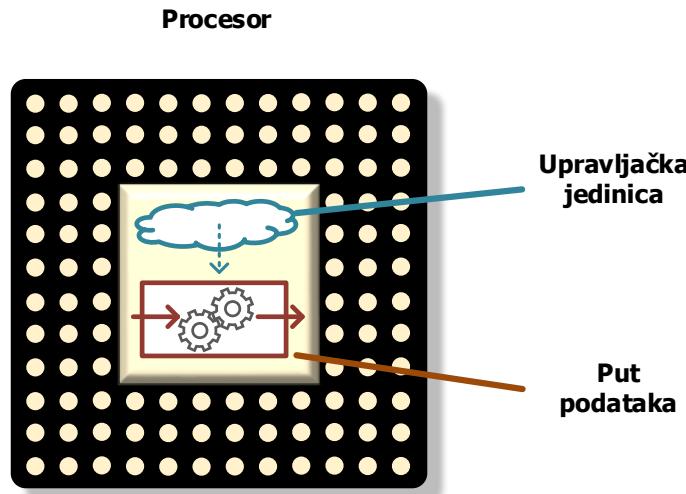


Arhitektura procesora FRISC

Sastavni dijelovi svakog računala

- Najvažniji dio računala je procesor
 - Procesor je aktivni dio računala koji upravlja radom računala i obavlja različite operacije nad podatcima u sustavu
 - Procesor se sastoji od puta podataka (datapath) i upravljačke jedinice
 - Procesor se naziva i CPU



Osnovni dijelovi računala

- Put podataka
 - Dio procesora koji:
 - obavlja određene operacije (npr. aritmetičke) nad podatcima
 - privremeno pamti podatke i međurezultate
 - prenosi podatke između dijelova za pamćenje i obradu
- Upravljačka jedinica
 - Dio procesora koji upravlja radom puta podataka, memorije i ulazno/izlaznih sklopova na temelju naredaba programa koji se izvodi

Osnovni dijelovi računala

- Memorija
 - Dio računalnog sustava u kojem se spremaju programi koji se izvode na procesoru te podatci potrebni za izvođenje tih programa
- Ulaz i Izlaz
 - Dijelovi računalnog sustava namijenjeni povezivanju s vanjskim svijetom
 - **Ulaz:** dio namijenjen primanju podataka iz vanjskog svijeta u računalo
 - **Izlaz:** dio namijenjen slanju podataka iz računala prema vanjskom svijetu

