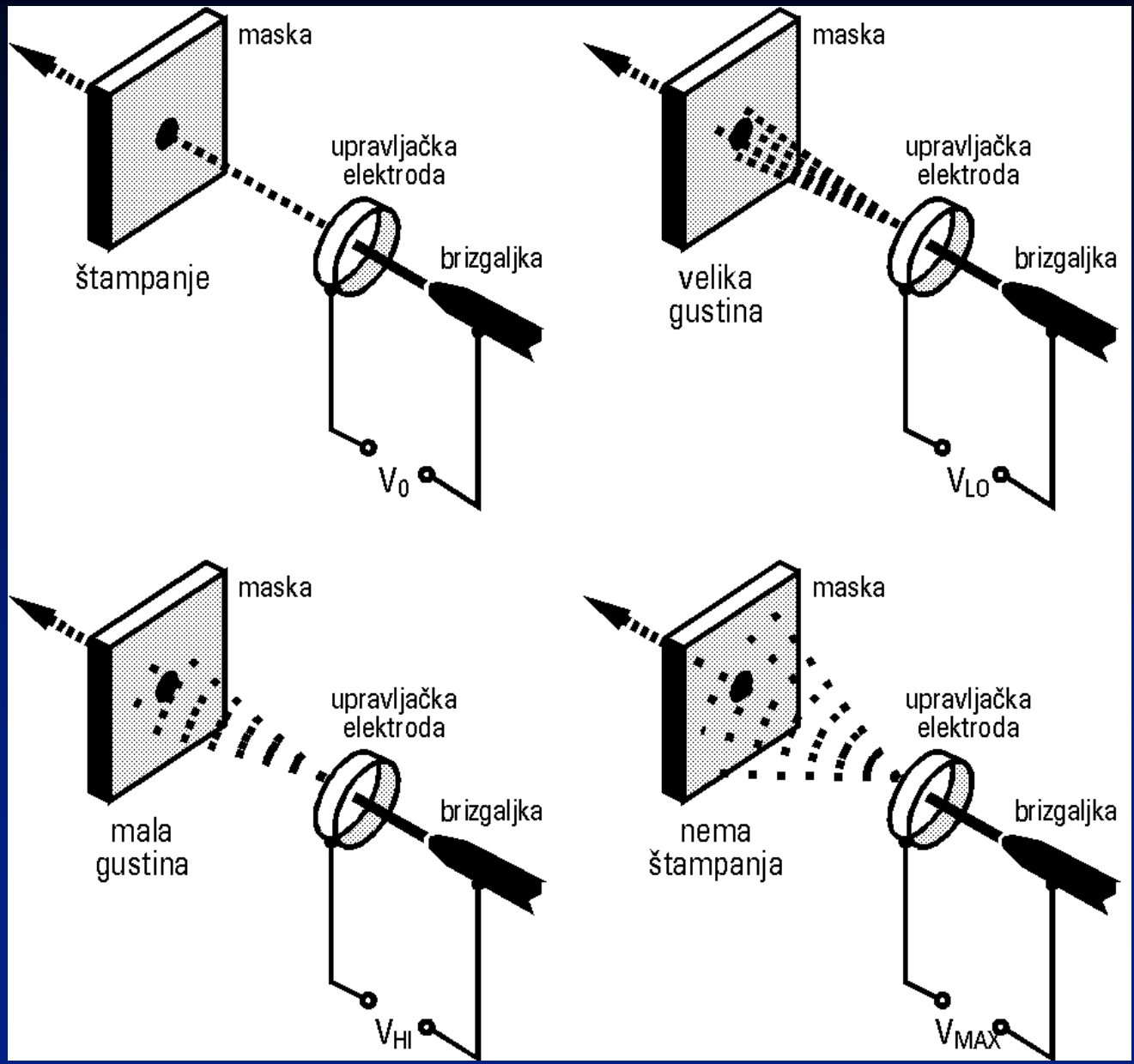
Ink-Jet štampanje sa termičkom pobudom



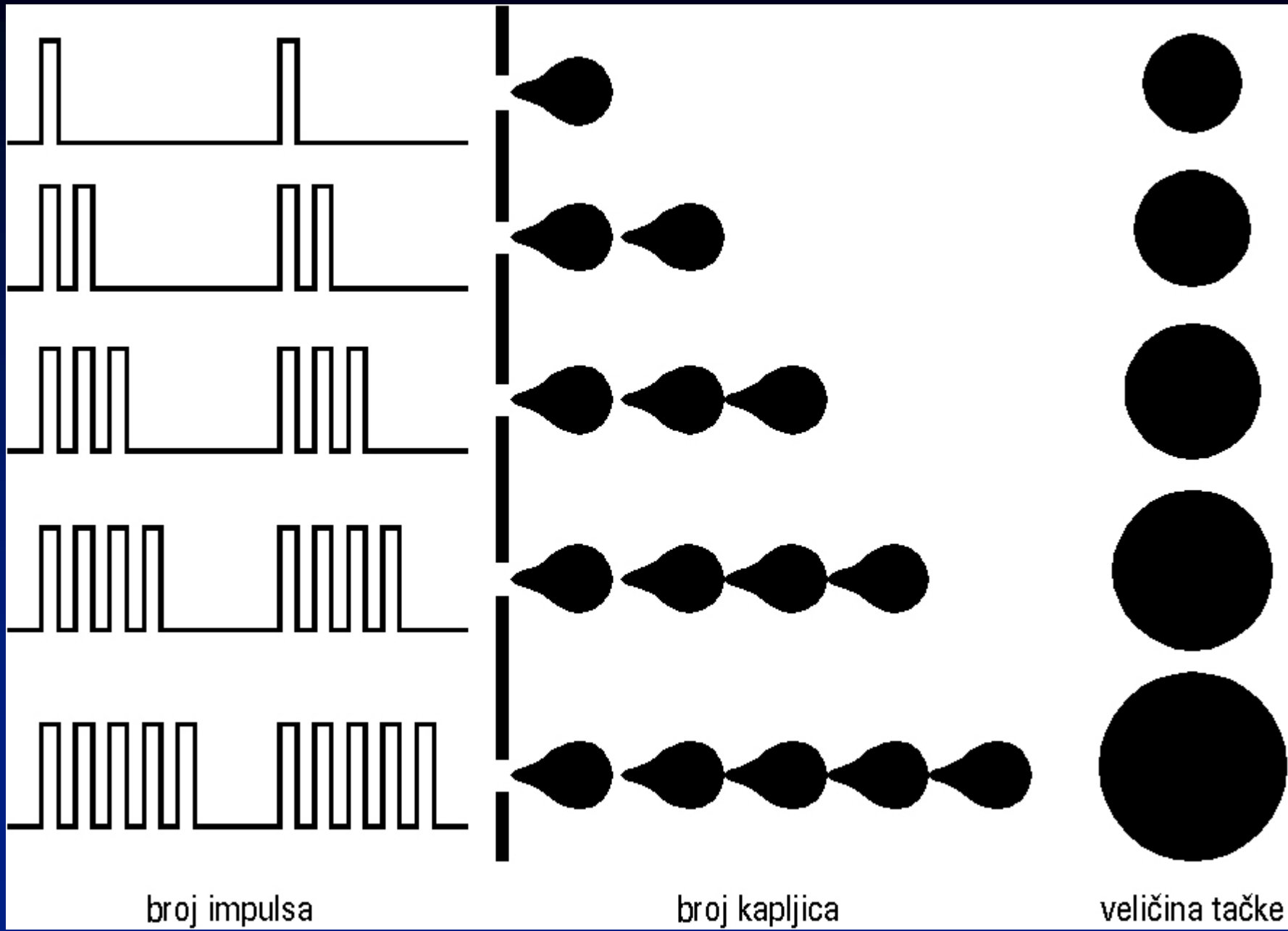
Elektrostatički *ink-jet*



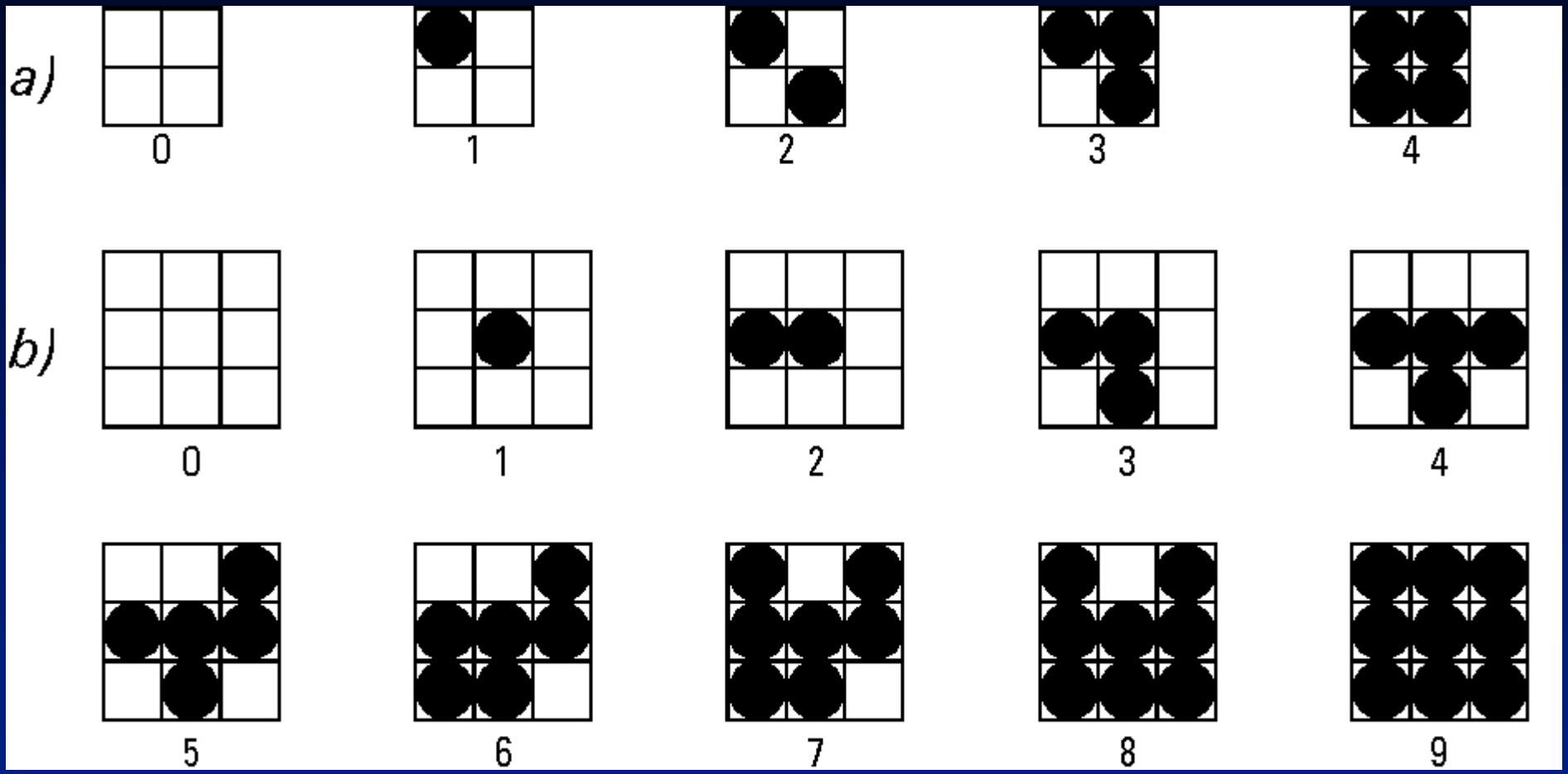
Tehnika *Hertz*-ovog rasipanja



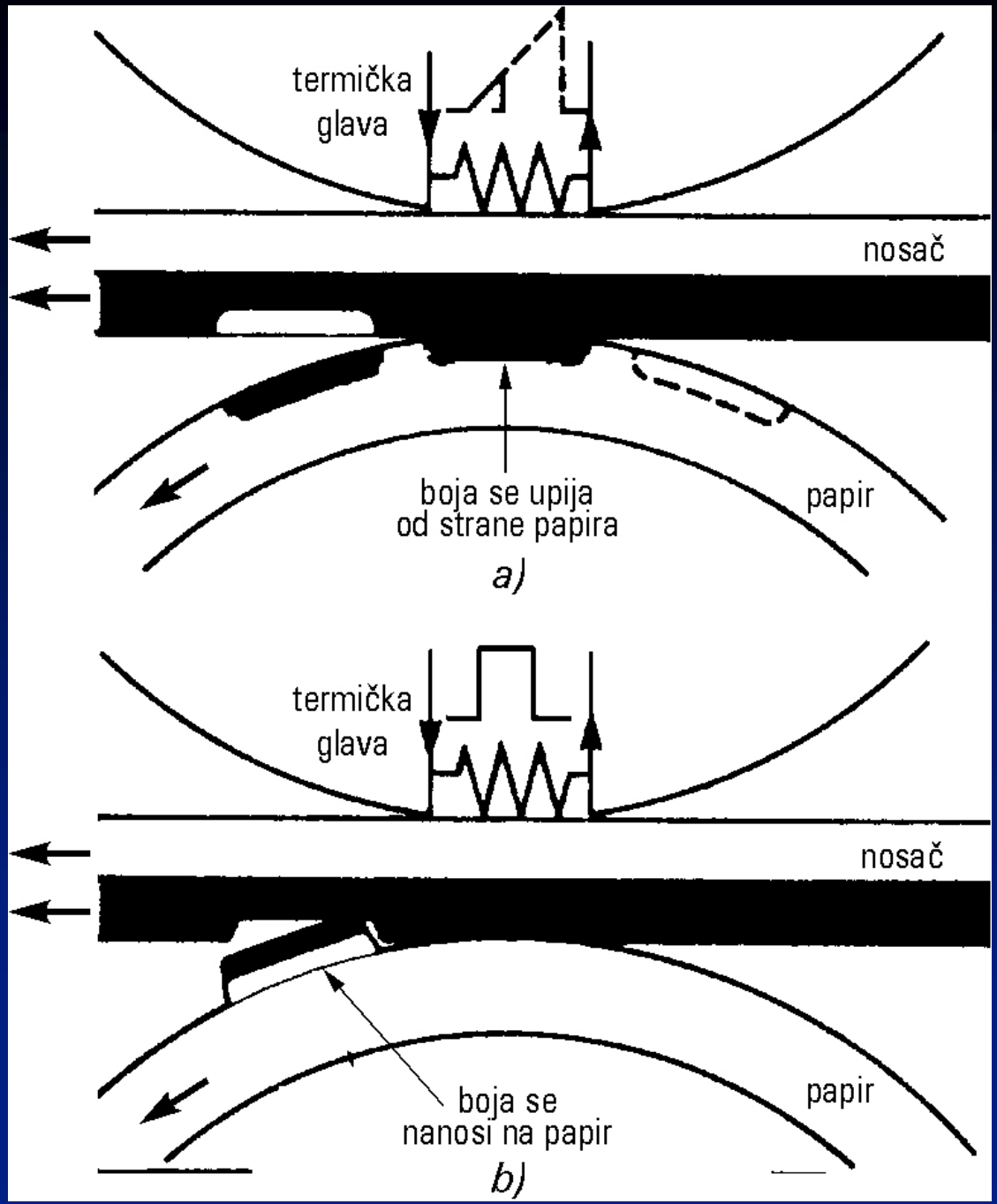
Digitalna regulacija veličine tačke



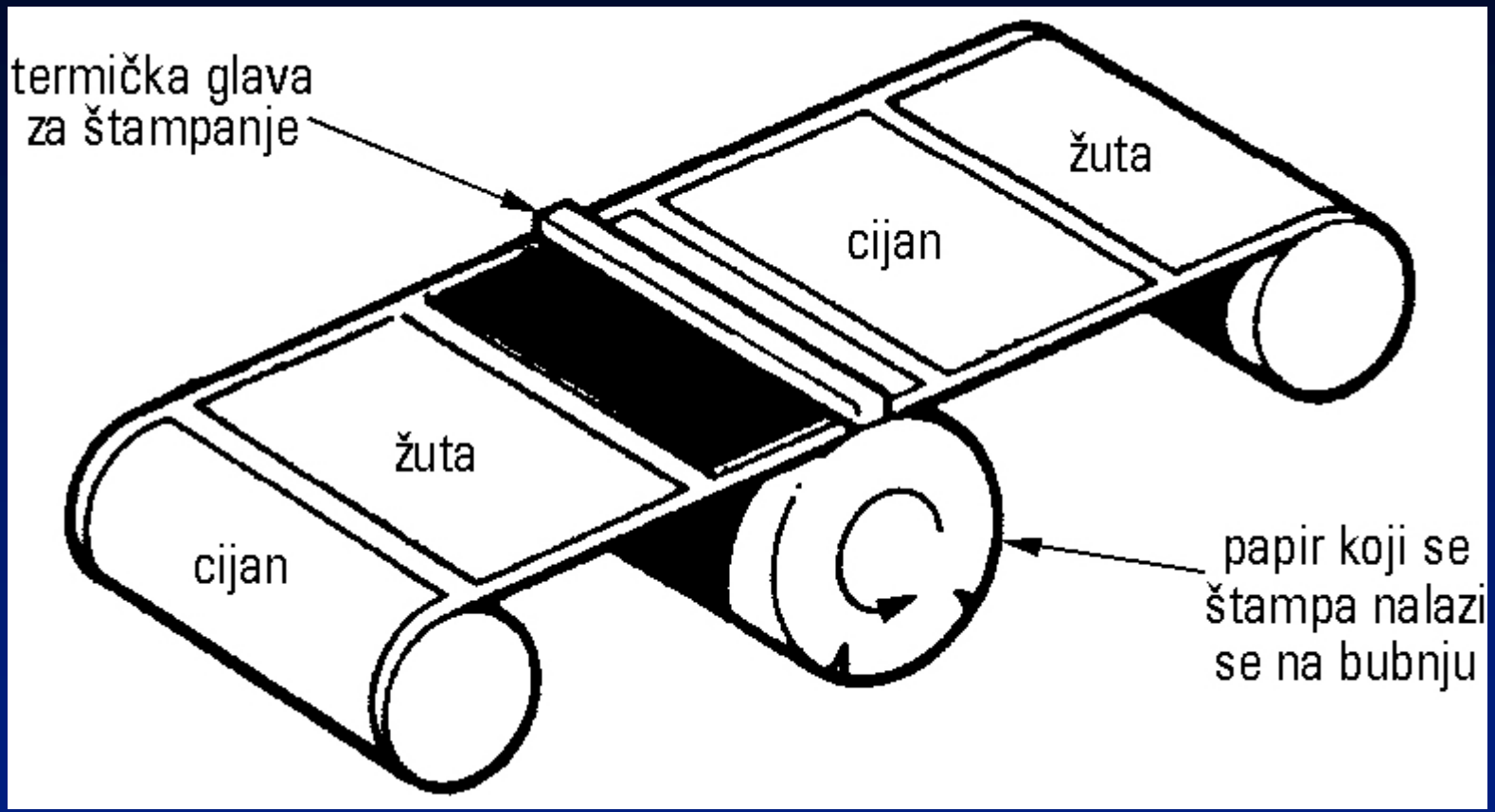
Grupisanje većeg broja *pixel-a*



Princip štampanja kod termičkog bojenja

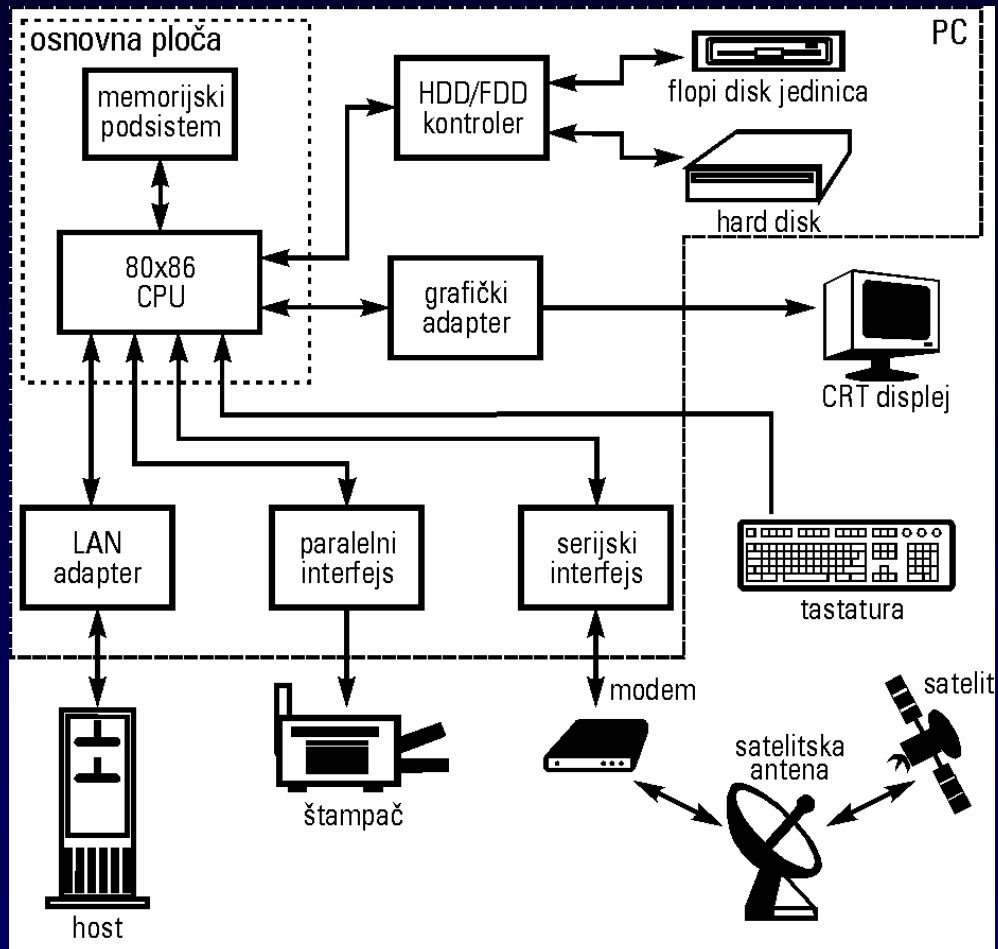


Štampački mehanizam



Organizacija sistema zasnovana na mikropocesoru 80x86

Blok dijagram PC mašina zasnovanih na Intel-ovim procesorima



memorijski podsistem	*CPU*	*U/I podsistem*
DRAM	8086	štampač
SRAM	8088	serijske komunikacije
Keš	80186	flopi disk
ROM	80286	miš
Fleš memorija	80386	CD ROM
EEPROM	80486	ploter
	Pentium	tastatura
	Pentium Pro	monitor (CRT prikaz)
		jedinica magnetne trake
		skener

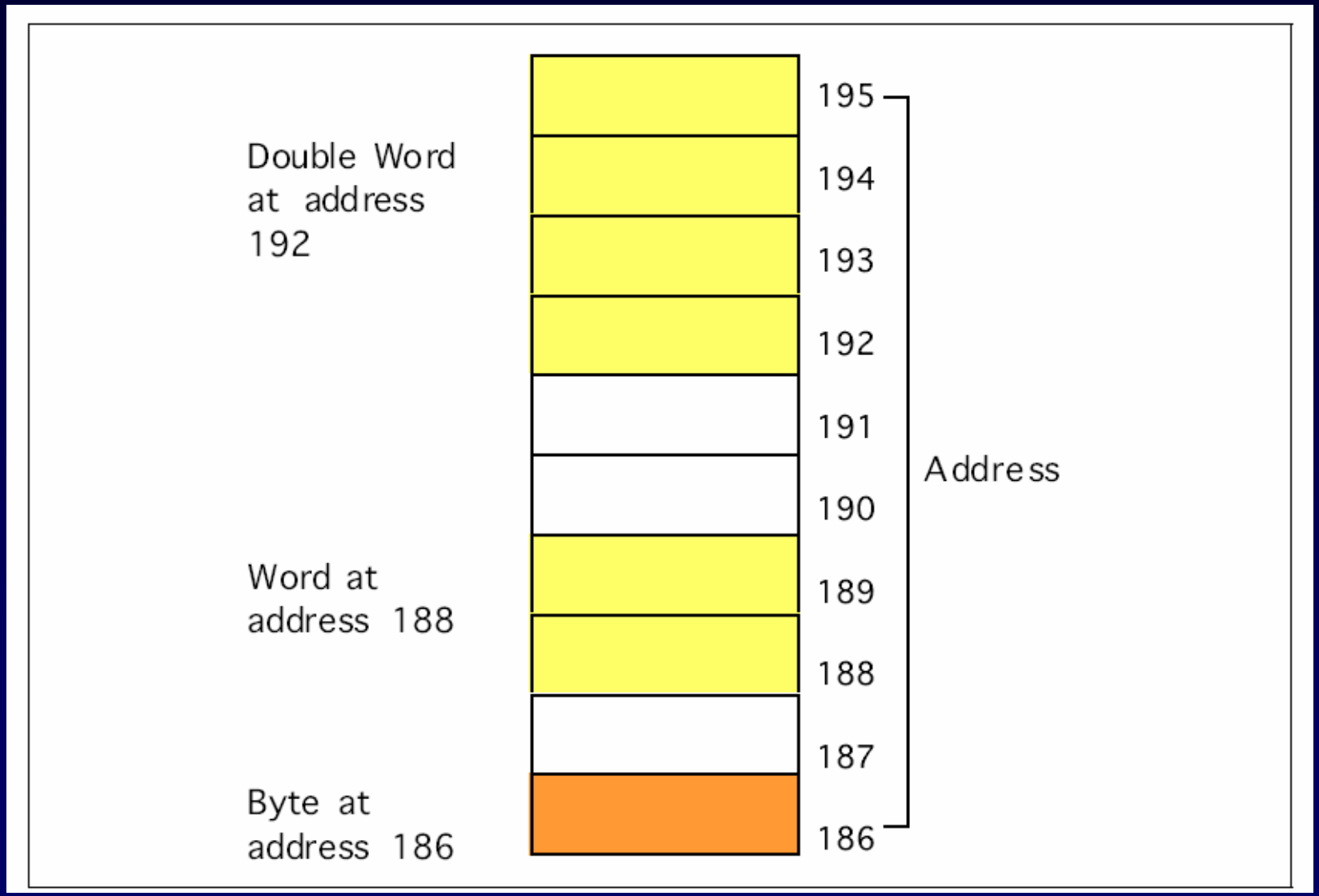
Obim magistrale podataka kod 80x86

Processor	Data Bus Size
8088	8
80188	8
8086	16
80186	16
80286	16
80386sx	16
80386dx	32
80486	32
80586 class/ Pentium (Pro)	64

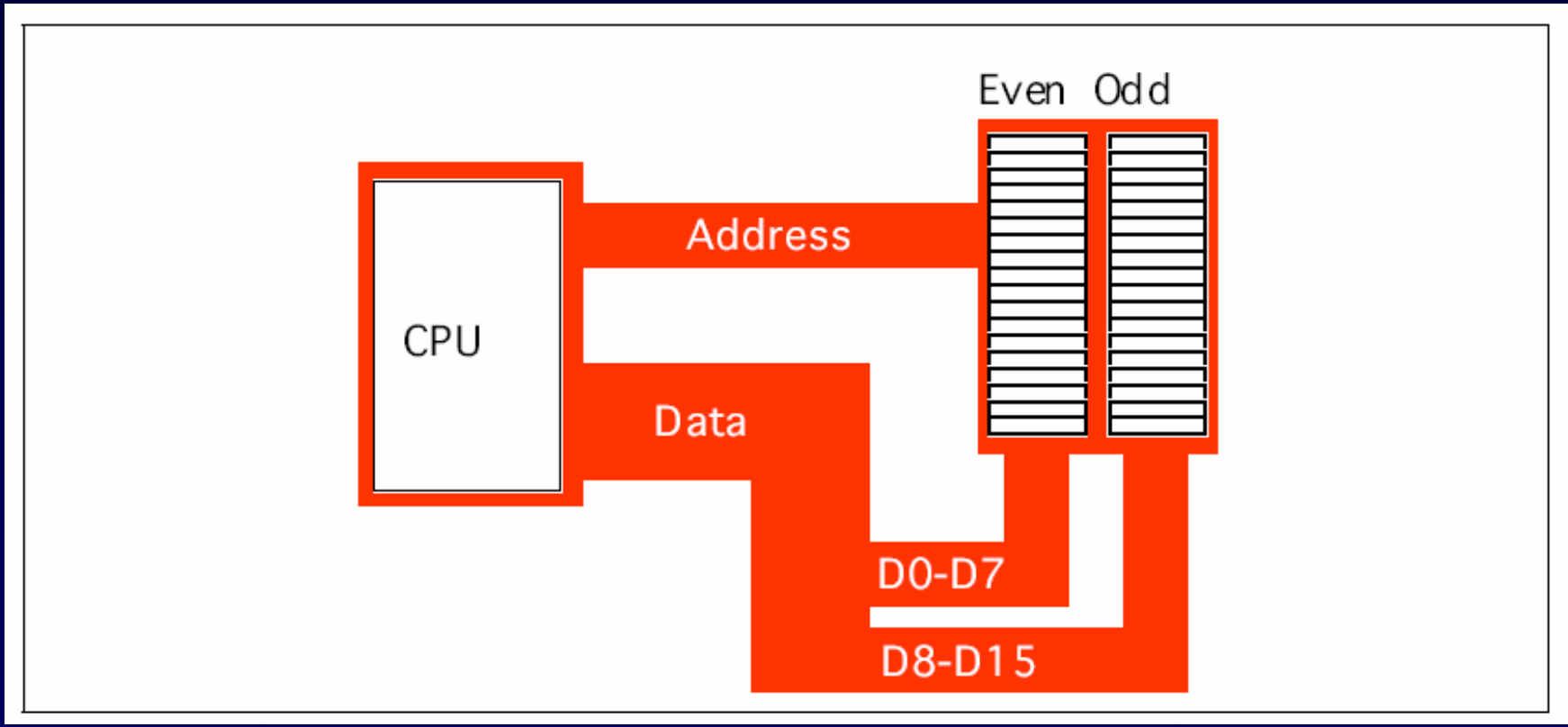
Obim adresne magistrale kod 80x86

Processor	Address Bus Size	Max Addressable Memory	In English!
8088	20	1,048,576	One Megabyte
8086	20	1,048,576	One Megabyte
80188	20	1,048,576	One Megabyte
80186	20	1,048,576	One Megabyte
80286	24	16,777,216	Sixteen Megabytes
80386sx	24	16,777,216	Sixteen Megabytes
80386dx	32	4,294,976,296	Four Gigabytes
80486	32	4,294,976,296	Four Gigabytes
80586 / Pentium (Pro)	32	4,294,976,296	Four Gigabytes

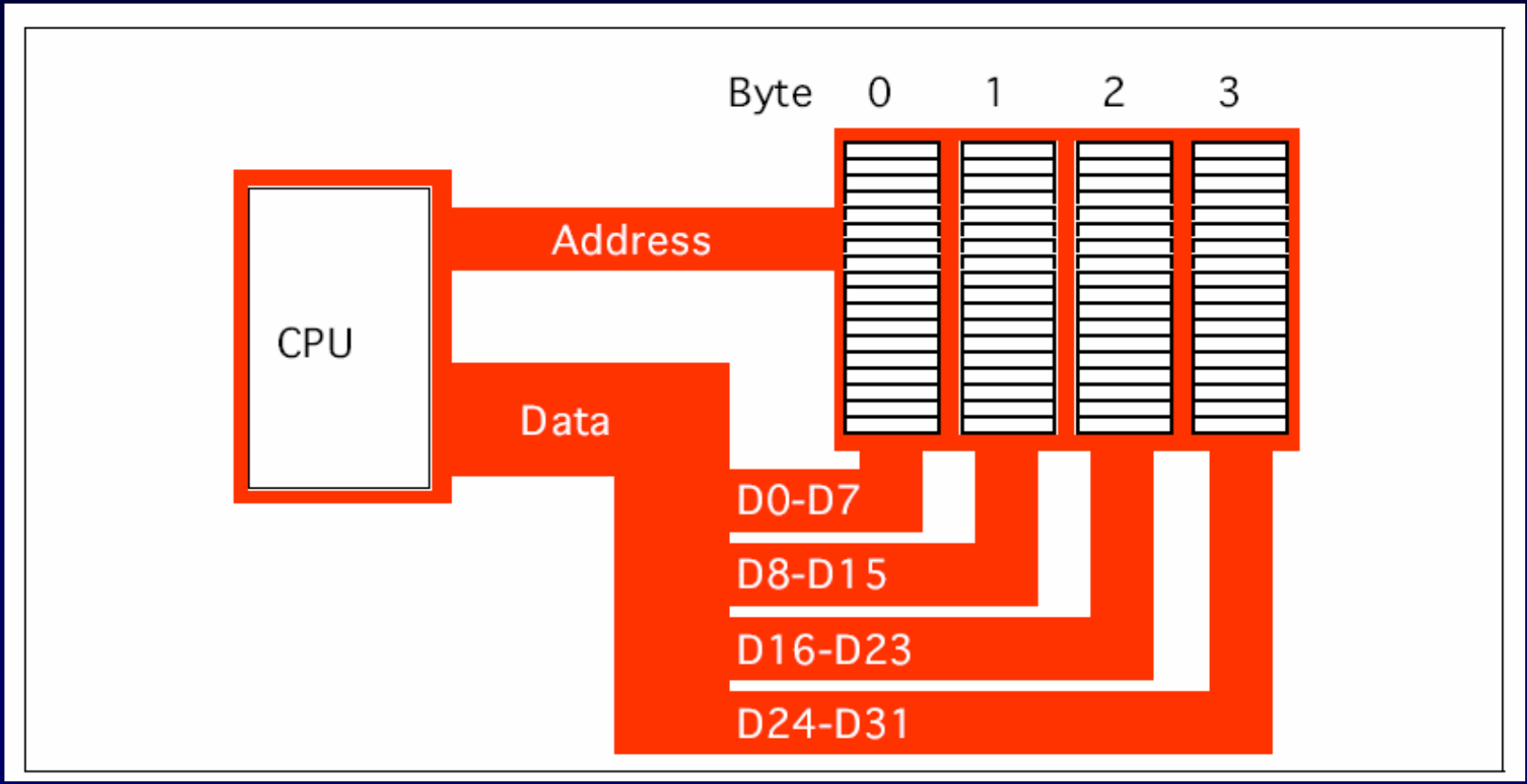
Memorisanje bajta, reči i duple reči kod 80x86



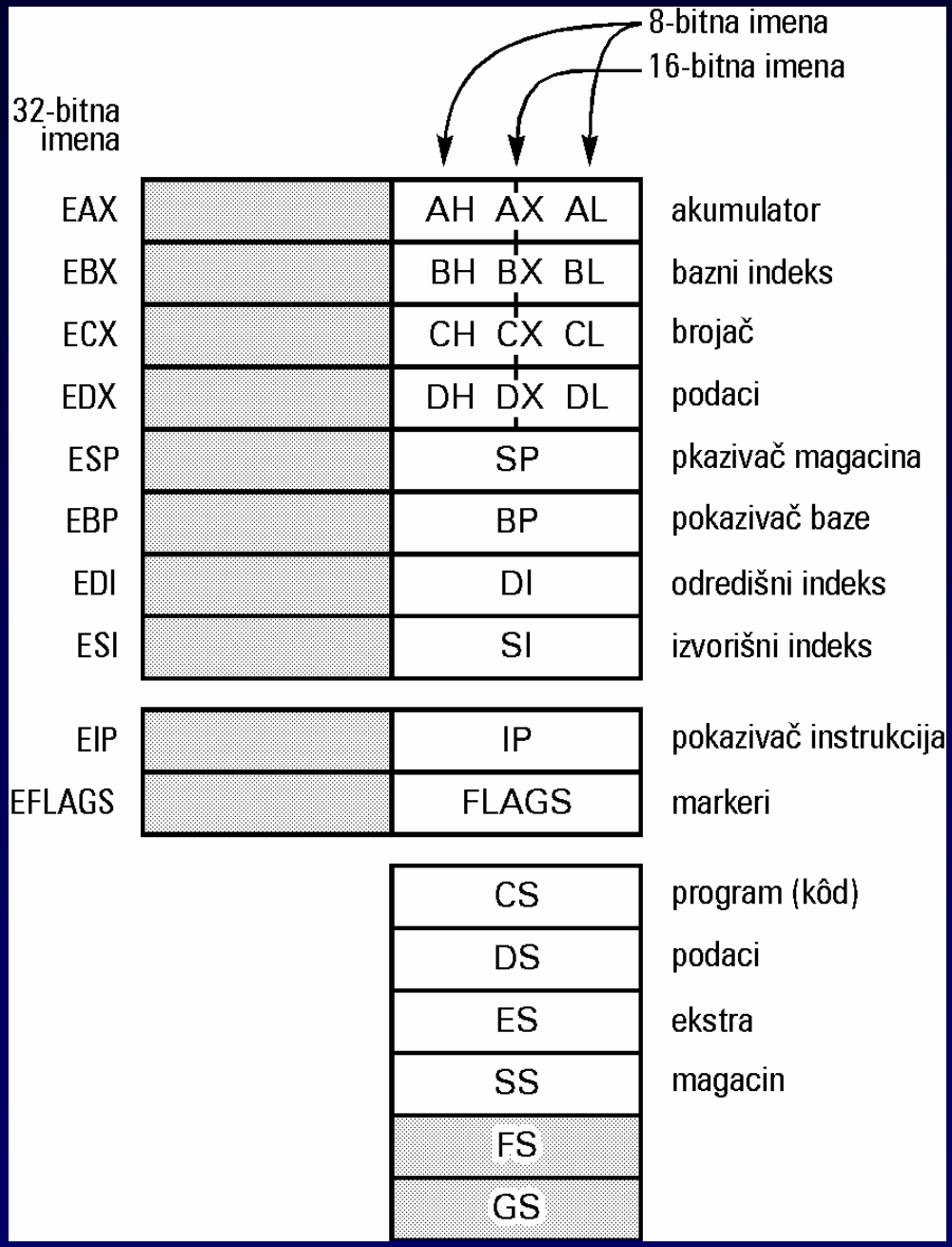
Organizacija memorije kod 16-bitnog procesora (8086, 80186, 80286, 80386sx)



Organizacija memorije kod 32-bitnog procesora (80386, 80486, Pentium, ...)



Programski model mikroprocesora počev od 8086 do Pentium



8086 register set

Figure 4.1 8086 Register Set

8086 Flags Register

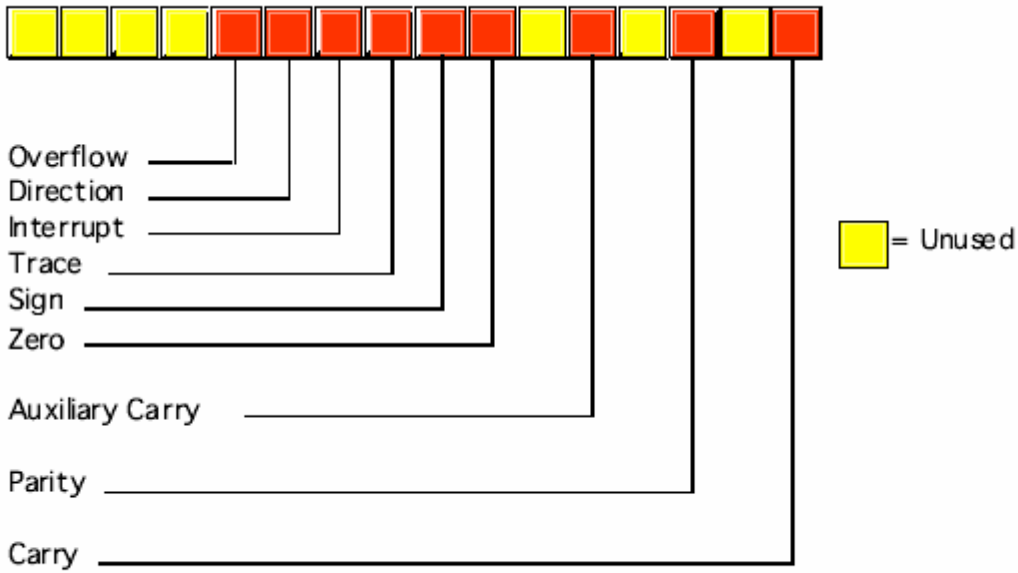


Figure 4.2 8086 Flags Register

80386 Registers (Application Programmer Visible)

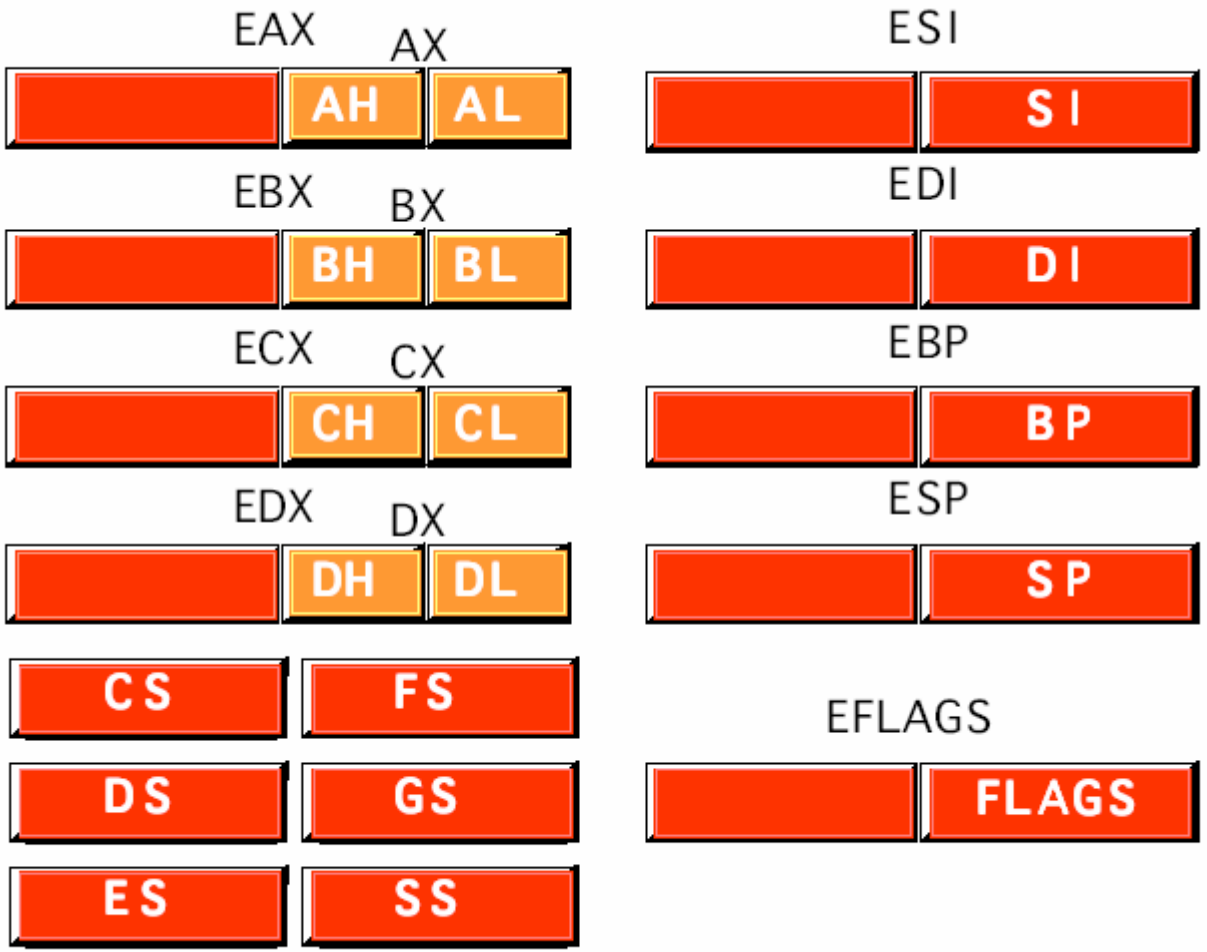
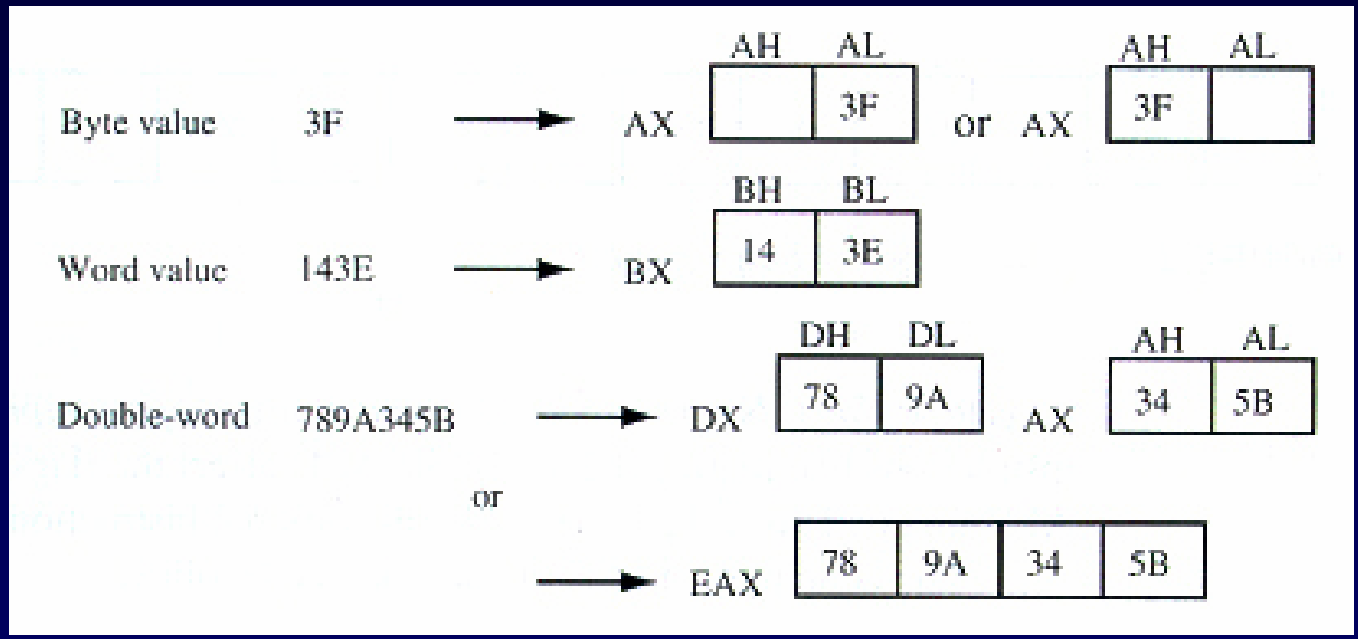
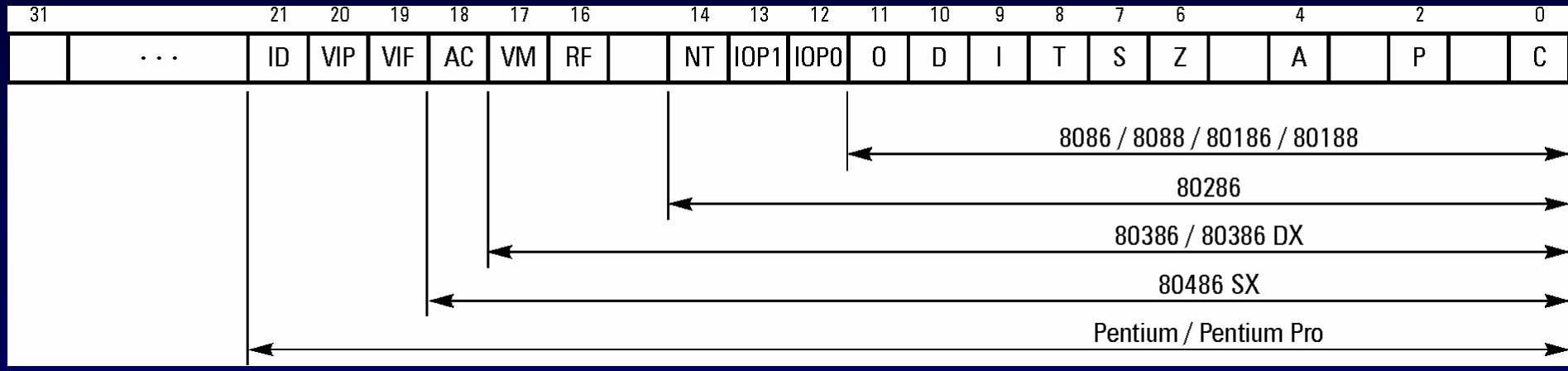


Figure 4.3 80386 Registers (Application Programmer Visible)