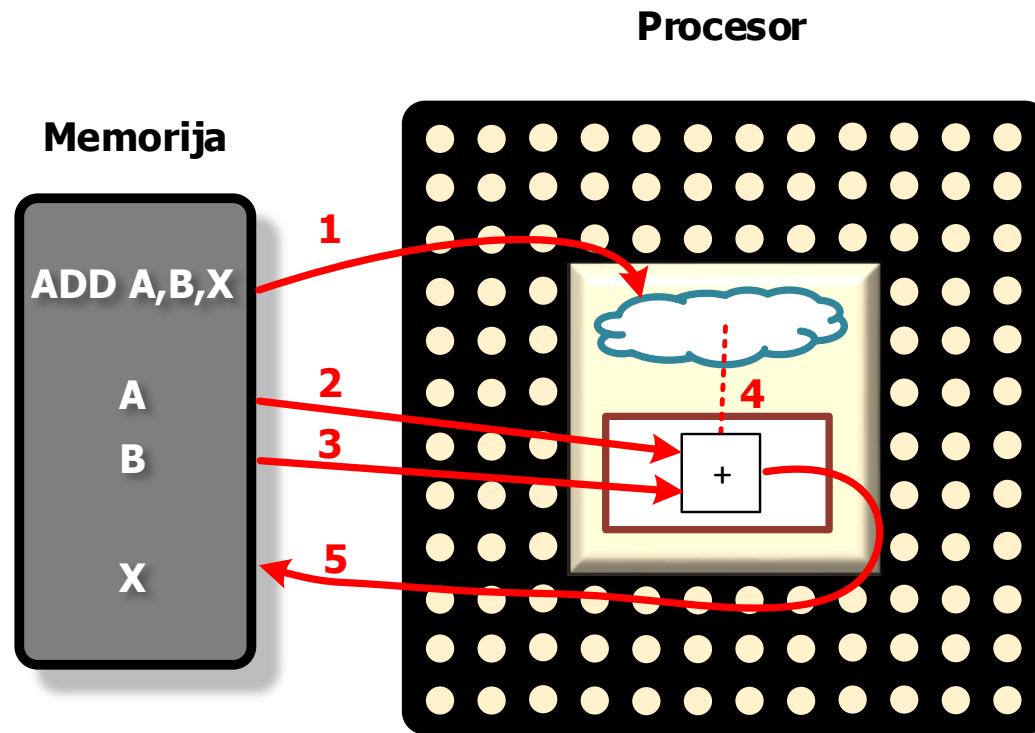
Osnovni način rada procesora

- **Osnovna funkcija procesora je izvođenje programa**
- Program se nalazi u memoriji i sastoji se od niza naredaba
 - svaka vrsta naredbi ima posebni oblik koji je razlikuje od drugih vrsta
- Procesor mora dohvaćati naredbe iz memorije, a dohvaćenu naredbu mora prepoznati da bi je mogao izvesti
- Dakle, rad procesora se odvija u tri osnovna koraka koji se stalno ponavljaju: >>>
 - dohvati naredbu (fetch)
 - dekodiranje ili prepoznavanje naredbe (decode)
 - izvođenje naredbe (execute)

Osnovni način rada procesora

- **Dohvat** naredbe uvijek se sastoji od operacije čitanja iz memorije
 - može biti više operacija čitanja, ako se naredba sastoji od više memorijskih riječi
- **Dekodiranje** naredbe odvija se internu unutar procesora
- **Izvođenje** se odvija na različite načine, ovisno o vrsti naredbe
 - može se odvijati internu, ako procesor u sebi ima sve podatke za izvođenje
 - može uključivati operacije čitanja iz memorije, ako se podatci za izvođenje naredbe nalaze u memoriji
 - može uključivati pisanja u memoriju, ako rezultate izvođenja naredbe treba spremiti u memoriju

Osnovni način rada procesora



Razine apstrakcije

- Koliko detaljno trebamo razmatrati naše računalo? Do koje dubine ići?
- Pri razmatranju kompleksnih sustava kao što je procesor vrlo često se koriste različite razine apstrakcije
- Razine apstrakcije omogućuju pojedinim sudionicima u procesu projektiranja ili korištenja da se mogu koncentrirati na njihov dio zadatka bez potrebe da brinu o detaljima koji im nisu potrebni

Razine apstrakcije

Primjeri nekih razina apstrakcije kod procesora su:

- Sistemska razina
 - Visoka razina apstrakcije koju koriste programeri aplikacija koji ne moraju znati mnogo o načinu kako procesor izvodi program već se koncentriraju na funkcionalnost algoritma i cjelokupne programske podrške.
- Arhitektura skupa naredaba (Instruction set architecture level)
 - Najčešće spominjana razina apstrakcije između razine programa i sklopolja.
 - Uključuje sve podatke o procesoru (registri, pristup memoriji, naredbe, pristup vanjskim sklopopovima i ostalo) koji su potrebni da bi se napisali programi u strojnom jeziku koji će se ispravno izvoditi.
 - Sa strane sklopolja ova razina opisuje funkcionalnost koju sklopolje treba omogućiti.

Razine apstrakcije

- Mikroarhitektura
 - Detaljan sklo povski opis arhitekture procesora. Opisuje načine povezivanja pojedinih dijelova procesora te signale potrebne za njihovo upravljanje.
- Razina logičkih vrata (gate level)
 - Potpun sklo povski opis procesora koji, između ostalog, često uključuje i podatke o svim vremenskim kašnjenjima signala unutar sklopa.
- Razina rasporeda (layout level)
 - Fizički opis svih sklopova procesora koji koristi definiranu tehnologiju izvedbe. Prikazuje sve detalje potrebne za preslikavanje ove razine u fizičko sklopovlje u postupku proizvodnje.