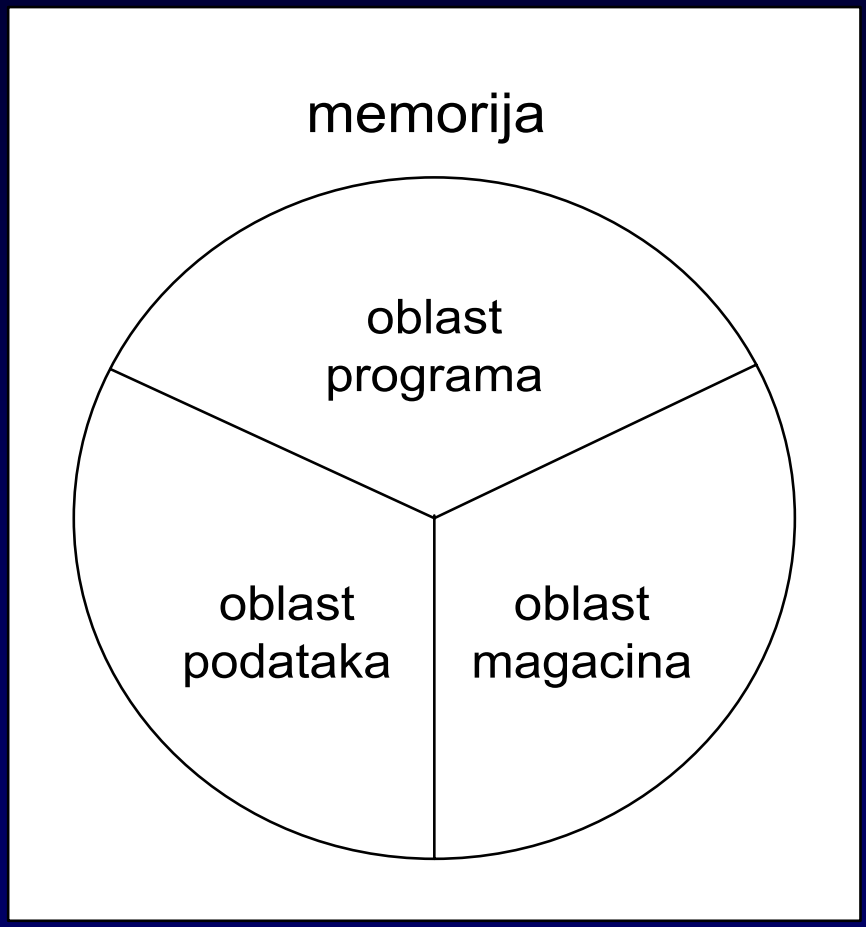
Način memorisanja podataka u registrima



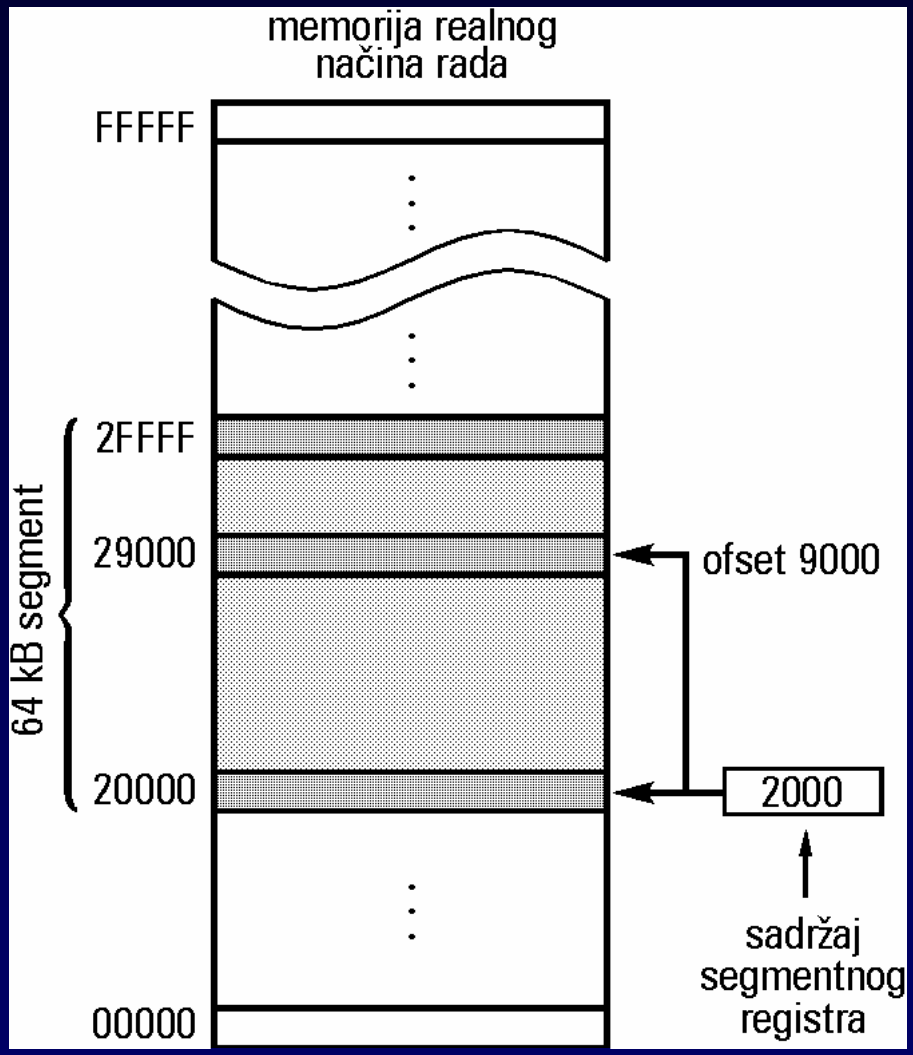
Registri FLAG i EFLAGS za familiju mikroprocesora 80x86 i Pentium

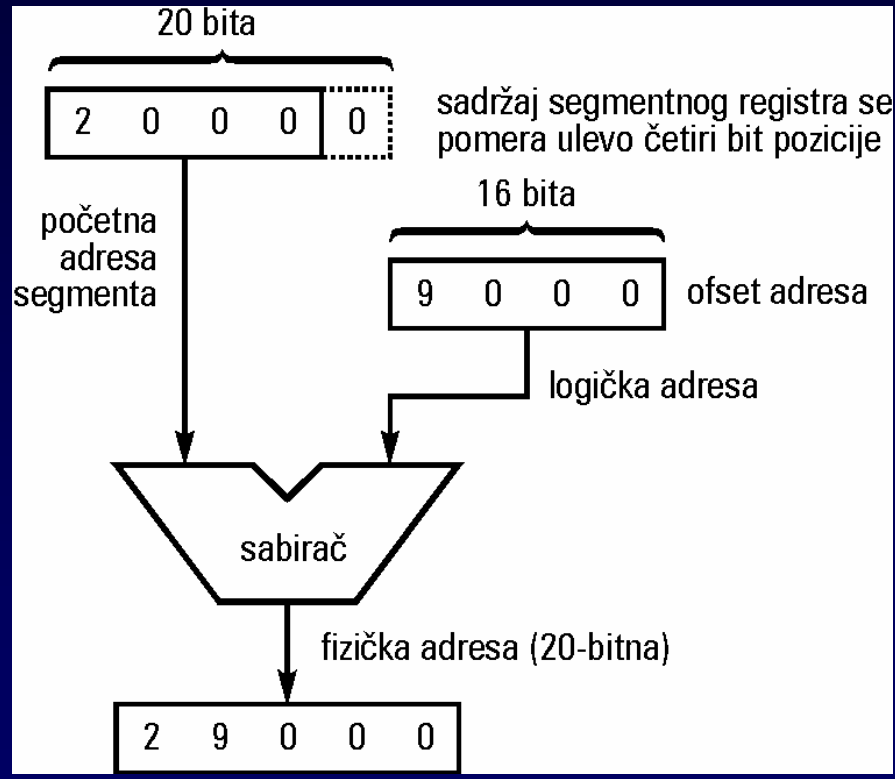


Pogled na memorijski sistem

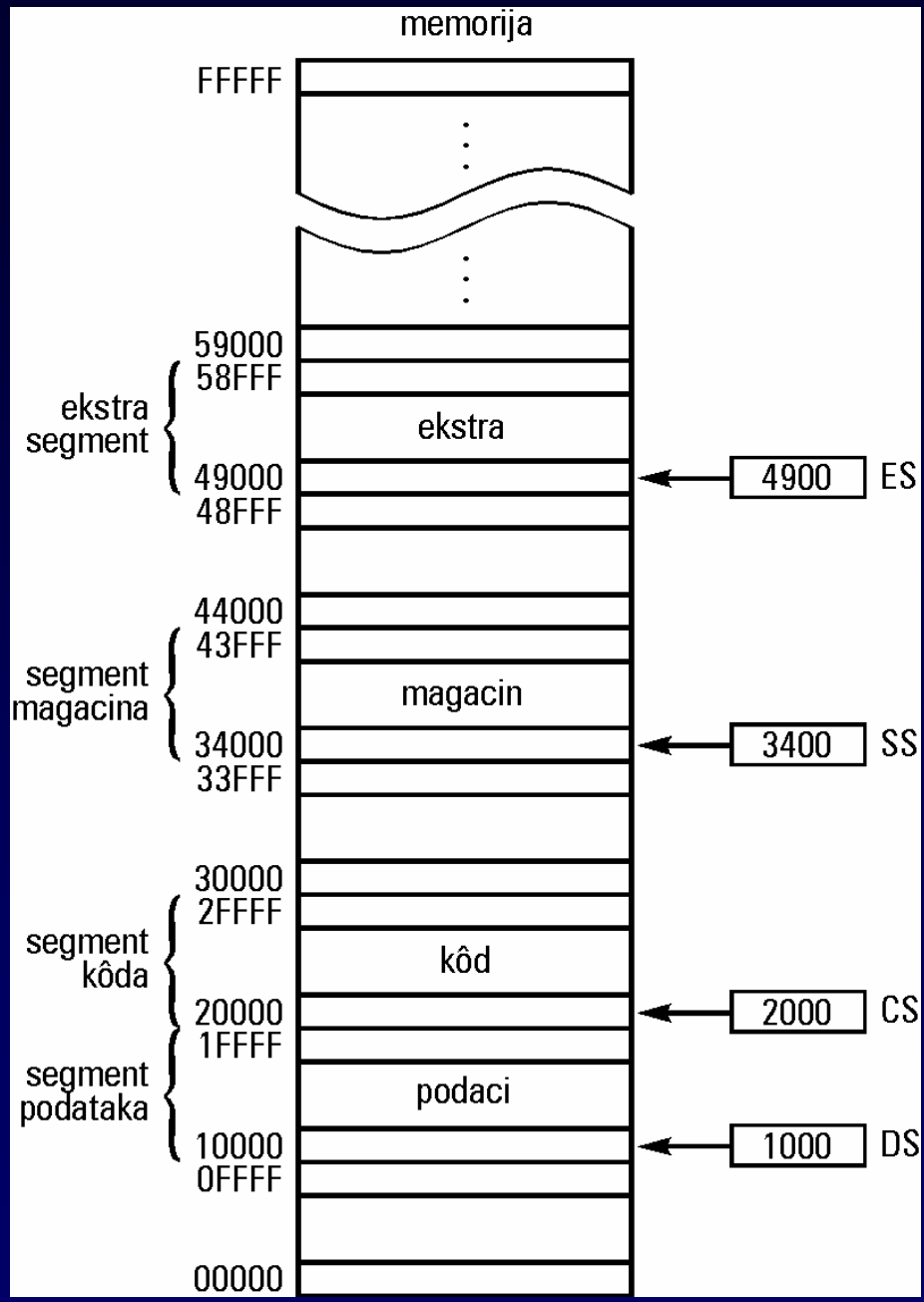


Adresiranje memorije korišćenjem adrese segment plus ofset

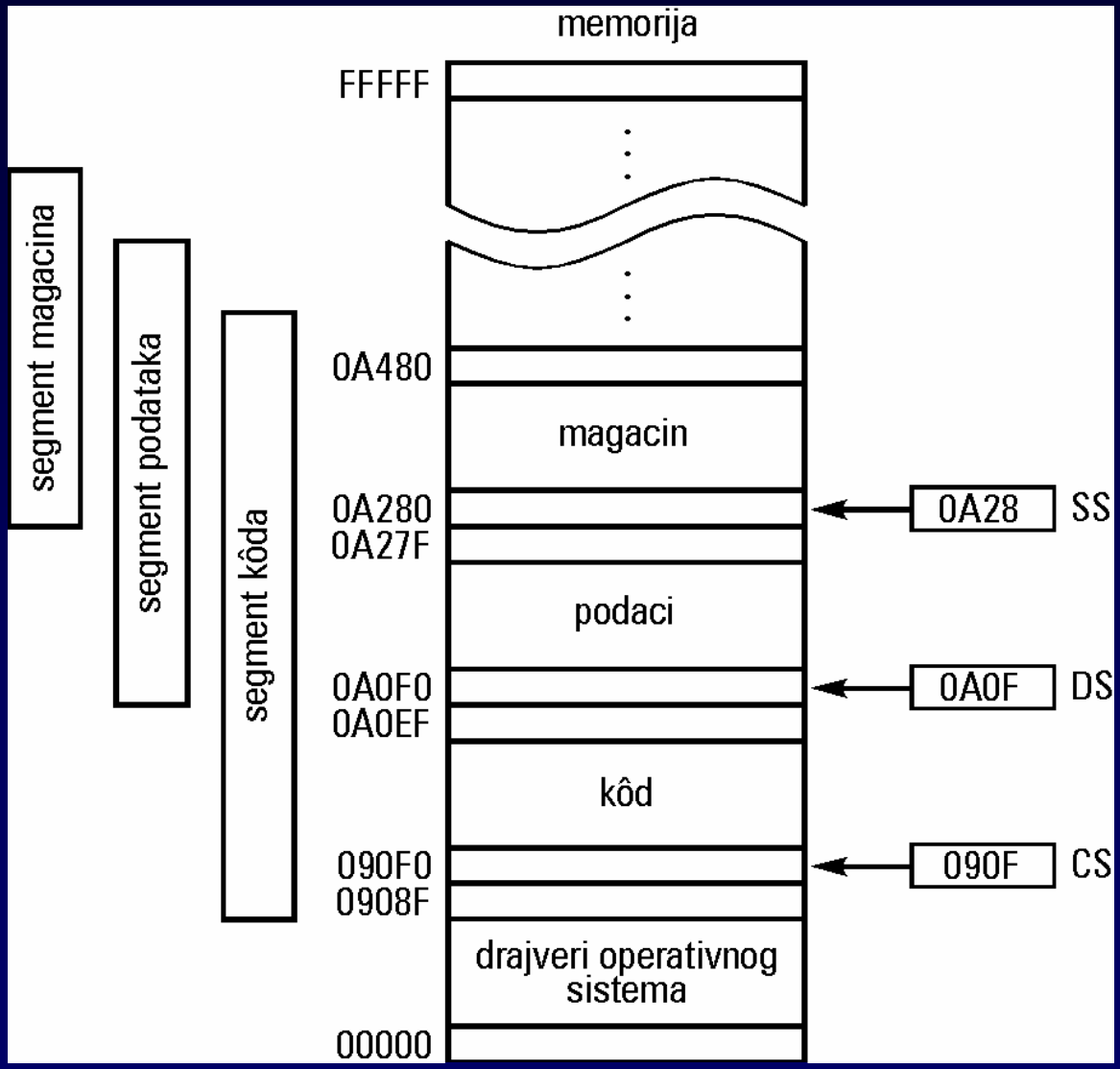




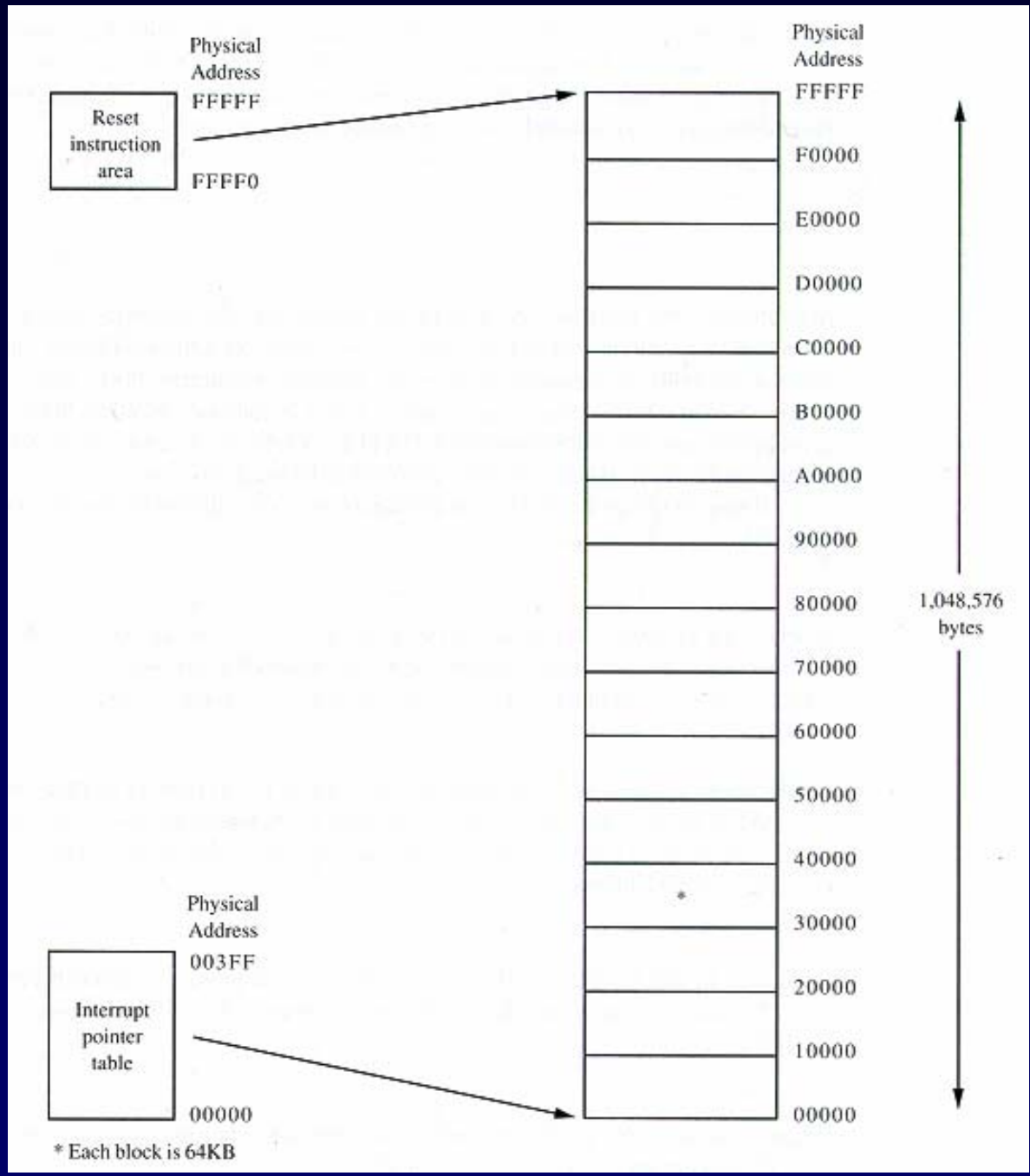
Konkretan primer rasporeda memorijskih segmenata bez preklapanja



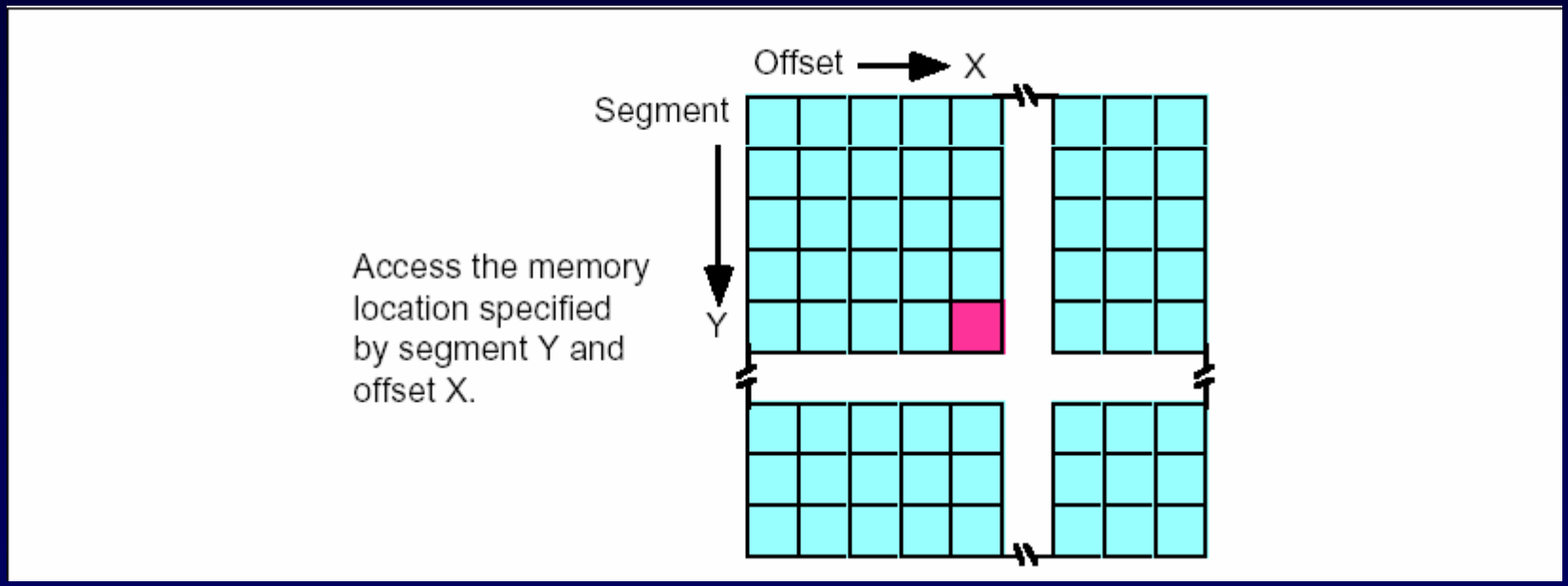
Konkretan primer rasporeda memorijskih segmenata sa preklapanjem



16 segmenata koji se ne preklapaju kod 8086



Segmented Addressing as a Two-Dimensional Process



Segmented Addressing in Physical Memory in Real Mode

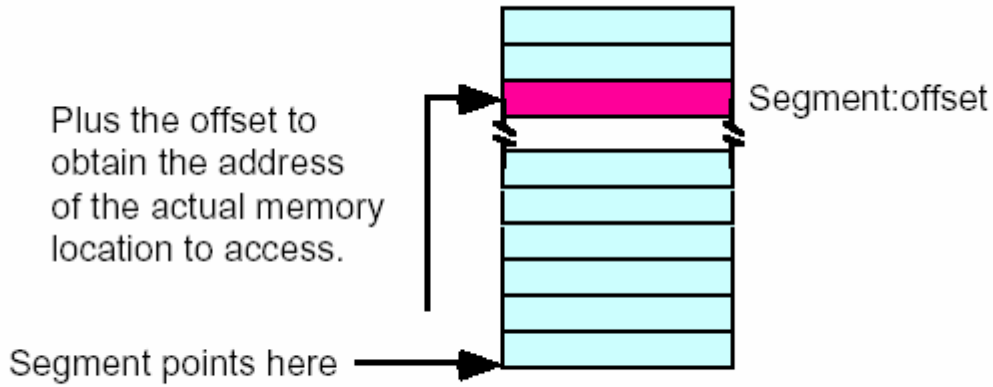


Figure 4.5 Segmented Addressing in Physical Memory

Converting a Logical Address to a Physical Address in Real Mode

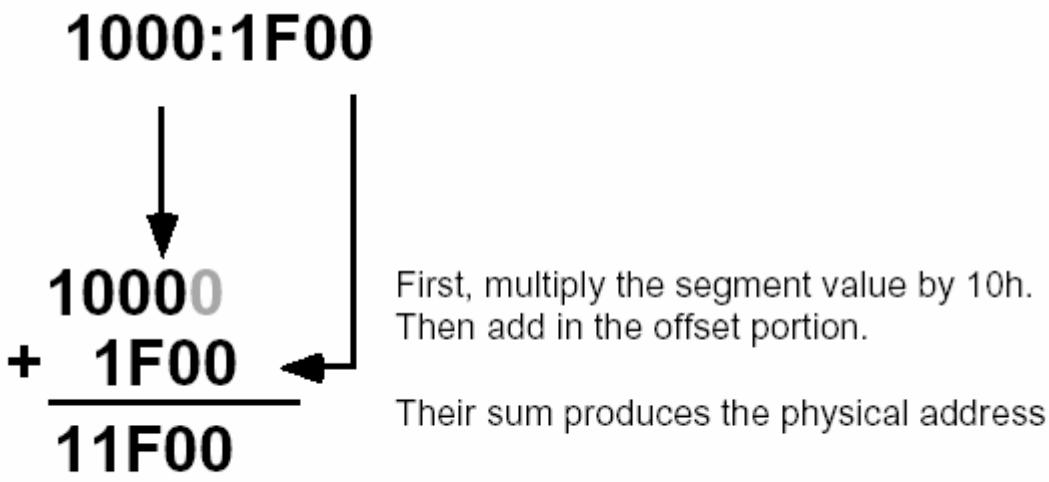


Figure 4.6 Converting a Logical Address to a Physical Address

Converting a Logical Address to a Physical Address in Protected Mode

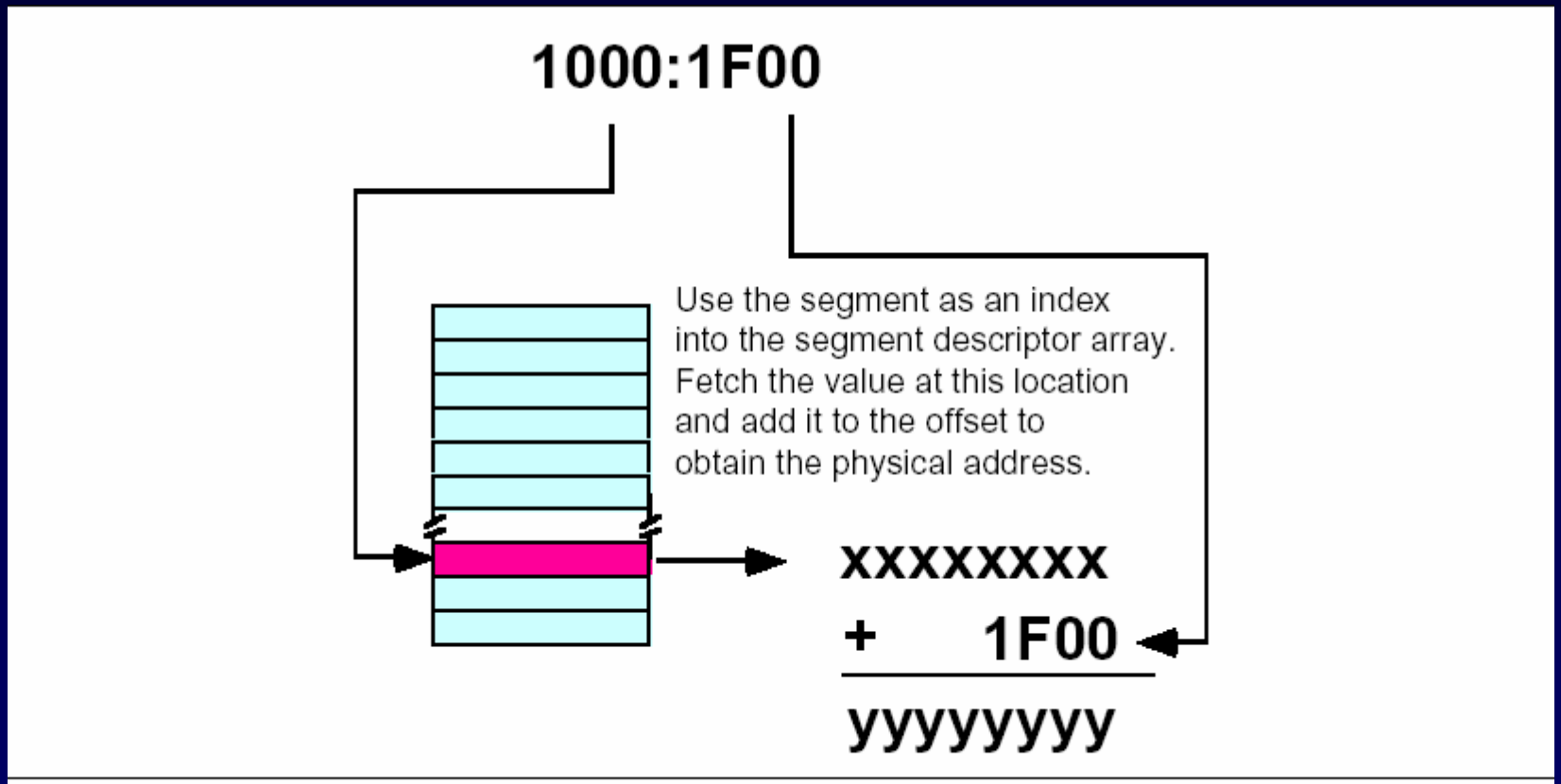
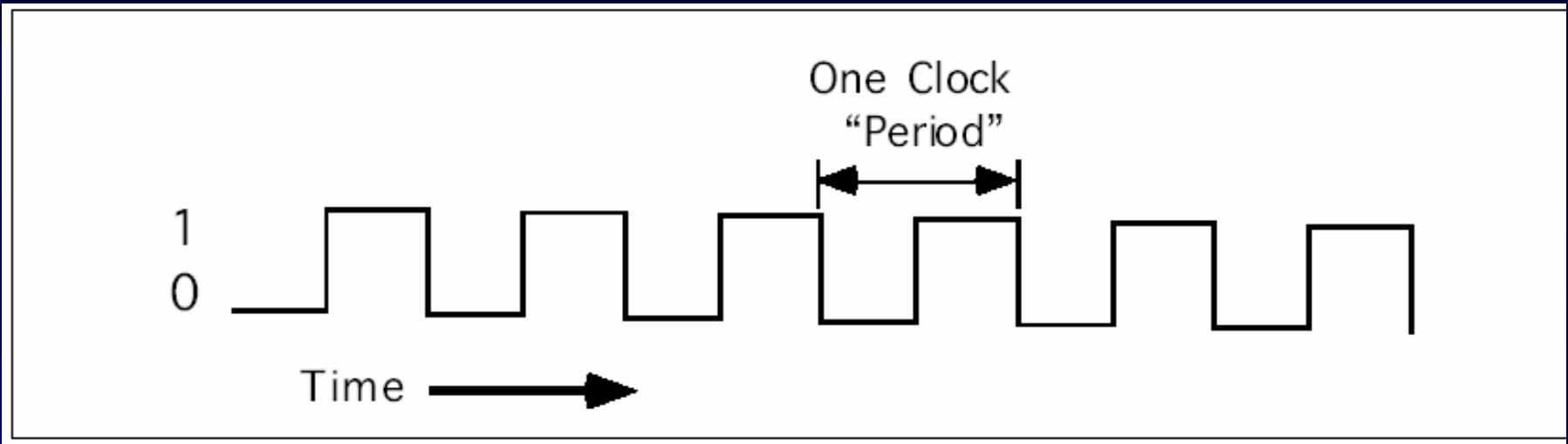
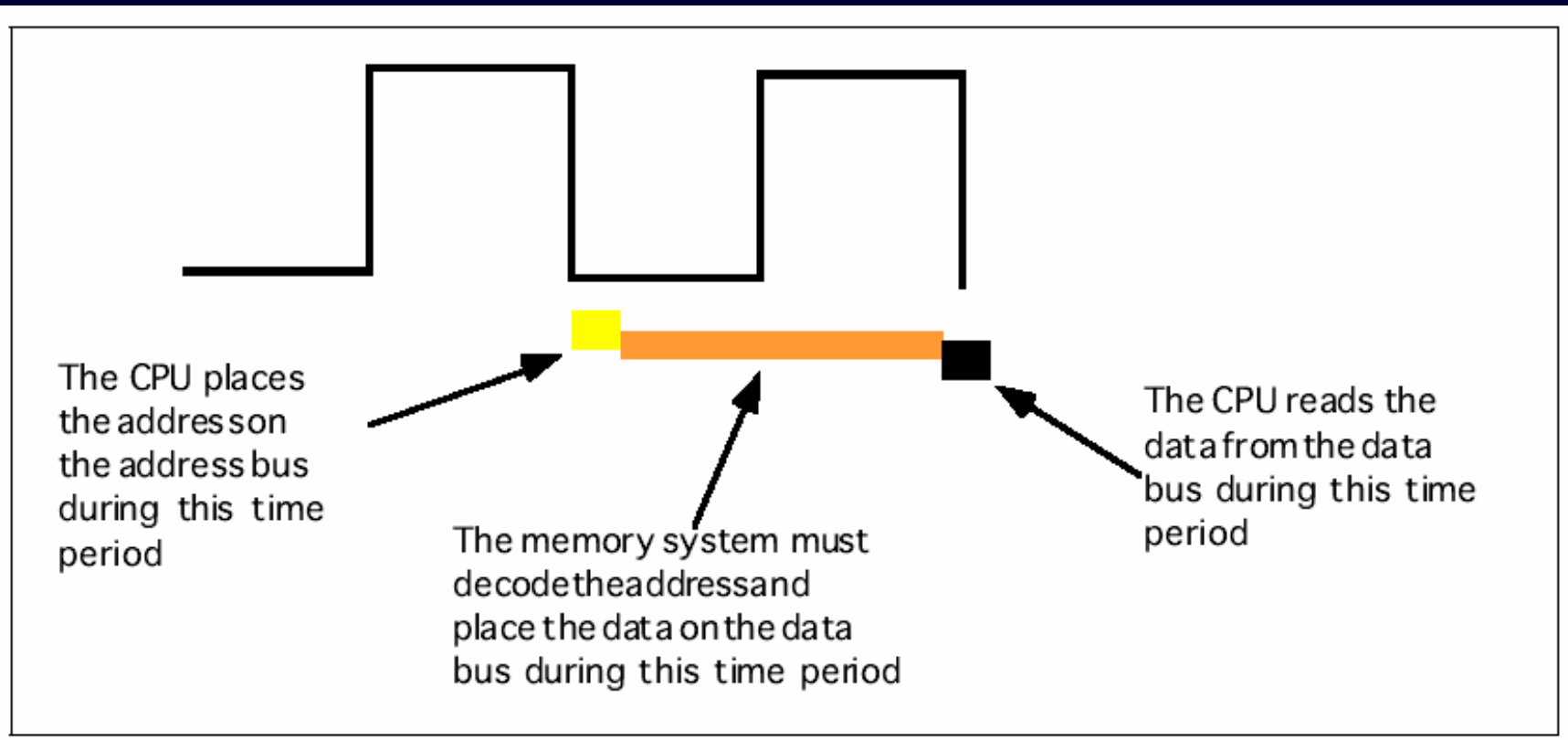


Figure 4.7 Converting a Logical Address to a Physical Address in Protected Mode

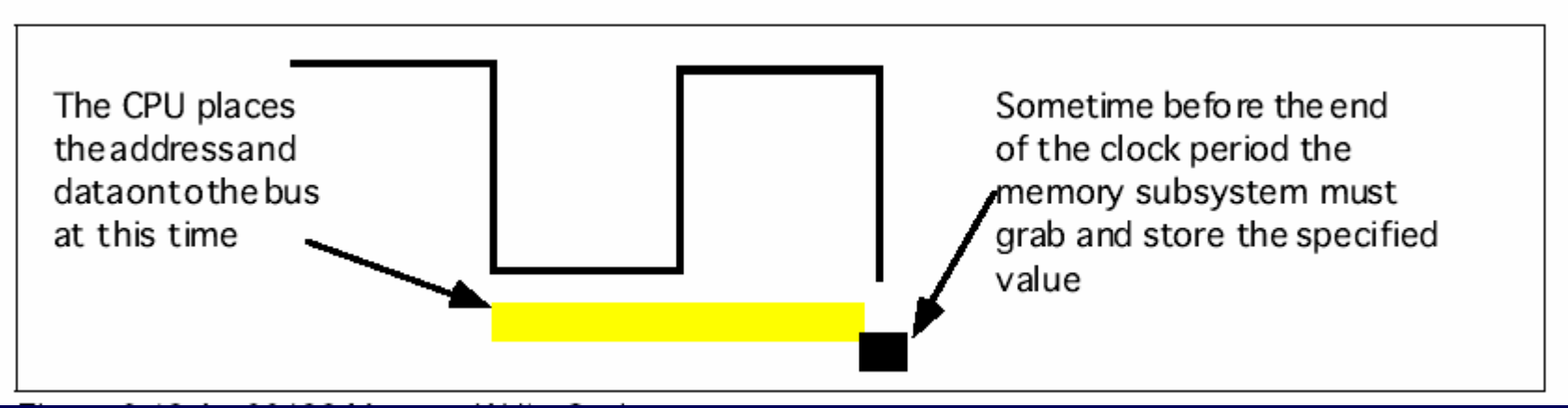
Sistemski takt



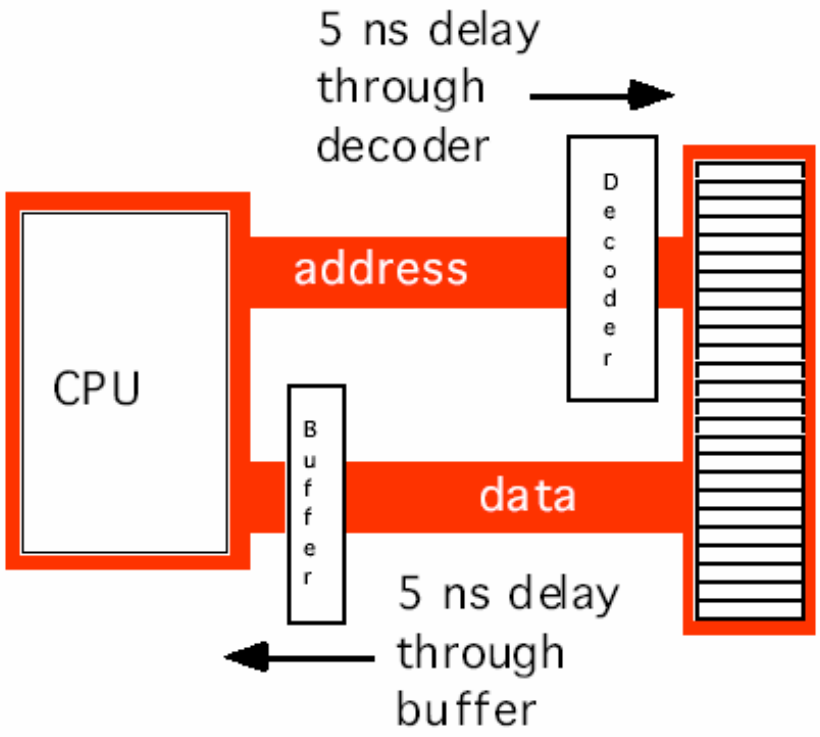
Ciklus čitanja memorije kod 80486



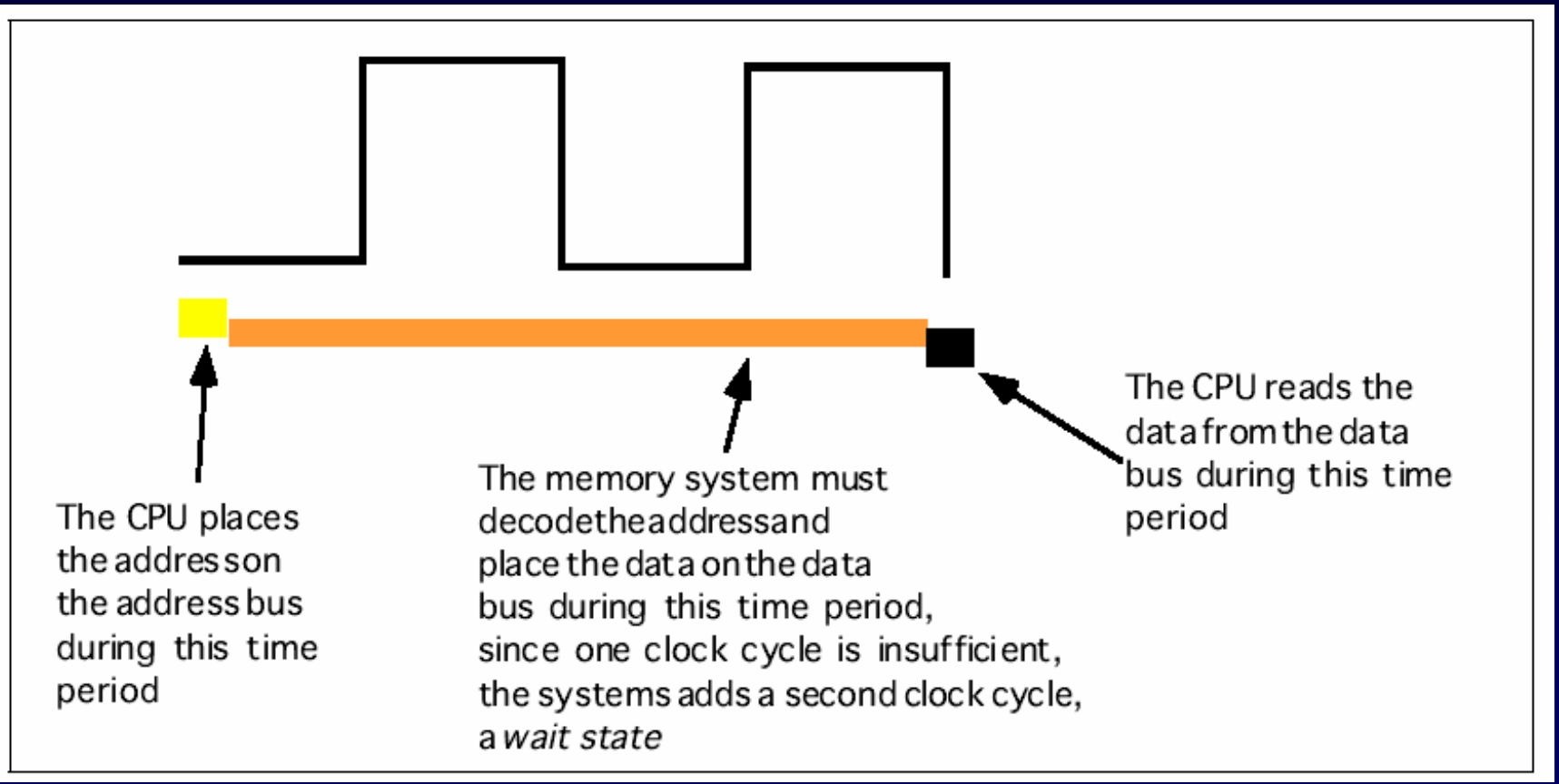
Ciklus upisa u memoriju kod 80486



Dekodiranje i kašnjenje bafera



Ubacivanje stanja čekanaj u Memory Read operaciju



Unutrašnja struktura 8486 CPU-a

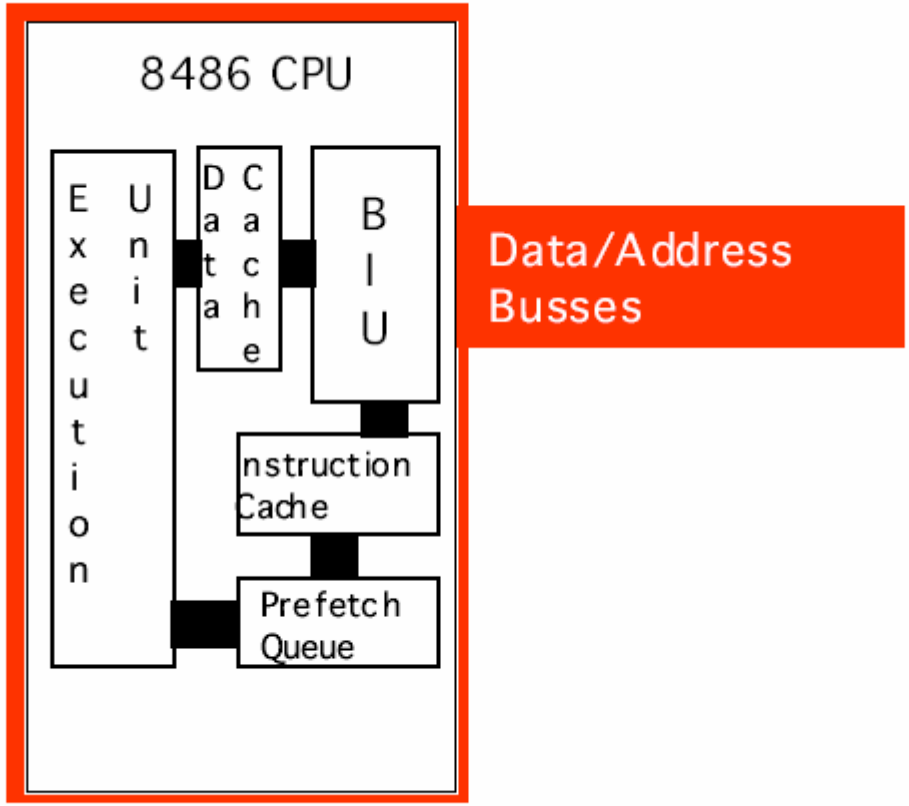
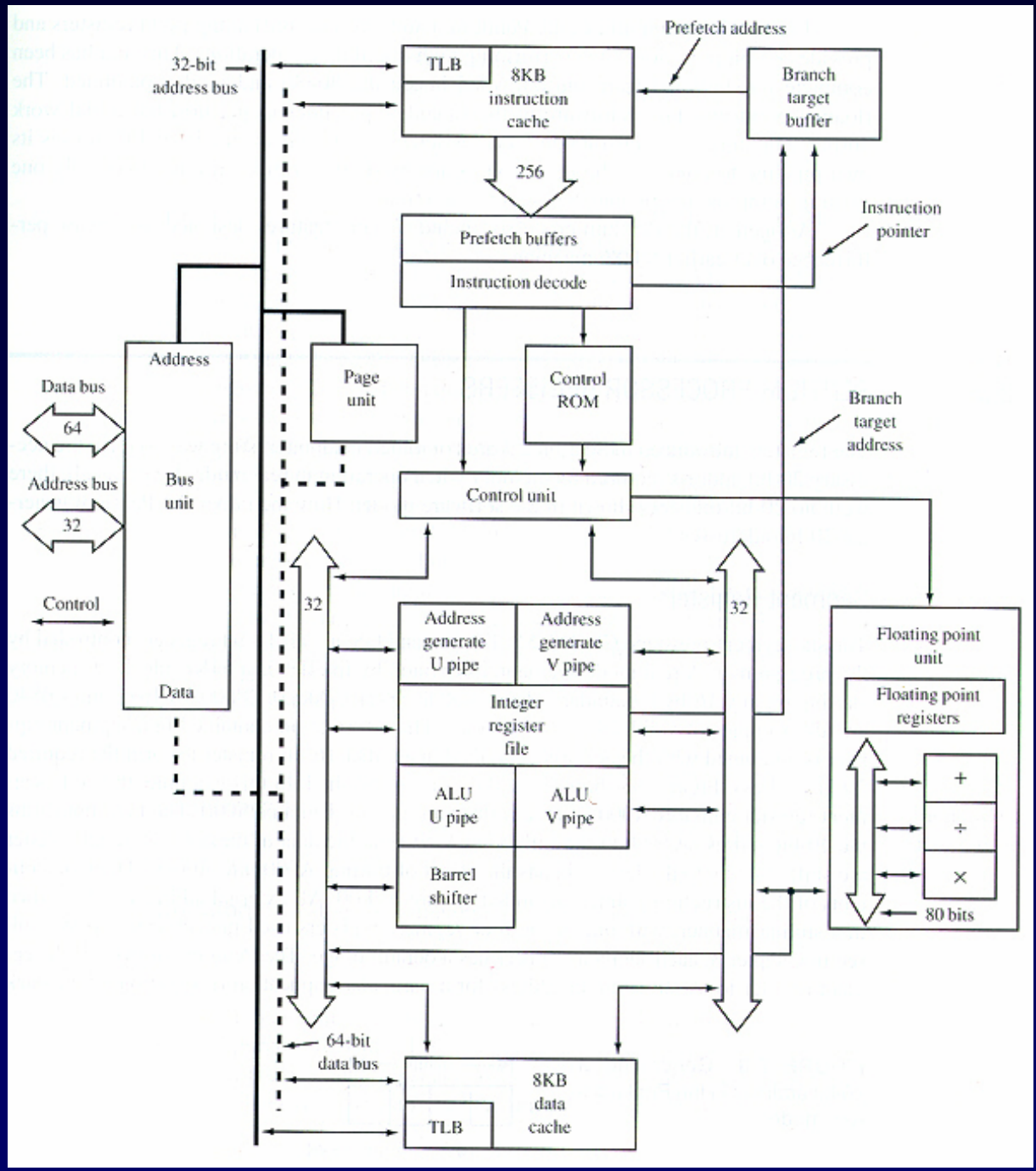


Figure 3.28 Internal Structure of the 8486 CPU

Interna struktura procesora Pentium



Hazardi kod 8486

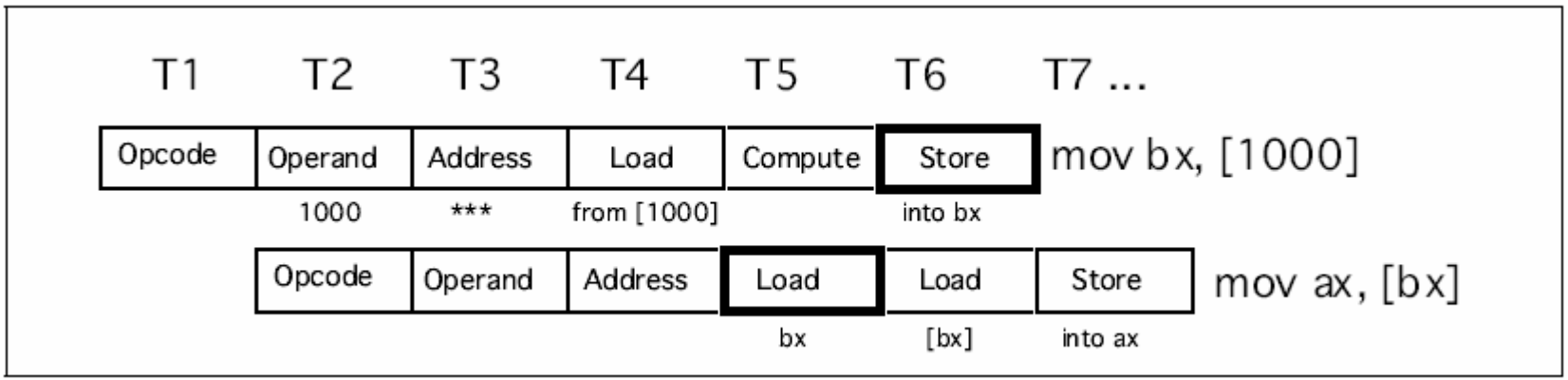


Figure 3.29 A Hazard on the 8486

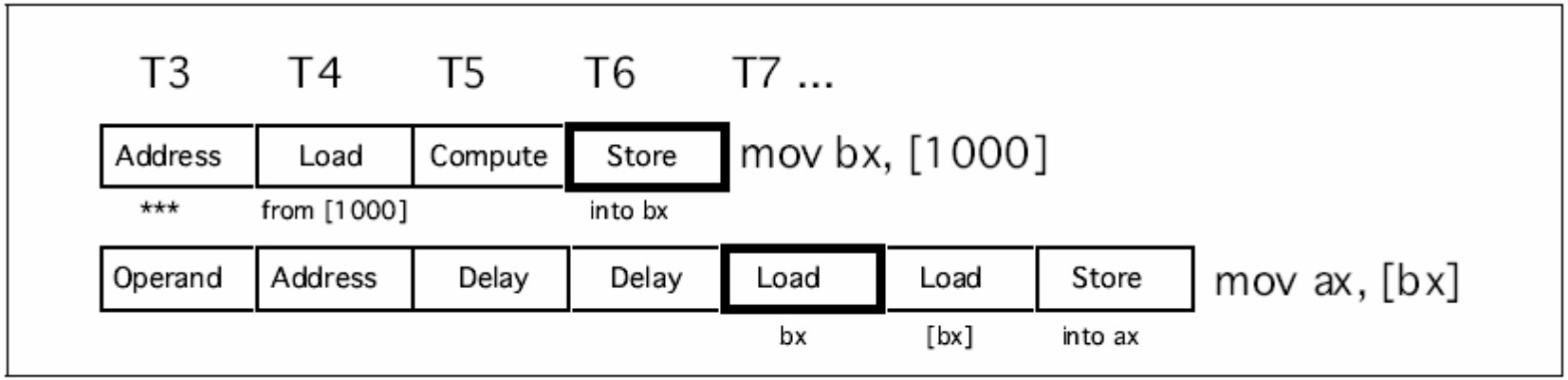


Figure 3.30 A Hazard on the 8486

Unutrašnja struktura kod 8686

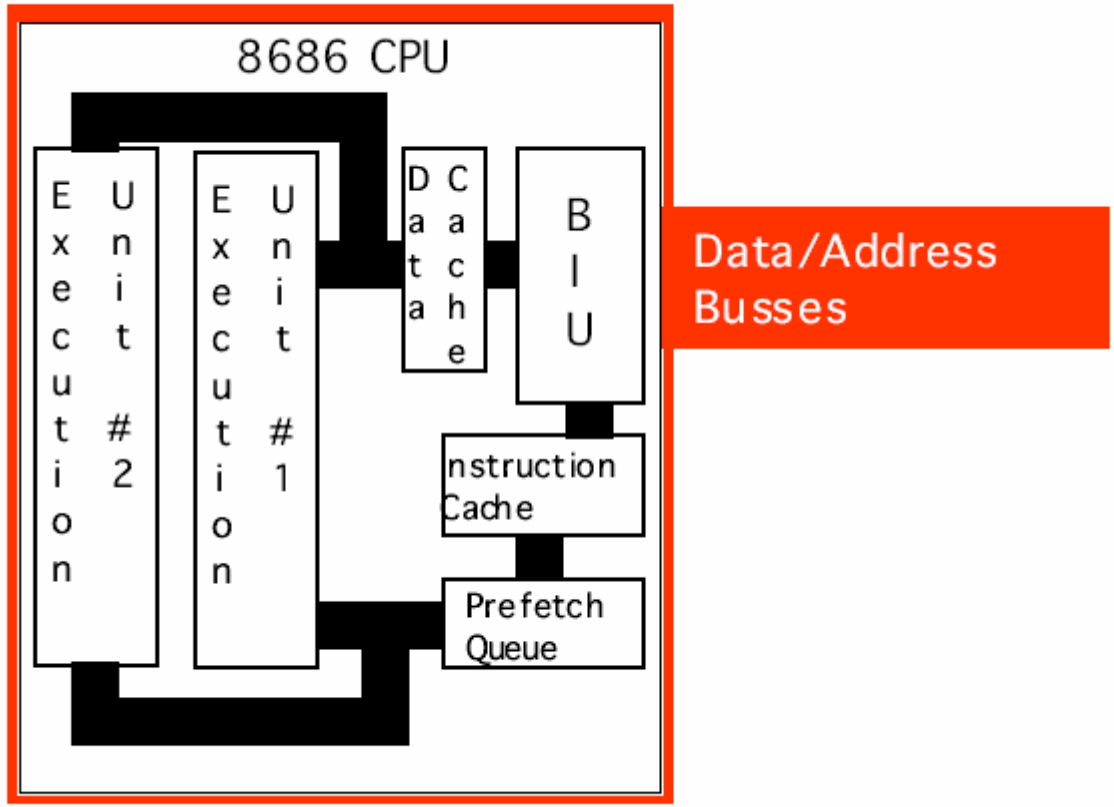


Figure 3.31 Internal Structure of the 8686 CPU

Izlazni port kreiran pomoću jednog leča

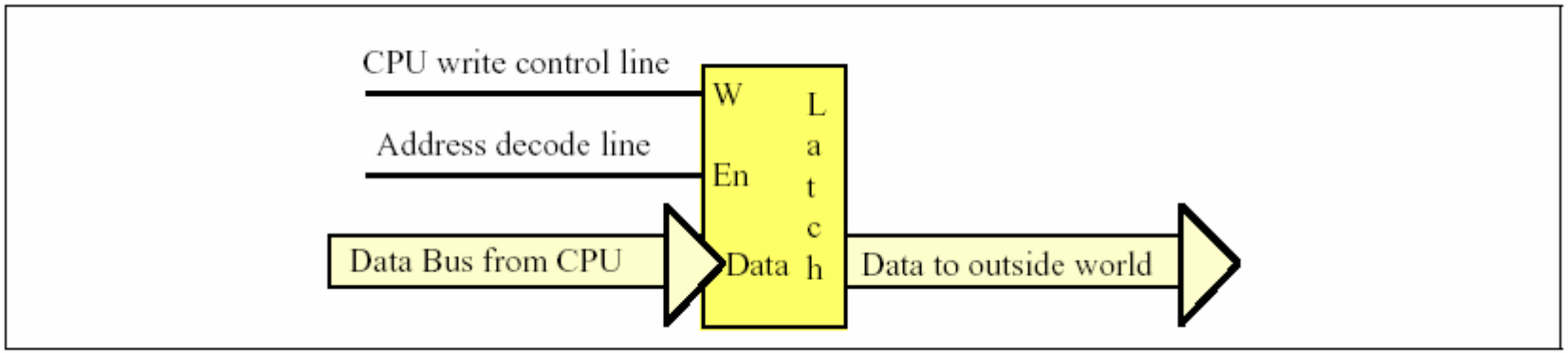


Figure 3.32 An Output Port Created with a Single Latch

Ulazno/izlazni port zahteva dva leča

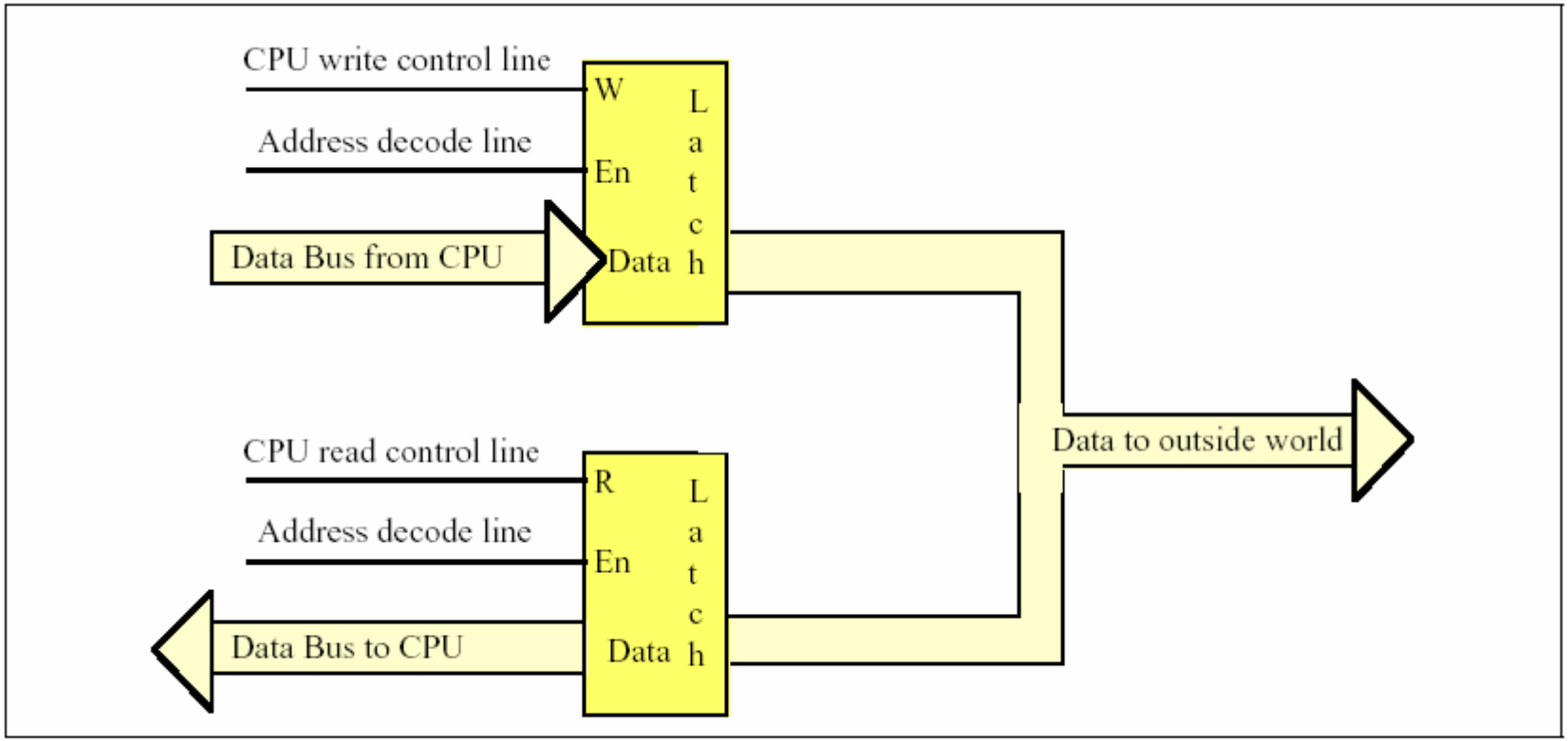
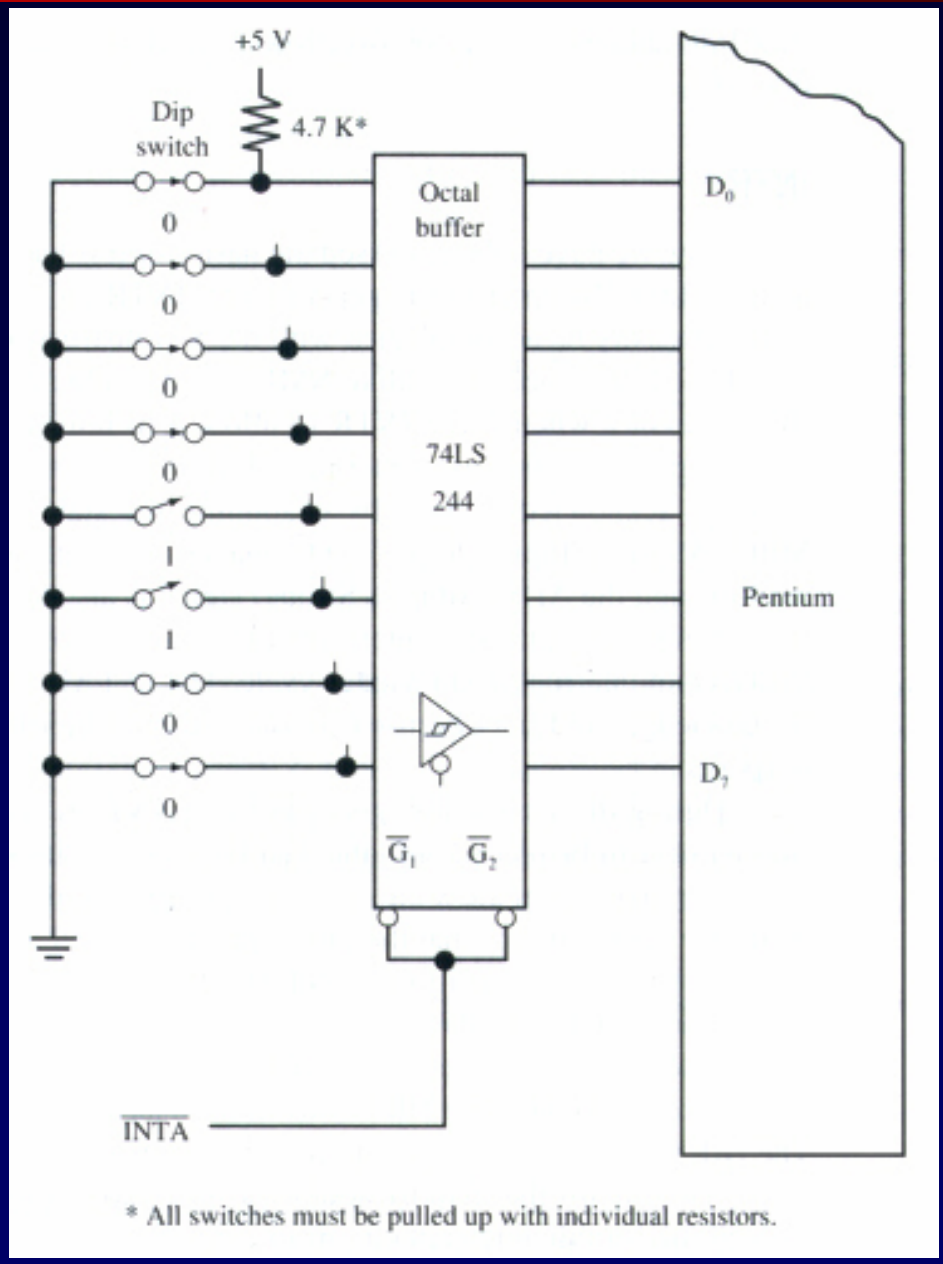
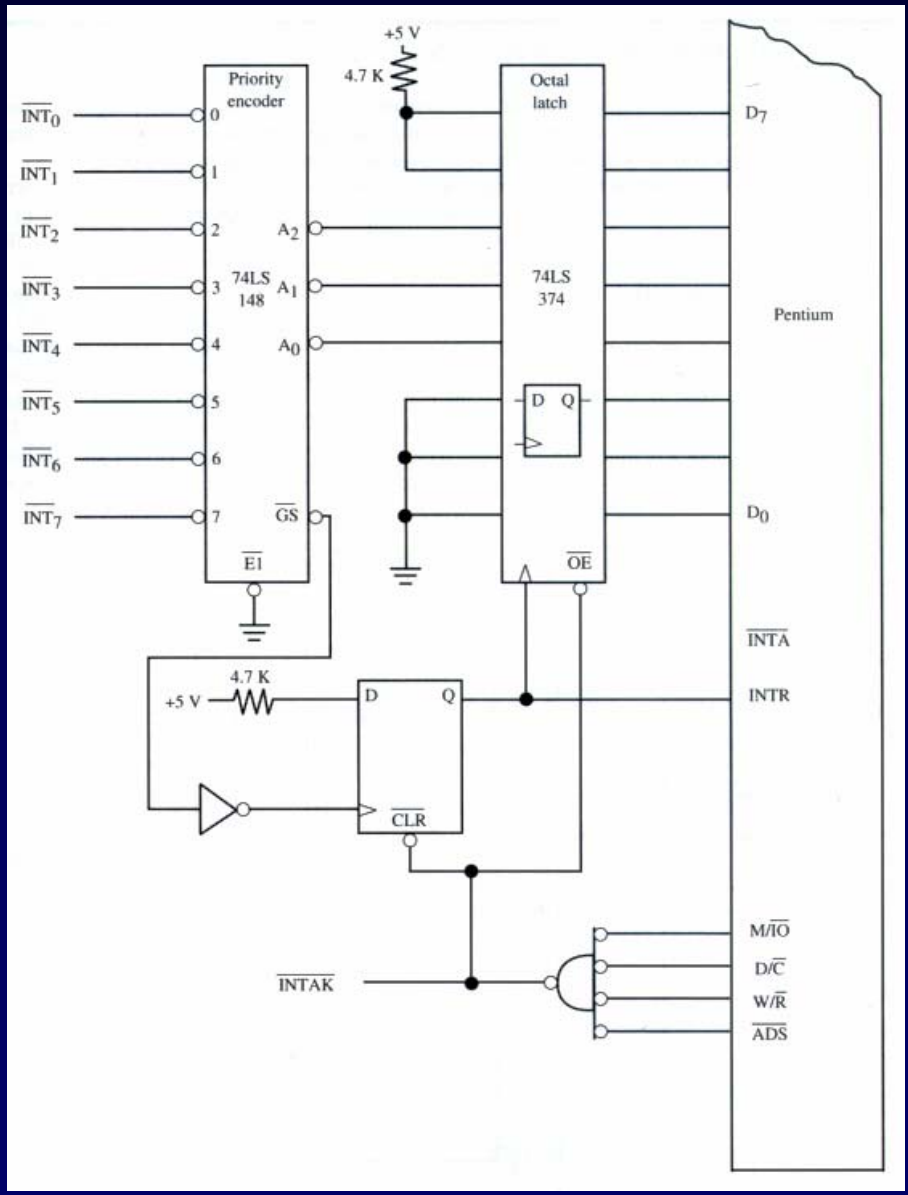


Figure 3.33 An Input/Output Port Requires Two Latches

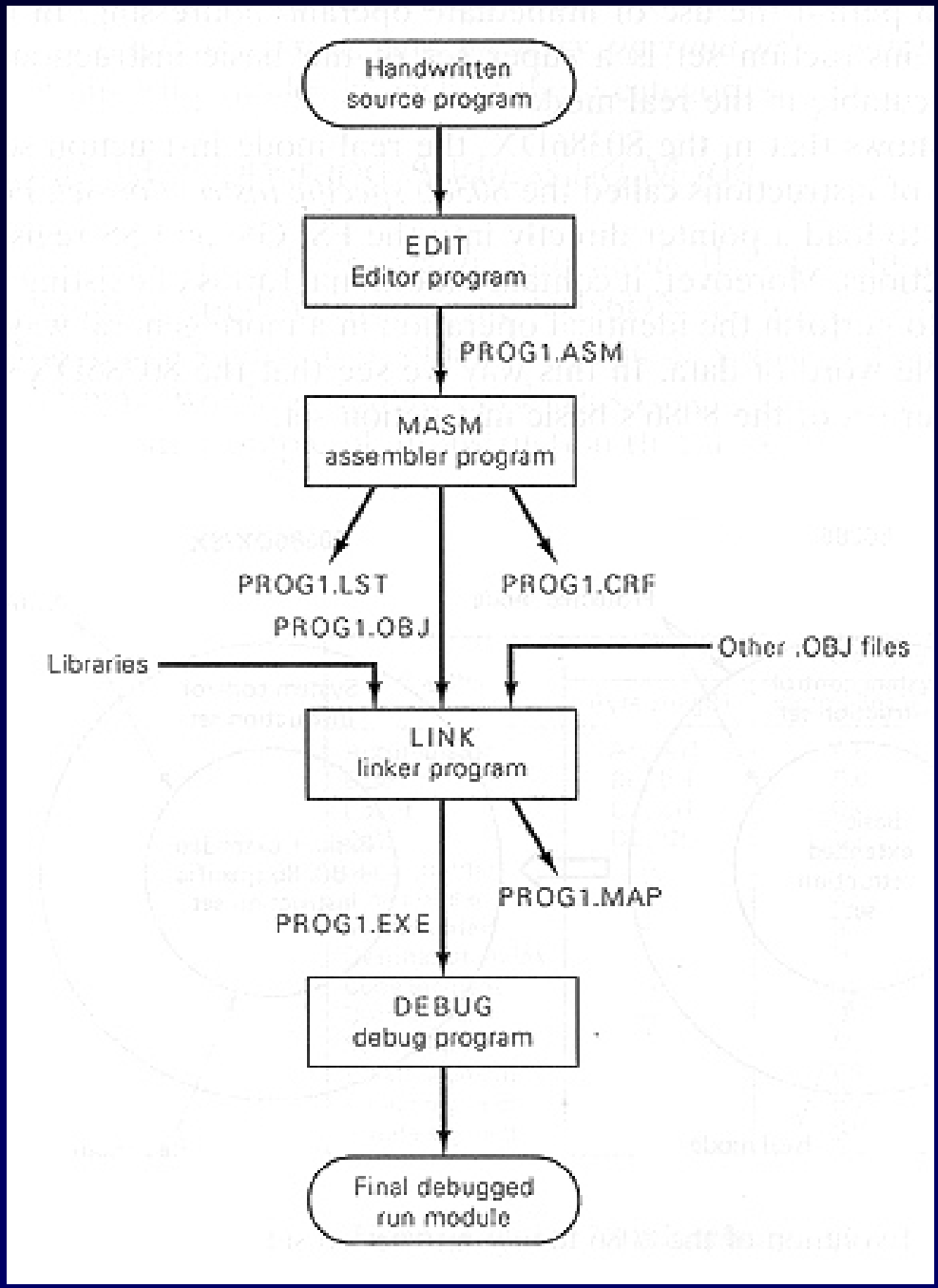
Inputing Data Vector Number



Inputing Data Vector Coded



Program Developement



Adresni načini rada

kod 8086

Addressing Modes on the x86

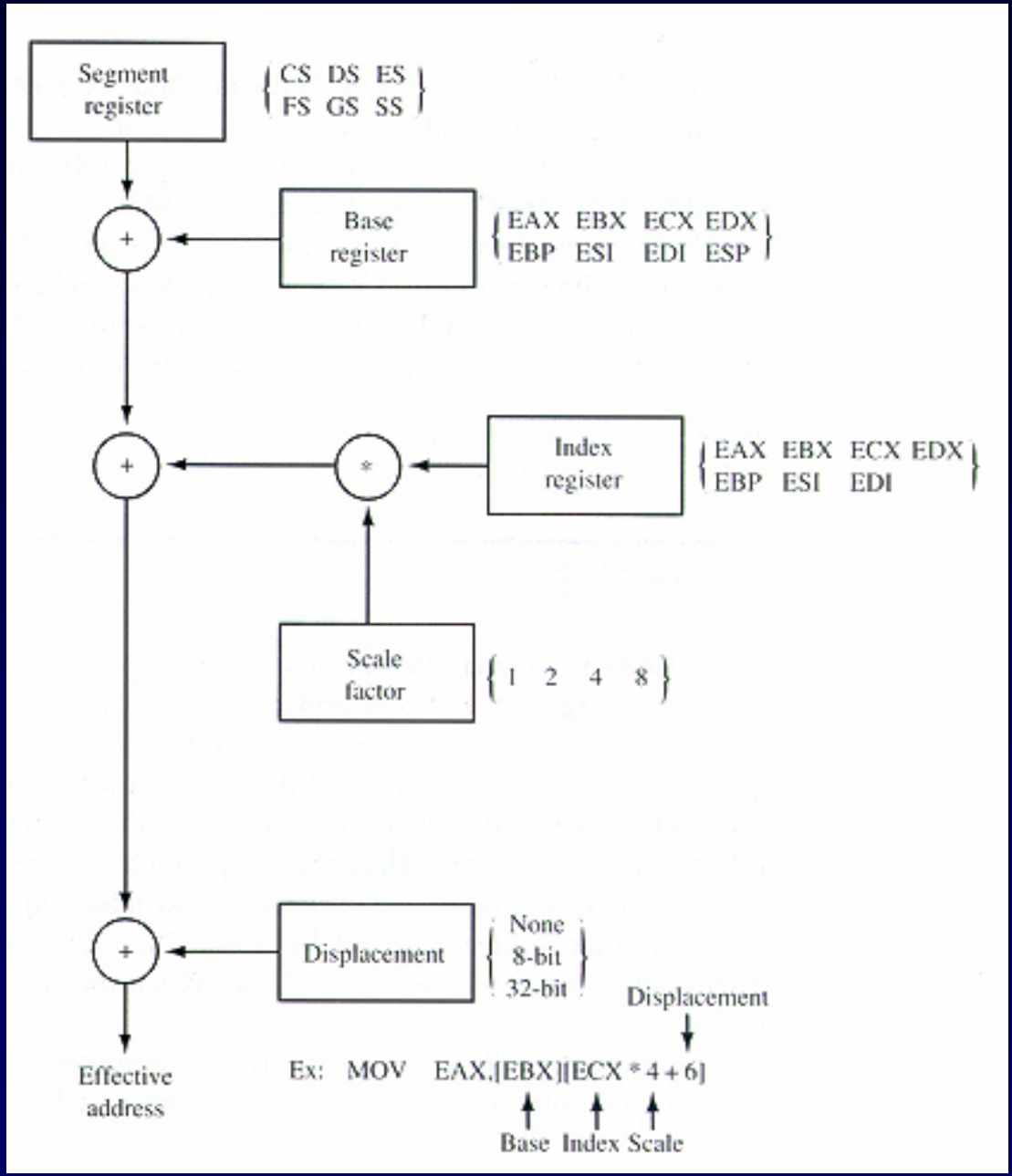
- The x86 instructions use five different operand types:
  - registers,
  - constants,
  - three memory addressing schemes.
- Each form is called an *addressing mode*.
- The x86 processors support:
  - the *register* addressing mode,
  - the *immediate* addressing mode,
  - the *indirect* addressing mode,
  - the *indexed* addressing mode, and
  - the *direct* addressing mode.

Adresni načini rada za pristup podacima kod x86

TIP	INSTRUKCIJA	IZVORIŠTE	GENERISANJE ADRESE	ODREDIŠTE
Registarsko	**MOV** BX, AX	registar BX		registar AX
Neposredno	**MOV** CH, 3AH	podatak 3AH		registar CH
Direktno	**MOV** [1234H], AX	registar AX	DS*10H+DISP 10000H+1234H	memorijska adresa 11234H
Registarsko indirektno	**MOV** [BX], CL	registar CL	DS*10H+BX 10000H+0300H	memorijska adresa 10300H
Bazno-plus-indeksno	**MOV** [BX+SI], BP	registar BP	DS*10H+BX+SI 10000H+0300H+0200H	memorijska adresa 10500H
Registarsko relativno	**MOV** CL, [BX+4]	memorijska adresa 10304H	DS*10H+BX+4 10000H+0300H+4	registar CL
Bazno relativno-plus-indeksirano	**MOV** POLJE [BX+SI], DX	registar DX	DS*10H+BX+SI 10000H+1000H+0300H+0200H	memorijska adresa 11500H
Skaliranje-sa-indeksiranjem	**MOV** [EBX+2*ESI], AX	registar AX	DS*10H+EBX+2*ESI 10000H+00000300H+00000400H	memorijska adresa 10700H

EBX=00000300h; ESI=00000200h; POLJE=1000h; DS=1000h

Generisanje 32-bitne adrese



Displacement Only (Direct) Addressing Mode

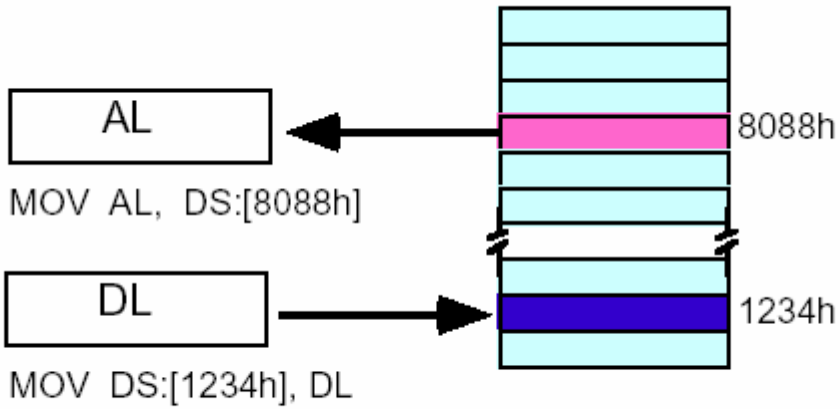


Figure 4.8 Displacement Only (Direct) Addressing Mode

Accessing a Word

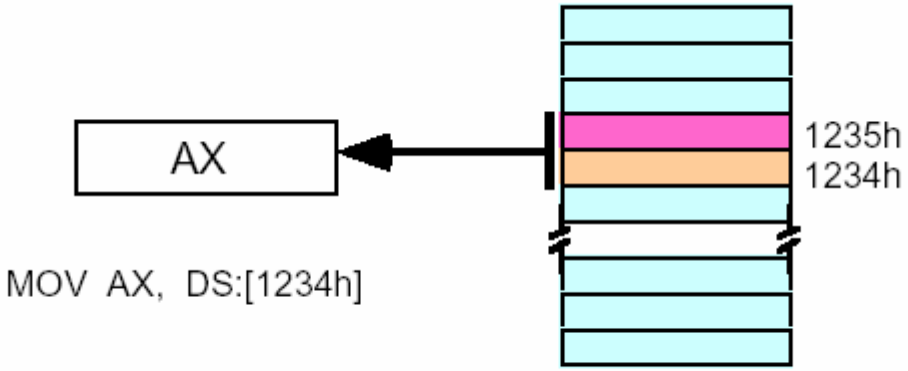


Figure 4.9 Accessing a Word

[BX] Addressing Mode

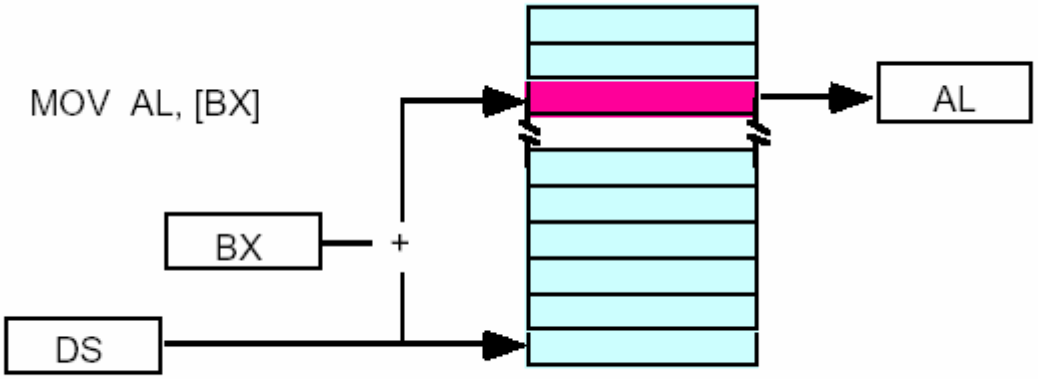


Figure 4.10 [BX] Addressing Mode

[BP] Addressing Mode

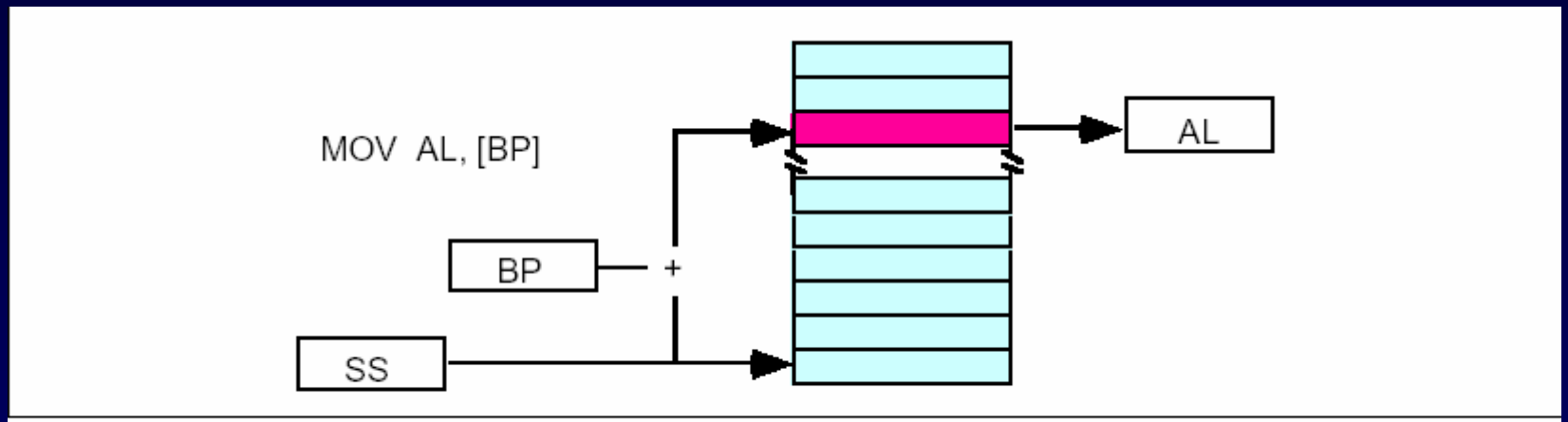


Figure 4.11 [BP] Addressing Mode

[BX+disp] Addressing Mode

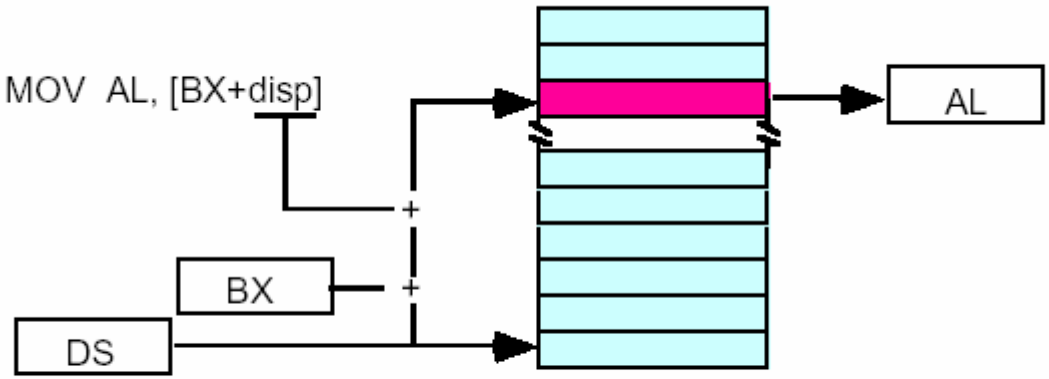


Figure 4.12 [BX+disp] Addressing Mode

[BP+disp] Addressing Mode

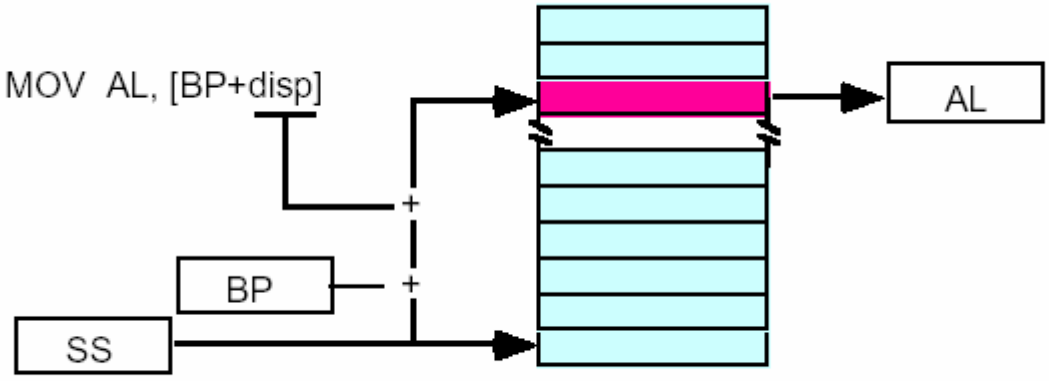


Figure 4.13 [BP+disp] Addressing Mode

[BX+SI] Addressing Mode

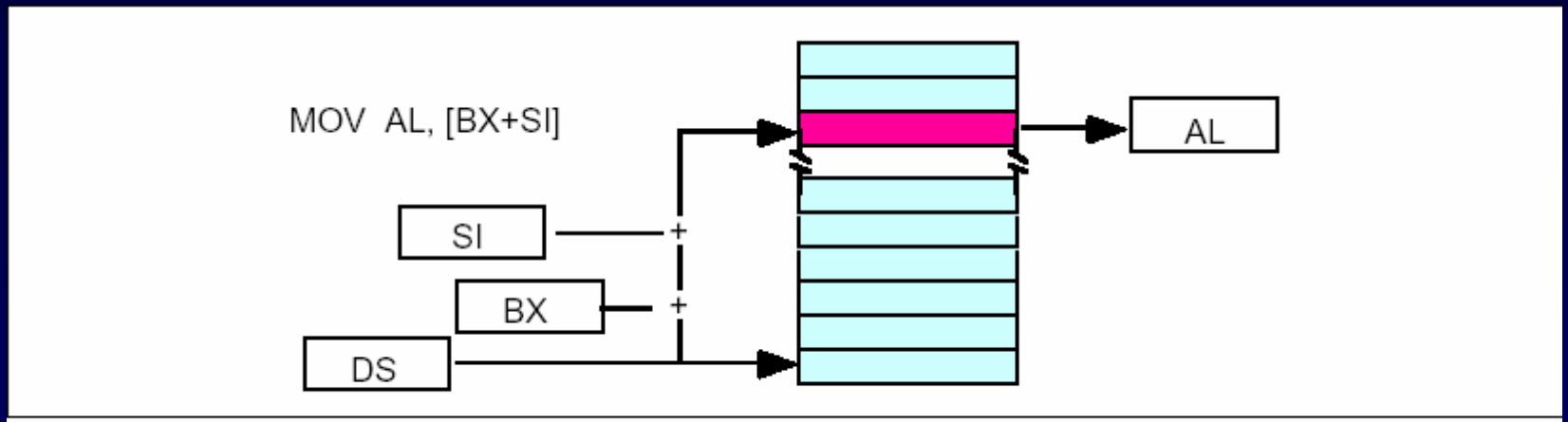


Figure 4.14 [BX+SI] Addressing Mode

[BP+SI] Addressing Mode

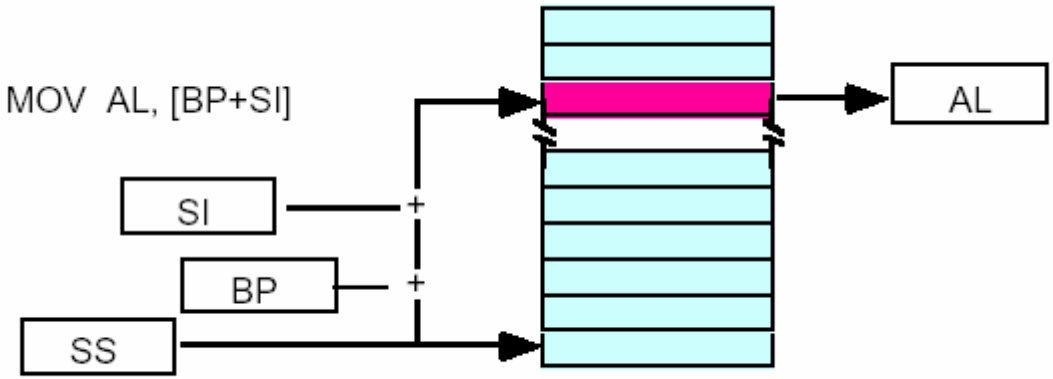


Figure 4.15 [BP+SI] Addressing Mode

[BX + SI + disp] Addressing Mode

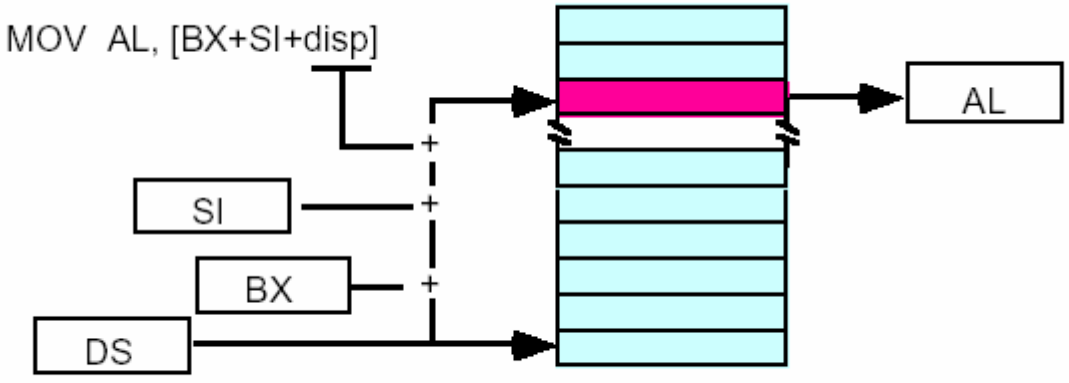


Figure 4.16 [BX + SI + disp] Addressing Mode

[BP + SI + disp] Addressing Mode

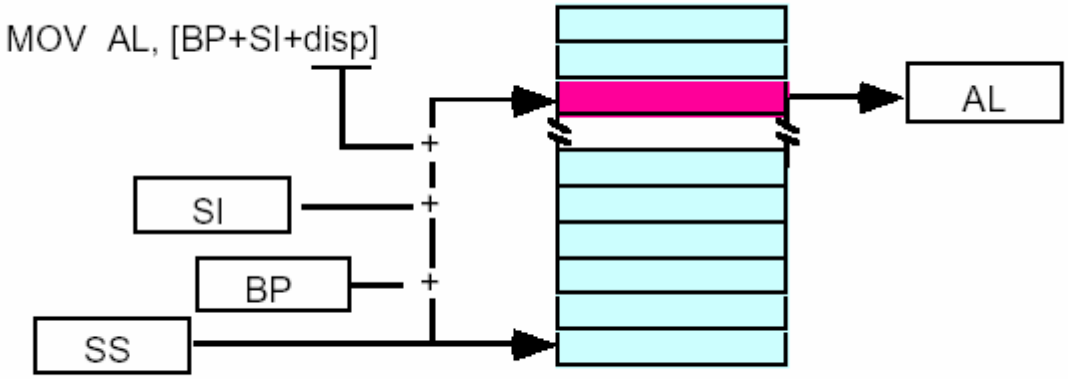


Figure 4.17 [BP + SI + disp] Addressing Mode

Table to Generate Valid 8086 Addressing Modes

DISP	[BX]	[SI]
	[BP]	[DI]