Razine apstrakcije

- Razine apstrakcije koje ćemo koristiti na ovom predmetu su:
 - Arhitektura skupa naredaba (Instruction set architecture - ISA)
 - Mikroarhitektura

Arhitektura računala s obzirom na memorijski pristup

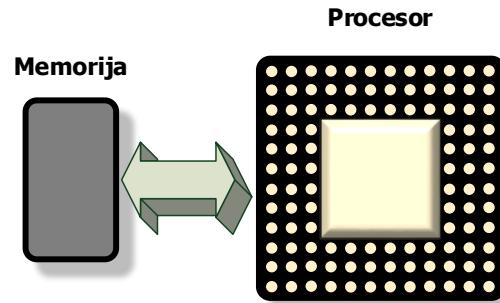
- Izvođenje zadatka na procesoru zahtijeva čitanje naredaba koje je zadao programer te čitanje i pisanje podataka koji se obrađuju (kao što smo vidjeli u jednostavnom primjeru ranije)
- Naredbe i podatci nalaze se u **memoriji**
- Da bi se obavila jednostavna operacija zbrajanja, kao u našem prethodnom primjeru, procesor mora više puta pristupiti memoriji:
 - dohvati naredbu ADD iz memorije
 - dohvati prvog operanda A
 - dohvati drugog operanda B
 - spremanje rezultata X

Arhitektura računala s obzirom na memorijski pristup

- Pri definiranju našeg procesora stoje nam na raspolaganju dvije arhitekture s obzirom na način dovođenja naredaba i podataka iz memorije u procesor:
 - von Neumannova arhitektura
 - Harvardska arhitektura

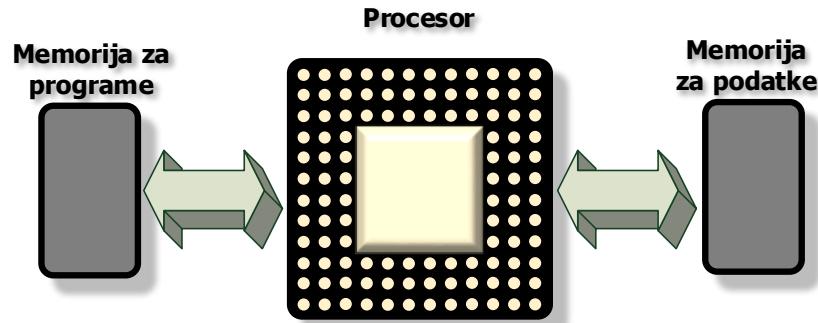
>>>

Von Neumannova arhitektura



- Značajka von Neumannove arhitekture je u tome da su **program i podatci** smješteni u **jedinstvenu memoriju** koja ima samo jednu vezu prema procesoru
- Jednostavna (i jeftina) arhitektura
- Nedostatak: Procesor ne može dohvatiti naredbu i podatke u istom trenutku => "**Von Neumannovo usko grlo**"
- Usko grlo predstavlja značajno ograničenje za brzinu obrade podataka kod računala koja jednostavnim operacijama obrađuju puno podataka, a trebaju biti efikasna

Harvardska arhitektura



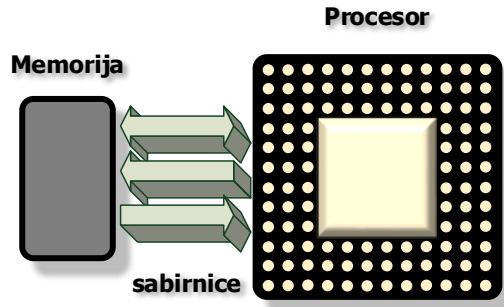
- Jednostavno rješenje von Neumannovog uskog grla je **razdvajanje memorije za pohranu programa od memorije za podatke**
- Procesor ima neovisne veze prema tim memorijama
- Ova arhitektura dozvoljava istovremeno dohvaćanje naredbe i jednog operanda čime se efikasnost znatno poboljšava*
- Skuplja od von Neumannove arhitekture te se koristi samo kad je potrebno povećanje performansi