Figure 4.18 Table to Generate Valid 8086 Addressing Modes

Mapping a 4 x 4 Array to Memory

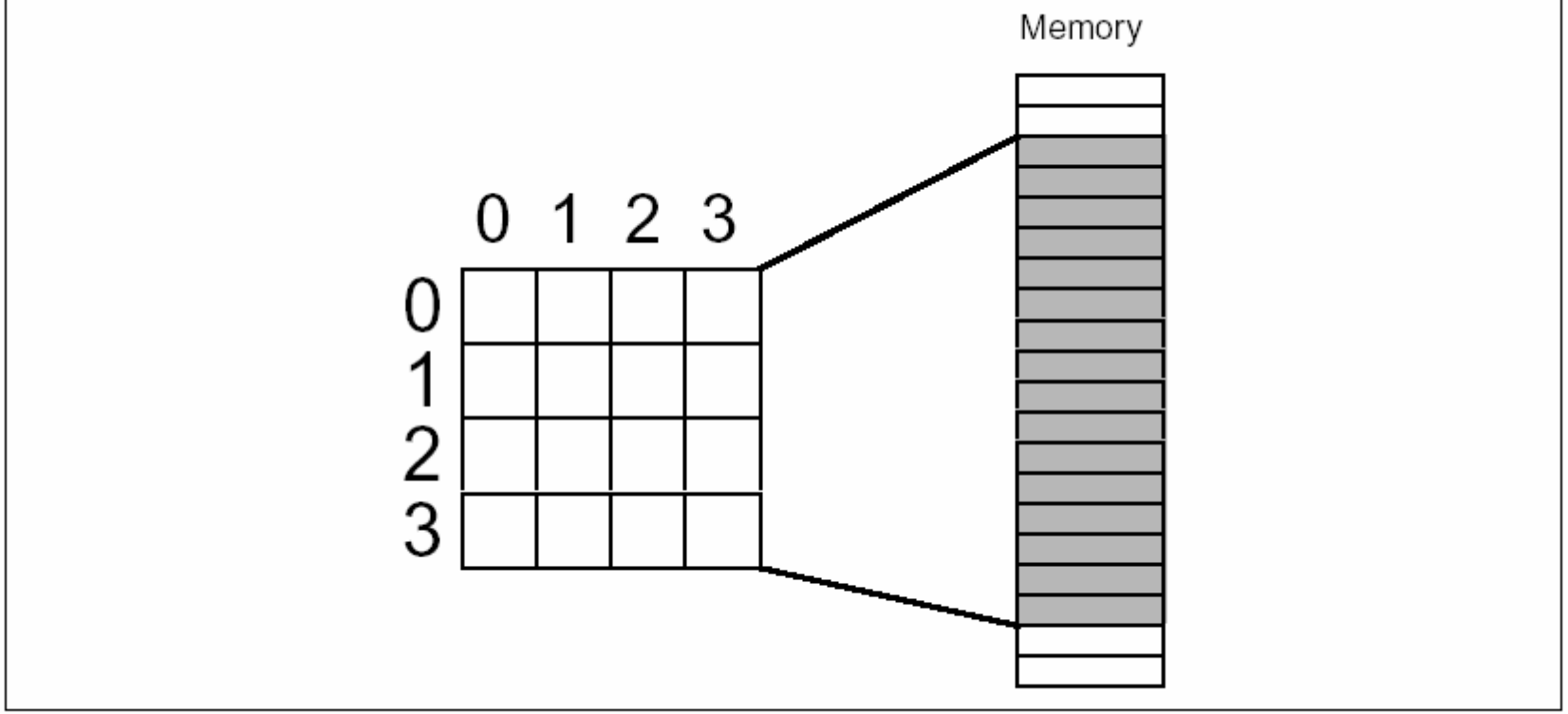


Figure 5.2 Mapping a 4 x 4 Array to Memory

Row Major Element Ordering

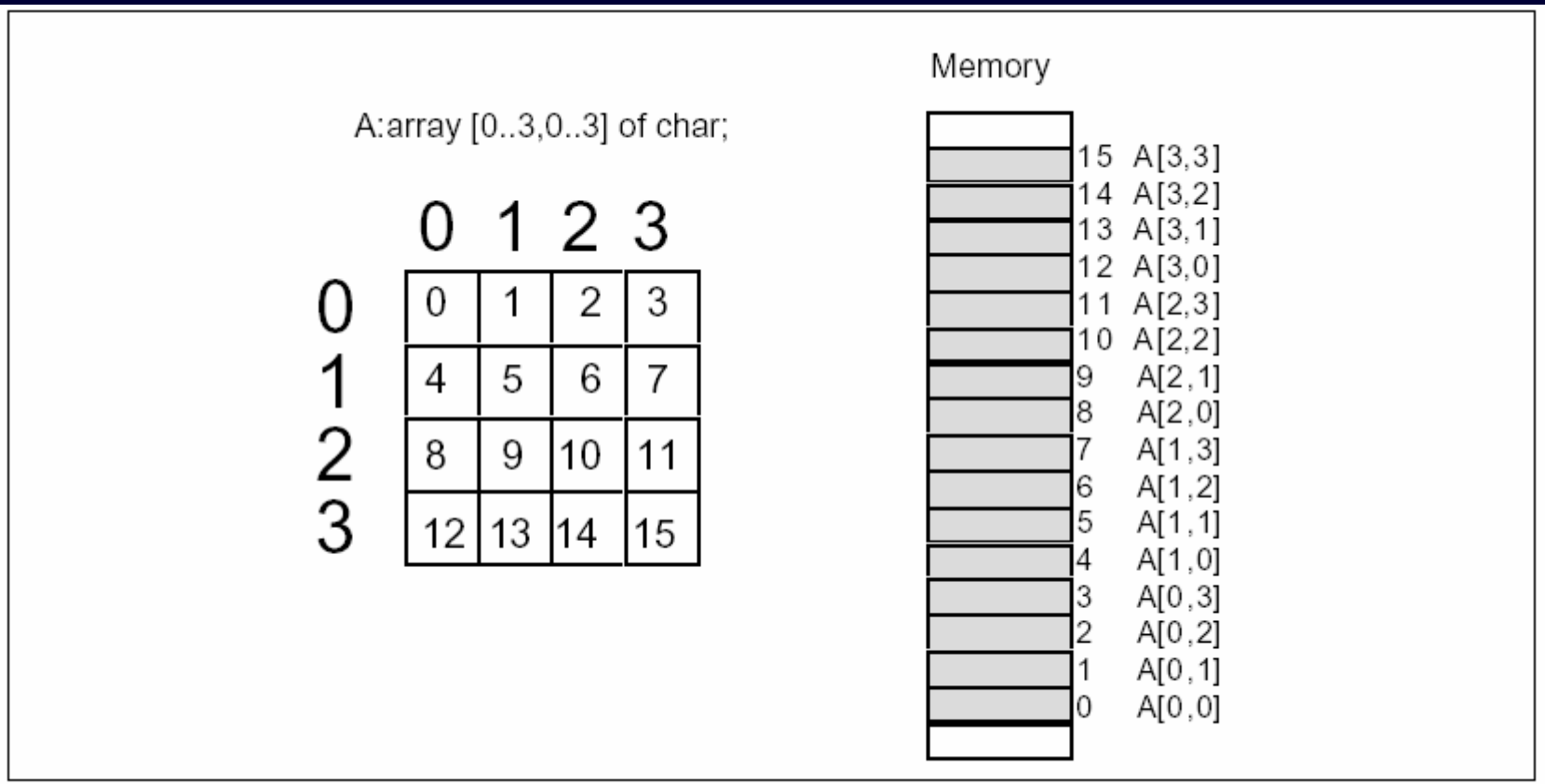


Figure 5.3 Row Major Element Ordering

Another View of Row Major Ordering for a 4x4 Array

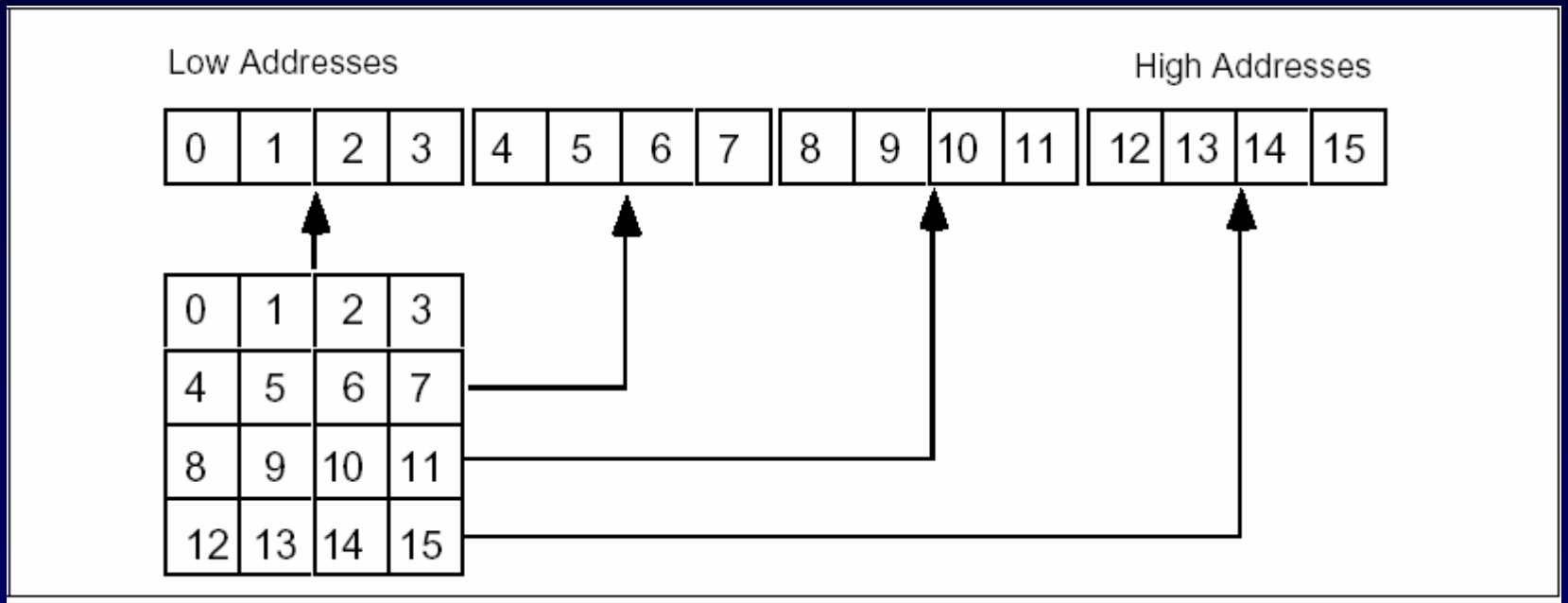


Figure 5.4 Another View of Row Major Ordering for a 4x4 Array

Column Major Element Ordering

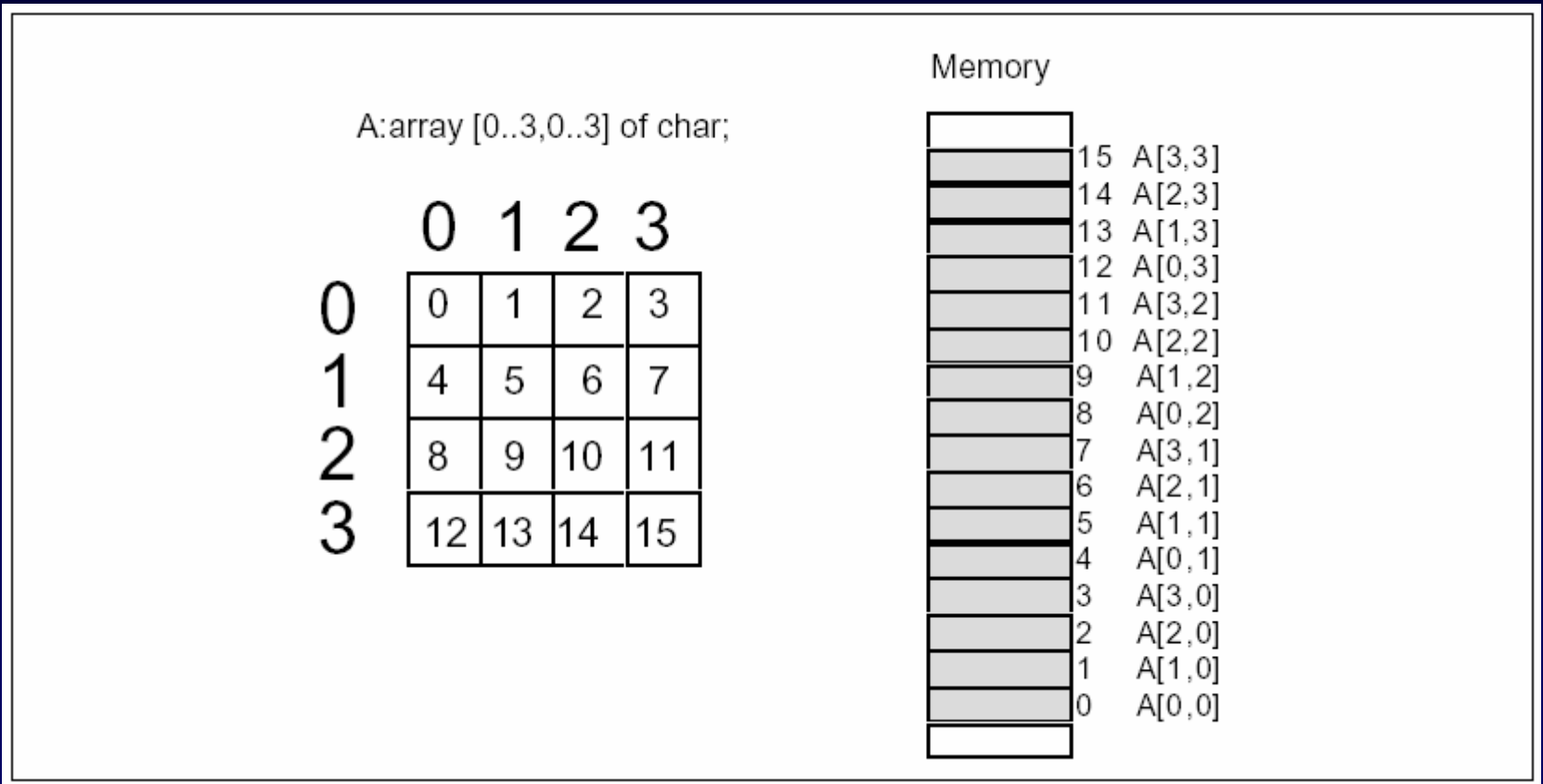
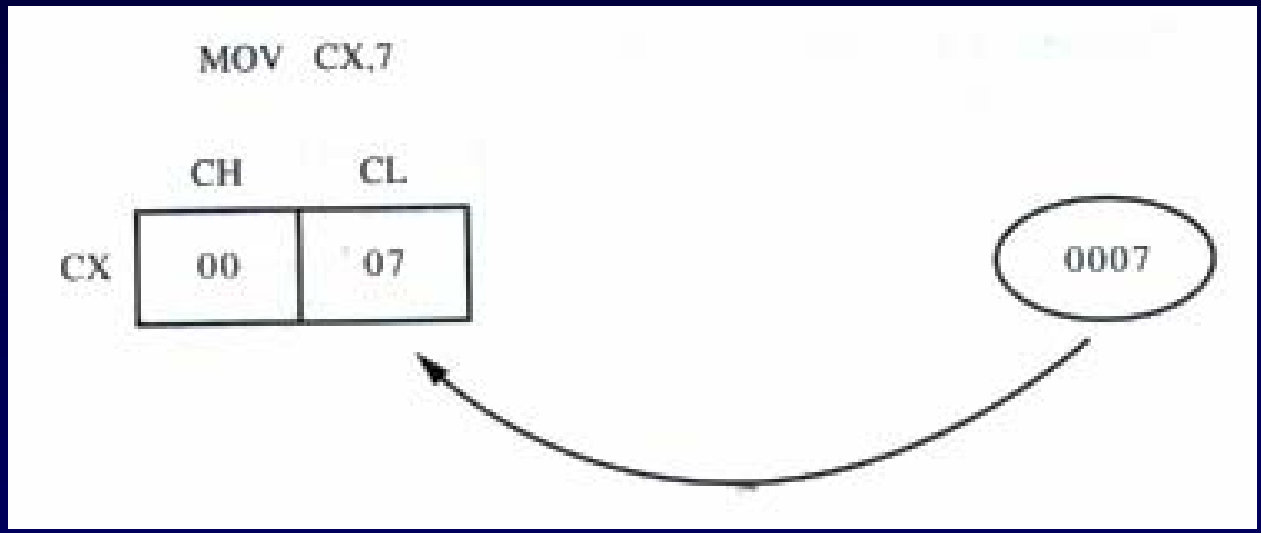
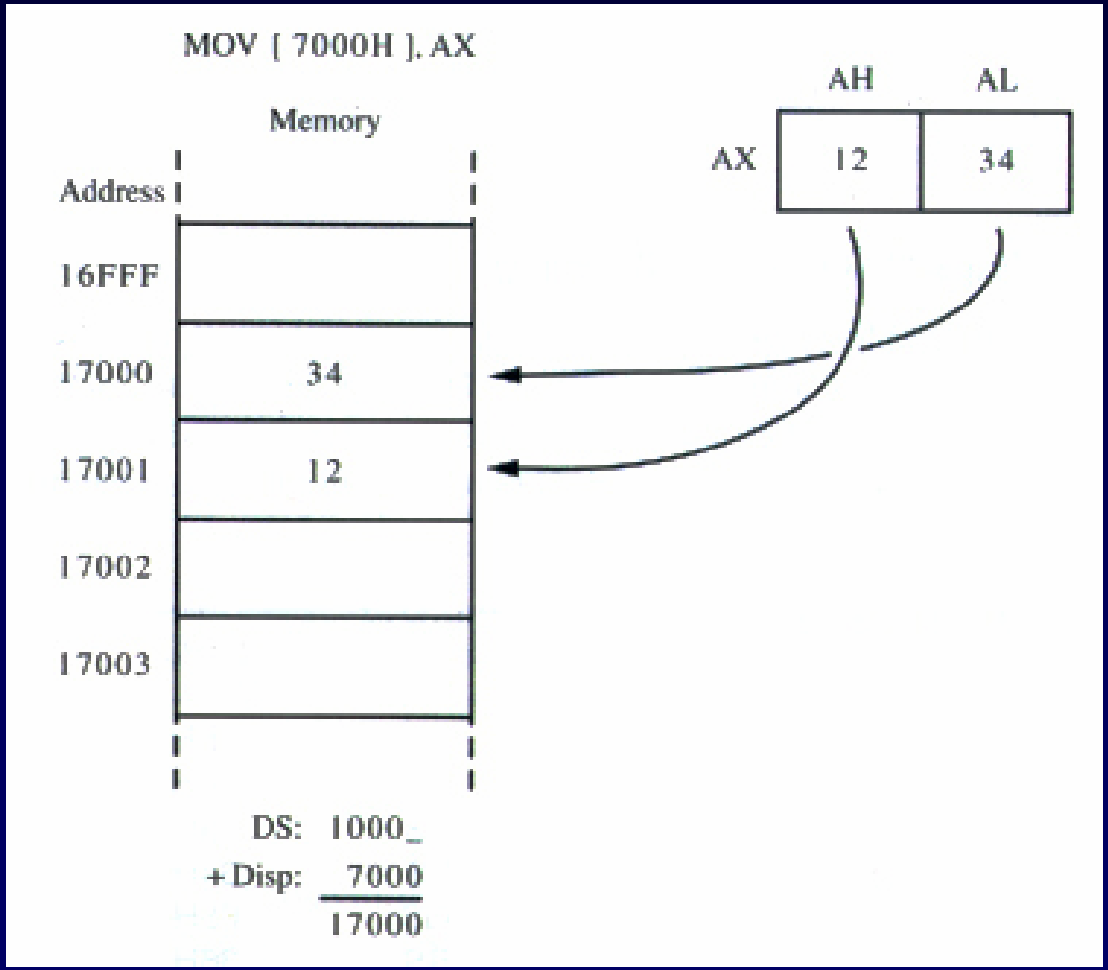


Figure 5.6 Column Major Element Ordering

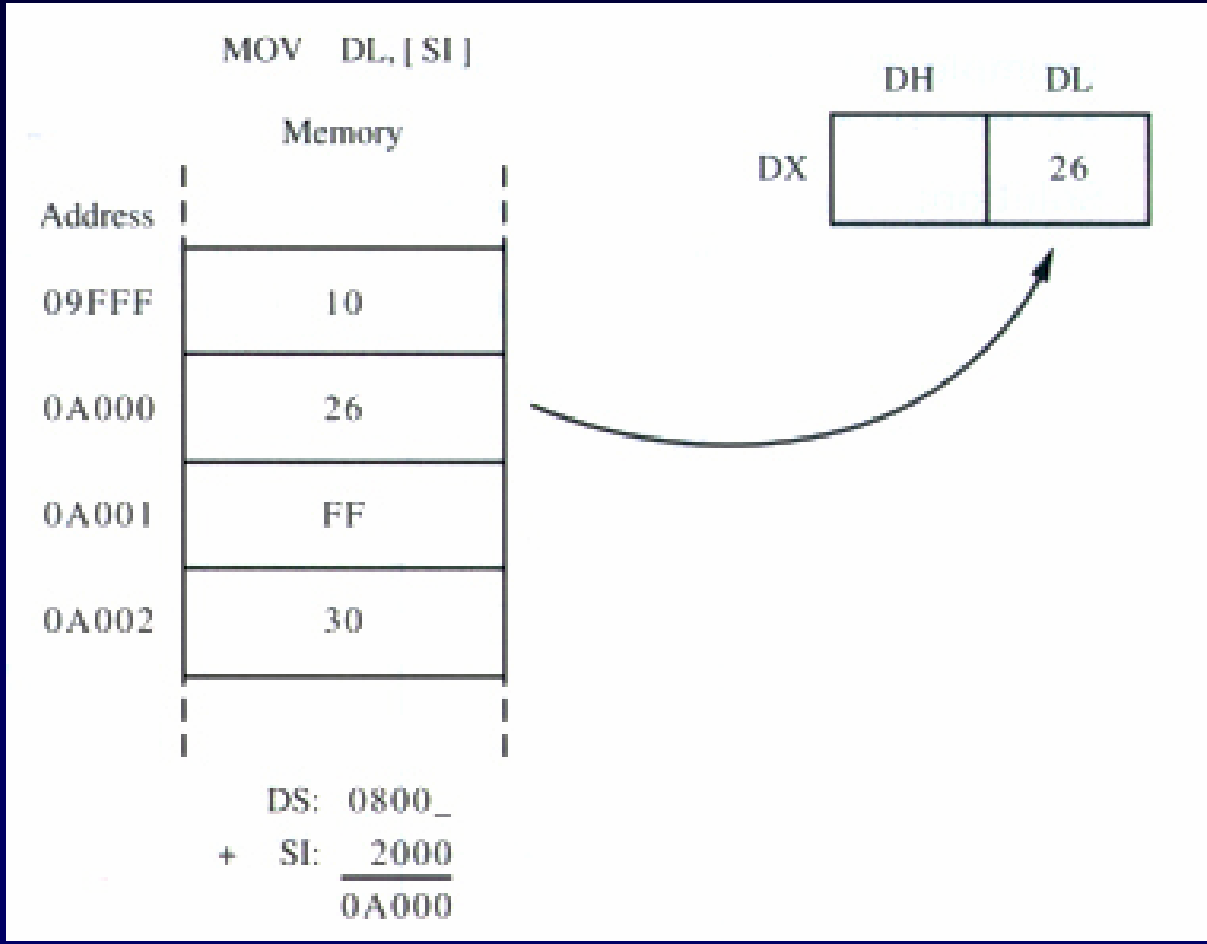
Primer neposrednog adresiranja



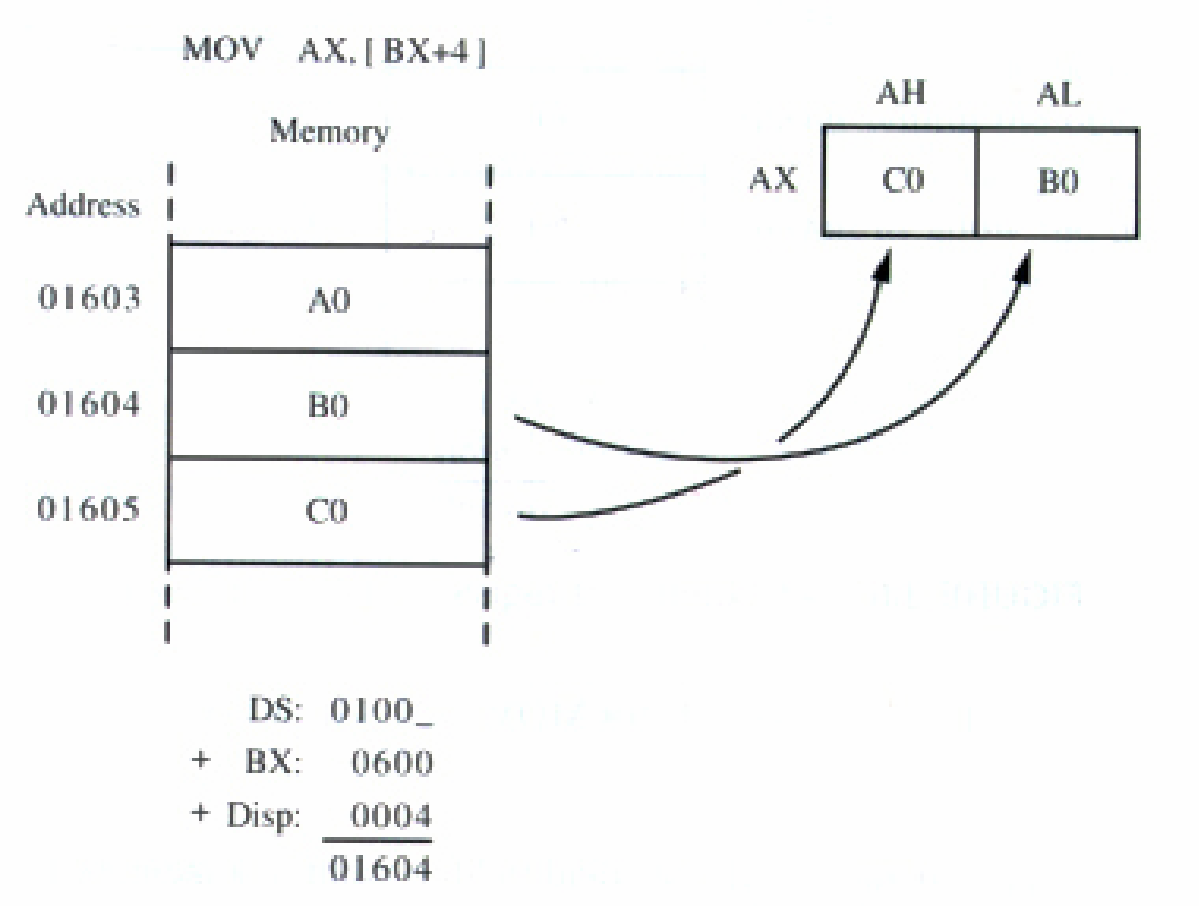
Primer direktnog adresiranja



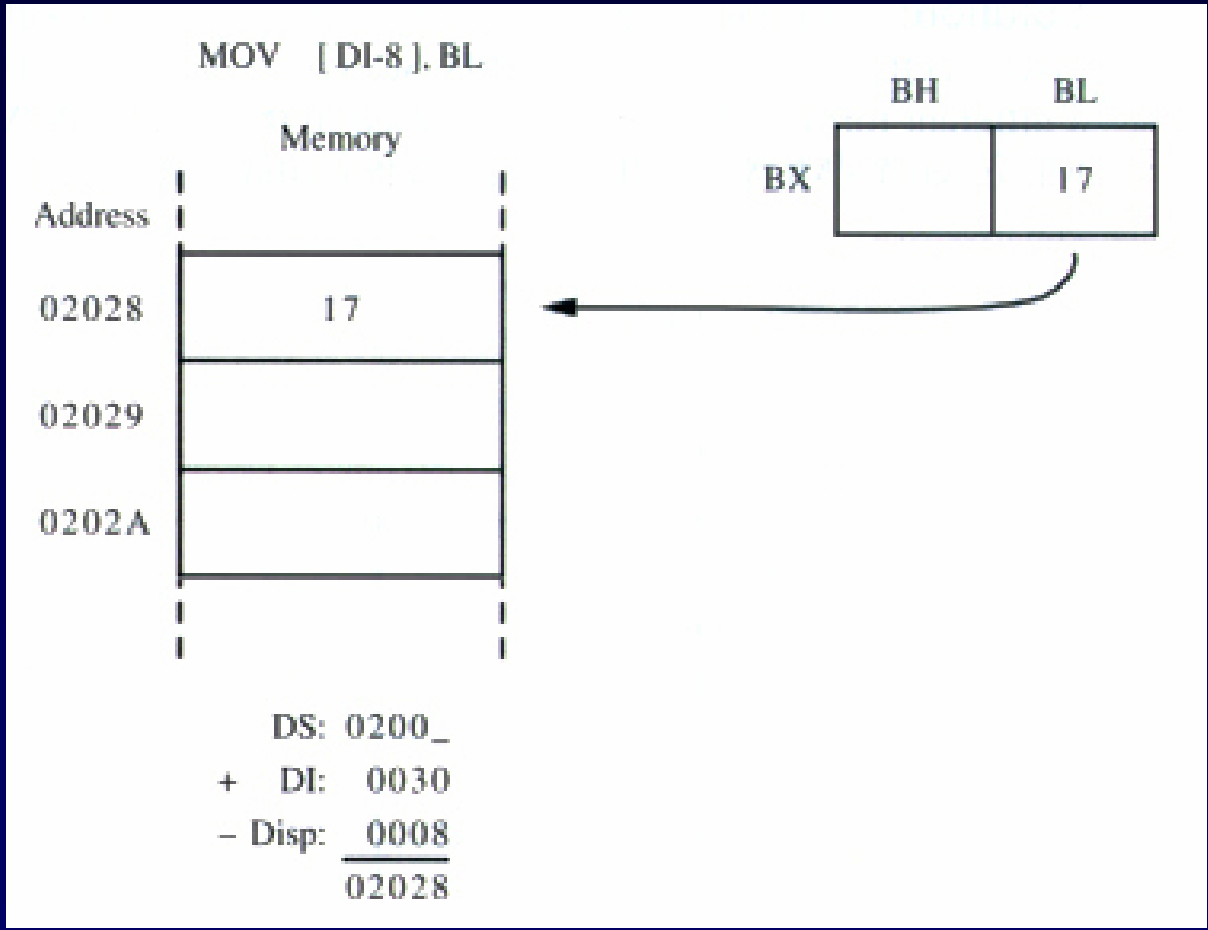
Primer registarsko-indirektnog adresiranja



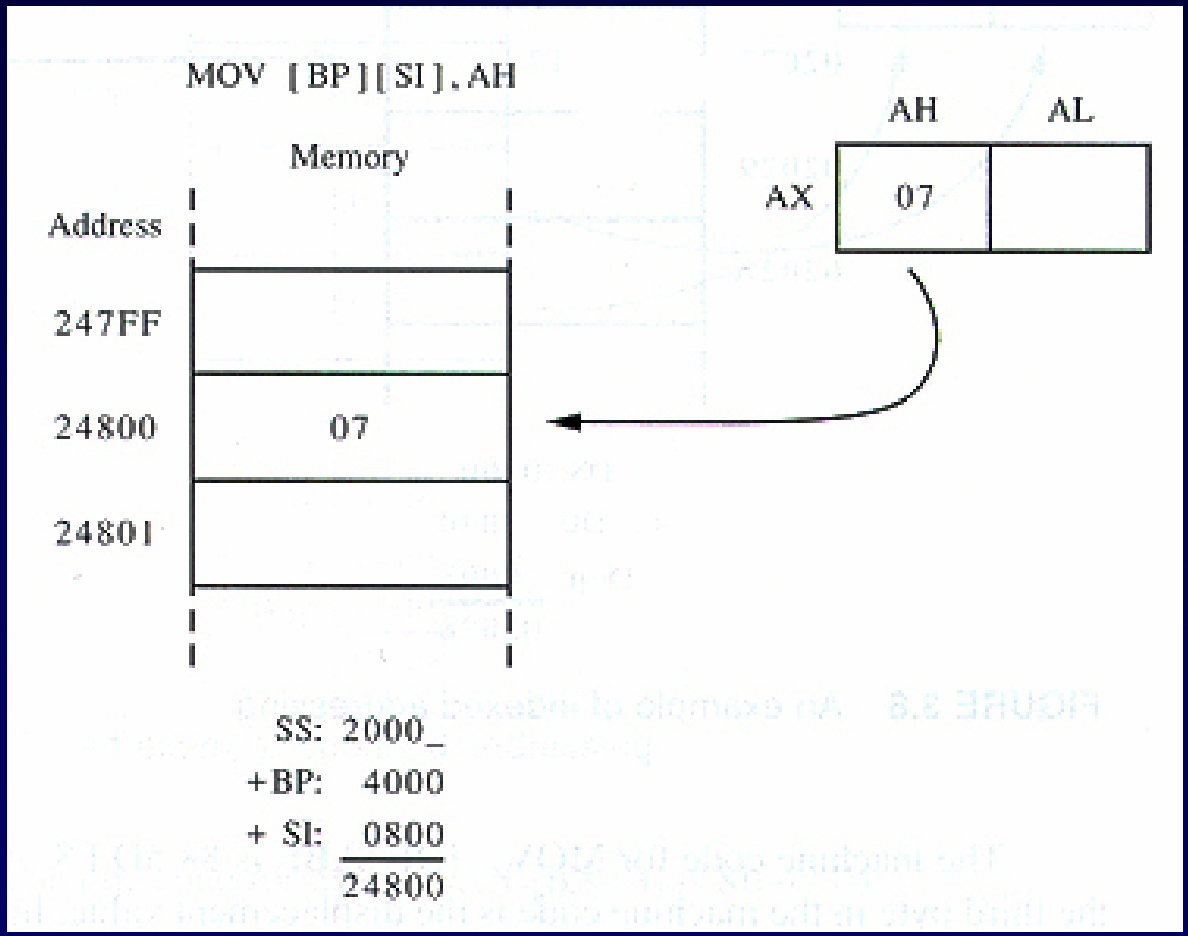
Primer baznog adresiranja



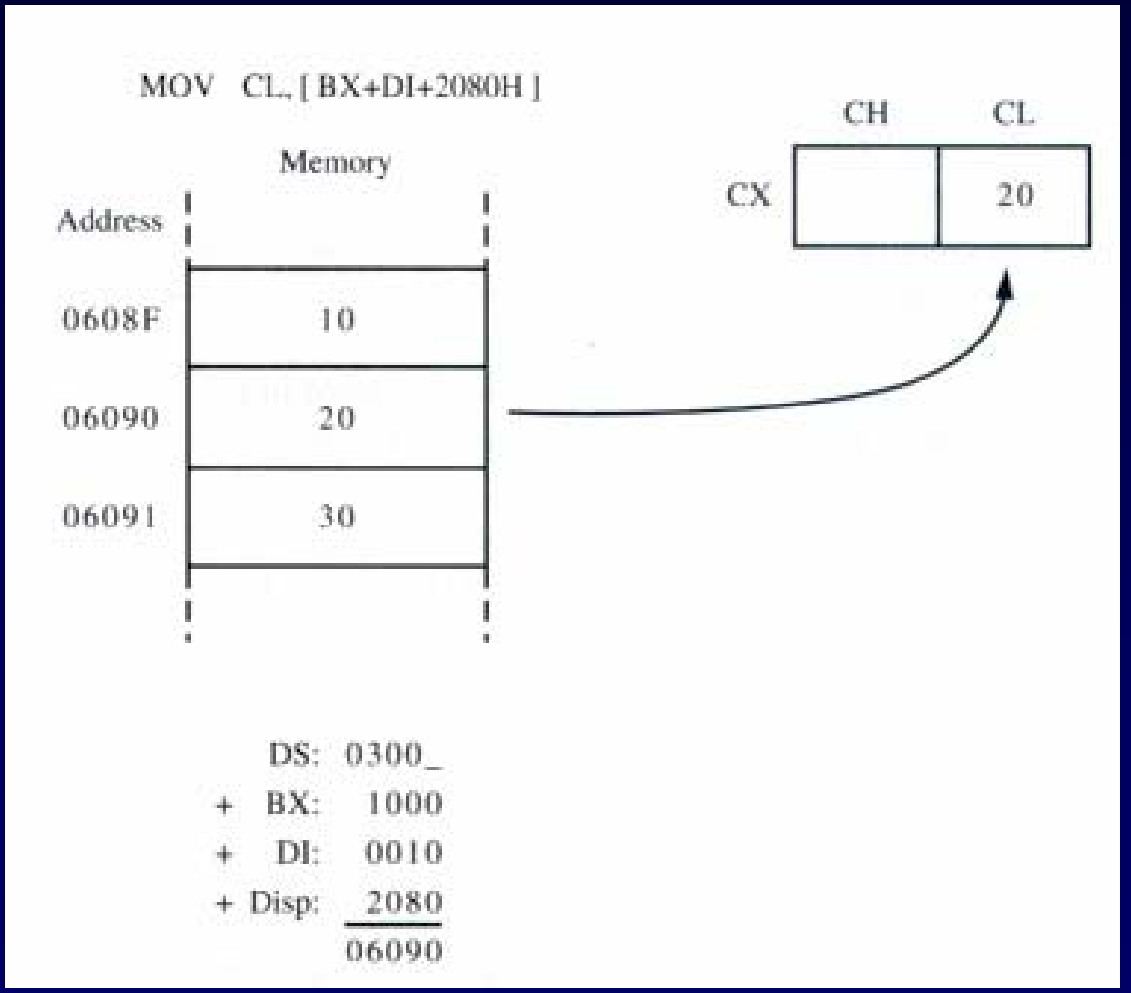
Primer indeksnog adrsiranja



Primer bazno-indeksnog adresiranja



Primer bazno-indeksnog adresiranja sa razmeštajem

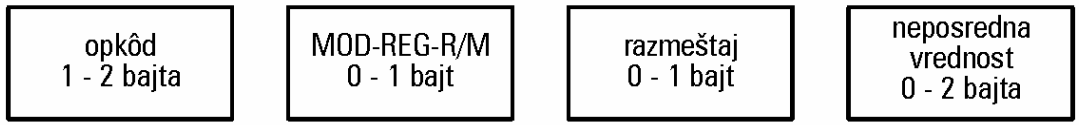


Format

instrukcija

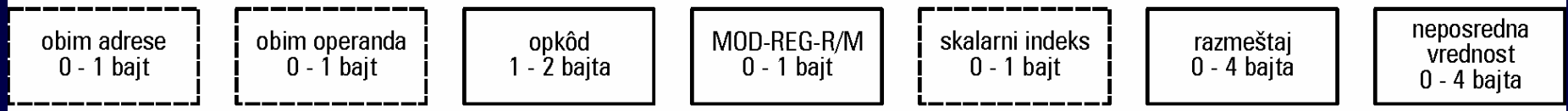
Format instrukcija od 8086 do Pentium

16-bitni instrukcioni način rada

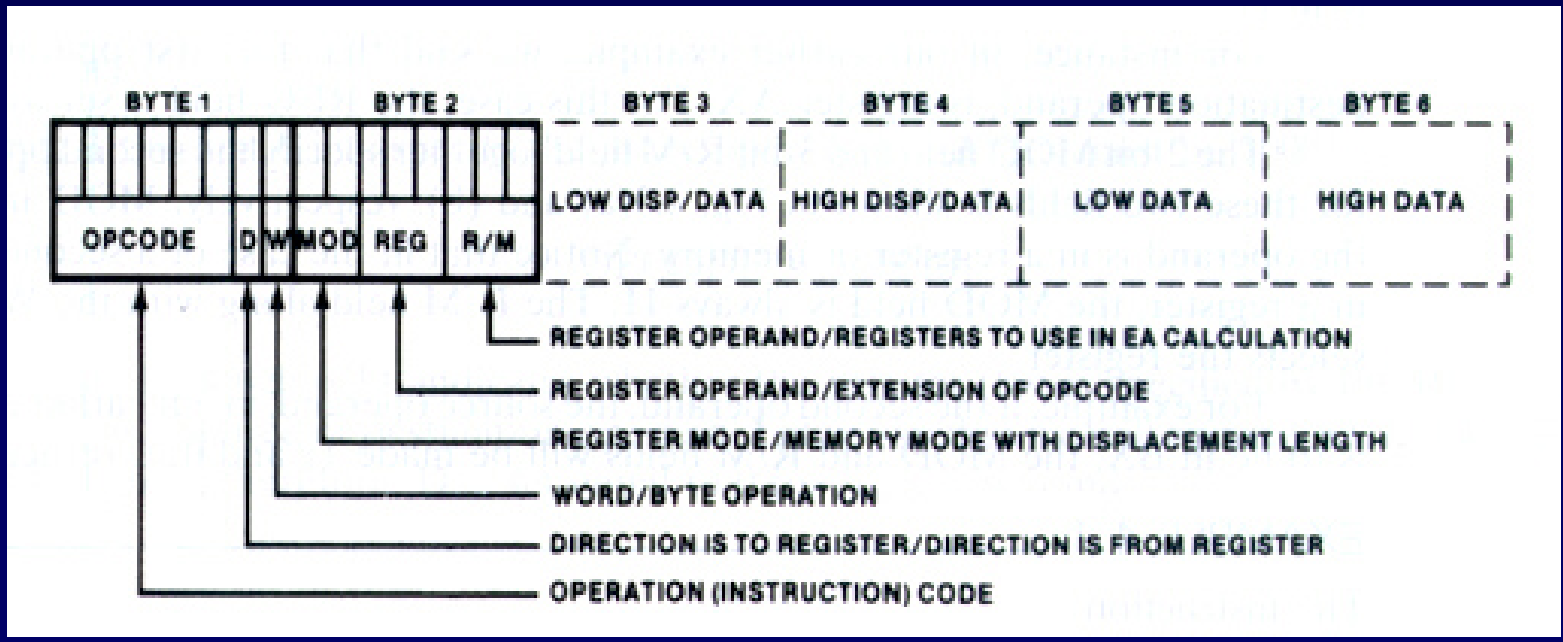


a) 16-bitni format

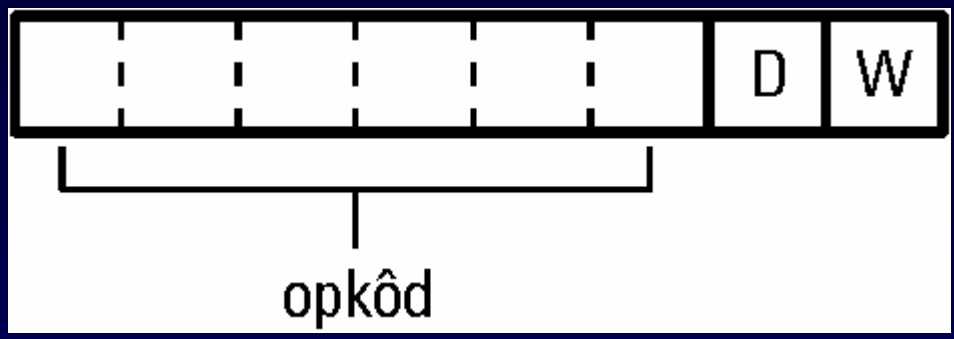
32-bitni instrukcioni način rada (80386/.../Pentium Pro)



b) 32-bitni format



Prvi bajt opkoda, pozicija polja MOD, REG i R/M



Tipovi instrukcija kod 8086

Podela instrukcija

- Prenos podataka
- Aritmetičke
- Logičke
- Pomeračke
- Rotiranje
- Testiranje i analiza bita
- Manipulisanje markerima
- Kompariranje i postavljanje
- Grananje
- Poziv potprograma
- LOOP
- Manipulisanje nizovima

Instrukcije za prenos podataka

Mnemonic	Meaning	Format	Operation	Flags affected
MOV	Move	MOV D, S	(S) → (D)	None

(a)

Destination	Source
Memory	Accumulator
Accumulator	Memory
Register	Register
Register	Memory
Memory	Register
Register	Immediate
Memory	Immediate
Seg-reg	Reg16
Seg-reg	Mem16
Reg16	Seg-reg
Mem16	Seg-reg
Register	Spec-reg
Spec-reg	Register

(b)

Sign-extend and Zero-extend Move Instructions

Mnemonic	Meaning	Format	Operation	Flags affected
MOVSX	Move with sign-extend	MOVSX D, S	(S) → (D) MSBs of D are filled with sign bit of S	None
MOVZX	Move with zero-extend	MOVZX D, S	(S) → (D) MSBs of D are filled with 0	None

(a)

Destination	Source
Reg16	Reg8
Reg32	Reg8
Reg32	Reg16
Reg16	Mem8
Reg32	Mem8
Reg32	Mem16

(b)

Exchange Data-Transfer Instructions

Mnemonic	Meaning	Format	Operation	Flags affected
XCHG	Exchange	XCHG D, S	(D) ↔ (S)	None

(a)

Destination	Source
Accumulator	Reg16
Accumulator	Reg32
Memory	Register
Register	Register

(b)

Translate Data-Transfer Instructions

Mnemonic	Meaning	Format	Operation	Flags affected
XLAT XLATB	Translate Translate	XLAT Source-table XLATB	((AL) + (BX) + (DS)0) → (AL) ((AL) + (BX) + (DS)0) → (AL)	None None

Load Effective Address and Full Pointer Data-Transfer Instructions

Mnemonic	Meaning	Format	Operation	Flags affected
LEA	Load effective address	LEA Reg16, EA	(EA) → (Reg16)	None
		LEA Reg32, EA	(EA) → (Reg32)	None
LDS	Load register and DS	LDS Reg16, EA	(EA) → (Reg16) (EA + 2) → (DS)	None
		LDS Reg32, EA	(EA) → (Reg32) (EA + 4) → (DS)	None
LSS	Load register and SS	LSS Reg16, EA	(EA) → (Reg16) (EA + 2) → (SS)	None
		LSS Reg32, EA	(EA) → (Reg32) (EA + 4) → (SS)	None
LES	Load register and ES	LES Reg16, EA	(EA) → (Reg16) (EA + 2) → (ES)	None
		LES Reg32, EA	(EA) → (Reg32) (EA + 4) → (DS)	None
LFS	Load register and FS	LFS Reg16, EA	(EA) → (Reg16) (EA + 2) → (FS)	None
		LFS Reg32, EA	(EA) → (Reg32) (EA + 4) → (FS)	None
LGS	Load register and GS	LGS Reg16, EA	(EA) → (Reg16) (EA + 2) → (GS)	None
		LGS Reg32, EA	(EA) → (Reg32) (EA + 4) → (GS)	None

Arithmetic Instructions

Addition	
ADD	Add byte or word
ADC	Add byte or word with carry
INC	Increment byte or word by 1
AAA	ASCII adjust for addition
DAA	Decimal adjust for addition
Subtraction	
SUB	Subtract byte or word
SBB	Subtract byte or word with borrow
DEC	Decrement byte or word by 1
NEG	Negate byte or word
AAS	ASCII adjust for subtraction
DAS	Decimal adjust for subtraction
Multiplication/Division	
MUL	Multiply byte or word unsigned
IMUL	Integer multiply byte or word
AAM	ASCII adjust for multiply
DIV	Divide byte or word unsigned
IDIV	Integer divide byte or word
AAD	ASCII adjust for division
CBW	Convert byte to word
CWDE	Convert word to double word in EAX
CWD	Convert word to double word in DX and AX
CDQ	Convert double word to quad word

Arithmetic and Logical Instructions

The *arithmetic and logical instructions* take the following forms:

```
add     reg, reg/memory/constant
sub     reg, reg/memory/constant
cmp     reg, reg/memory/constant
and     reg, reg/memory/constant
or      reg, reg/memory/constant
not     reg/memory
```

Addition Arithmetic Instructions

Mnemonic	Meaning	Format	Operation	Flags affected
ADD	Addition	ADD D, S	(S) + (D) → (D) carry → (CF)	OF, SF, ZF, AF, PF, CF
ADC	Add with carry	ADC D, S	(S) + (D) + (CF) → (D) carry → (CF)	OF, SF, ZF, AF, PF, CF
INC	Increment by 1	INC D	(D) + 1 → (D)	OF, SF, ZF, AF, PF
DAA	Decimal adjust for addition	DAA		SF, ZF, AF, PF, CF OF undefined
AAA	ASCII adjust for addition	AAA		AF, CF OF, SF, ZF, PF undefined

(a)

Destination	Source
Register	Register
Register	Memory
Memory	Register
Register	Immediate
Memory	Immediate
Accumulator	Immediate

(b)

Destination
Reg16
Reg8
Memory

(c)

Subtraction Arithmetic Instructions

Mnemonic	Meaning	Format	Operation	Flags affected
SUB	Subtract	SUB D,S	(D) − (S) → (D) Borrow → (CF)	OF, SF, ZF, AF, PF, CF
SBB	Subtract with borrow	SBB D,S	(D) − (S) − (CF) → (D)	OF, SF, ZF, AF, PF, CF
DEC	Decrement by 1	DEC D	(D) − 1 → (D)	OF, SF, ZF, AF, PF
NEG	Negate	NEG D	0 − (D) → (D) 1 → (CF)	OF, SF, ZF, AF, PF, CF
DAS	Decimal adjust for subtraction	DAS		SF, ZF, AF, PF, CF OF undefined
AAS	ASCII adjust for subtraction	AAS		AF, CF OF, SF, ZF, PF undefined

(a)

Destination	Source
Register	Register
Register	Memory
Memory	Register
Accumulator	Immediate
Register	Immediate
Memory	Immediate

(b)

Destination
Register
Memory

(c)

Multiplication and Division Arithmetic Instructions

Mnemonic	Meaning	Format	Operation	Flags affected
MUL	Multiply (unsigned)	MUL S	(AL) • (S8) → (AX) (AX) • (S16) → (DX), (AX) (EAX) • (S32) → (EDX), (EAX)	OF, CF SF, ZF, AF, PF undefined
DIV	Division (unsigned)	DIV S	(1) Q ((AX)/(S8)) → (AL) R ((AX)/(S8)) → (AH) (2) Q((DX, AX)/(S16)) → (AX) R ((DX, AX)/(S16)) → (DX) (3) Q ((EDX, EAX)/(S32)) → (EAX) R ((EDX, EAX)/(S32)) → (EDX) If Q is FF_{16} in case (1), $FFFF_{16}$ in case (2), or $FFFFFFFF_{16}$ in case (3), then type 0 interrupt occurs	All flags undefined
IMUL	Integer multiply (signed)	IMUL S	(AL) • (S8) → (AX) (AX) • (S16) → (DX), (AX) (EAX) • (S32) → (EDX), (EAX)	OF, CF SF, ZF, AF, PF undefined
		IMUL R, I	(R16) • (Imm8) → (R16) (R32) • (Imm8) → (R32) (R16) • (Imm16) → (R16) (R32) • (Imm32) → (R32)	OF, CF SF, ZF, AF, PF undefined
		IMUL R, S, I	(S16) • (Imm8) → (R16) (S32) • (Imm8) → (R32) (S16) • (Imm16) → (R16) (S32) • (Imm32) → (R32)	OF, CF SF, ZF, AF, PF undefined
		IMUL R, S	(R16) • (S16) → (R16) (R32) • (S32) → (R32)	OF, CF SF, ZF, AF, PF undefined
IDIV	Integer divide (signed)	IDIV S	(1) Q ((AX)/(S8)) → (AL) R ((AX)/(S8)) → (AH) (2) Q ((DX, AX)/(S16)) → (AX) R ((DX, AX)/(S16)) → (DX) (3) Q ((EDX, EAX)/(S32)) → (EAX) R ((EDX, EAX)/(S32)) → (EDX) If Q is positive and exceeds $7FFF_{16}$ or if Q is negative and becomes less than 8001_{16}, then type 0 interrupt occurs	All flags undefined
AAM	Adjust AL after multiplication	AAM	Q ((AL)/10) → (AH) R ((AL)/10) → (AL)	SF, ZF, PF OF, AF, CF undefined
AAD	Adjust AX before division	AAD	(AH) • 10 + (AL) → (AL) 00 → (AH)	SF,ZF, PF OF, AF, CF undefined
CBW	Convert byte to word	CBW	(MSB of AL) → (All bits of AH)	None
CWDE	Convert word to double word	CWDE	(MSB of AX) → (16 MSBs of EAX)	None
CWD	Convert word to double word	CWD	(MSB of AX) → (All bits of DX)	None
CDQ	Convert double word to quad word	CDQ	(MSB of EAX) → (All bits of EDX)	None

(a)

Source
Reg8
Reg16
Reg32
Mem8
Mem16
Mem32

(b)

Destination	Source
	Reg8
	Reg16
	Reg32
Reg16	Imm8
Reg32	Imm8
Reg16	Imm16
Reg32	Imm32
Reg16	Reg16, Imm8
Reg16	Mem16, Imm8
Reg32	Reg32, Imm8
Reg32	Mem32, Imm8
Reg16	Reg16, Imm16
Reg16	Mem16, Imm16
Reg32	Reg16, Imm32
Reg32	Mem16, Imm32
Reg16	Reg16
Reg16	Mem16
Reg32	Reg32
Reg32	Mem32

(c)

Logic Instructions

Mnemonic	Meaning	Format	Operation	Flags Affected
AND	Logical AND	AND D,S	(S) · (D) → (D)	OF, SF, ZF, PF, CF AF undefined
OR	Logical Inclusive-OR	OR D,S	(S) + (D) → (D)	OF, SF, ZF, PF, CF AF undefined
XOR	Logical Exclusive-OR	XOR D,S	(S) ⊕ (D) → (D)	OF, SF, ZF, PF, CF AF undefined
NOT	Logical NOT	NOT D	$(\bar{D}) \rightarrow (D)$	None

(a)

Destination	Source
Register	Register
Register	Memory
Memory	Register
Register	Immediate
Memory	Immediate
Accumulator	Immediate

(b)

Destination
Register
Memory

(c)

Shift Instructions

Mnemonic	Meaning	Format	Operation	Flags affected
SAL/SHL	Shift arithmetic left/shift logical left	SAL/SHL D, Count	Shift the (D) left by the number of bit positions equal to Count and fill the vacated bit positions on the right with zeros. The last bit shifted out at MSB end is in CF.	SF, ZF, PF, CF AF undefined OF undefined if Count ≠1
SHR	Shift logical right	SHR D, Count	Shift the (D) right by the number of bit positions equal to Count and fill the vacated bit positions on the left with zeros. The last bit shifted out at the LSB end is in CF.	SF, ZF, PF, CF AF undefined OF undefined if Count ≠1
SAR	Shift arithmetic right	SAR D, Count	Shift the (D) right by the number of bit positions equal to Count and fill the vacated bit positions on the left with the original most significant bit. The last bit shifted out of the LSB end is in CF.	SF, ZF, PF, CF AF undefined OF undefined if Count ≠1
SHLD	Double precision shift left	SHLD D1, D2, Count	Shift (D1) left by the number of bit positions equal to Count. (D2) supplies the bits to be loaded into D1. Bits are shifted from the MSB of D2 to the LSB of D1. The value in D2 is unchanged. The last bit shifted out of D1 is in CF.	SF, ZF, PF, CF OF, AF undefined
SHRD	Double precision shift right	SHLD D1, D2, Count	Shift (D1) right by the number of bit positions equal to Count. (D2) supplies the bits to be loaded into D1. Bits are shifted from the LSB of D2 into the MSB of D1. The value in D2 is unchanged. The last bit shifted out of D1 is in CF.	SF, ZF, PF, CF OF, AF undefined

(a)

Destination	Count
Register	1
Register	CL
Register	Imm8
Memory	1
Memory	CL
Memory	Imm8

(b)

Destination 1	Destination 2	Count
Reg16	Reg16	Imm8
Reg16	Reg16	CL
Reg32	Reg32	Imm8
Reg32	Reg32	CL
Mem16	Reg16	Imm8
Mem16	Reg16	CL
Mem32	Reg32	Imm8
Mem32	Reg32	CL

(c)

Rotate Instructions

Mnemonic	Meaning	Format	Operation	Flags affected
ROL	Rotate left	ROL D, Count	Rotate the (D) left by the number of bit positions equal to Count. Each bit shifted out from the leftmost bit goes back into the rightmost bit position	CF OF undefined if Count ≠1
ROR	Rotate right	ROR D, Count	Rotate the (D) right by the number of bit positions equal to Count. Each bit shifted out from the rightmost bit goes into the leftmost bit position	CF OF undefined if Count ≠1
RCL	Rotate left through carry	RCL D, Count	Same as ROL except carry is attached to (D) for rotation	CF OF undefined if Count ≠1
RCR	Rotate right through carry	RCR D, Count	Same as ROR except carry is attached to (D) for rotation	CF OF undefined if Count ≠1

(a)

Destination	Count
Register	1
Register	CL
Register	Imm8
Memory	1
Memory	CL
Memory	Imm8

(b)