Odluka: Arhitektura mem. pristupa

- S obzirom da želimo projektirati jednostavan sustav i da nam performanse sustava nisu u popisu primarnih zahtjeva, odlučujemo se za jednostavniju i jeftiniju arhitekturu memorijskog pristupa:

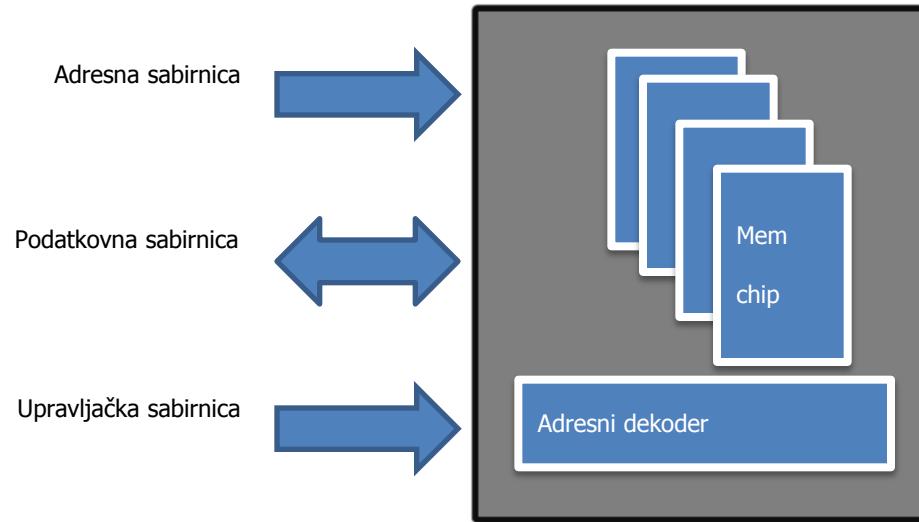
Von Neumannova arhitektura

Memorijski podsustav (osnovno)



- Memorijski podsustav načelno se spaja na procesor pomoću **sabirnica**
- Sabirnice su spojni putovi (vodovi) koji povezuju dijelove računala
- Adresna sabirnica – određuje mem. lokaciju kojoj se pristupa
- Podatkovna sabirnica – služi za prijenos podataka između procesora i memorije
- Upravljačka sabirnica – upravlja prijenosom podataka

Memorijski podsustav



- Organizacija memorijske riječi razlikuje se od procesora do procesora: jedna adresa može odgovarati memorijskoj lokaciji širine 8, 16, 32 ili više bita.

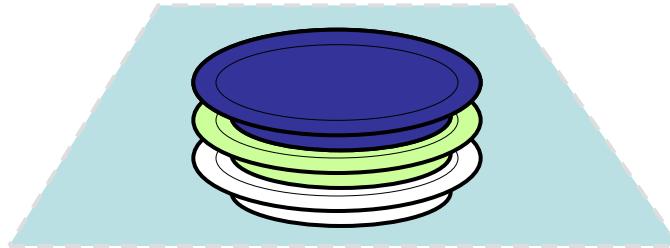
Povezivanje memorije i procesora

- Naš primjer povezivanja MEM-CPU:
 - Von Neumannova arhitektura: jedna (zajednička) memorija za naredbe i podatke
 - Veza prema CPU preko tri sabirnice: adresna, podatkovna i upravljačka
 - Adresna sabirnica: služi za izbor memorijske lokacije
 - Podatkovna sabirnica: prijenos podataka prema/iz memorije
 - Upravljačka sabirnica:
 - određivanje smjera toka podataka, tj. operacije čitanja ili pisanja - RD i WR (read, write)
 - Određivanje širine podatka – SIZE
- Koristeći gore navedeno možemo reći da smo definirali najjednostavniji sustav za pristup memoriji

Mala digresija: Stog

- Struktura podataka koja radi po načelu LIFO (engl. last in first out): zadnji spremljeni (stavljeni) podatak je prvi koji se čita (uzima)
- **Oprez: stog na razini arhitekture računala različit je od stoga na razini višeg programskog jezika (povezana lista i alokacija memorije) !!!**
- Ova struktura se može slikovito zamisliti kao niz tanjura složenih jedan na drugi: 