Arithmetic Shift Right Operation

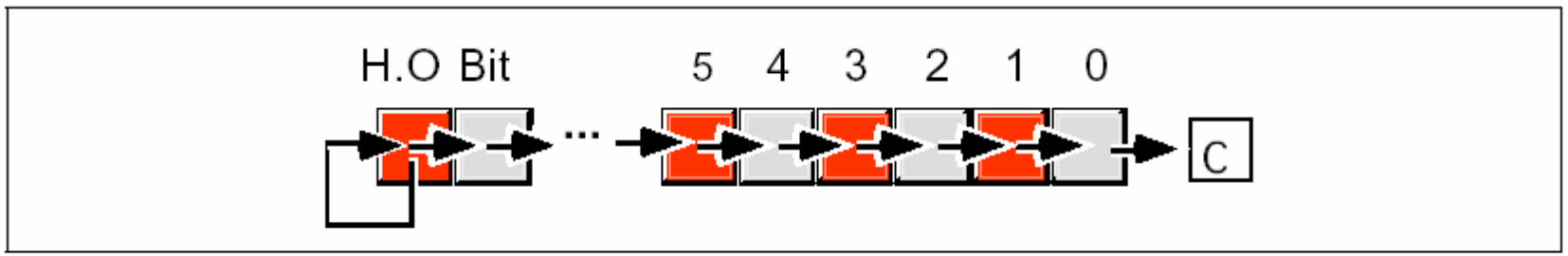


Figure 6.3 Arithmetic Shift Right Operation

Shift Right Operation

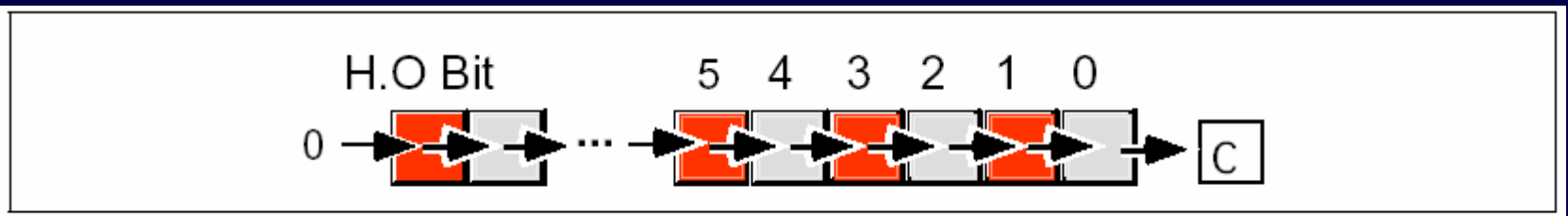


Figure 6.4 Shift Right Operation

Double Precision Shift Left Operation

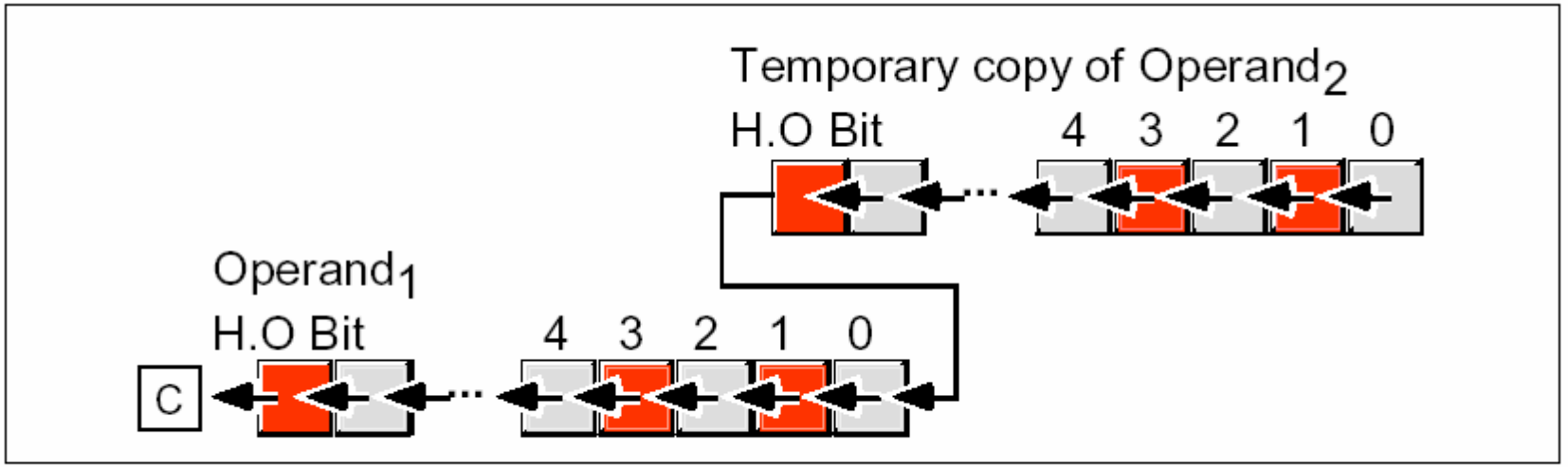


Figure 6.5 Double Precision Shift Left Operation

Double Precision Shift Right Operation

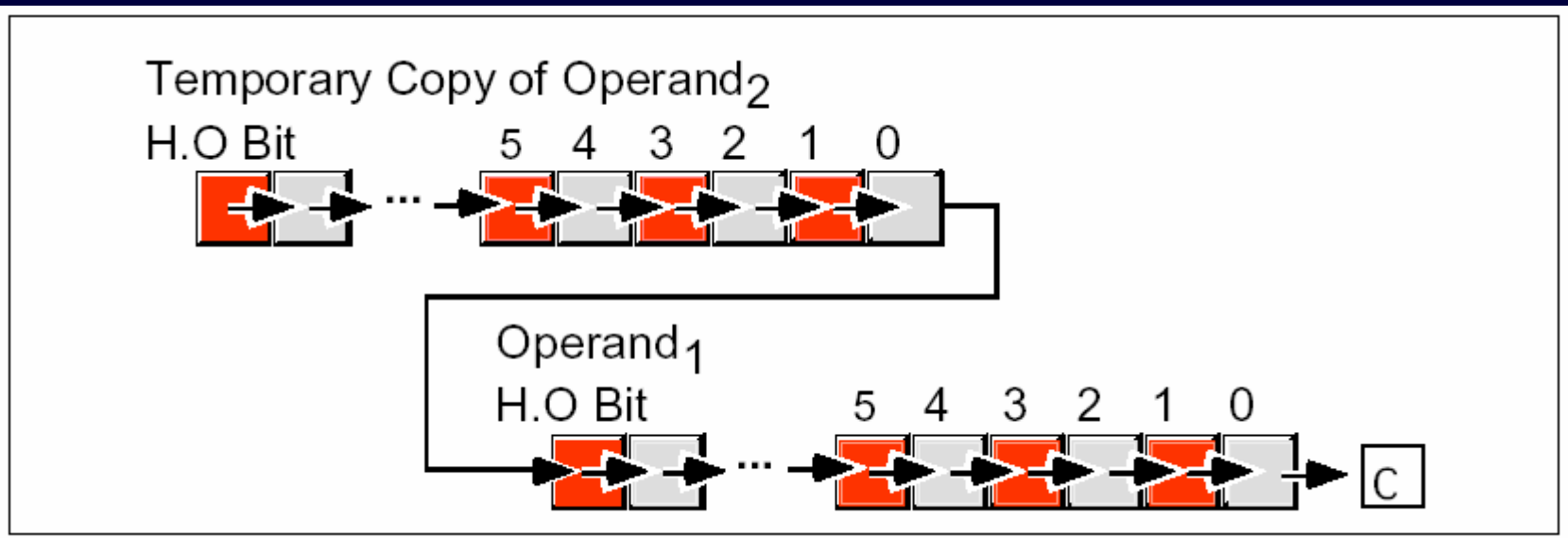


Figure 6.6 Double Precision Shift Right Operation

Rotate Through Carry Left Operation

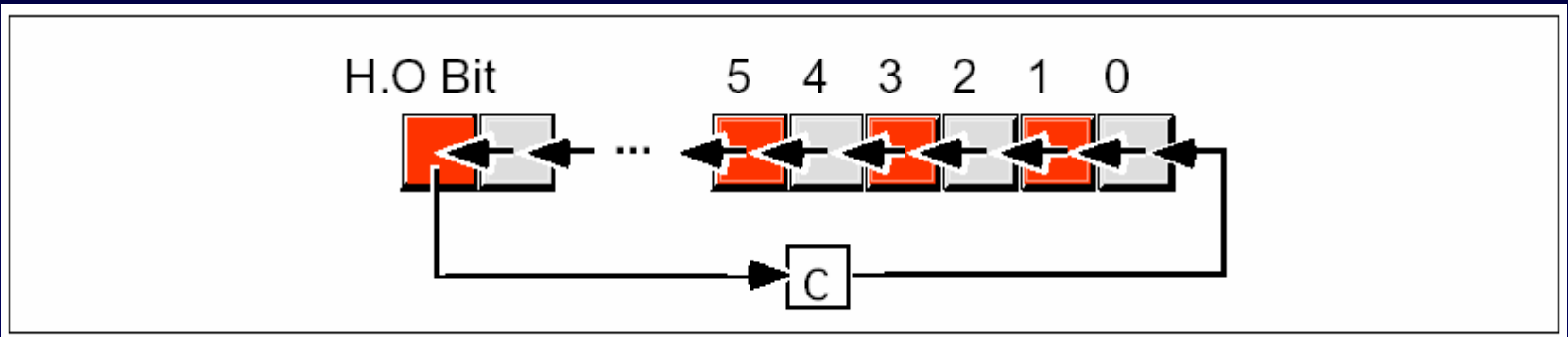


Figure 6.8 Rotate Through Carry Left Operation

Rotate Through Carry Right Operation

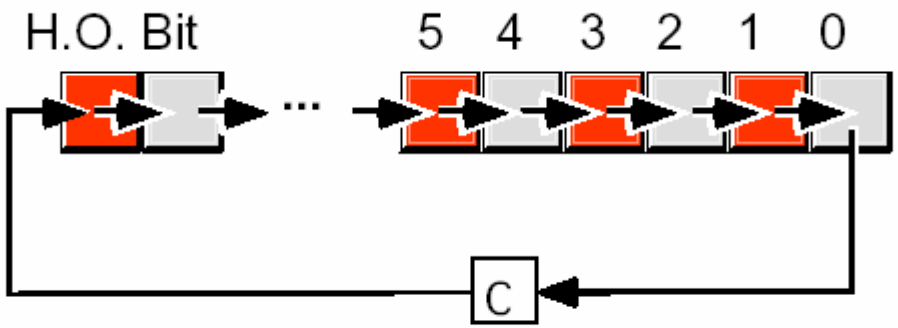


Figure 6.9 Rotate Through Carry Right Operation

Rotate Left Operation

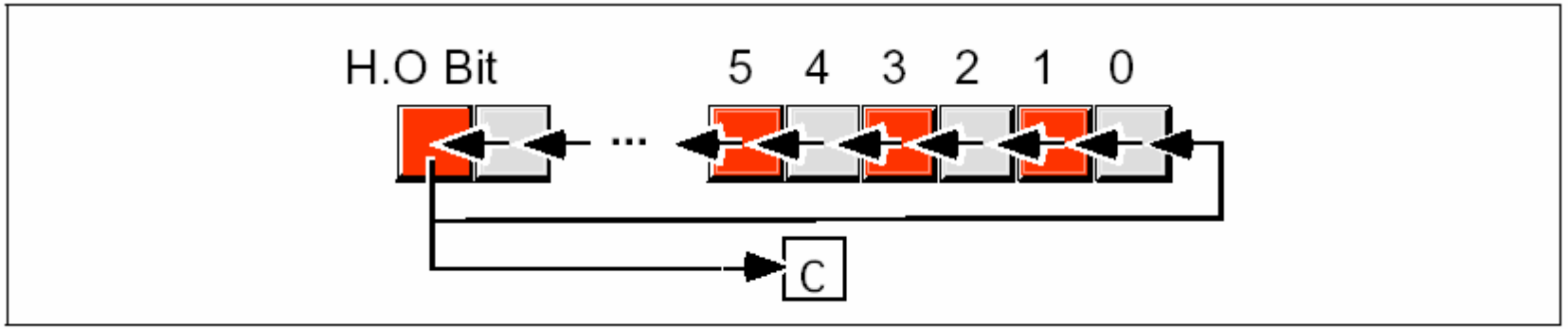


Figure 6.10 Rotate Left Operation

Rotate Right Operation

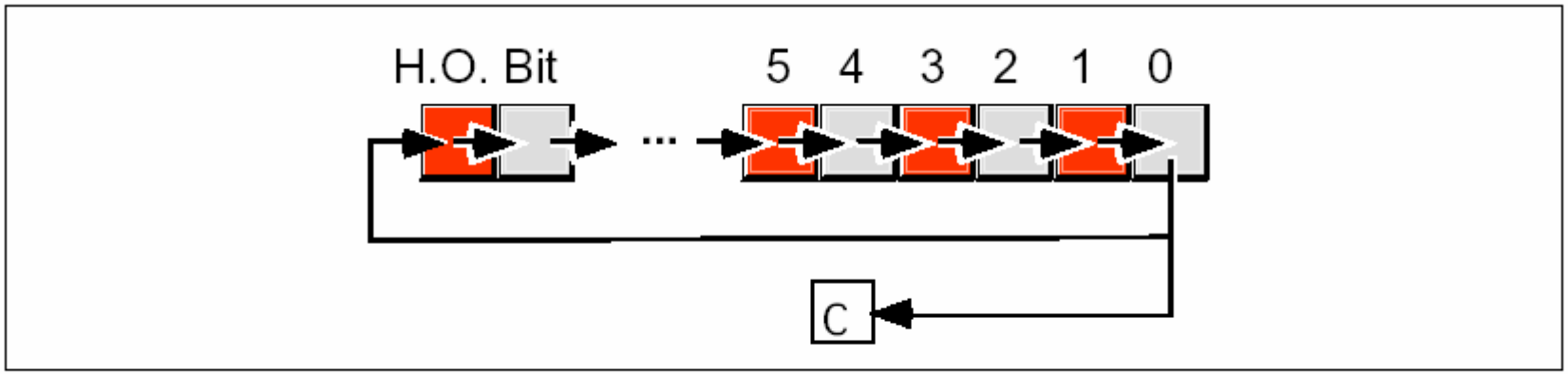


Figure 6.11 Rotate Right Operation

Bit-Test and Bit-Scan Instructions

Destination	Source
Reg16	Reg16
Reg16	Imm8
Reg32	Reg32
Reg32	Imm8
Mem16	Reg16
Mem16	Imm8
Mem32	Reg32
Mem32	Imm8

Destination	Source
Reg16	Reg16
Reg16	Mem16
Reg32	Reg32
Reg32	Mem32

Mnemonic	Meaning	Format	Operation	Flags affected
BT	Bit test	BT D, S	Saves the value of the bit in D specified by the value in S in CF.	CF OF, SF, ZF, AF, PF undefined
BTR	Bit test and reset	BTR D, S	Saves the value of the bit in D specified by the value in S in CF and then resets the bit in D.	CF OF, SF, ZF, AF, PF undefined
BTS	Bit test and set	BTS D, S	Saves the value of the bit in D specified by the value in S in CF and then sets the bit in D.	CF OF, SF, ZF, AF, PF undefined
BTC	Bit test and complement	BTC D, S	Saves the value of the bit in D specified by the value in S in CF and then complements the bit in D.	CF OF, SF, ZF, AF, PF undefined
BSF	Bit scan forward	BSF D, S	Scan the source operand starting from bit 0. ZF = 0 if all bits are 0, else ZF = 1 and the destination operand is loaded with the bit index of the first set bit.	ZF OF, SF, AF, PF, CF undefined
BSR	Bit scan reverse	BSR D, S	Scan the source operand starting from the MSB. ZF = 0 if all bits are 0, else ZF = 1 and the destination operand is loaded with the bit index of the first set bit.	ZF OF, SF, AF, PF, CF undefined

Flag-Control Instructions

Mnemonic	Meaning	Operation	Flags affected
LAHF	Load AH from flags	(AH) ← (Flags)	None
SAHF	Store AH into flags	(Flags) ← (AH)	SF,ZF,AF,PF,CF
CLC	Clear carry flag	(CF) ← 0	CF
STC	Set carry flag	(CF) ← 1	CF
CMC	Complement carry flag	(CF) ← ($\overline{CF}$)	CF
CLI	Clear interrupt flag	(IF) ← 0	IF
STI	Set interrupt flag	(IF) ← 1	IF

(a)

SF = Sign flag
ZF = Zero flag
AF = Auxiliary
PF = Parity flag
CF = Carry flag
– = Undefined (do not use)

(b)

Compare Instructions

Mnemonic	Meaning	Format	Operation	Flags affected
CMP	Compare	CMP D,S	(D) − (S) is used in setting or resetting the flags	CF,AF,OF,PF,SF,ZF

(a)

Destination	Source
Register	Register
Register	Memory
Memory	Register
Register	Immediate
Memory	Immediate
Accumulator	Immediate

(b)

Byte Set on Condition Instruction

Mnemonic	Meaning	Format	Operation	Affected flags
SETcc	Byte set on condition	SETcc D	11111111 → D if cc is true 00000000 → D if cc is false	None

(a)

Instruction	Meaning	Conditions code relationship
SETA *r/m8*	Set byte if above	CF = 0 · ZF = 0
SETAE *r/m8*	Set byte if above or equal	CF = 0
SETB *r/m8*	Set byte if below	CF = 1
SETBE *r/m8*	Set byte if below or equal	CF = 1 + ZF = 1
SETC *r/m8*	Set if carry	CF = 1
SETE *r/m8*	Set byte if equal	ZF = 1
SETG *r/m8*	Set byte if greater	ZF = 0 + SF = OF
SETGE *r/m8*	Set byte if greater	SF = OF
SETL *r/m8*	Set byte if less	SF <> OF
SETLE *r/m8*	Set byte if less or equal	ZF = 1 · SF <> OF
SETNA *r/m8*	Set byte if not above	CF = 1
SETNAE *r/m8*	Set byte if not above	CF = 1
SETNB *r/m8*	Set byte if not below	CF = 0
SETNBE *r/m8*	Set byte if not below	CF = 0 · ZF = 0
SETNC *r/m8*	Set byte if not carry	CF = 0
SETNE *r/m8*	Set byte if not equal	ZF = 0
SETNG *r/m8*	Set byte if not greater	ZF = 1 + SF <> OF
SETNGE *r/m8*	Set if not greater or equal	SF <> OF
SETNL *r/m8*	Set byte if not less	SF = OF
SETNLE *r/m8*	Set byte if not less or equal	ZF = 1 · SF <> OF
SETNO *r/m8*	Set byte if not overflow	OF = 0
SETNP *r/m8*	Set byte if not parity	PF = 0
SETNS *r/m8*	Set byte if not sign	SF = 0
SETNZ *r/m8*	Set byte if not zero	ZF = 0
SETO *r/m8*	Set byte if overflow	OF = 1
SETP *r/m8*	Set byte if parity	PF = 1
SETPE *r/m8*	Set byte if parity even	PF = 1
SETPO *r/m8*	Set byte if parity odd	PF = 0
SETS *r/m8*	Set byte if sign	SF = 1
SETZ *r/m8*	Set byte if zero	ZF = 1

(b)

Source
Reg8 Mem8

(c)

Unconditional Jump Instruction

Mnemonic	Meaning	Format	Operation	Affected flags
JMP	Unconditional jump	JMP Operand	Jump is initiated to the address specified by the operand	None

(a)

Operands
Short-label
Near-label
Far-label
Memptr16
Regptr16
Memptr32
Regptr32

(b)

Coditional Jump Instructions

Mnemonic	Meaning	Format	Operation	Flags Affected
Jcc	Conditional jump	Jcc Operand	If the specific condition cc is true, the jump to the address specified by the Operand is initiated; otherwise, the next instruction is executed	None

Mnemonic	Meaning	Condition
JA	above	CF = 0 and ZF = 0
JAE	above or equal	CF = 0
JB	below	CF = 1
JBE	below or equal	CF = 1 or ZF = 1
JC	carry	CF = 1
JCXZ	CX register is zero	CX = 0000H
JECXZ	ECX register is zero	ECX = 00000000H
JE	equal	ZF = 1
JG	greater	ZF = 0 and SF = OF
JGE	greater or equal	SF = OF
JL	less	(SF xor OF) = 1
JLE	less or equal	((SF xor OF) or ZF) = 1
JNA	not above	CF = 1 or ZF = 1
JNAE	not above nor equal	CF = 1
JNB	not below	CF = 0
JNBE	not below nor equal	CF = 0 and ZF = 0
JNC	not carry	CF = 0
JNE	not equal	ZF = 0
JNG	not greater	((SF xor OF) or ZF) = 1
JNGE	not greater nor equal	(SF xor OF) = 1
JNL	not less	SF = OF
JNLE	not less nor equal	ZF = 0 and SF = OF
JNO	not overflow	OF = 0
JNP	not parity	PF = 0
JNS	not sign	SF = 0
JNZ	not zero	ZF = 0
JO	overflow	OF = 1
JP	parity	PF = 1
JPE	parity even	PF = 1
JPO	parity odd	PF = 0
JS	sign	SF = 1
JZ	zero	ZF = 1

The control transfer instructions

These instructions include the following:

```
ja          dest            -- Jump if above
jae         dest            -- Jump if above or equal
jb          dest            -- Jump if below
jbe         dest            -- Jump if below or equal
je          dest            -- Jump if equal
jne         dest            -- Jump if not equal
jmp         dest            -- Unconditional jump
iret                        -- Return from an interrupt
```

Other instructions:

The **get** and **put** instructions let you read and write integer values.
The remaining instructions do not require any operands, they are **halt** and **brk**.

Jcc Instructions That Test Flags

Table 32: Jcc Instructions That Test Flags

Instruction	Description	Condition	Aliases	Opposite
JC	Jump if carry	Carry = 1	JB, JNAE	JNC
JNC	Jump if no carry	Carry = 0	JNB, JAE	JC
JZ	Jump if zero	Zero = 1	JE	JNZ
JNZ	Jump if not zero	Zero = 0	JNE	JZ
JS	Jump if sign	Sign = 1		JNS
JNS	Jump if no sign	Sign = 0		JS
JO	Jump if overflow	Ovrflw=1		JNO
JNO	Jump if no Ovrflw	Ovrflw=0		JO
JP	Jump if parity	Parity = 1	JPE	JNP
JPE	Jump if parity even	Parity = 1	JP	JPO
JNP	Jump if no parity	Parity = 0	JPO	JP
JPO	Jump if parity odd	Parity = 0	JNP	JPE

Jcc Instructions for Unsigned Comparisons

Table 33: Jcc Instructions for Unsigned Comparisons

Instruction	Description	Condition	Aliases	Opposite
JA	Jump if above (>)	Carry=0, Zero=0	JNBE	JNA
JNBE	Jump if not below or equal (not <=)	Carry=0, Zero=0	JA	JBE
JAE	Jump if above or equal (>=)	Carry = 0	JNC, JNB	JNAE
JNB	Jump if not below (not <)	Carry = 0	JNC, JAE	JB
JB	Jump if below (<)	Carry = 1	JC, JNAE	JNB
JNAE	Jump if not above or equal (not >=)	Carry = 1	JC, JB	JAE
JBE	Jump if below or equal (<=)	Carry = 1 or Zero = 1	JNA	JNBE
JNA	Jump if not above (not >)	Carry = 1 or Zero = 1	JBE	JA
JE	Jump if equal (=)	Zero = 1	JZ	JNE
JNE	Jump if not equal (≠)	Zero = 0	JNZ	JE

Jcc Instructions for Signed Comparisons

Table 34: Jcc Instructions for Signed Comparisons

Instruction	Description	Condition	Aliases	Opposite
JG	Jump if greater (>)	Sign = Ovrflw or Zero=0	JNLE	JNG
JNLE	Jump if not less than or equal (not <=)	Sign = Ovrflw or Zero=0	JG	JLE
JGE	Jump if greater than or equal (>=)	Sign = Ovrflw	JNL	JGE
JNL	Jump if not less than (not <)	Sign = Ovrflw	JGE	JL
JL	Jump if less than (<)	Sign ≠ Ovrflw	JNGE	JNL
JNGE	Jump if not greater or equal (not >=)	Sign ≠ Ovrflw	JL	JGE
JLE	Jump if less than or equal (<=)	Sign ≠ Ovrflw or Zero = 1	JNG	JNLE
JNG	Jump if not greater than (not >)	Sign ≠ Ovrflw or Zero = 1	JLE	JG
JE	Jump if equal (=)	Zero = 1	JZ	JNE
JNE	Jump if not equal (≠)	Zero = 0	JNZ	JE

IF..THEN and IF..THEN..ELSE Statement Flow

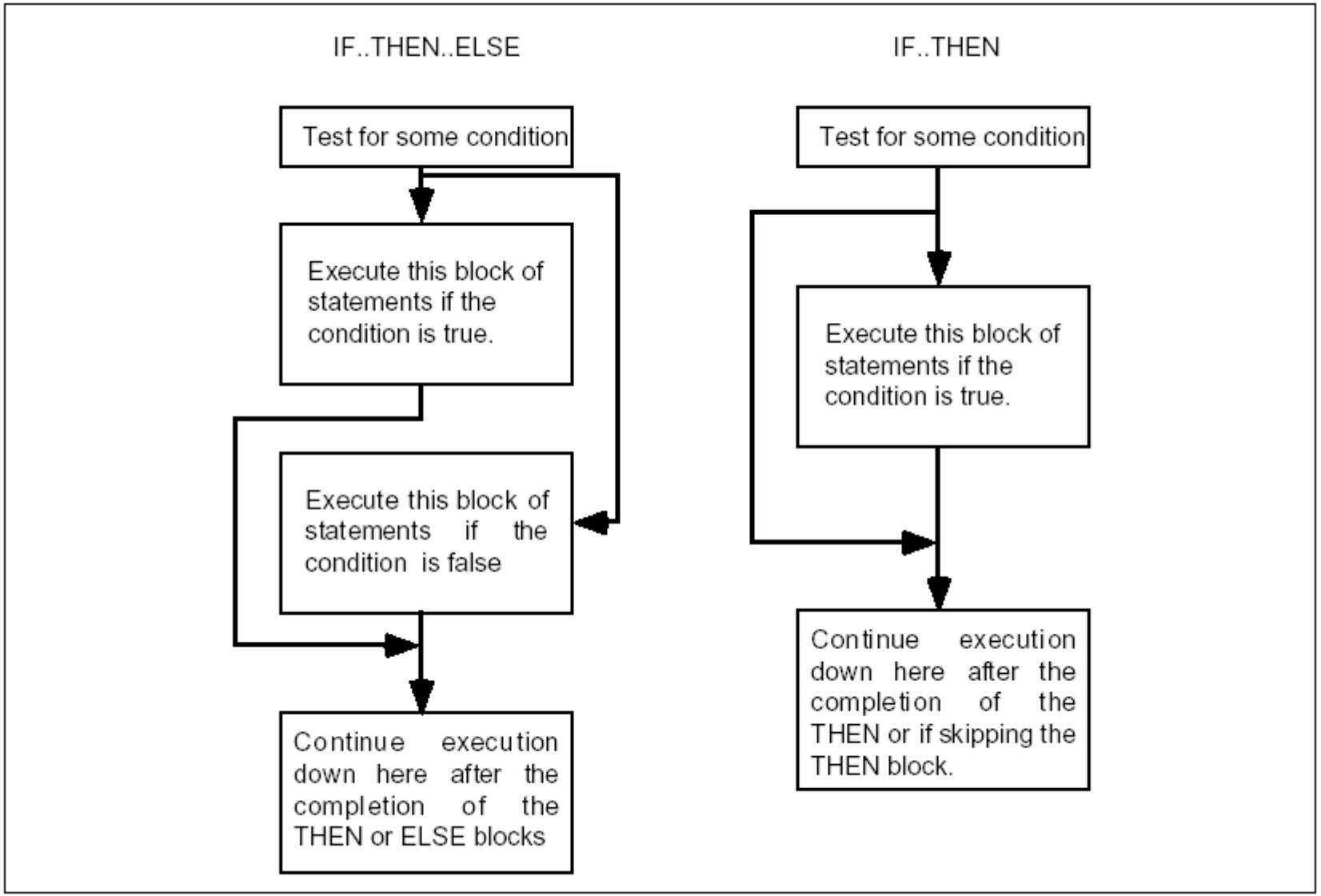
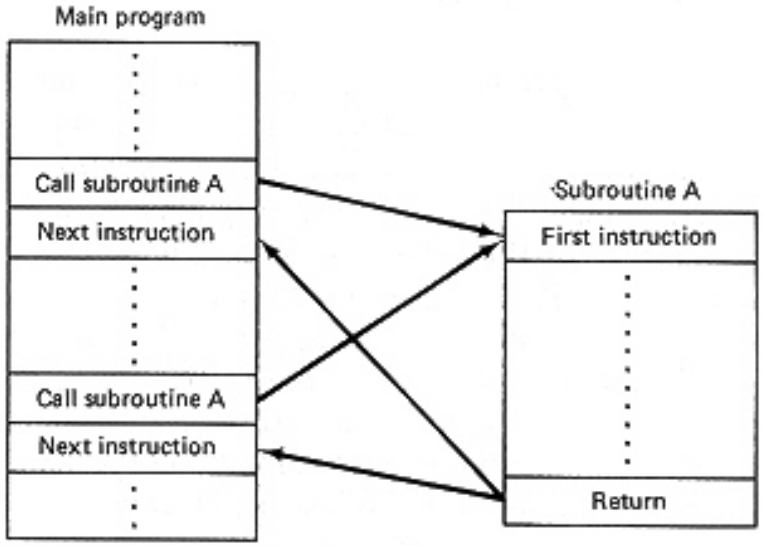


Figure 10.1 IF..THEN and IF..THEN..ELSE Statement Flow

Subroutine Call Instruction



(a)

Mnemonic	Meaning	Format	Operation	Affected flags
CALL	Subroutine call	CALL Operand	Execution continues from the address of the subroutine specified by the operand. Information required to return back to the main program such as IP and CS are saved on the stack.	None

(b)

Operands
Near-proc
Far-proc
Memptr16
Regptr16
Memptr32
Regptr32

(c)

Return Instruction

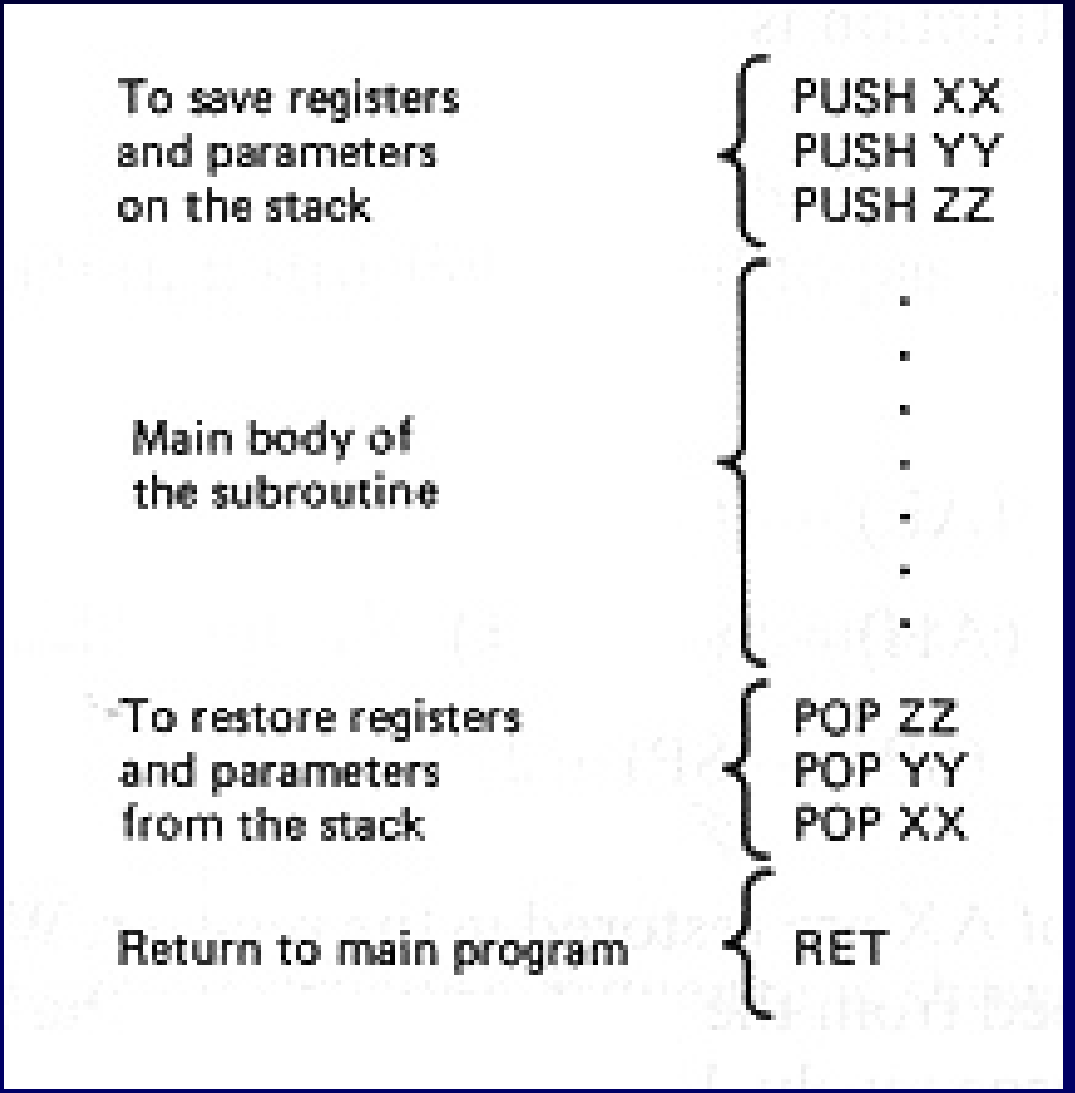
Mnemonic	Meaning	Format	Operation	Affected flags
RET	Return	RET or RET Operand	Return to the main program by restoring IP (and CS for far-procedure). If operand is present it is added to the content of SP.	None

(a)

Operands
None Disp16

(b)

Structure of a Subroutine



Push and Pop Instructions

Mnemonic	Meaning	Format	Operation	Flags Affected
PUSH	Push word onto stack	PUSH S	((SP)) ← (S) (SP) ← (SP) – 2	None
POP	Pop word off stack	POP D	(D) ← ((SP)) (SP) ← (SP) + 2	None

(a)

Operands (S or D)
Immediate (PUSH only)
Register
Seg-Reg
Memory

(b)

Push Flags and Pop Flags Instructions

Mnemonic	Meaning	Operation	Flags Affected
PUSHF	Push flags onto stack	((SP)) ← (Flags) (SP) ← (SP) – 2	None
POPF	Pop flags from stack	(Flags) ← ((SP)) (SP) ← (SP) + 2	OF, DF, IF, TF, SF, ZF, AF, PF, CF

Push all and Pop all Instructions

Mnemonic	Meaning	Operation	Flags Affected
PUSHA	Push all 16-bit general registers onto stack	((SP)) ← (AX) ((SP–2)) ← (CX) ((SP–4)) ← (DX) ((SP–6)) ← (BX) ((SP–8)) ← (OLD SP) ((SP–A)) ← (BP) ((SP–C)) ← (SI) ((SP–E)) ← (DI)	None
PUSHAD	Push all 32-bit general registers onto stack	((SP)) ← (EAX) ((SP–4)) ← (ECX) ((SP–8)) ← (EDX) ((SP–C)) ← (EBX) ((SP–10)) ← (OLD ESP) ((SP–14)) ← (EBP) ((SP–18)) ← (ESI) ((SP–1C)) ← (EDI)	None
POPA	Pop all 16-bit general registers from stack	(DI) ← ((SP + 2)) (SI) ← ((SP + 4)) (BP) ← ((SP + 6)) (OLD SP) ← ((SP + 8)) (BX) ← ((SP + A)) (DX) ← ((SP + C)) (CX) ← ((SP + E)) (AX) ← ((SP + 10))	None
POPAD	Pop all 32-bit general registers from stack	(EDI) ← ((SP + 4)) (ESI) ← ((SP + 8)) (EBP) ← ((SP + C)) (OLD ESP) ← ((SP + 10)) (EBX) ← ((SP + 14)) (EDX) ← ((SP + 18)) (ECX) ← ((SP + 1C)) (EAX) ← ((SP + 20))	

Enter and Leave Instructions

Mnemonic	Meaning	Format	Operation

ENTER	Make stack frame	ENTER Imm16,Imm8	Push BP
			TEMP ← (SP)
			If Imm8 > 0 then
			Repeat (Imm8 − 1) times
			(BP) ← (BP) − 2
			Push ((BP))
			End repeat
			PUSH TEMP
			End if
			(BP) ← TEMP
			(SP) ← (SP) − Imm16

LEAVE	Release stack frame	LEAVE	(SP) ← (BP) Pop to BP

Loop Instructions

Mnemonic	Meaning	Format	Operation
LOOP	Loop	LOOP Short-label	(CX) ← (CX) − 1 Jump is initiated to location defined by Short-label if (CX) ≠ 0; otherwise, execute next sequential instruction.
LOOPE/LOOPZ	Loop while equal/ loop while zero	LOOPE/LOOPZ Short-label	(CX) ← (CX) − 1 Jump to location defined by Short-label if (CX) ≠ 0 and (ZF) = 1; otherwise, execute next sequential instruction.
LOOPNE/LOOPNZ	Loop while not equal/ loop while not zero	LOOPNE/LOOPNZ Short-label	(CX) ← (CX) − 1 Jump to location defined by Short-label if (CX) ≠ 0 and (ZF) = 0; otherwise, execute next sequential instruction.

Basic String Instructions

Mnemonic	Meaning	Format	Operation	Flags affected
MOVSB/ MOVSW/ MOVSD	Move string	MOVSB MOVSW MOVSD	((ES)0 + (DI) ← ((DS)0 + (SI)) (SI) ← (SI) ± 1, 2, or 4 (DI) ← (DI) ± 1, 2, or 4	None
CMPSB/ CMPSW/ CMPSD	Compare strings	CMPSB CMPSW CMPSD	Set flags as per ((DS)0 + (SI)) – ((ES)0 + (DI)) (SI) ← (SI) ± 1, 2, or 4 (DI) ← (DI) ± 1, 2, or 4	CF, PF, AF, ZF, SF, OF
SCASB/ SCASW/ SCASD	Scan string	SCASB SCASW SCASD	Set flags as per (AL, AX, or EAX) – ((ES)0 + (DI)) (DI) ← (DI) ± 1, 2, or 4	CF, PF, AF, ZF, SF, OF
LODSB/ LODSW/ LODSD	Load string	LODSB LODSW LODSD	(AL, AX, or EAX) ← ((DS)0 + (SI) (SI) ← (SI) ± 1, 2, or 4	None
STOSB/ STOSW/ STOSD	Store string	STOSB STOSW STOSD	((ES)0 + (DI) ← (AL, AX, or EAX) ± 1, 2, or 4 (DI) ← (DI) ± 1, 2, or 4	None

Prefixes for Use with Basic String Instructions

Prefix	Used with	Meaning
REP	MOVS STOS	Repeat while not end of string CX ≠ 0
REPE/REPZ	CMPS SCAS	Repeat while not end of string and strings are equal CX ≠ 0 and ZF = 1
REPNE/REPNZ	CMPS SCAS	Repeat while not end of string and strings are not equal CX ≠ 0 and ZF = 0

Instructions for Selecting Autoincrementing and Autodecrementing in String Instructions

Mnemonic	Meaning	Format	Operation	Affected flags
CLD	Clear DF	CLD	(DF) ← 0	DF
STD	Set DF	STD	(DF) ← 1	DF

Symbols for Specifying Register Operands

Symbol	Register
AX	Accumulator register
AH	Accumulator register high byte
AL	Accumulator register lower byte
BX	Base register
BH	Base register high byte
BL	Base register low byte
CX	Count register
CH	Count register high byte
CL	Count register low byte
DX	Data register
DH	Data register high byte
DL	Data register low byte
SI	Source index register
DI	Destination index register
SP	Stack pointer register
BP	Base pointer register
CS	Code segment register
DS	Data segment register
SS	Stack segment register
ES	Extra segment register
FS	F segment register
GS	G segment register

Examples of Using Various Addressing Modes

Addressing mode	Operand	Example	Segment
Register	Destination	MOV AX, LABEL	–
Immediate	Source	MOV AL, 15H	–
Direct	Destination	MOV LABEL, AX	Data
Register indirect	Source	MOV AX, [SI]	Data
		MOV AX, [BP]	Stack
		MOV AX, [DI]	Data
		MOV AX, [BX]	Data
Based	Destination	MOV [BX] +DISP, AL	Data
		MOV [BP] +DISP, AL	Stack
Indexed	Source	MOV AL, [SI]	Data
		MOV AL, [DI]	Data
Based indexed	Destination	MOV [BX] [SI] +DISP, AH	Data
		MOV [BX] [DI] +DISP, AH	Data
		MOV [BP] [SI] +DISP, AH	Stack
		MOV [BP] [DI] +DISP, AH	Stack

Arithmetic, Relational, and Logical Operators

Type	Operator	Example	Function
Arithmetic	*	A * B	Multiplies A with B and makes the operand equal to the product
	/	A / B	Divides A by B and makes the operand equal to the quotient
	MOD	A MOD B	Divides A by B and assigns the remainder to the operand
	SHL	A SHL n	Shifts the value in A left by n bit positions and assigns this shifted value to the operand
	SHR	A SHR n	Shifts the value in A right by n bit positions and assigns this shifted value to the operand
	+	A + B	Adds A to B and makes the operand equal to the sum
	–	A – B	Subtracts B from A and makes the operand equal to the difference
Relational	EQ	A EQ B	Compares value of A to that of B. If A equals B, the operand is set to FFFFH and if they are not equal it is set to 0H
	NE	A NE B	Compares value of A to that of B. If A is not equal to B, the operand is set to FFFFH and if they are equal it is set to 0H
	LT	A LT B	Compares value of A to that of B. If A is less than B, the operand is set to FFFFH and if it is equal or greater than it is set to 0H
	GT	A GT B	Compares value of A to that of B. If A is greater than B, the operand is set to FFFFH and if it is equal or less than it is set to 0H
	LE	A LE B	Compares value of A to that of B. If A is less than or equal to B, the operand is set to FFFFH and if it is greater than it is set to 0H
	GE	A GE B	Compares value of A to that of B. If A is greater than or equal to B, the operand is set to FFFFH and if it is less than it is set to 0H
Logical	NOT	NOT A	Takes the logical NOT of A and makes the value that results equal to the operand
	AND	A AND B	A is ANDed with B and makes the value that results equal to the operand
	OR	A OR B	A is ORed with B and makes the value that results equal to the operand
	XOR	A XOR B	A is XORed with B and makes the value that results equal to the operand

Value-Returning and Attribute Operators

Type	Operator	Example	Function
Value-returning	SEG	SEG A	Assigns the contents held in the segment register corresponding to the segment in which A resides to the operand
	OFFSET	OFFSET A	Assigns the offset of the location A in its corresponding segment to the operand
	TYPE	TYPE A	Returns to the operand a number representing the type of A; 1 for a byte variable and 2 for a word variable; NEAR or FAR for the label
	SIZE	SIZE A	Returns the byte count of variable A to the operand
	LENGTH	LENGTH A	Returns the number of units (as specified by TYPE) allocated for the variable A to the operand
Attribute	PTR	NEAR PTR A	Overrides the current type of label operand A and assigns a new pointer type: BYTE, WORD, NEAR, or FAR to A
	DS:,ES:,SS:	ES:A	Overrides the normal segment for operand A and assigns a new segment to A
	SHORT	JMP SHORT A	Assigns to operand A an attribute that indicates that it is within +127 or −128 bytes of the next instruction. This lets the instruction be encoded with the minimum number of bytes
	THIS	THIS BYTE A	Assigns to operand A a distance or type attribute: BYTE, WORD, NEAR, or FAR, and the corresponding segment attribute
	HIGH	HIGH A	Returns to the operand A the high byte of the word of A
	LOW	LOW A	Returns to the operand A the low byte of the word of A

Pseudo-ops of the Macroassembler

Type	Pseudo-ops		
Data	ASSUME	ENDS	NAME
	COMMENT	EQU	ORG
	DB	= (Equal	PROC
	DD	Sign)	PUBLIC
	DQ	EVEN	.RADIX
	DT	EXTRN	RECORD
	DW	GROUP	SEGMENT
	END	INCLUDE	STRUC
	ENDP	LABEL	
Conditional	ELSE	IFDEF	IFNB
	ENDIF	IFDIF	IFNDEF
	IF	IFE	IF1
	IFB	IFIDN	IF2
Macro	ENDM	IRPC	PURGE
	EXITM	LOCAL	REPT
	IRP	MACRO	
Listing	.CREF	PAGE	TITLE
	.LALL	.SALL	.XALL
	.LFCOND	.SFCOND	.XCREF
	.LIST	SUBTTL	.XLIST
	%OUT	.TFCOND	

Data Pseudo-ops

Pseudo-op	Meaning	Function
EQU	Equate	Assign a permanent value to a symbol
=	Equal to	Set or redefine the value of a symbol
DB	Define byte	Define or initialize byte size variables or locations
DW	Define word	Define or initialize word size (2 byte) variables or locations
DD	Define double word	Define or initialize double word size (4 byte) variables or locations

Segment Pseudo-ops

Pseudo-op	Function
SEGMENT	Defines the beginning of a segment and specifies its kind, at what type of address boundary it is to be stored in memory, and how it is to be positioned with respect to other similar segments in memory
ENDS	Specifies the end of a segment
ASSUME	Specifies the segment address register for a given segment

Align Type Attributes

PARA – paragraph alignment

BYTE – byte alignment

WORD – word alignment

PAGE – page alignment

INPAGE – inpage alignment

Combine Type Attributes

Attribute	Function
PUBLIC	Concatenates segments with the same name
COMMON	Overlaps from the beginning segments with the same name
AT [expression]	Locates the segment at the 16-bit paragraph number evaluated from the expression
STACK	The segment is part of the run-time stack segment
MEMORY	Locates the segment at an address above all other segments

Class Attributes

Attribute	Function
CODE	Specifies the code segment
DATA	Specifies the data segment
STACK	Specifies the stack segment
EXTRA	Specifies the extra segment

Modular Programming Pseudo-ops

Pseudo-op			Function
proc-name	PROC	[NEAR]	Defines the beginning of a near-proc procedure
proc-name	PROC	FAR	Defines the beginning of a far-proc procedure
proc-name	ENDP		Defines the end of a procedure
	PUBLIC	Symbol[. . . . . .]	The defined symbols can be referenced from other modules
	EXTRN	name:type[. . . .]	The specified symbols are defined in other modules and are to be used in this module

Org and End Pseudo-op

Pseudo-op	Function
ORG [expression]	Specifies the memory address starting from which the machine code must be placed
END [expression]	Specifies the end of the source program

Listing Control Pseudo-op

Pseudo-op		Function
PAGE	operand_1 operand_2	Selects the number of lines printed per page and the maximum number of characters printed per line in the listing
TITLE	text	Prints 'text' on the second line of each page of the listing
SUBTTL	text	Prints 'text' on the third line of each page of the listing

Definicija segmenata – izbor memorijskog modela

Uglavnom se koriste sledeći memorijski modeli:

SMALL model - jedan od najčešće korišćenih modela, koji koristi 64kB memorija za program i 64kB za podatke
MEDIUIM model - podaci moraju se smestiti u 64kB, a program može biti veći od 64kB.
COMPACT model - suprotan je modelu MEDIUM, tj. program se mora smestiti u 64kB, a podaci u prostor veći od 64kB.
LARGE model - kombinovanjem prethodna dva modela dobija se LARGE model, tj. i kôd i podatke mogu biti veći od 64kB. Jedinstvena struktura podataka kao što je polje (vektor, matrica,...) ne sme da premaši 64kB.
HUGE model - program i podaci mogu da premaše 64kB a takodje i jedinstvena struktura podataka može biti veća od 64kB.
TINY model - se koristi sa *COM* fajlovima kod kojih ukupna memorija program plus podaci ne sme da premaši 64kB. *TINY* model se ne može koristiti u sklopu pojednostavljene definicije segmenata.

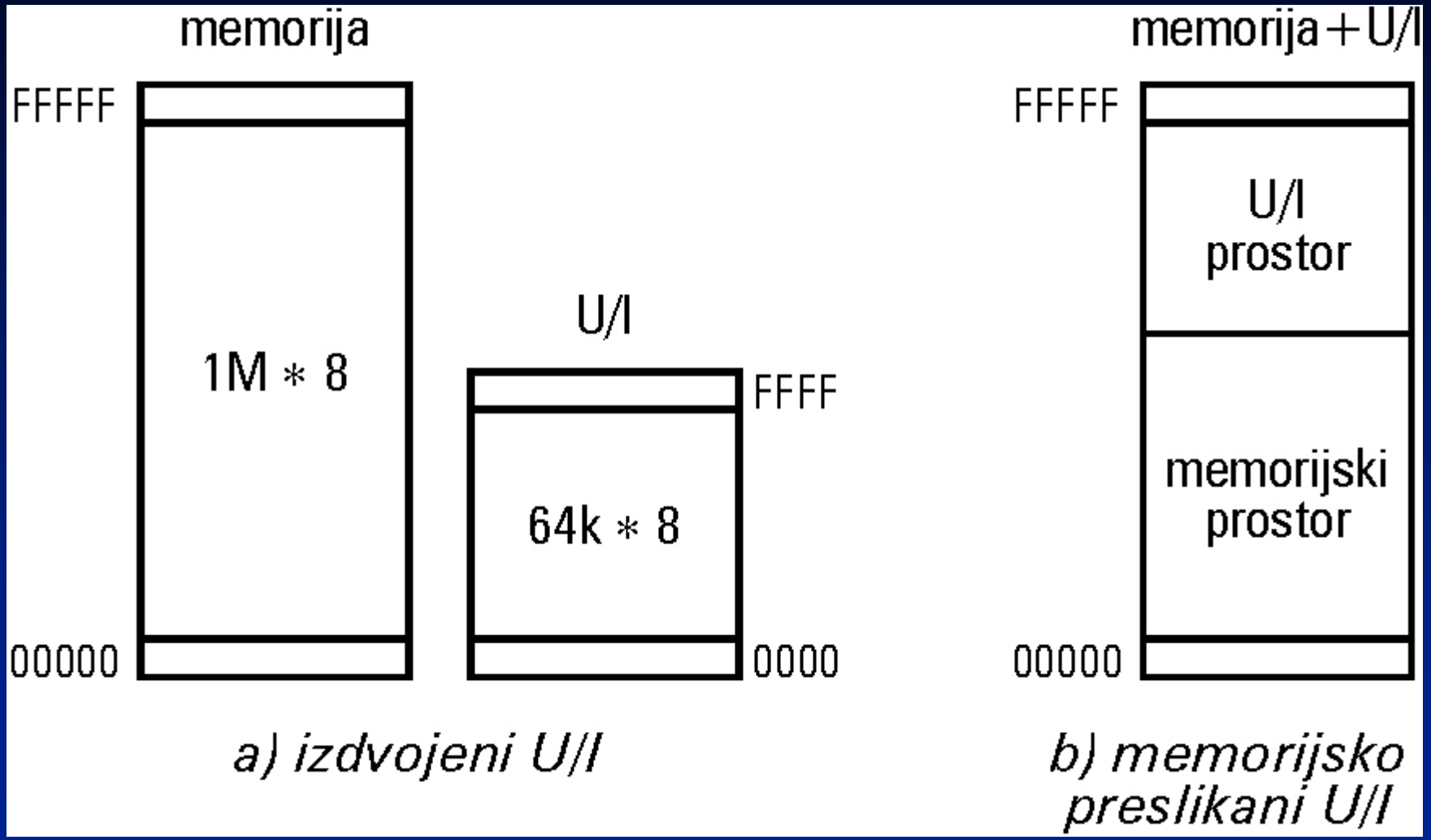
Potpuna u odnosu na pojednostavljenu definiciju segmenata

```
; potpuna definicija segmenata
; --------- segment magacina ----
SMAG        SEGMENT
            DB    64 DUP (?)
SMAG        ENDS
; -------- segment podataka -----
SPOD        SEGMENT
POD_1       DW    1234h
POD_2       DW    5678h
REZ         DW    ?
SPOD        ENDS
; -------- programski segment ____
SPROG       SEGMENT
SABERI      PROC    FAR
            ASSUME  CS:SPROG
            MOV     AX,SPOD
            MOV     DS,AX
            ...
SABERI      ENDP
SPROG       ENDS
            END         SABERI
```

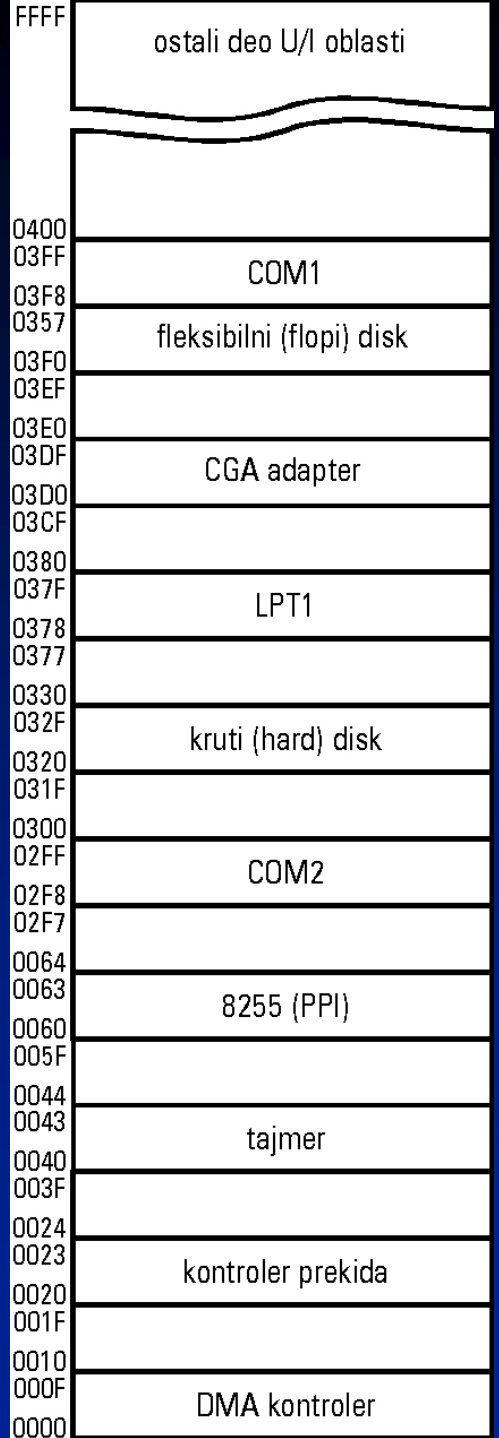
```
; pojednostavljena definicija segmenata
            .MODEL      SMALL
            .STACK      64
;
;
; ---------------------------------------------
            .DATA
POD_1       DW          1234h
POD_2       DW          5678h
REZ         DW          ?
;
; ---------------------------------------------
            .CODE
SABERI:     MOV     AX,@DATA
            MOV     DS,AX
            ...
            ...
            ...
            ...
            ...
            END         SABERI
```

SPREGA SA U/I UREDJAJIMA

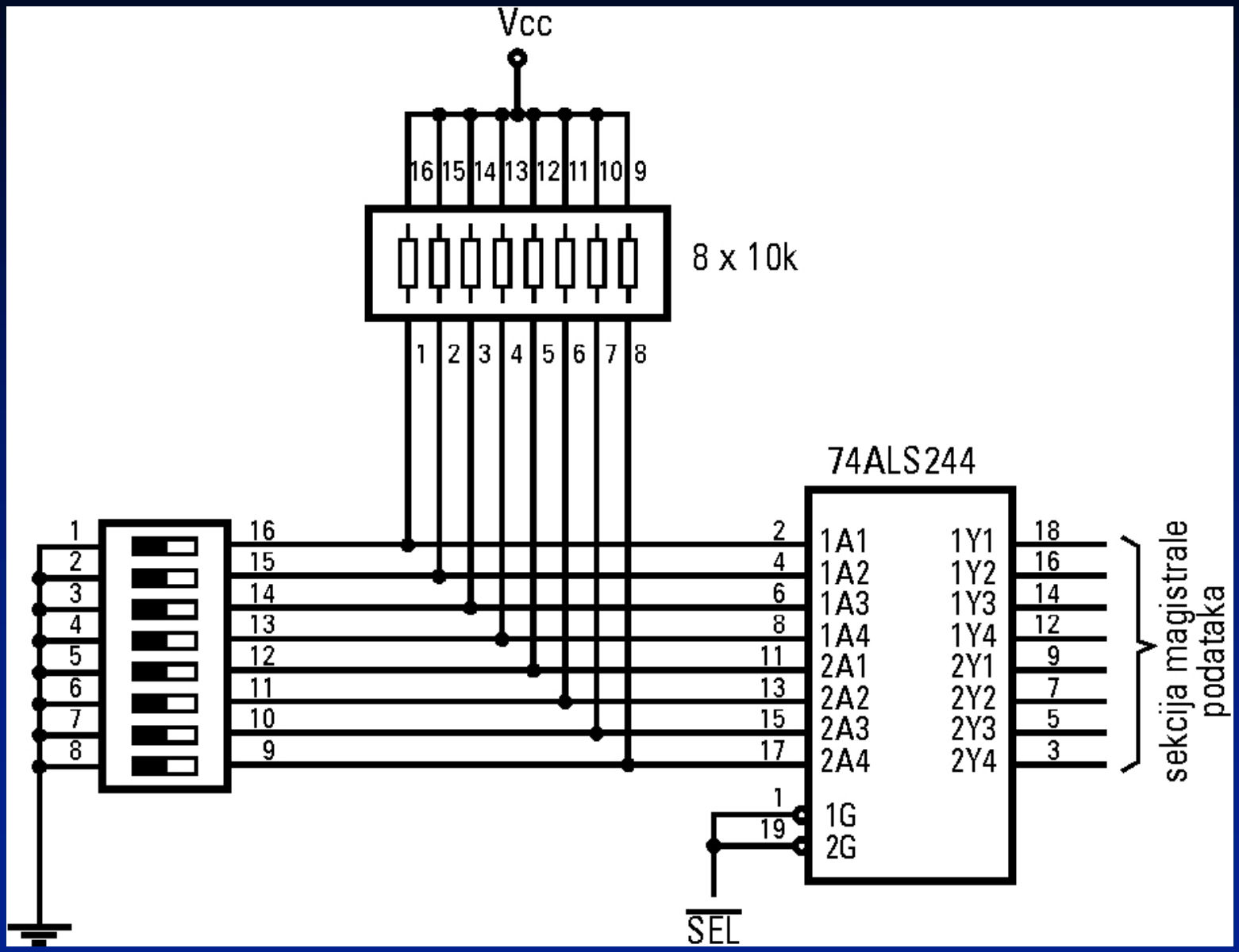
Memorijska i U/I mapa kod 8086



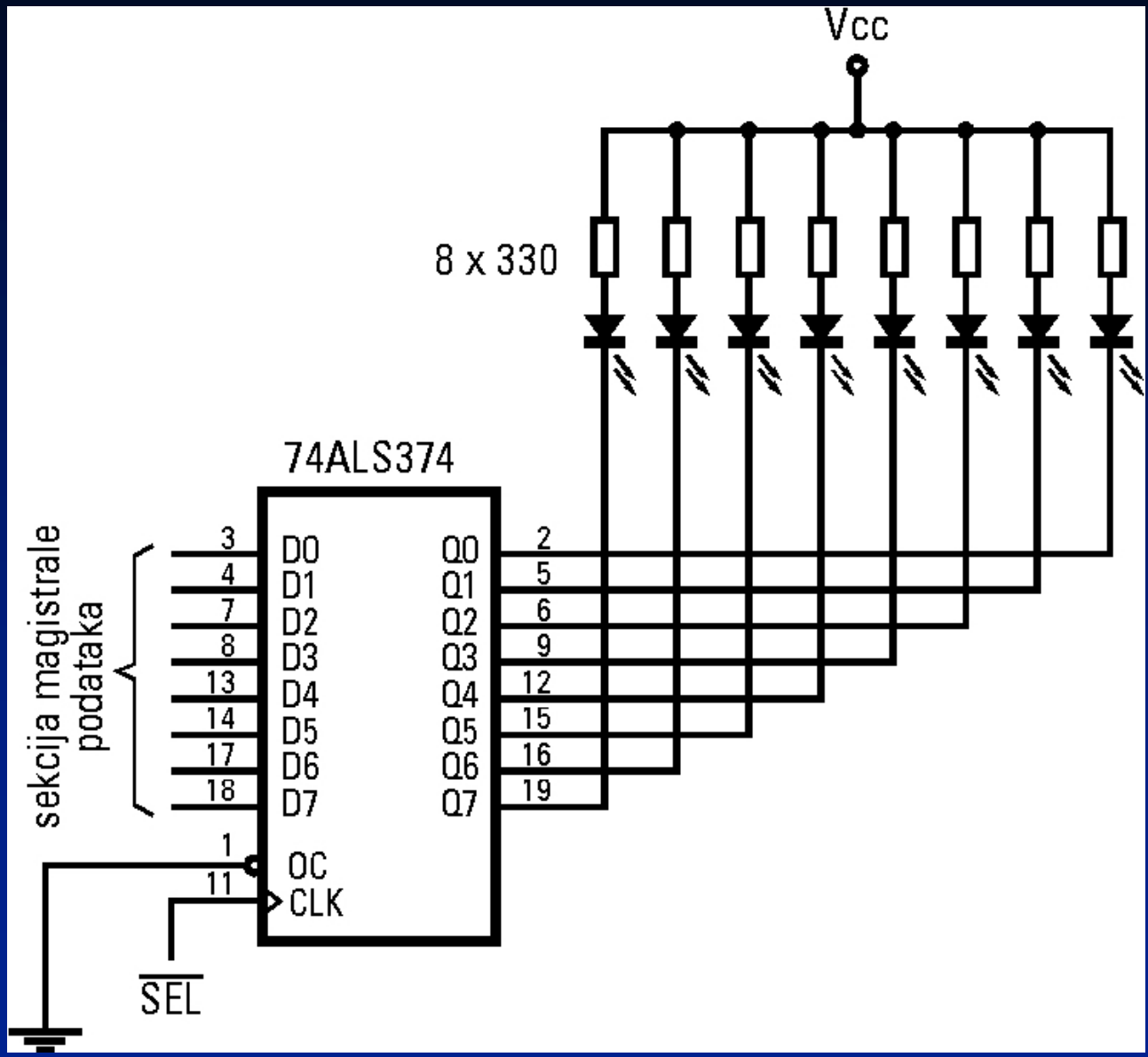
U/I mapa PC-a


FFFF
ostali deo U/I oblasti
0400
03FF
COM1
03F8
0357
fleksibilni (flopi) disk
03F0
03EF
03E0
03DF
CGA adapter
03D0
03CF
0380
037F
LPT1
0378
0377
0330
032F
kruti (hard) disk
0320
031F
0300
02FF
COM2
02F8
02F7
0064
0063
8255 (PPI)
0060
005F
0044
0043
tajmer
0040
003F
0024
0023
kontroler prekida
0020
001F
0010
000F
DMA kontroler
0000

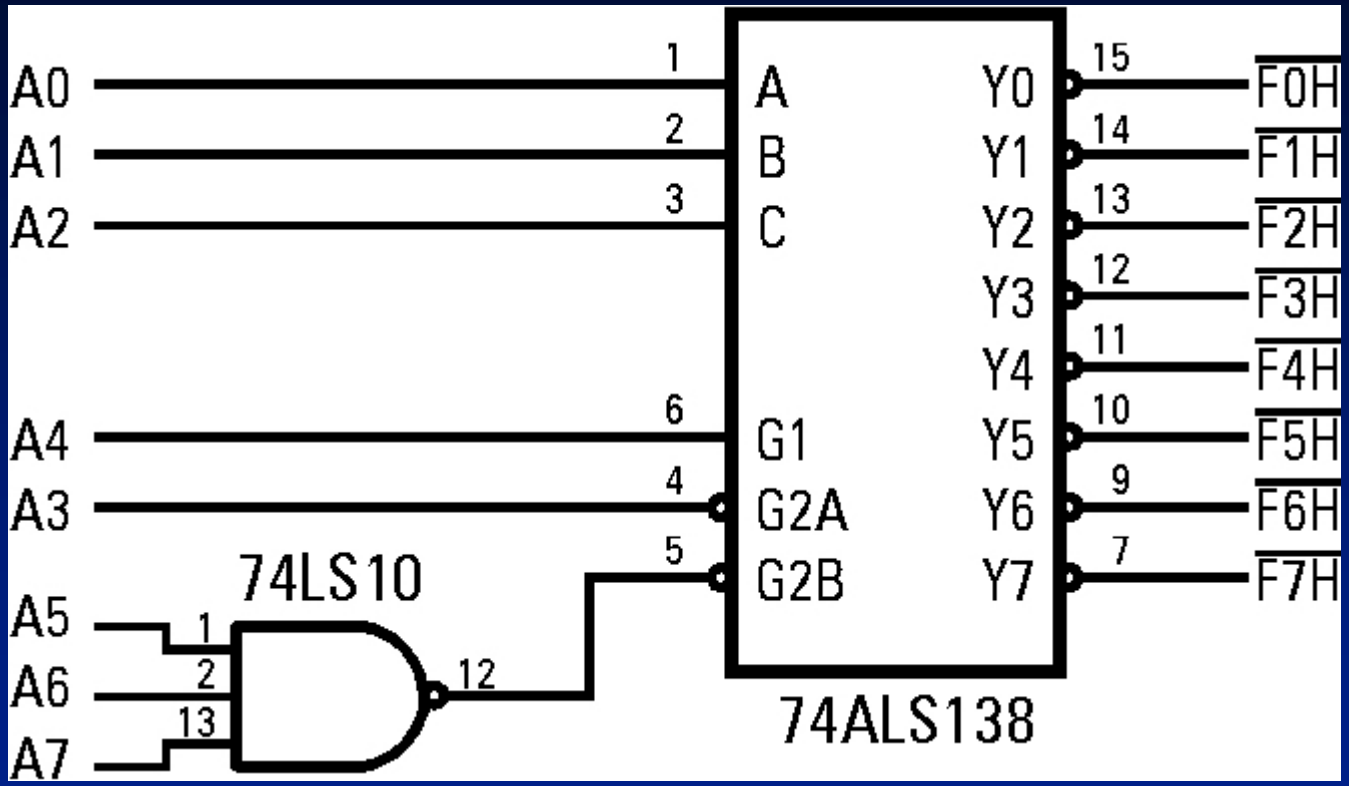
Jednostavni ulazni interfejs



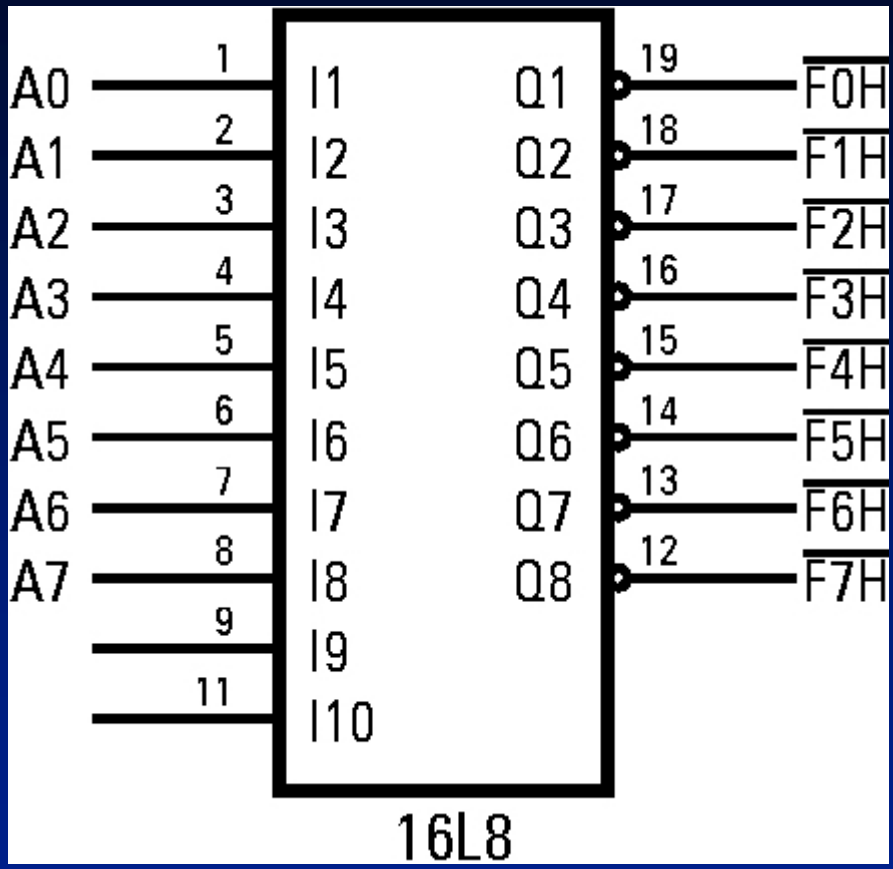
Jednostavni izlazni interfejs



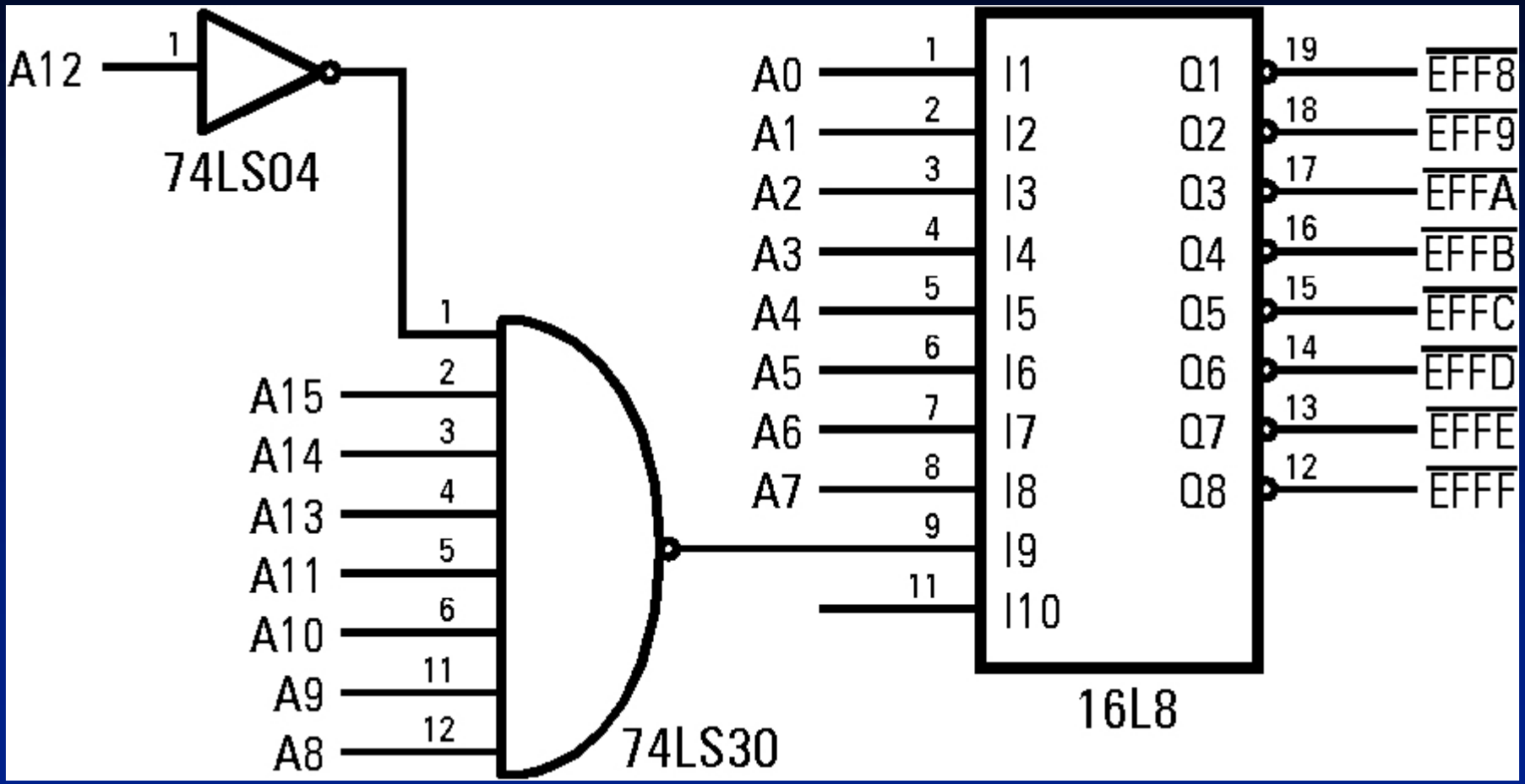
Dekoder 8 U/I portova čije se adrese nalaze u opsegu F0h do F7h



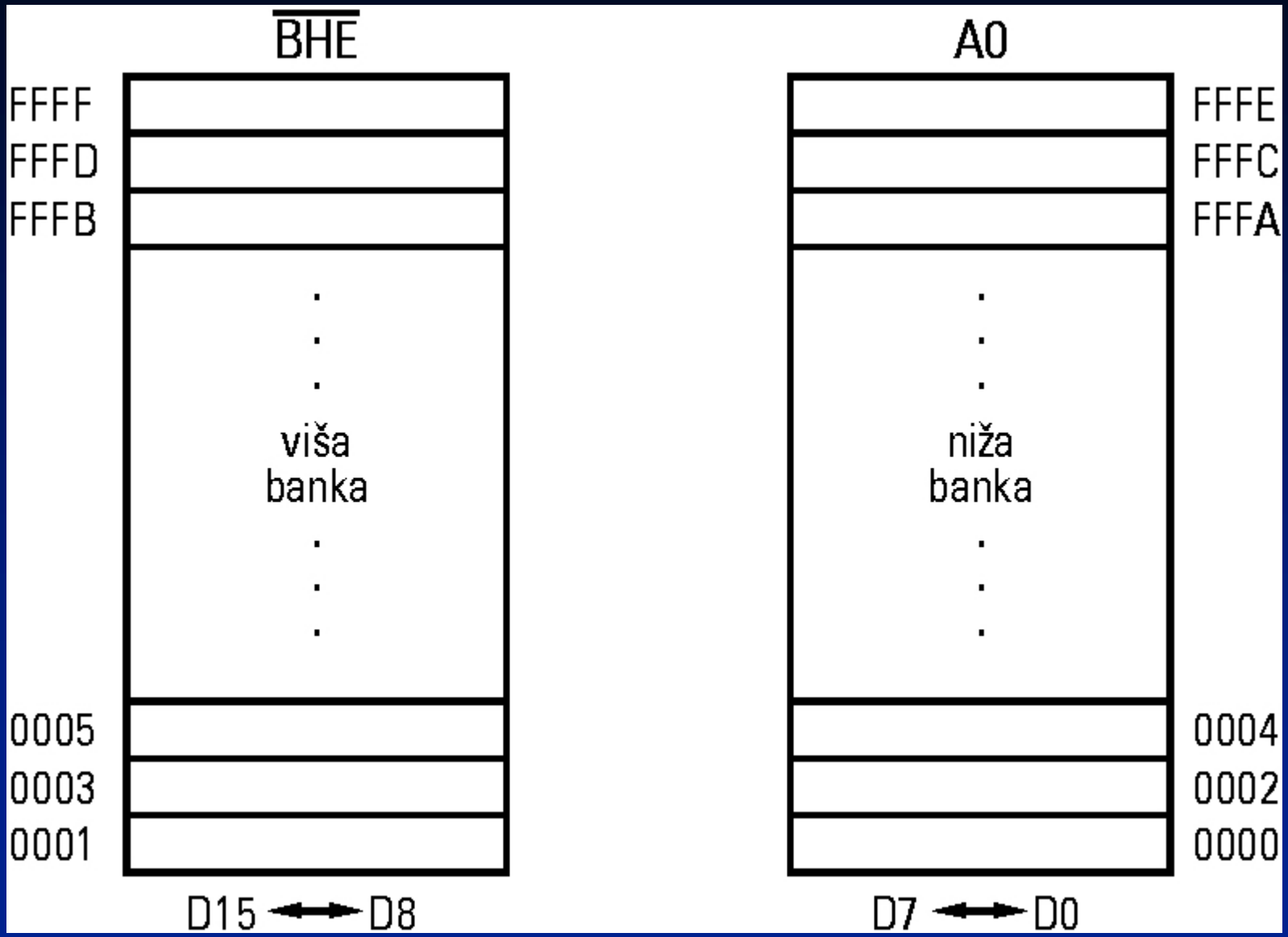
Dekoder


A0
A1
A2
A3
A4
A5
A6
A7
1
2
3
4
5
6
7
8
9
11
I1
I2
I3
I4
I5
I6
I7
I8
I9
I10
Q1
Q2
Q3
Q4
Q5
Q6
Q7
Q8
19
18
17
16
15
14
13
12
F0H
F1H
F2H
F3H
F4H
F5H
F6H
F7H
16L8

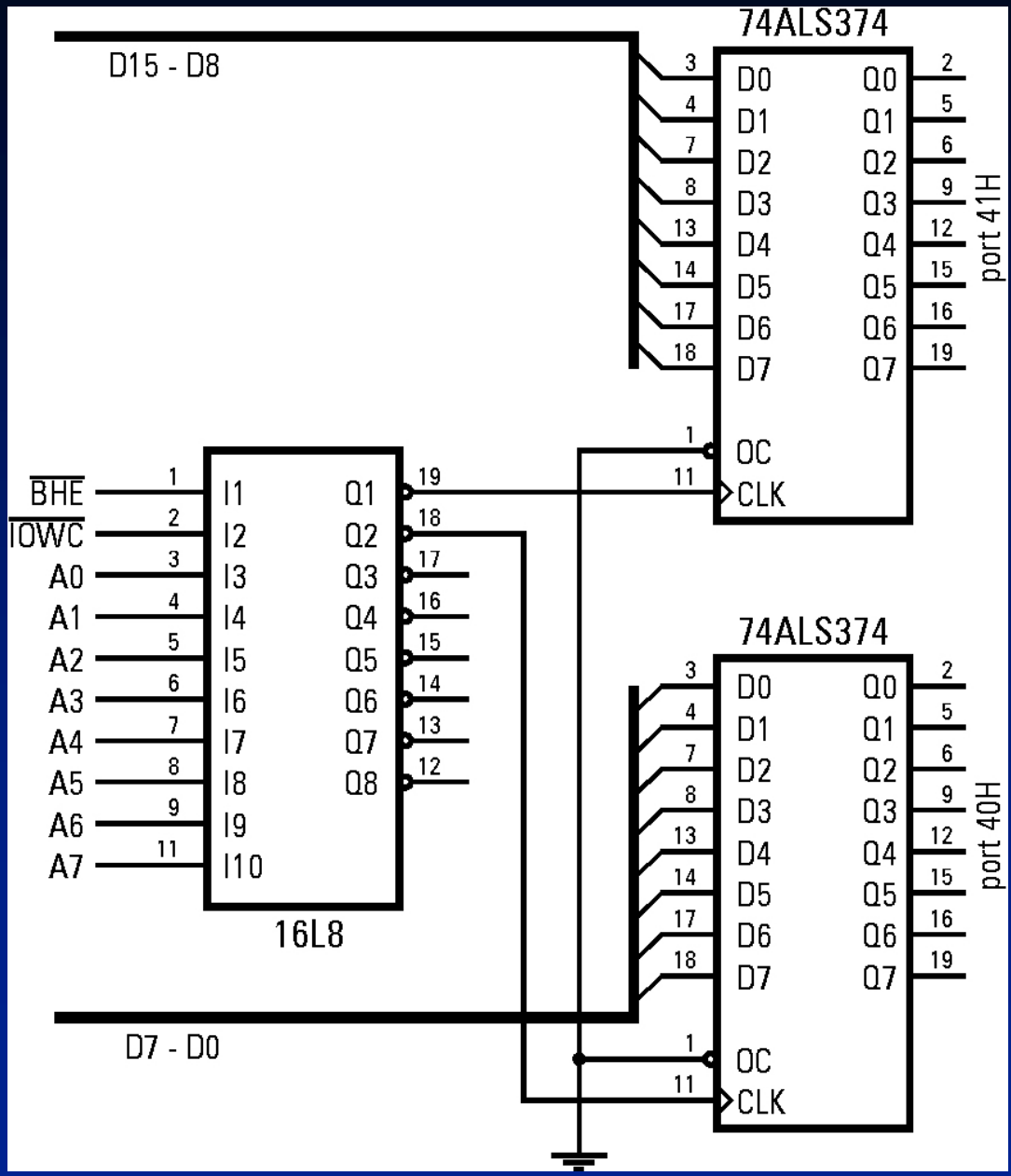
Struktura jednog dekodera



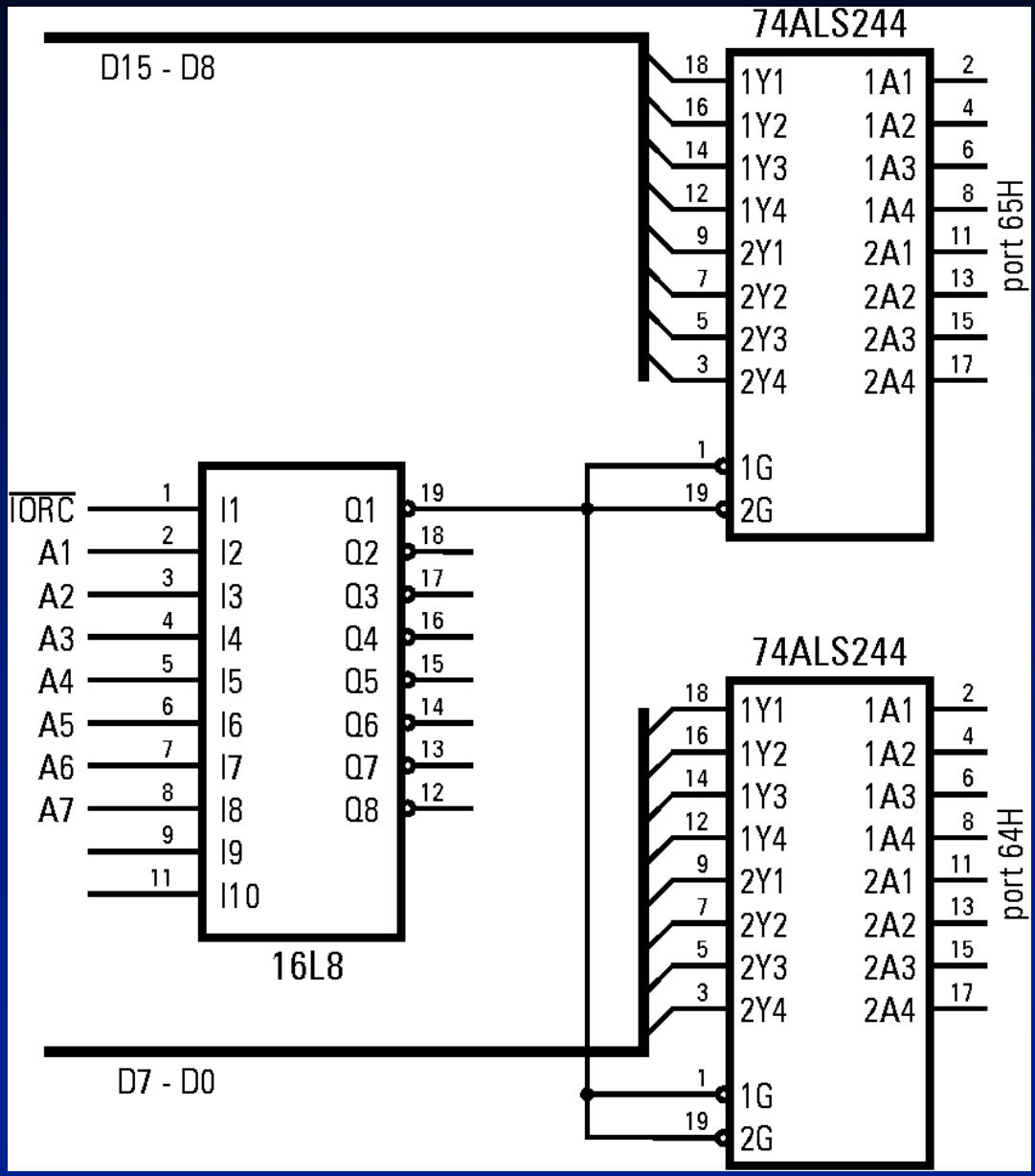
U/I banke kod 8086-80386 SX



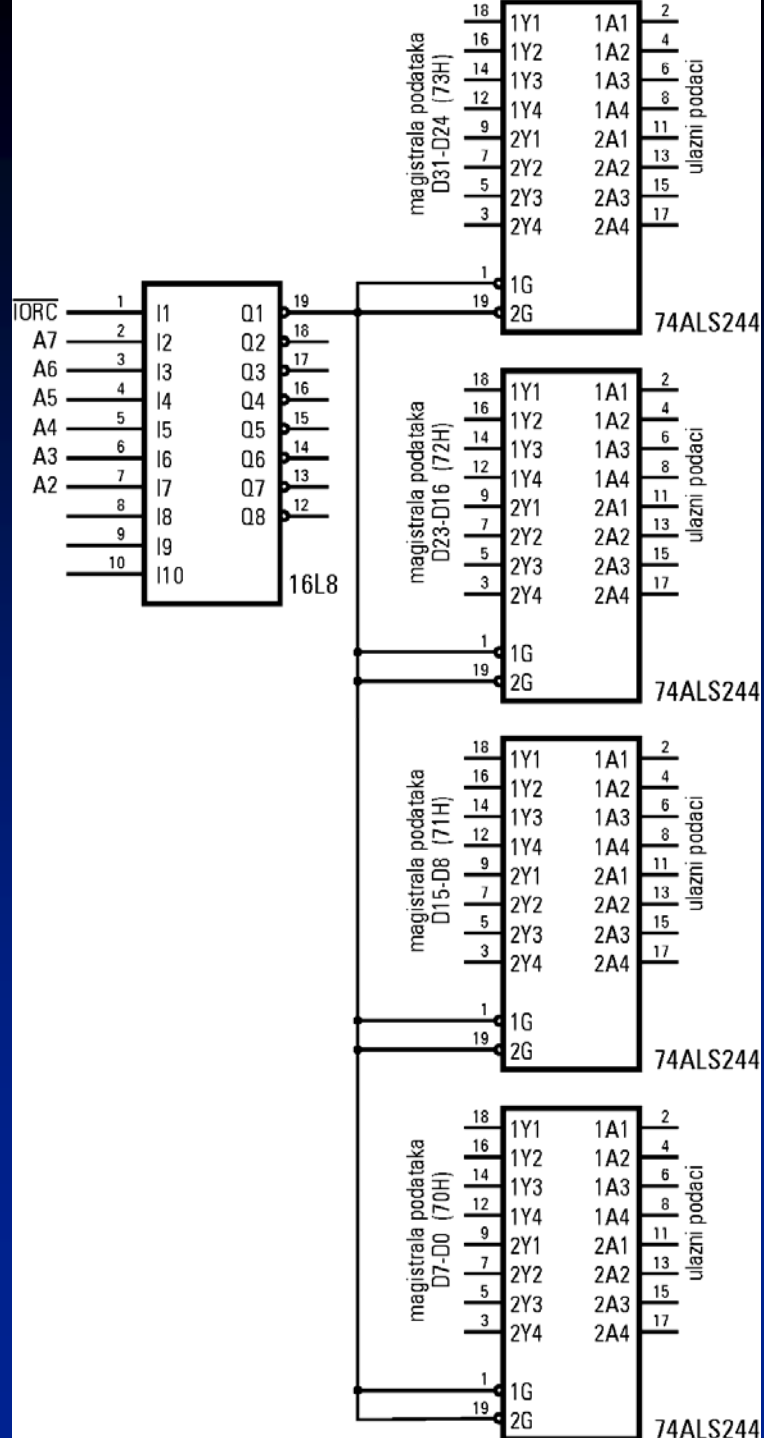
U/I port dekoder



Povezivanje 16-bitnog U/I porta



Povezivanje 32-bitnog ulaznog porta



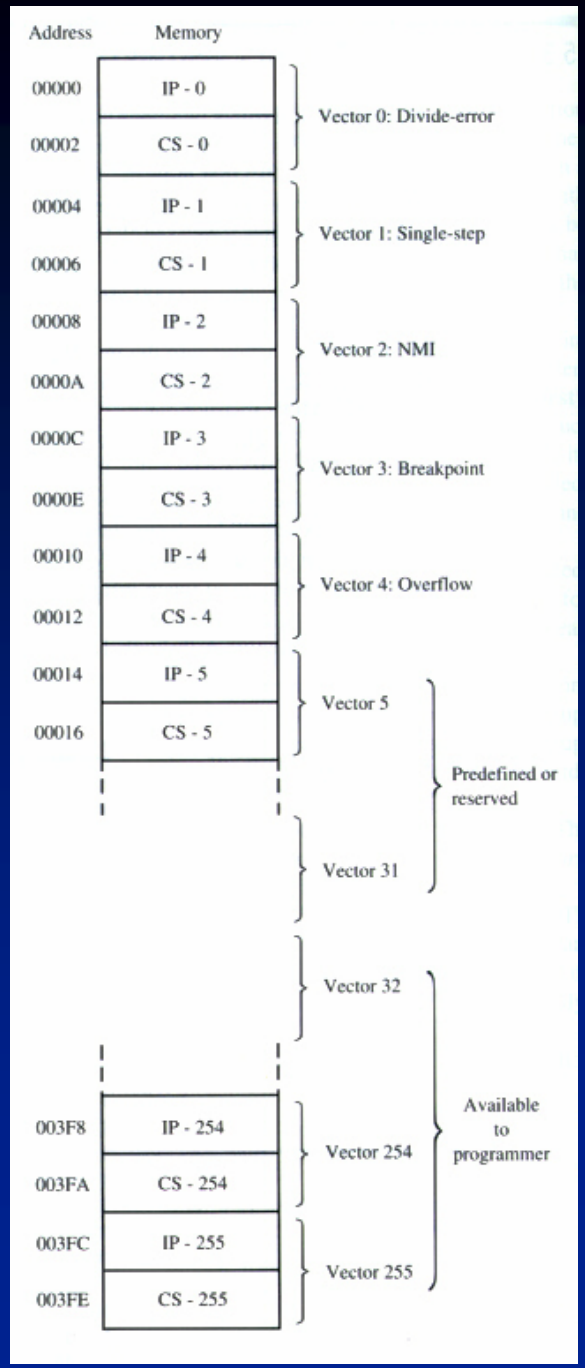
PREKIDI

Vektorska tabela

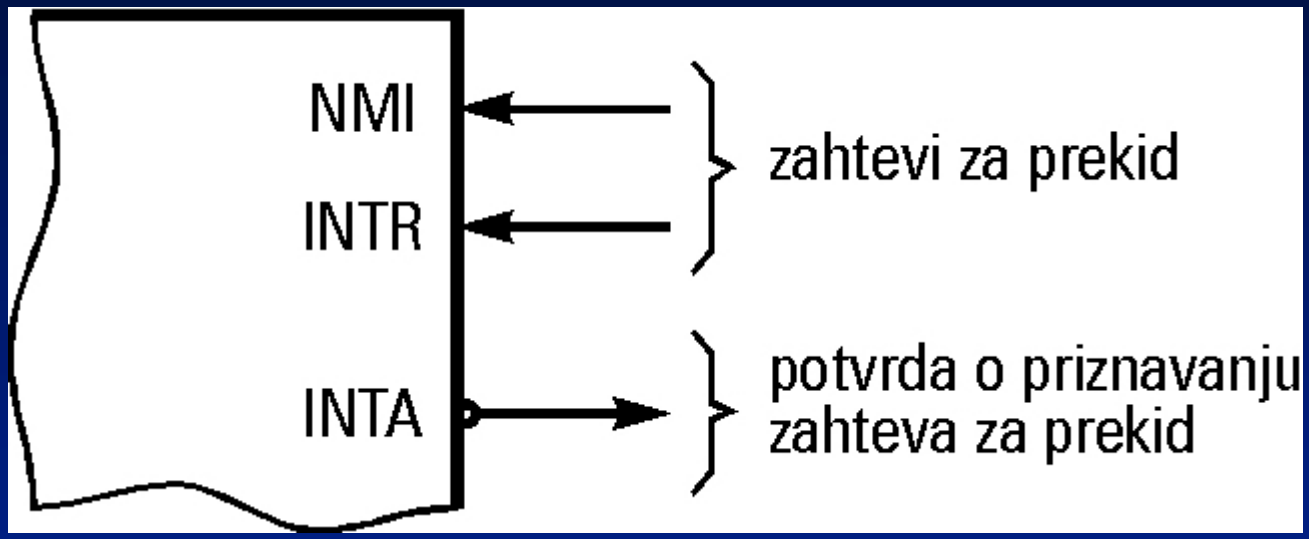
bilo koji vektor prekida

3	segment (viši deo)
2	segment (niži deo)
1	ofset (viši deo)
0	ofset (niži deo)

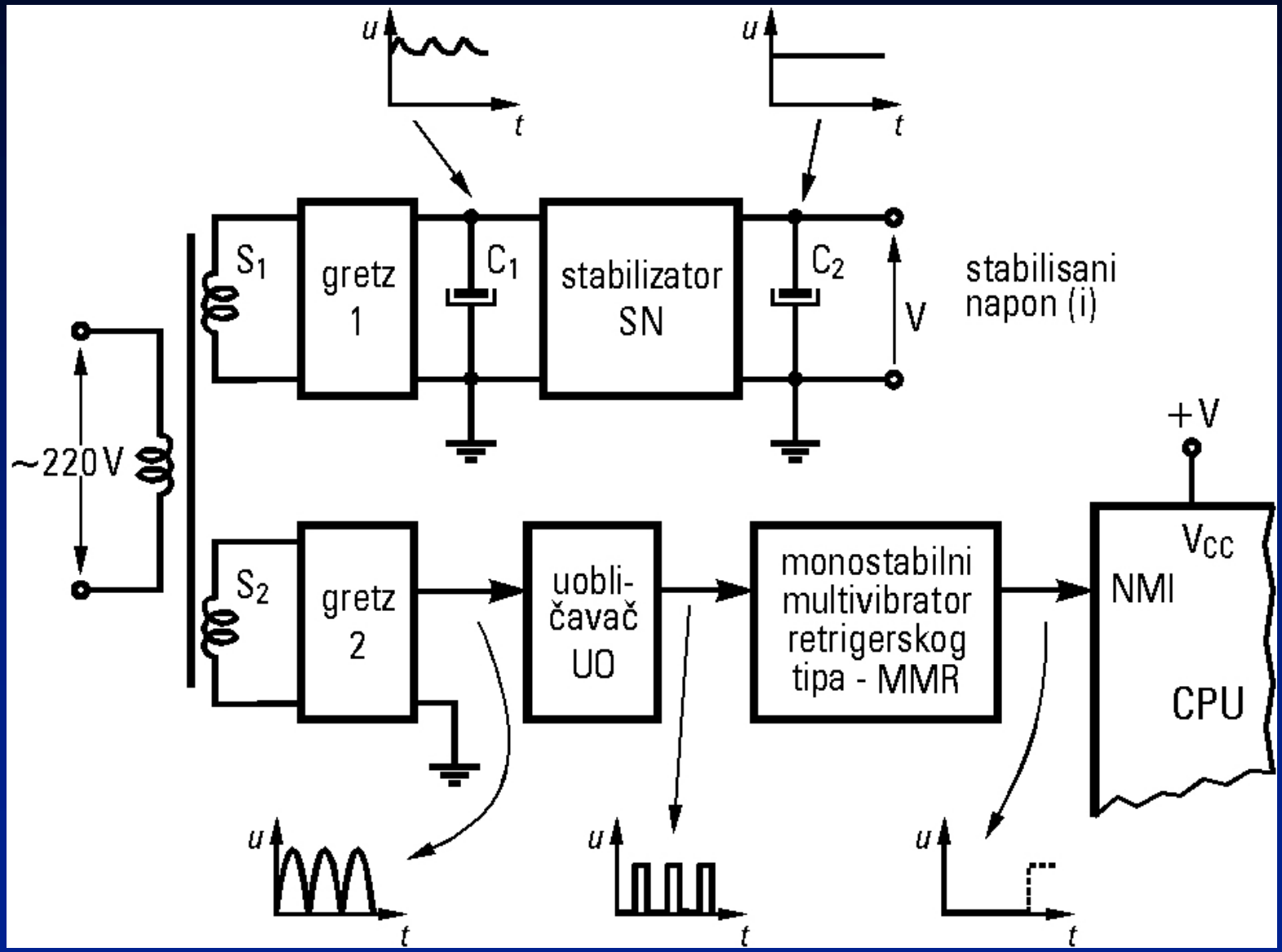
Vektorska tabela drugi pogled



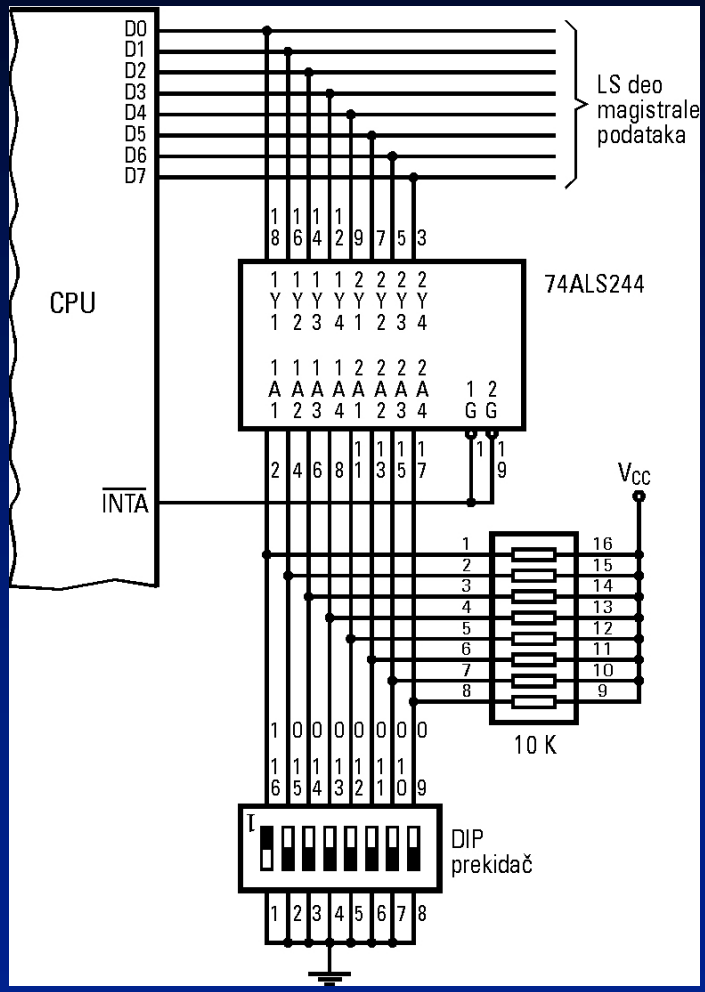
Pinovi prekida



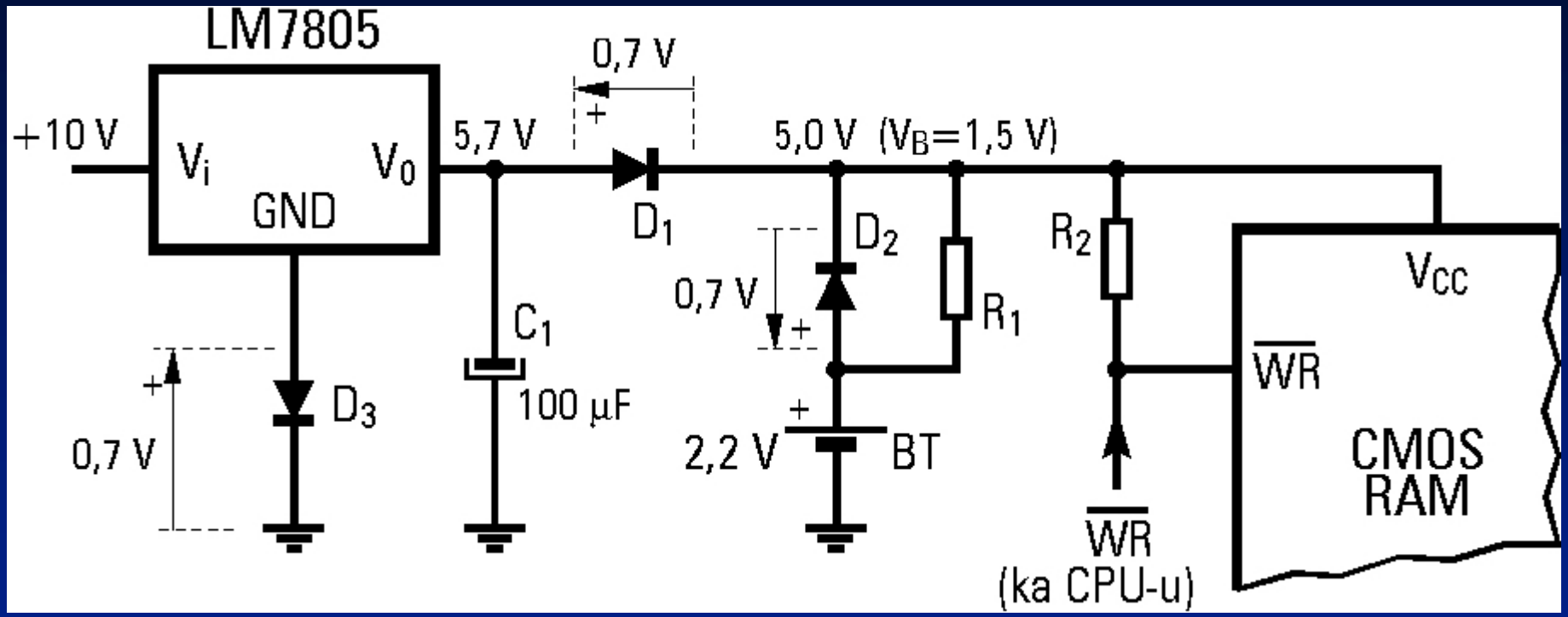
Kolo za detekciju nestanka napona napajanja



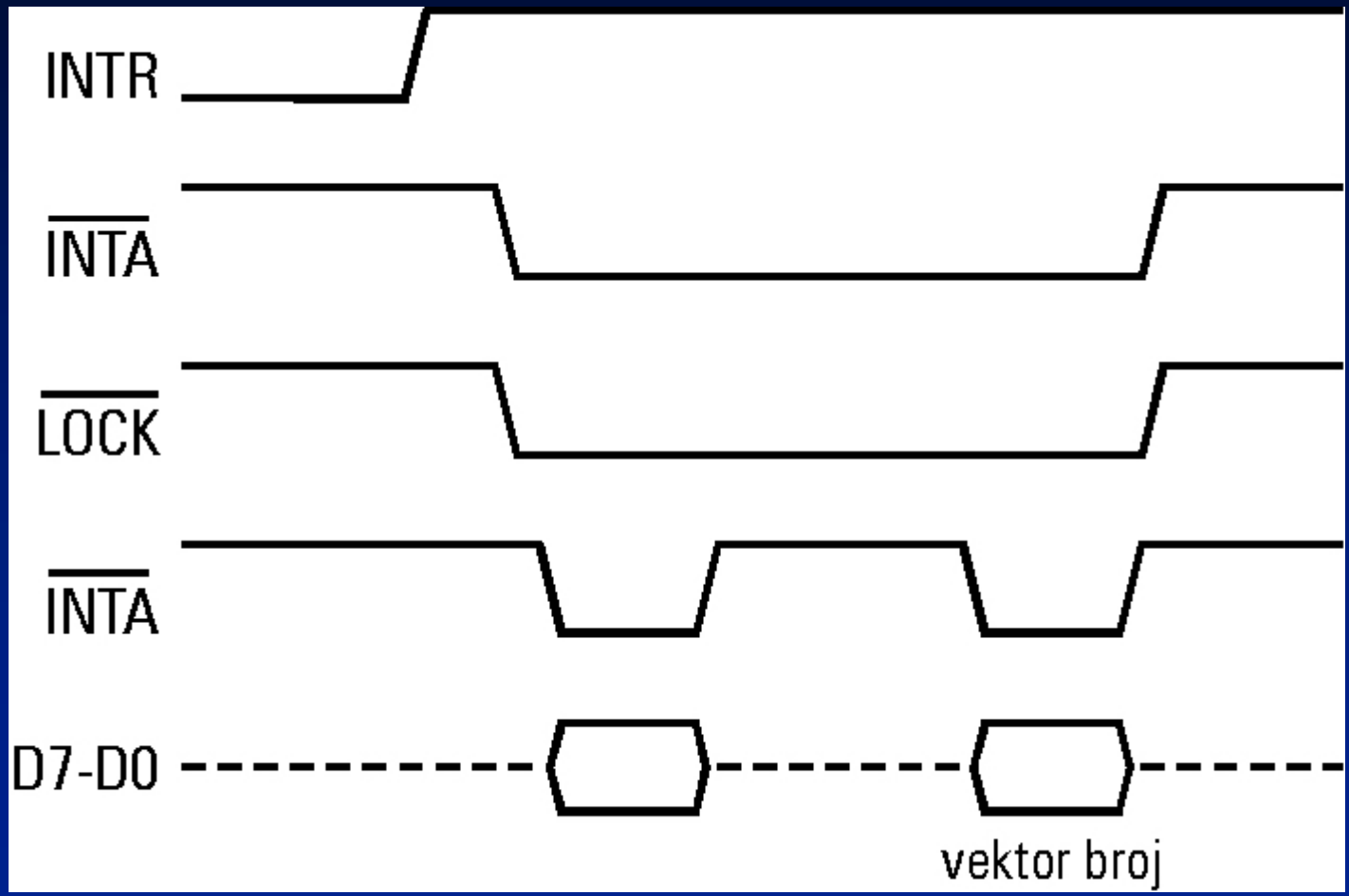
Povezivanje 16-bitnog U/I porta



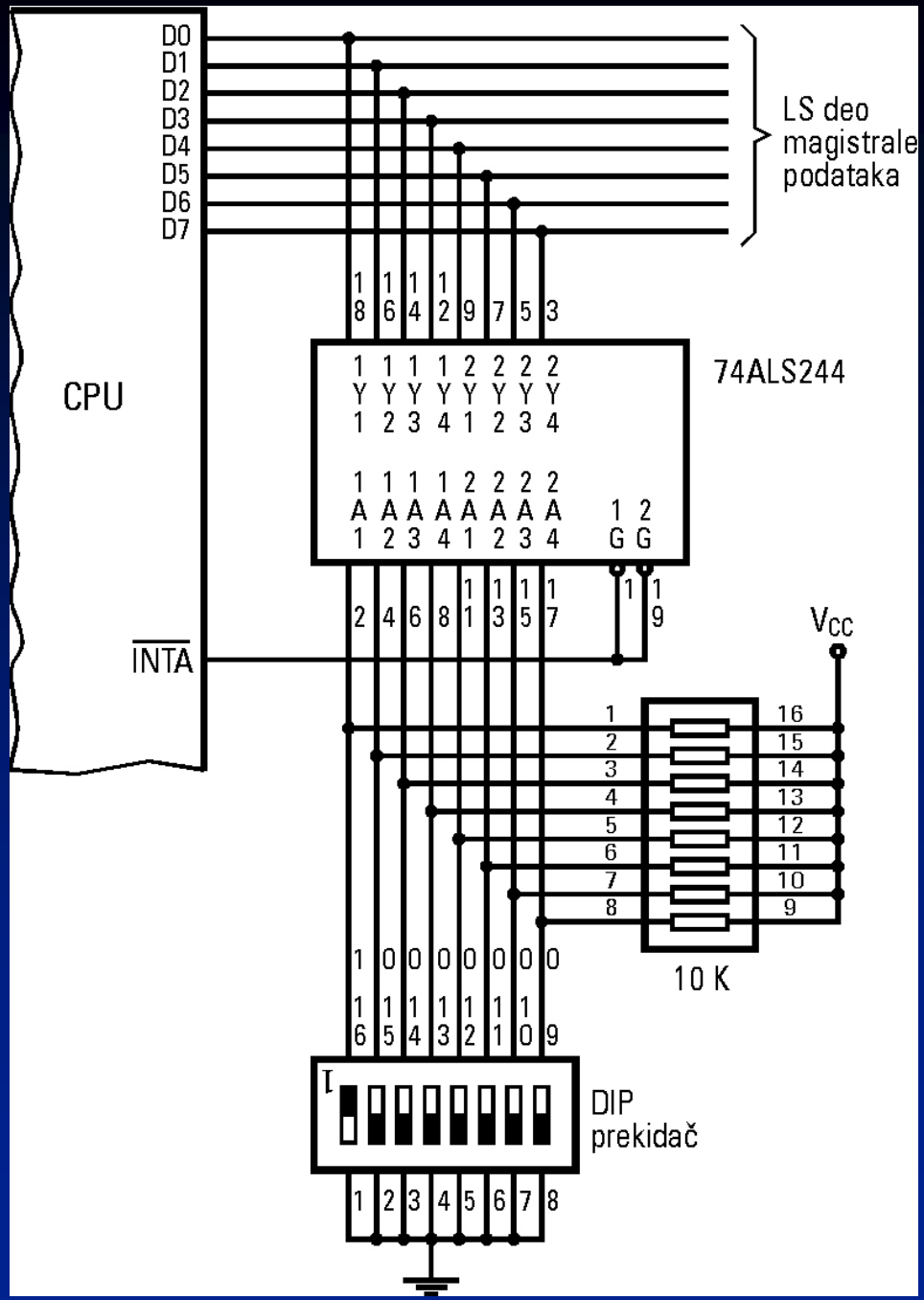
Rezervno baterijsko napajanje



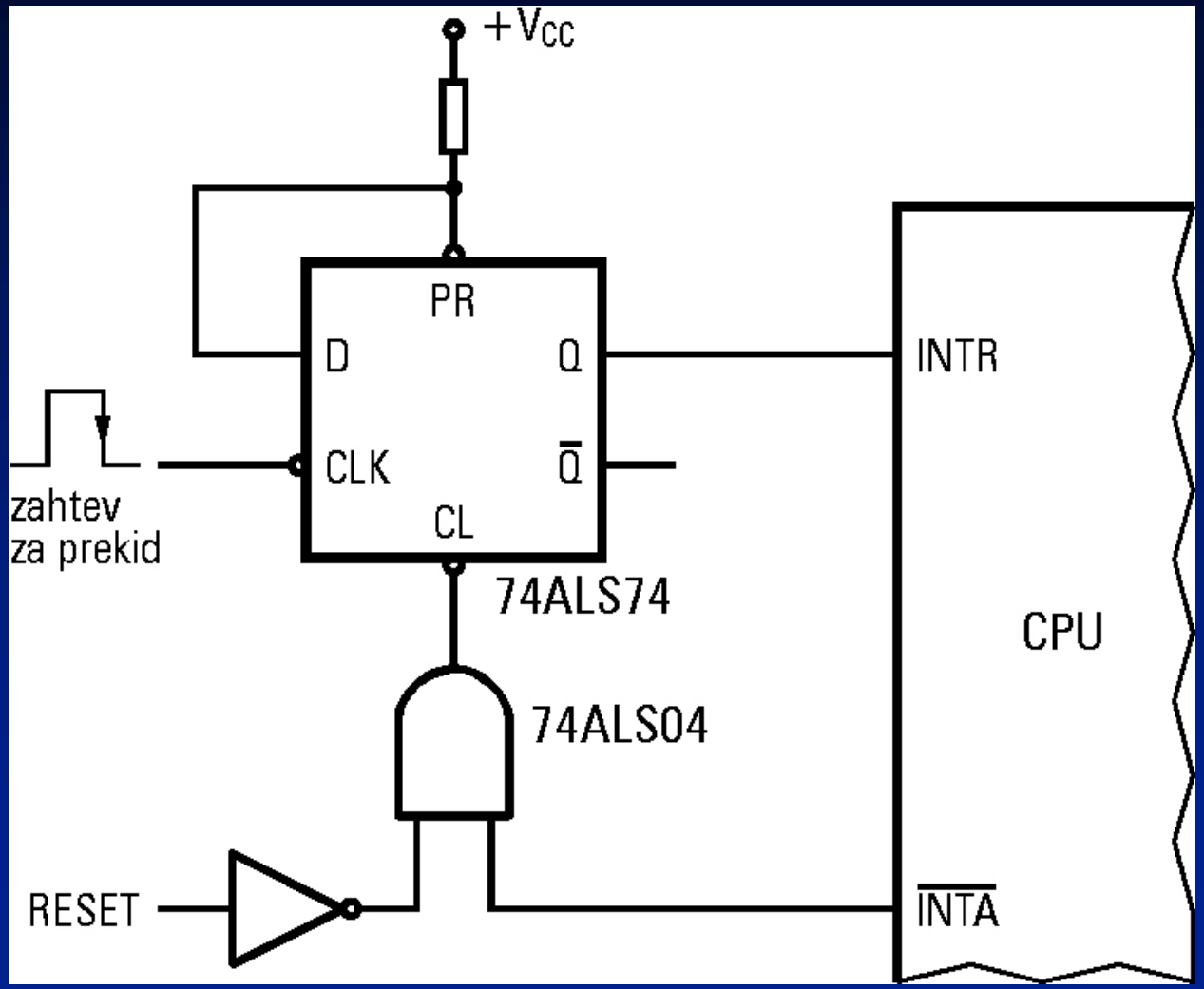
Vremenski dijagram INTR i INTA signala



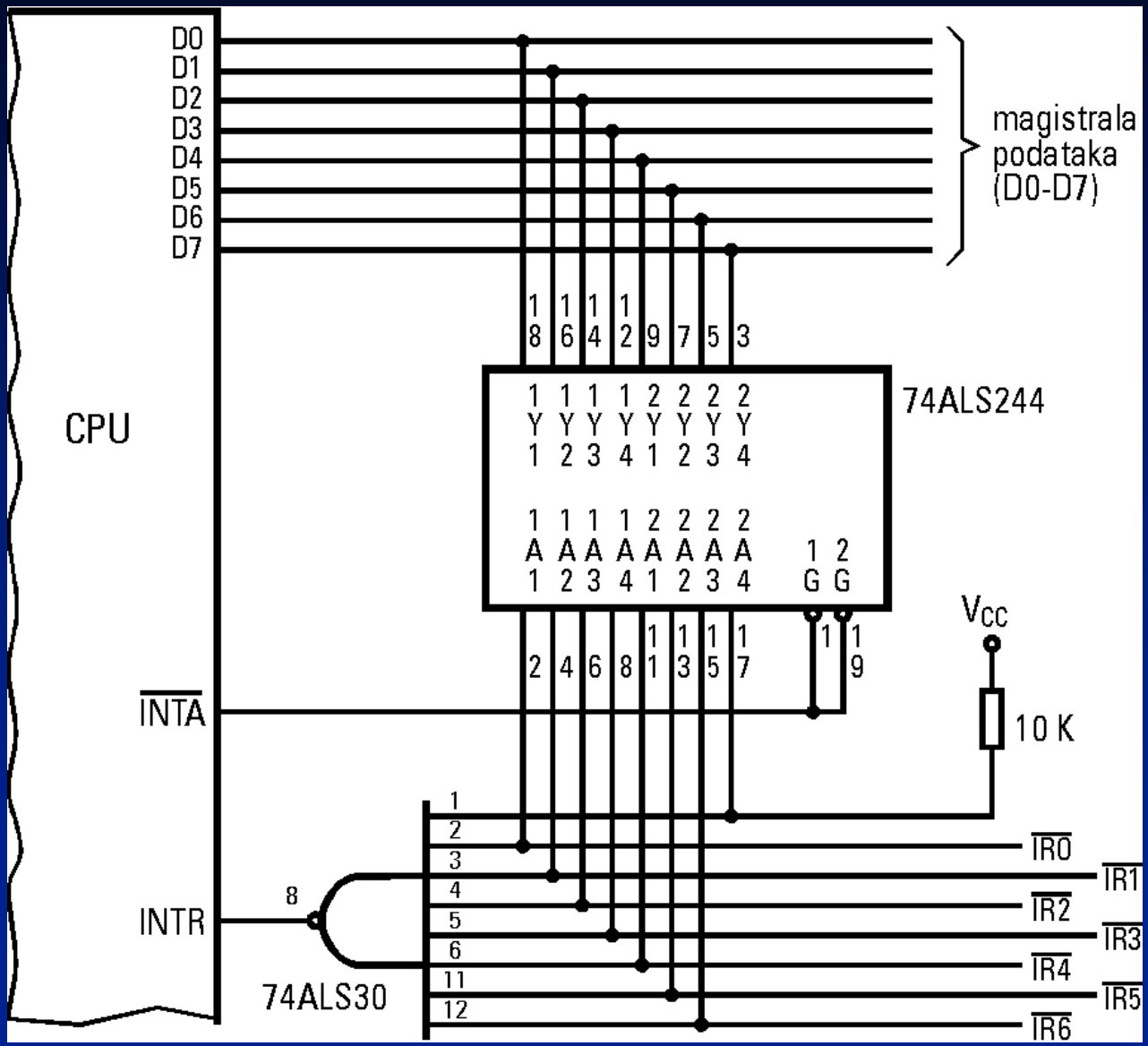
Struktura hardvera za prihvatanje vektor-broja 80h



Konverzija INTR u INTR koji se inicira ivicom

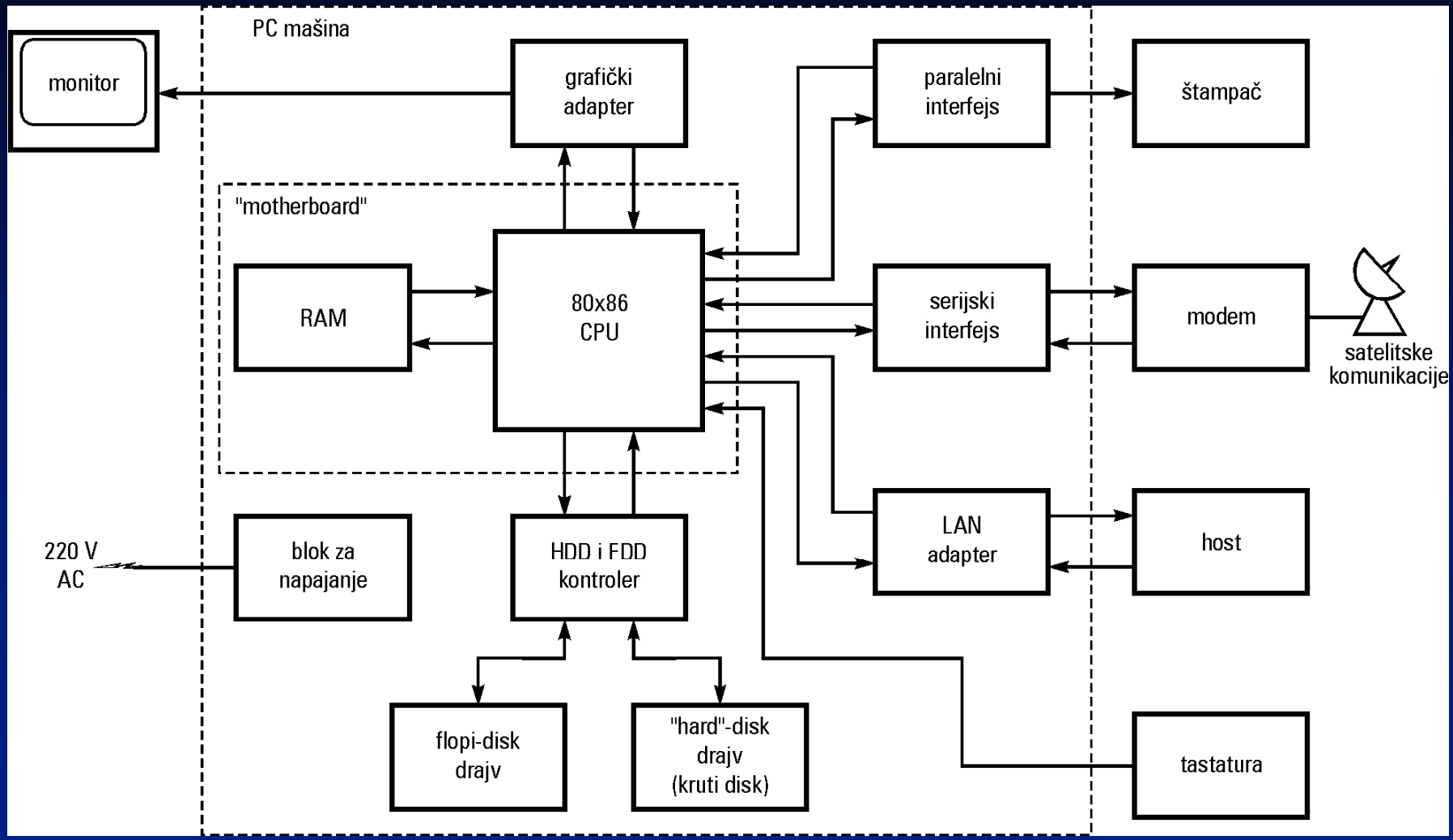


Proširenje INTR ulaza sa jedne na sedam linija zahteva za prekid

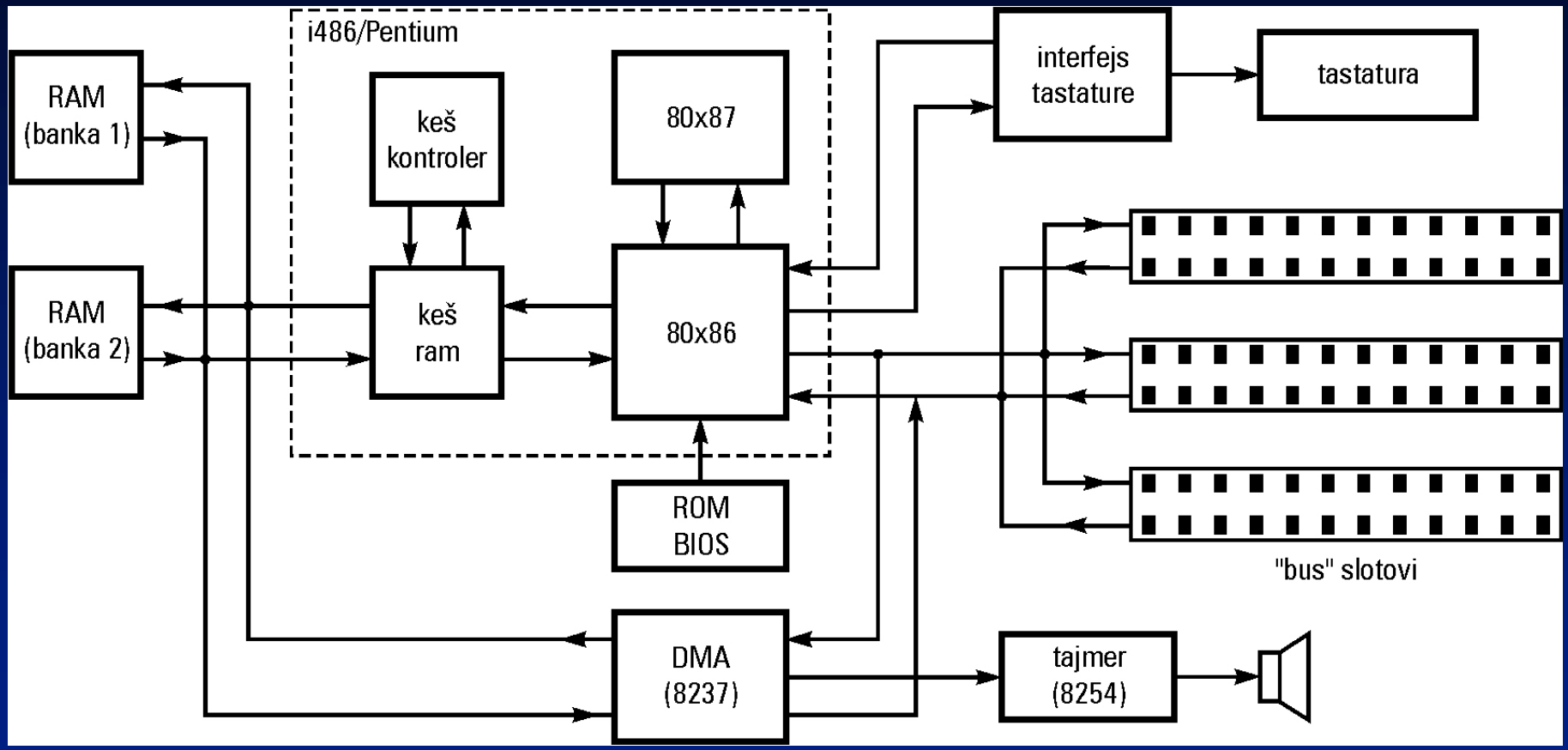


Magistrale kod PC mašina

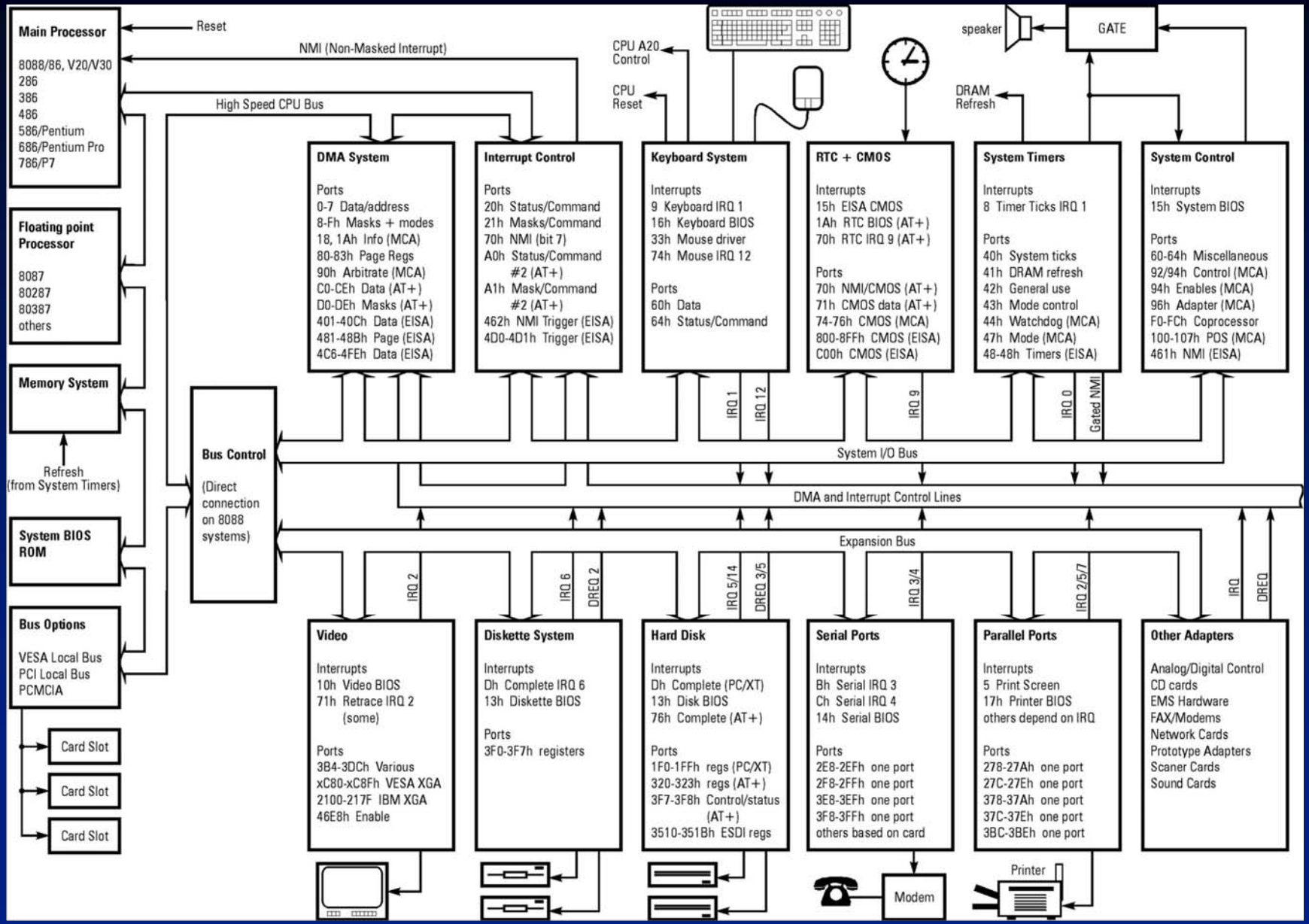
Blok dijagram PC mašine



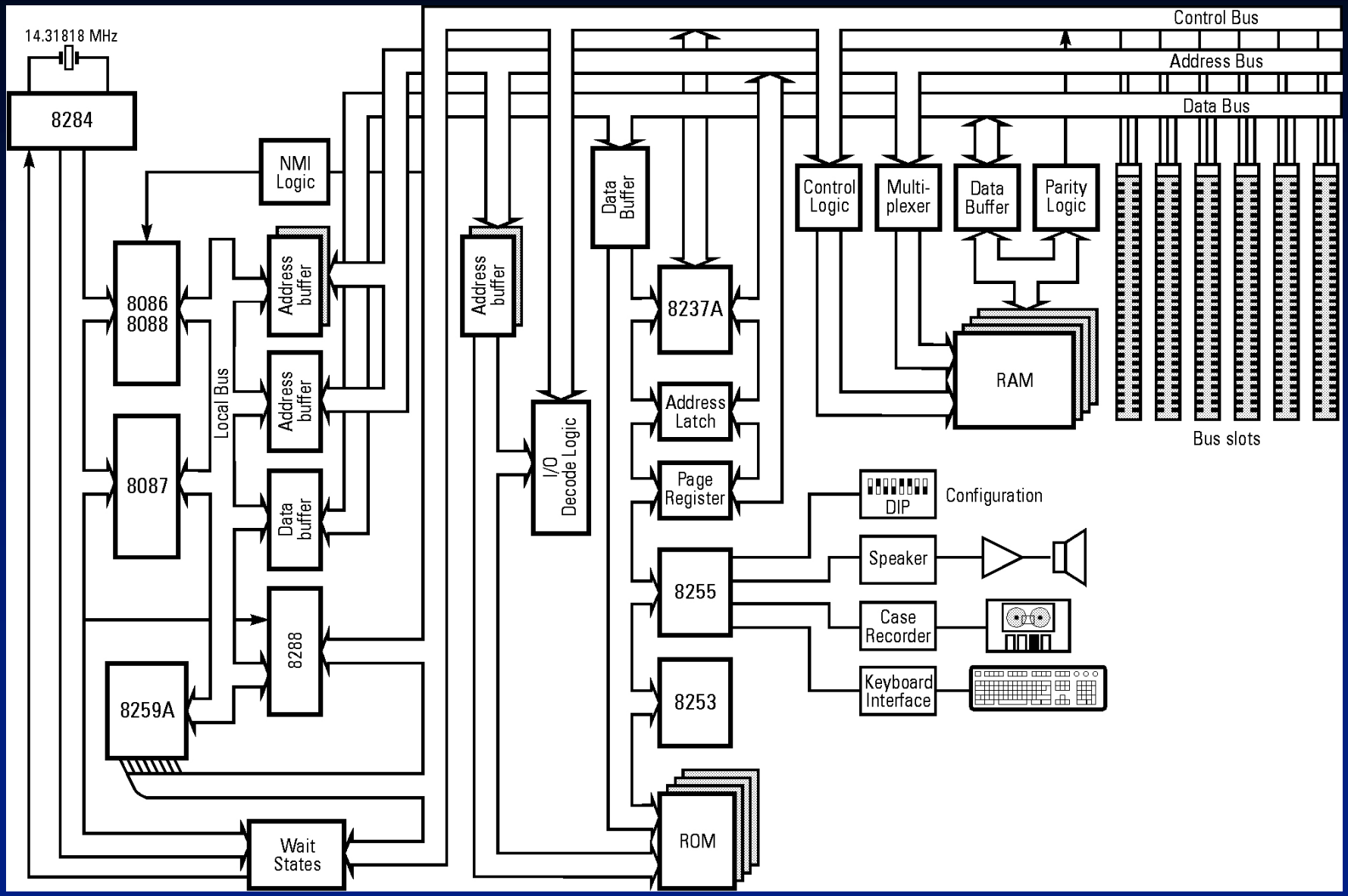
Blok dijagram *mother-board*-a



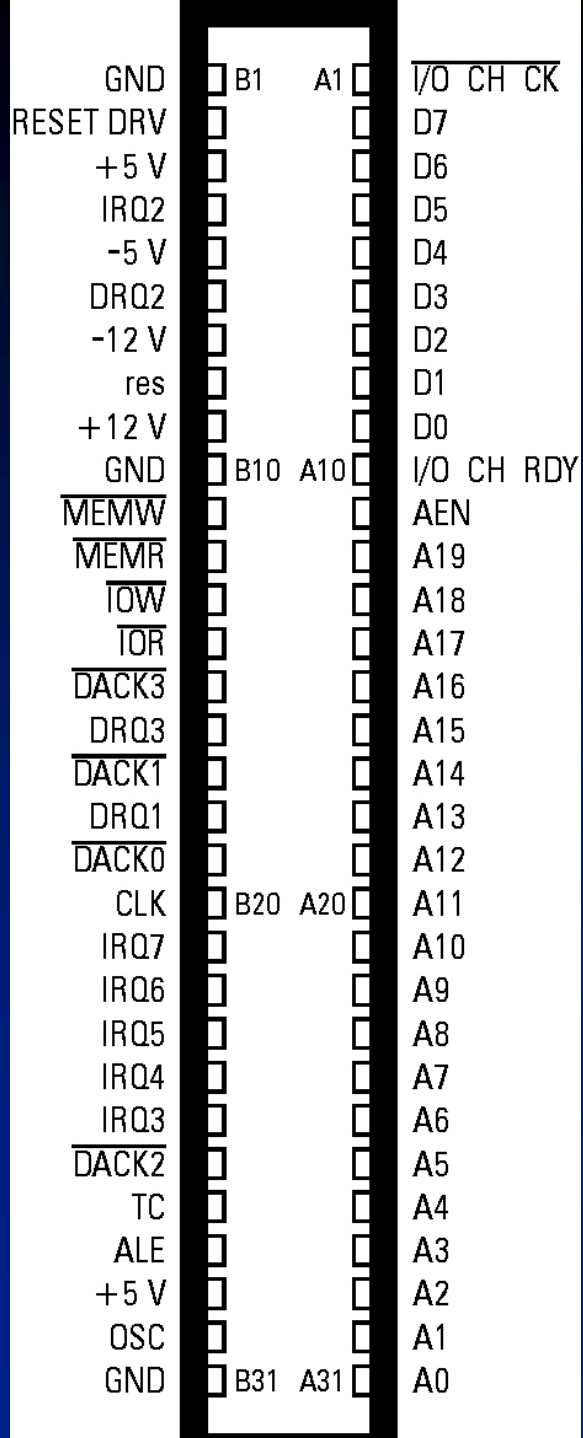
Pogled sistemskog programera na PC mašinu



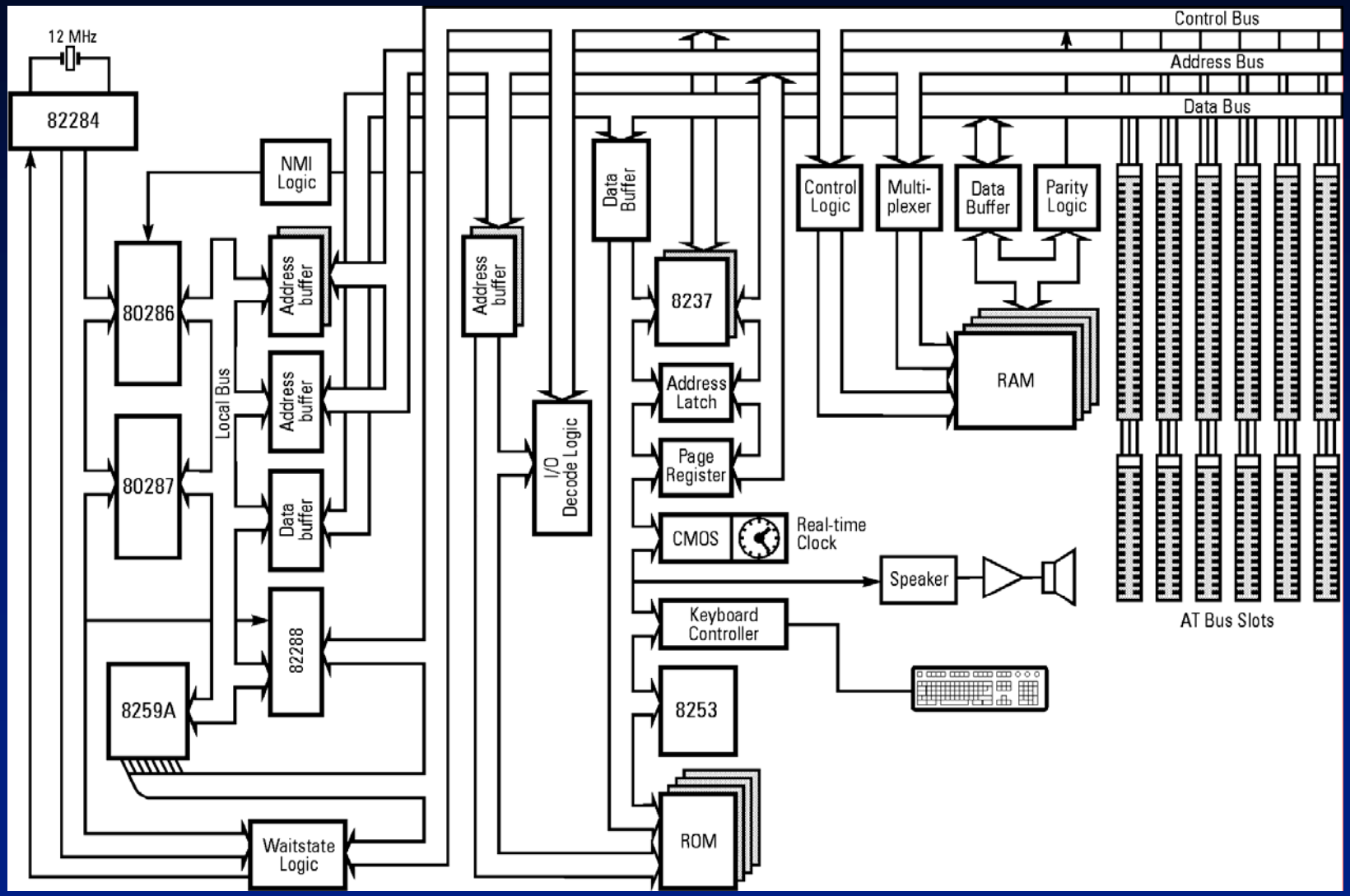
Arhitektura PC/XT mašine



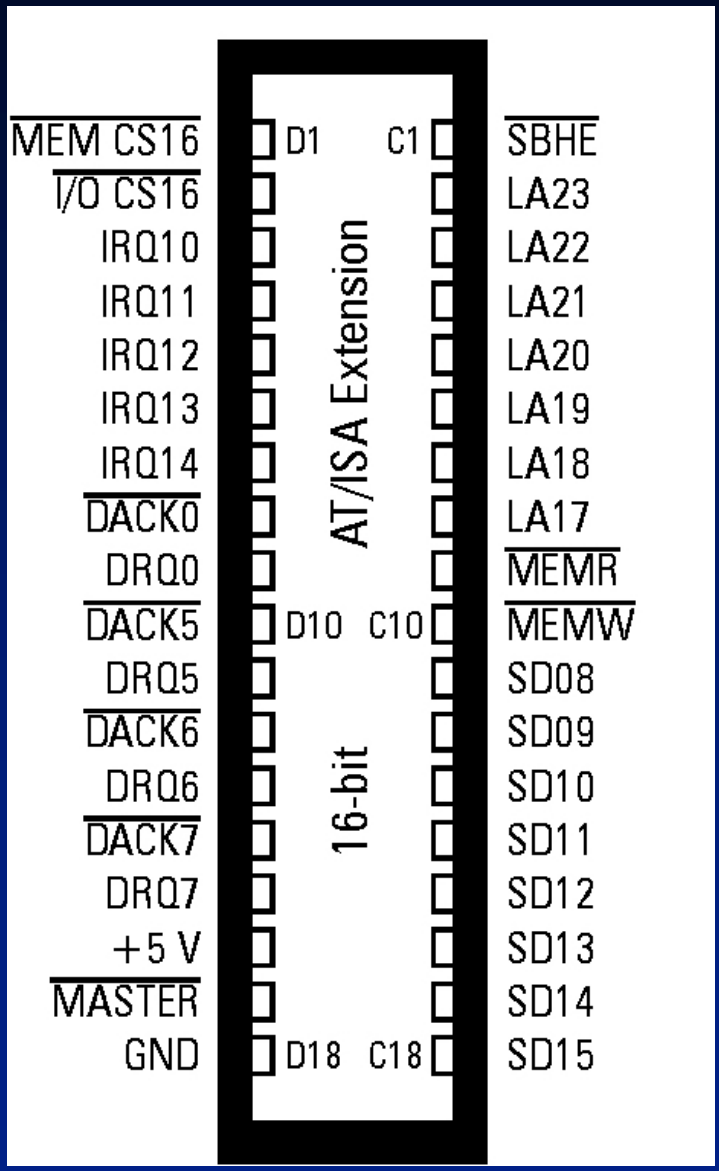
BUS slot kod PC/XT

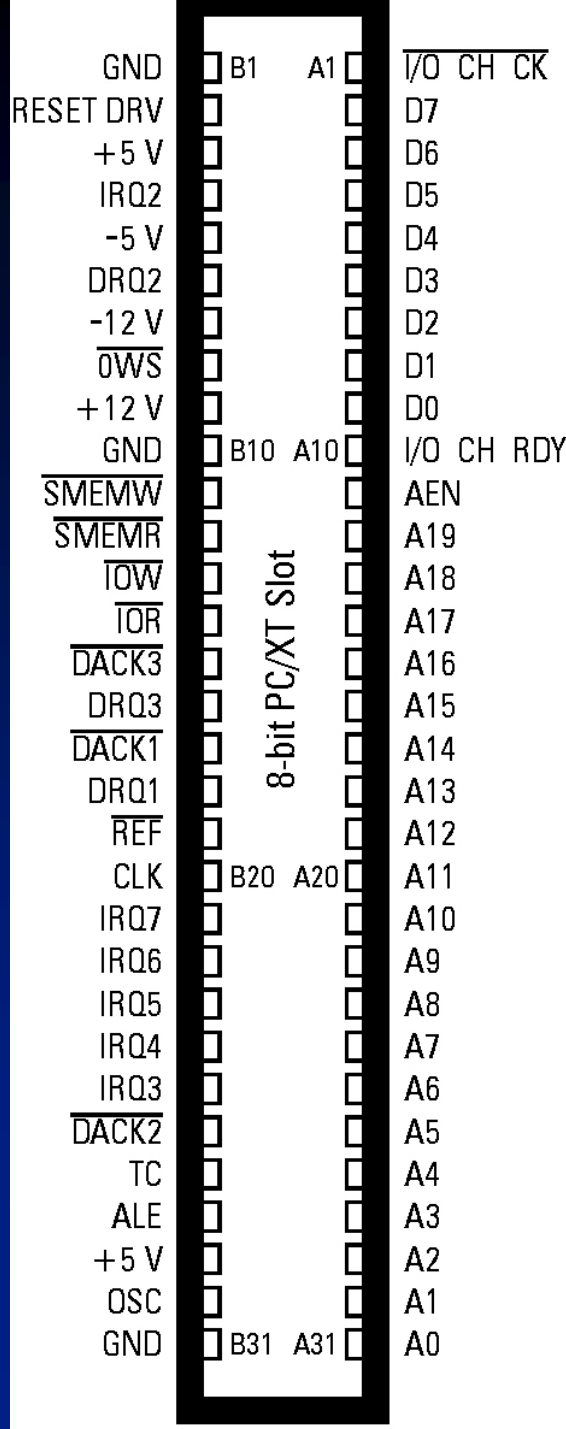


PC AT arhitektura

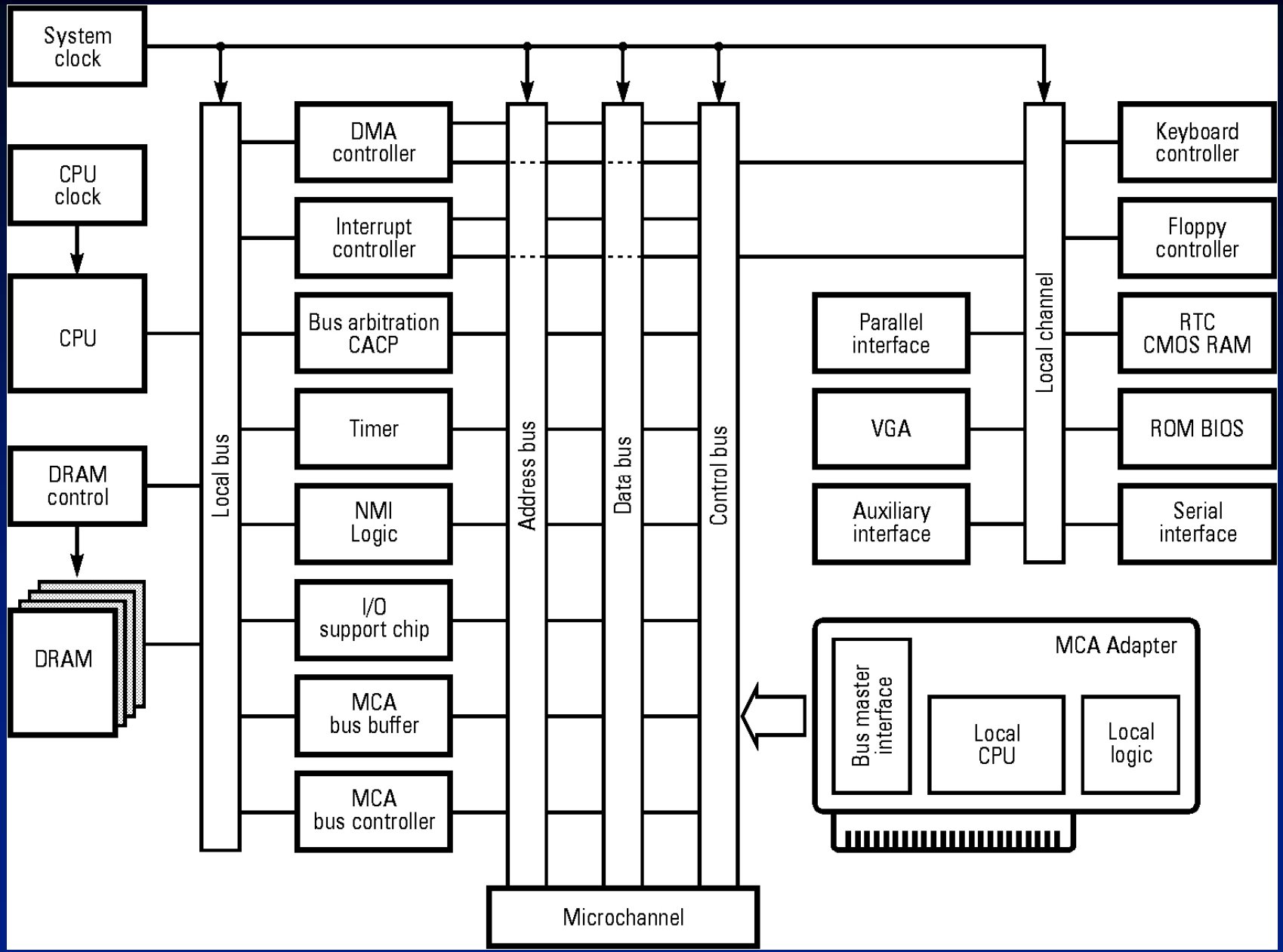


AT bus slot

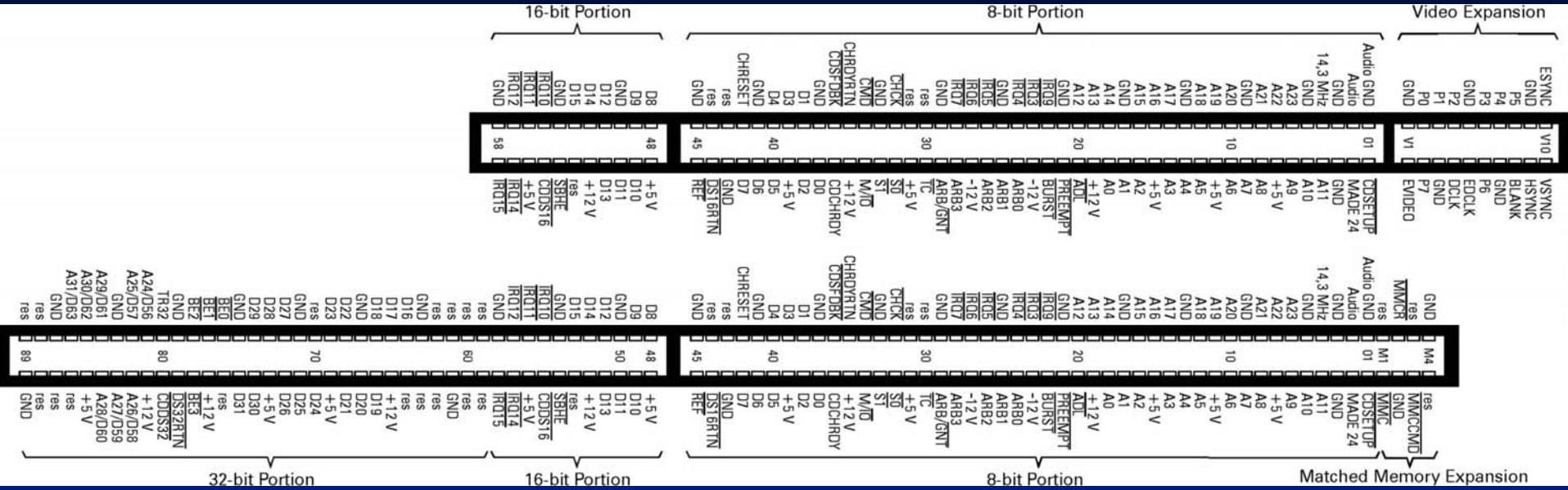

MEM CS16
I/O CS16
IRQ10
IRQ11
IRQ12
IRQ13
IRQ14
DACK0
DRQ0
DACK5
DRQ5
DACK6
DRQ6
DACK7
DRQ7
+5 V
MASTER
GND
D1
C1
D10
C10
D18
C18
16-bit
AT/ISA Extension
SBHE
LA23
LA22
LA21
LA20
LA19
LA18
LA17
MEMR
MEMW
SD08
SD09
SD10
SD11
SD12
SD13
SD14
SD15


GND
RESET DRV
+5 V
IRQ2
-5 V
DRQ2
-12 V
0WS
+12 V
GND
SMEMW
SMEMR
IOW
IOR
DACK3
DRQ3
DACK1
DRQ1
REF
CLK
IRQ7
IRQ6
IRQ5
IRQ4
IRQ3
DACK2
TC
ALE
+5 V
OSC
GND
B1
A1
B10
A10
B20
A20
B31
A31
8-bit PC/XT Slot
I/O CH CK
D7
D6
D5
D4
D3
D2
D1
D0
I/O CH RDY
AEN
A19
A18
A17
A16
A15
A14
A13
A12
A11
A10
A9
A8
A7
A6
A5
A4
A3
A2
A1
A0

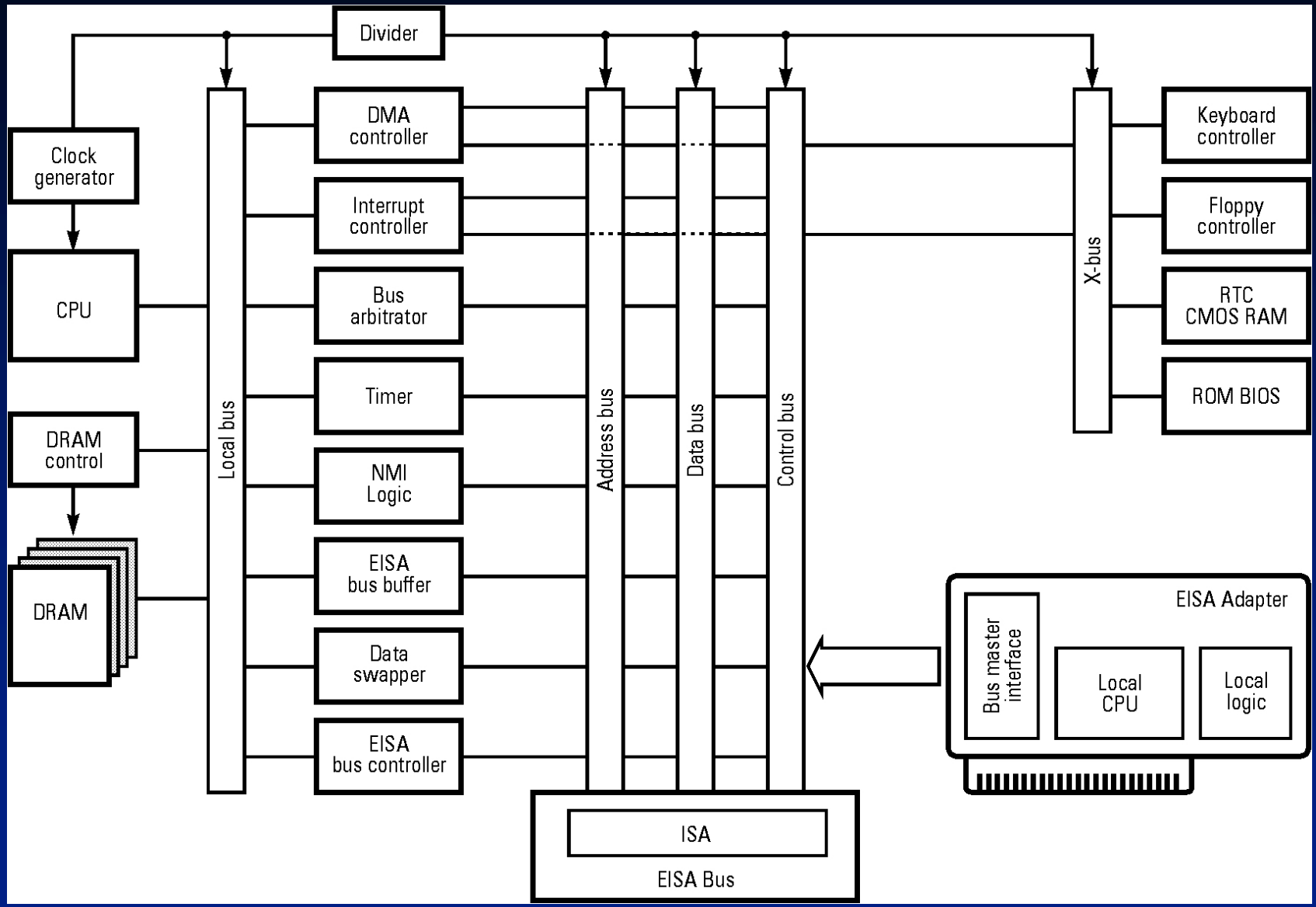
MCA arhitektura



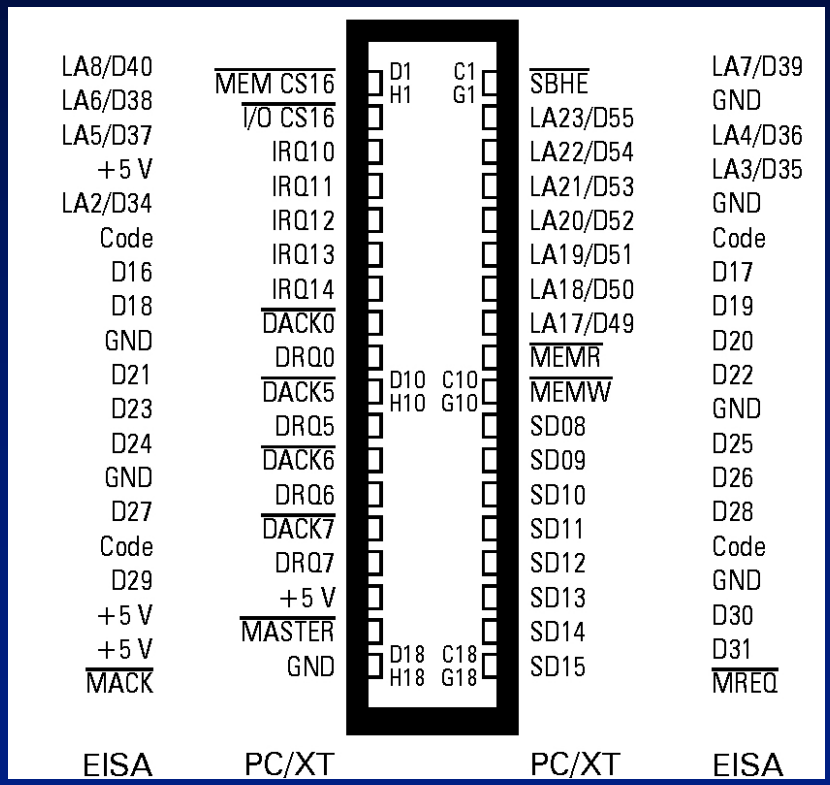
MCA slot

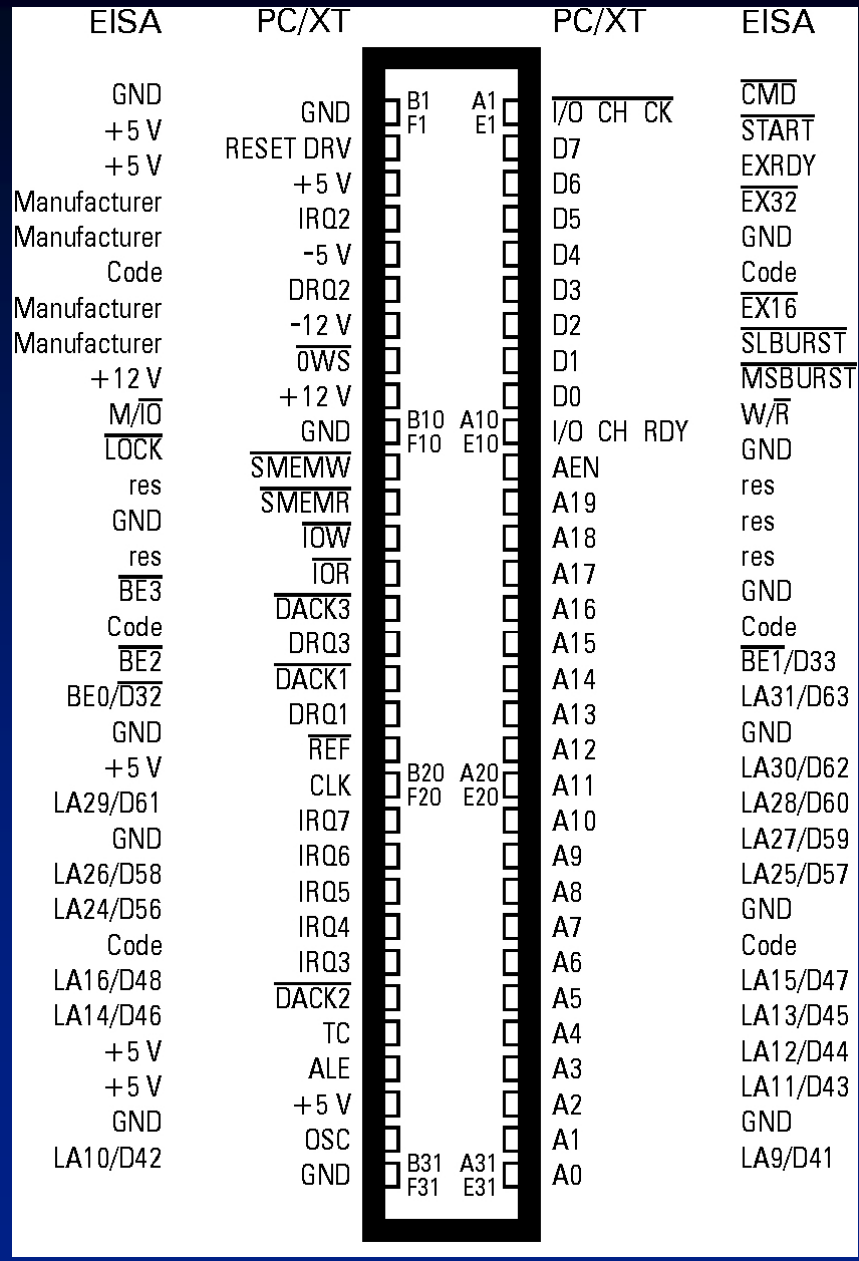

Video Expansion
8-bit Portion
16-bit Portion
Matched Memory Expansion
8-bit Portion
16-bit Portion
32-bit Portion

EISA arhitektura

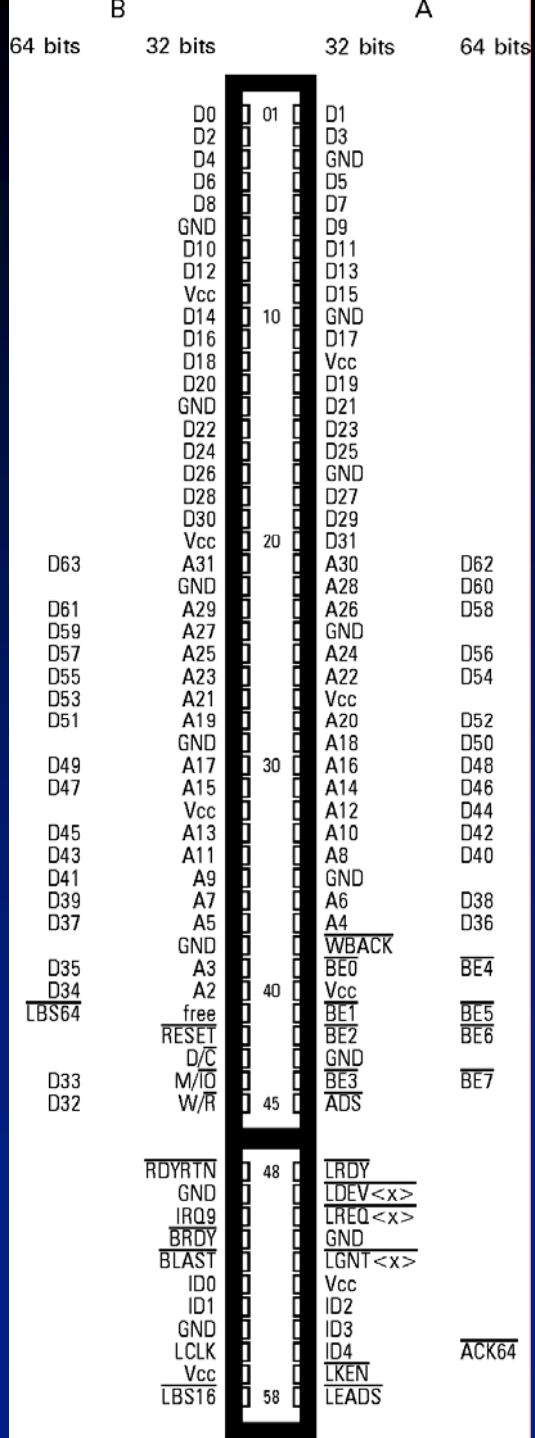


Slotovi EISA magistrale

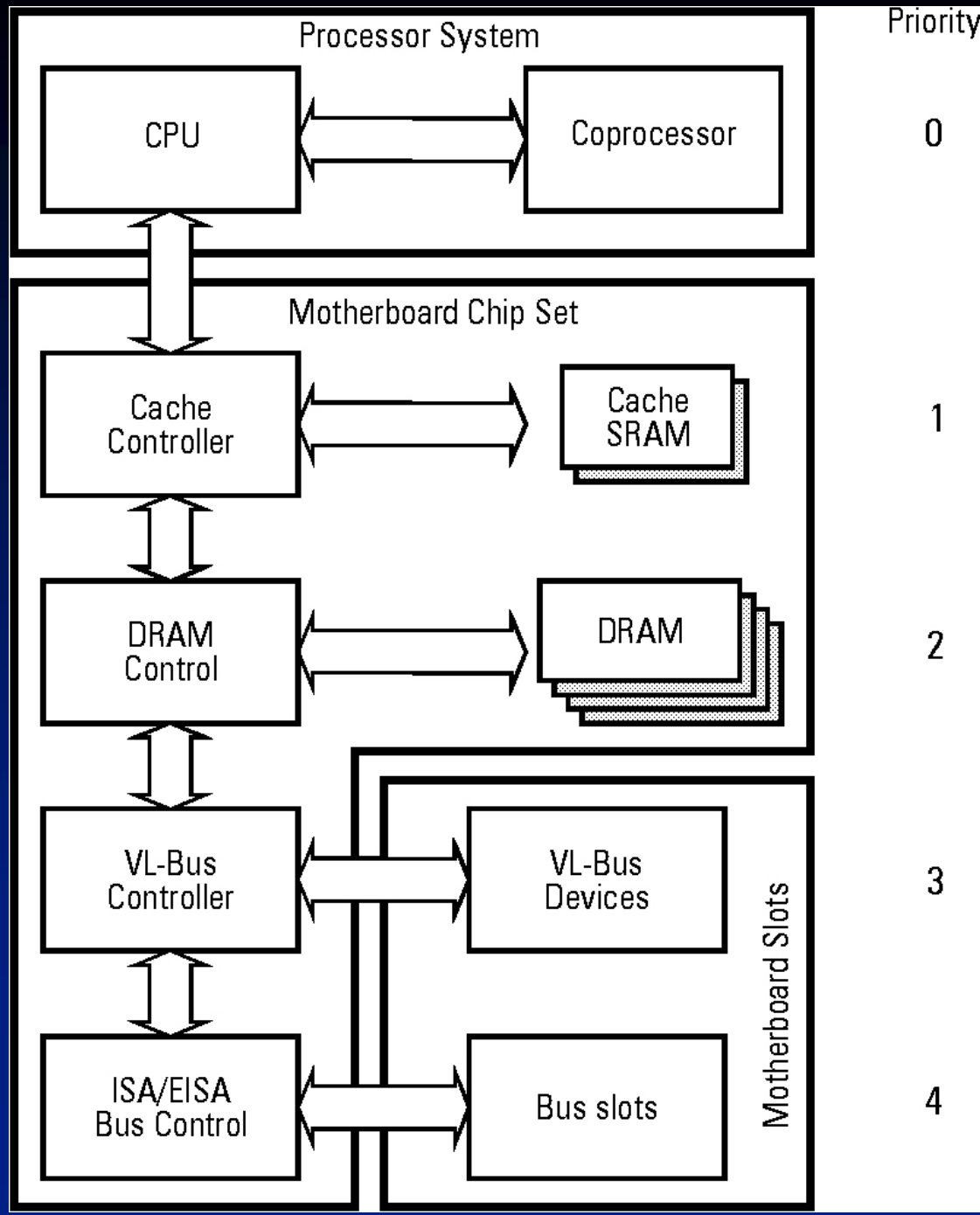




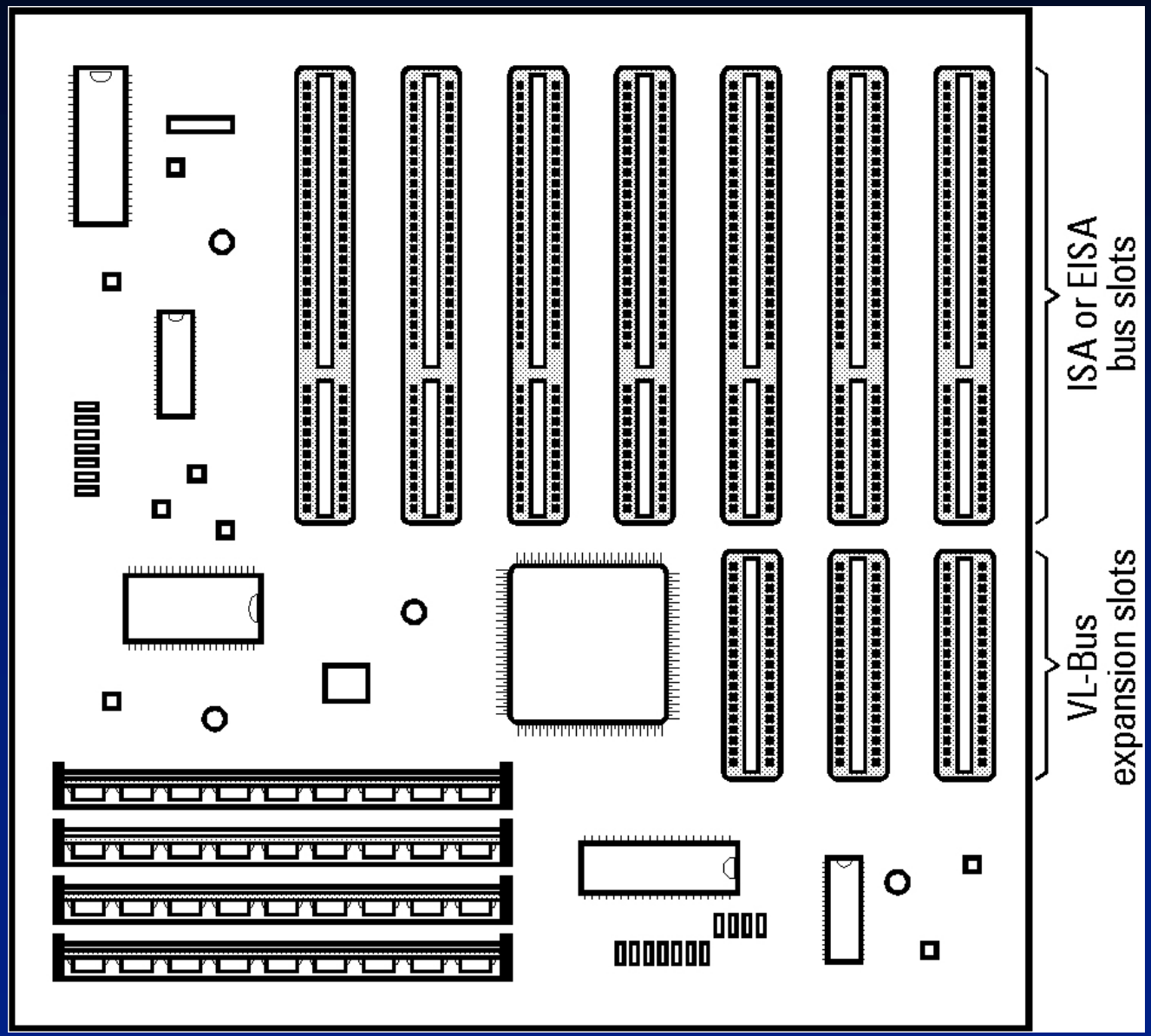
VLB slot


B
A
64 bits
32 bits
32 bits
64 bits
D0 01 D1
D2 D3
D4 GND
D6 D5
D8 D7
GND D9
D10 D11
D12 D13
Vcc D15
D14 10 GND
D16 D17
D18 Vcc
D20 D19
GND D21
D22 D23
D24 D25
D26 GND
D28 D27
D30 D29
Vcc 20 D31
D63 A31 A30 D62
GND A28 D60
D61 A29 A26 D58
D59 A27 GND
D57 A25 A24 D56
D55 A23 A22 D54
D53 A21 Vcc
D51 A19 A20 D52
GND A18 D50
D49 A17 30 A16 D48
D47 A15 A14 D46
Vcc A12 D44
D45 A13 A10 D42
D43 A11 A8 D40
D41 A9 GND
D39 A7 A6 D38
D37 A5 A4 D36
GND WBACK
D35 A3 BE0 BE4
D34 A2 40 Vcc
LBS64 free BE1 BE5
RESET BE2 BE6
D/C GND
D33 M/IO BE3 BE7
D32 W/R 45 ADS
RDYRTN 48 LRDY
GND LDEV<x>
IRQ9 LREQ<x>
BRDY GND
BLAST LGNT<x>
ID0 Vcc
ID1 ID2
GND ID3
LCLK ID4 ACK64
Vcc LKEN
LBS16 58 LEADS

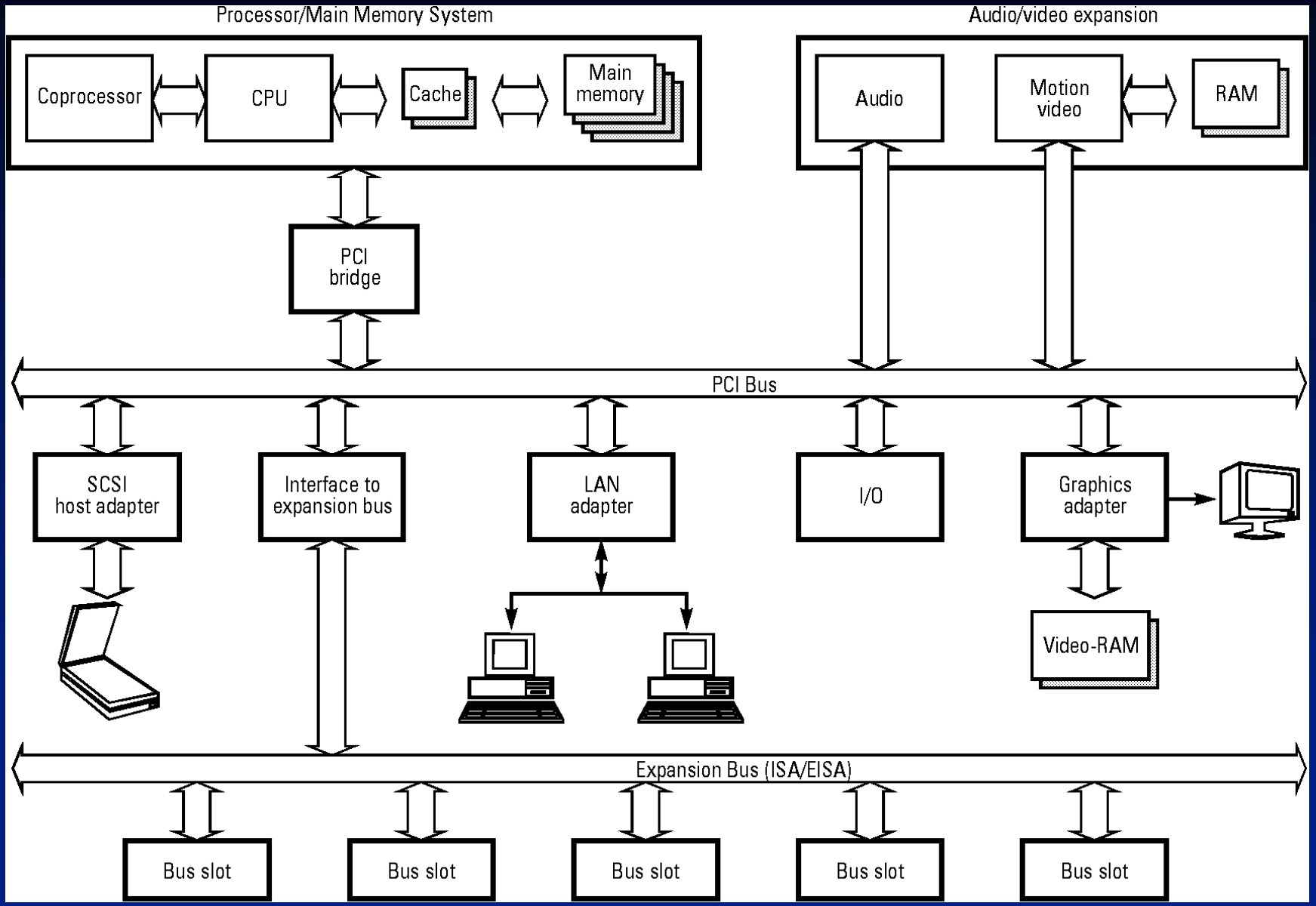
Pririteti opsluživanja i fizičke pozicije VLBI i ISA/EISA bus kontrolera kod PC mašine



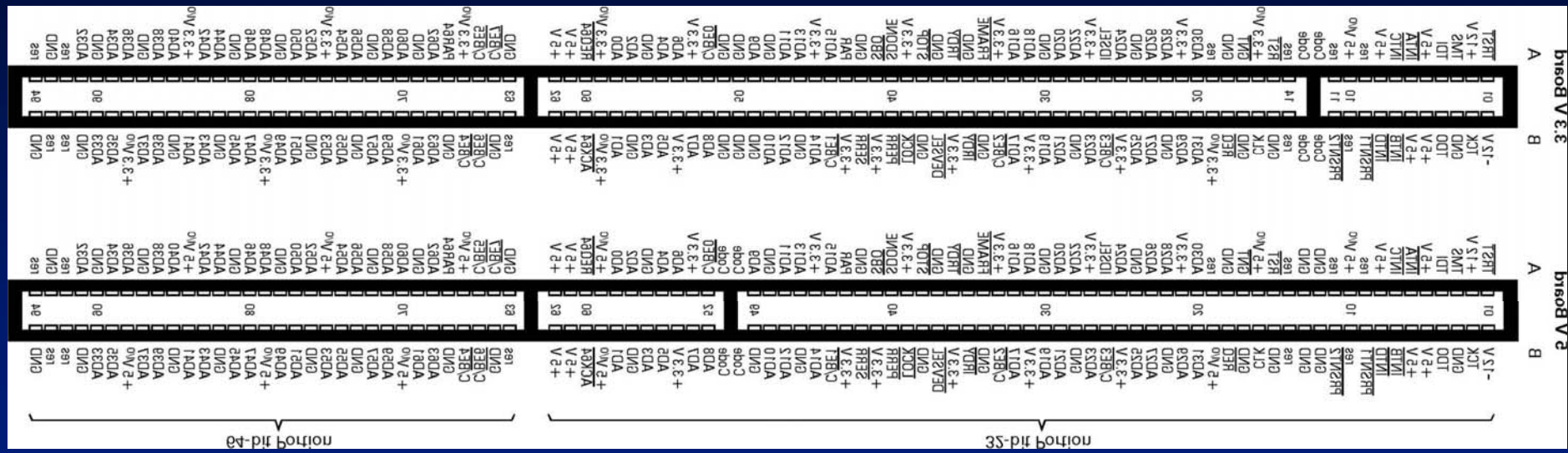
Fizička pozicija ISA/EISA bus slotova kao i VLB slotova kod osnovne PC ploče



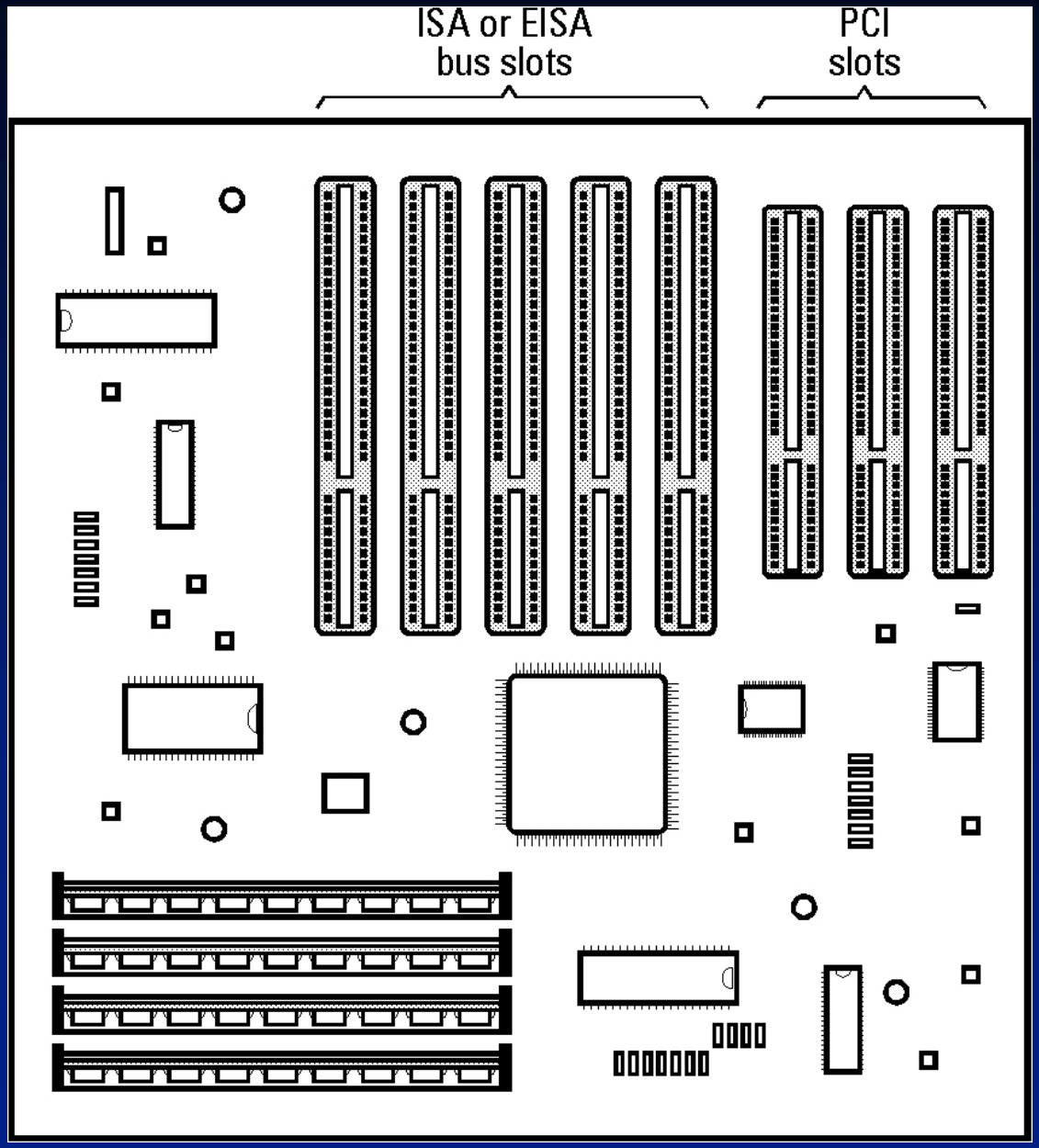
Arhitektura računara zasnovana na PCI magistrali



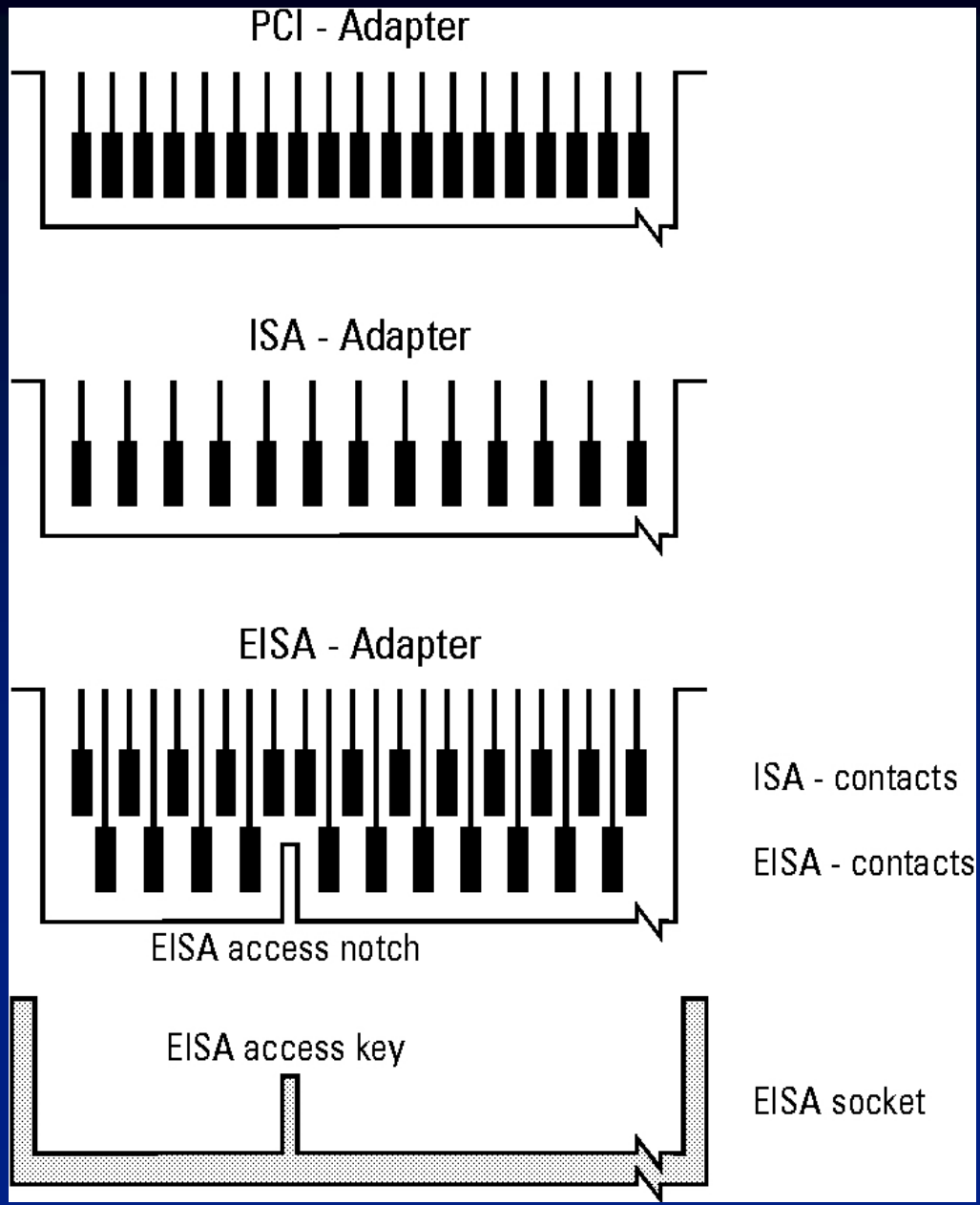
PCI slotovi za ploče 5 V i 3,3 V



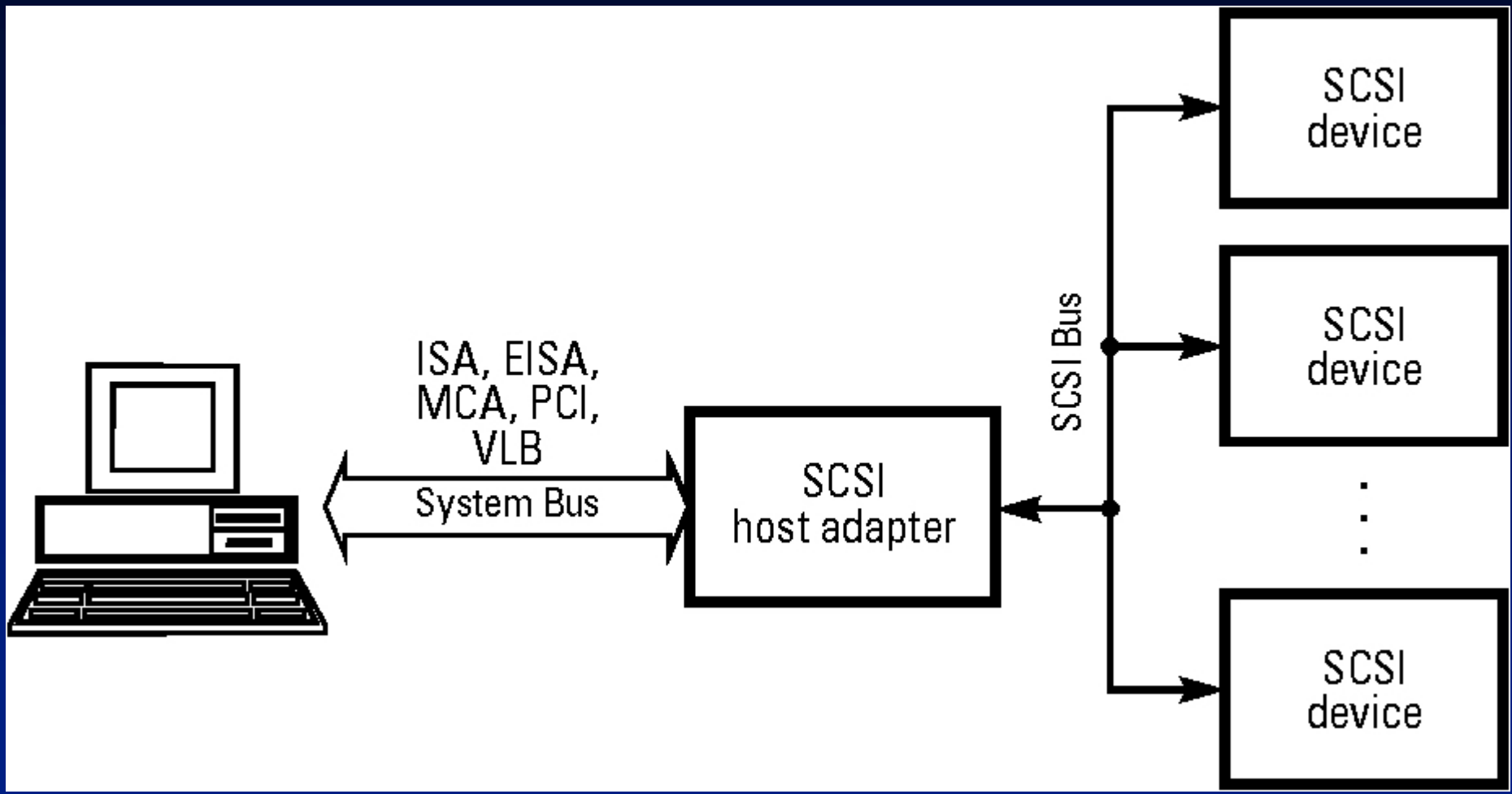
Fizička pozicija PCI, ISA/EISA slotova kod PC mašine



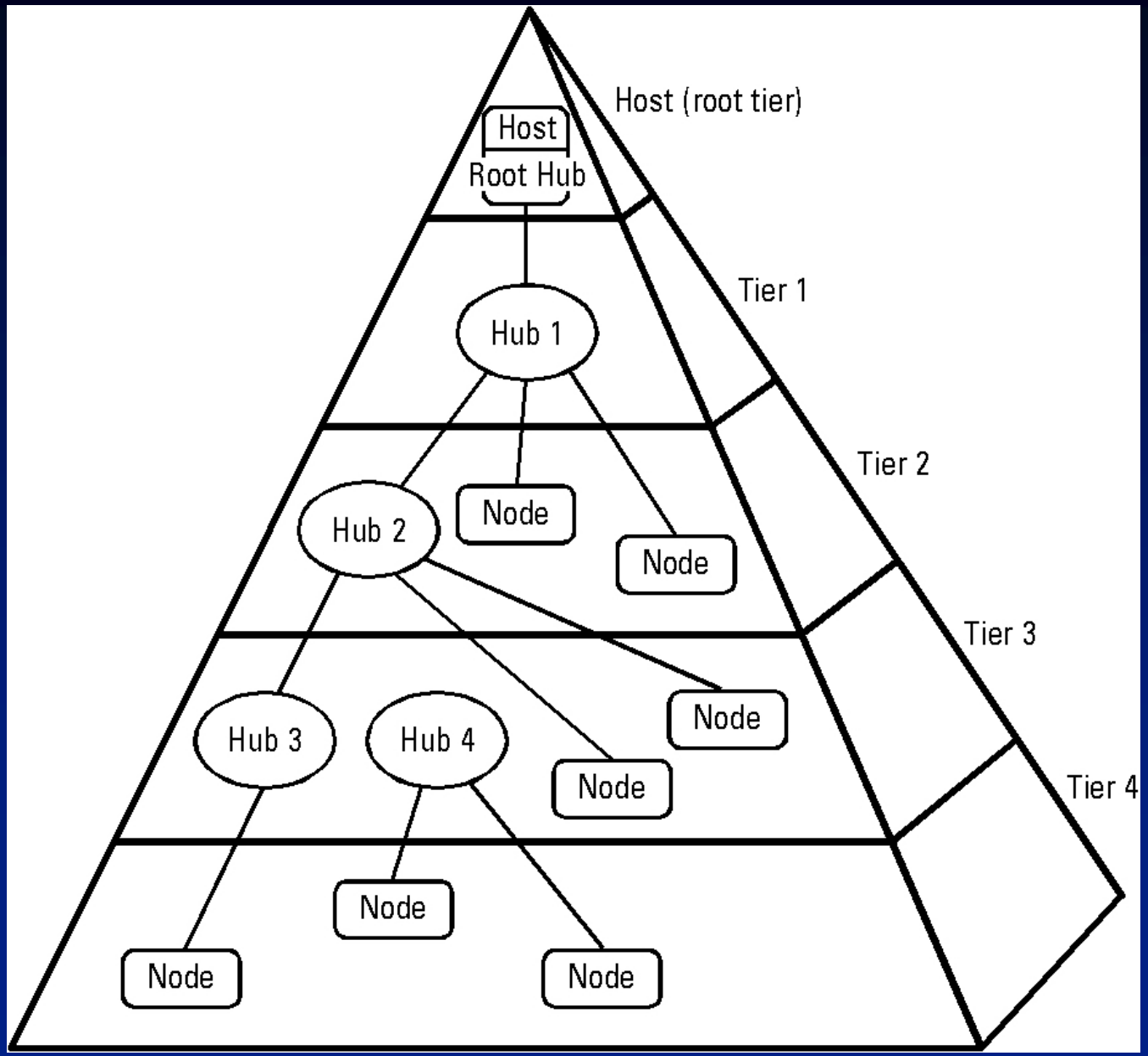
Izgledi PCI, ISA i EISA konektora



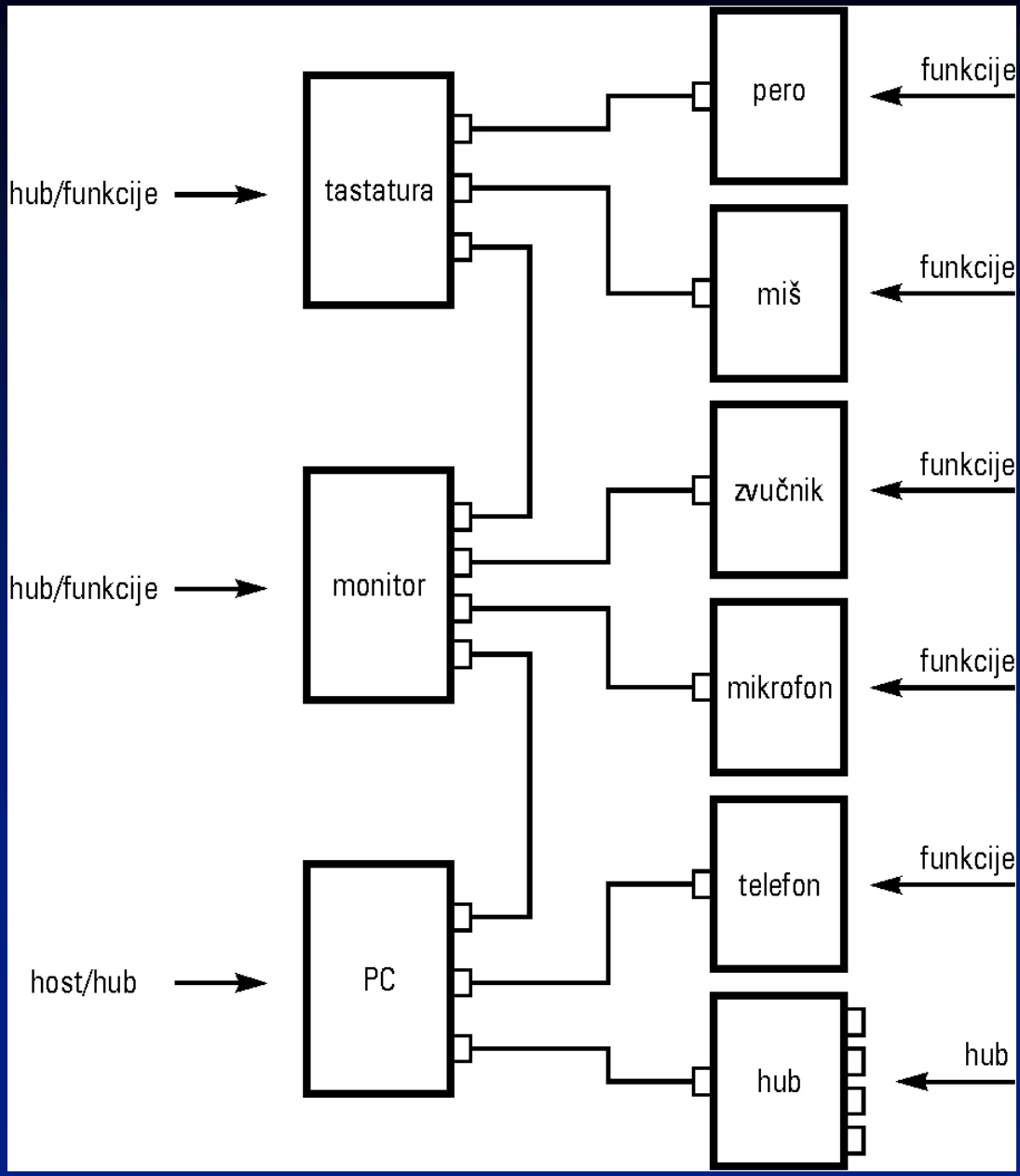
Sprezanje uredjaja tipa SCSI sa PC mašinom

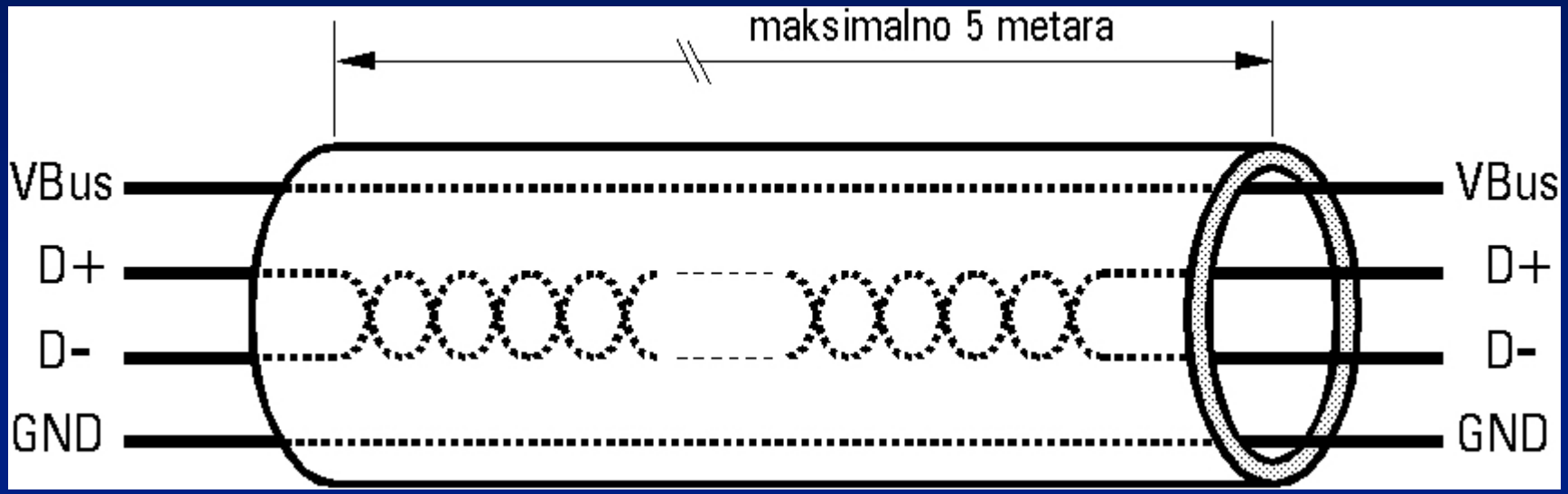


Povezivanje USB uredjaja preko Hub-ova



USB uredjaji - proširenje sistema



Format USB paketa i četvorožilni kabl kod USB-a

SYNC	PID	informacija	CRC