STAVI



UZMI

Stog

- S tanjurima je lako: uvijek znamo gdje stavljamo tanjur ili od kuda uzimamo tanjur
- Kako znati u koju memorijsku lokaciju treba stavljati ili iz koje uzimati podatak?
- Svaka lokacija se može adresirati, a adresa (položaj) vrha stoga (ToS=Top of Stack) pamti se u pokazivaču stoga (SP=Stack Pointer)
- SP nije ništa drugo, nego obični register u kojem se pamti adresa vrha stoga

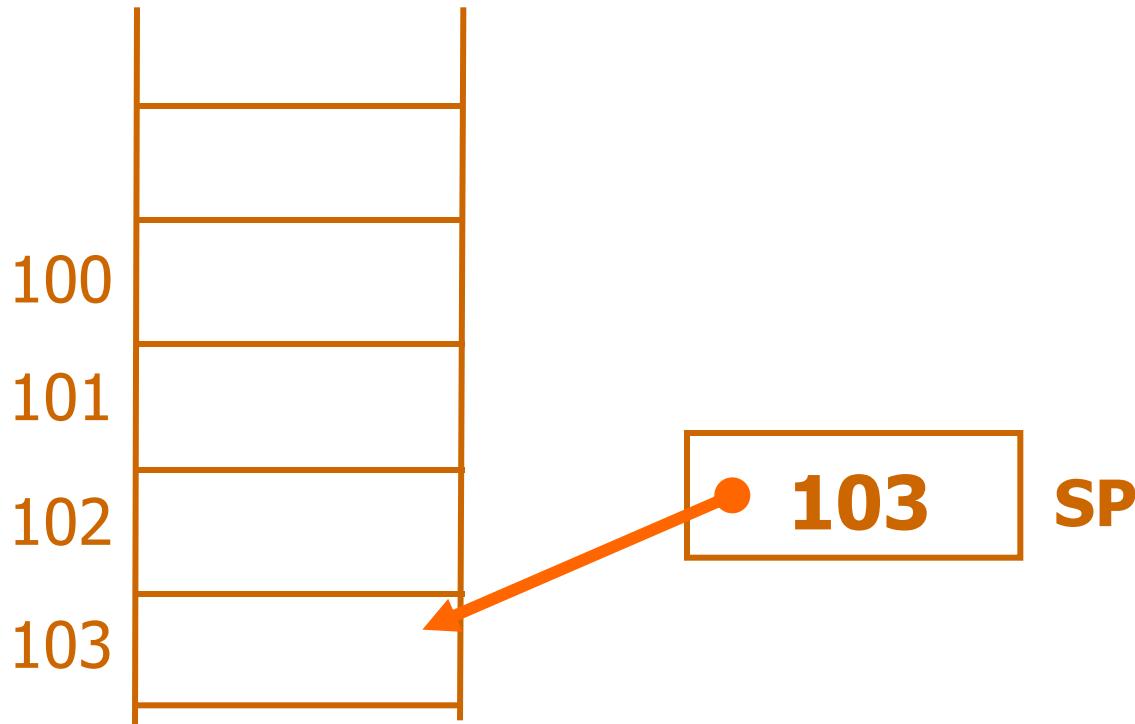
Stog

- Podatci se stavljaju na stog i uzimaju sa stoga pomoću posebnih naredaba:
 - PUSH (stavi podatak na vrh stoga) i
 - POP (uzmi podatak s vrha stoga)
- Ove naredbe koriste pokazivač stoga (SP) da bi znale odakle uzimaju (čitaju) ili gdje stavljaju (pišu) podatak
- Kod rada sa stogom uvijek moramo paziti da:
 - ne pokušamo staviti više podataka nego što ima mesta na stogu (da se stog ne "prepuni")
 - ne pokušamo uzimati podatke s praznog stoga (jer bi čitali podatke iz dijela memorije koja nije dio stoga)
 - ukratko: koliko se stavi na stog, toliko se treba uzeti

Stog

- Kako izgleda stog u memoriji računala?
- Određene memorijske lokacije koriste se kao područje za stog
- Registar SP se obično nalazi u procesoru

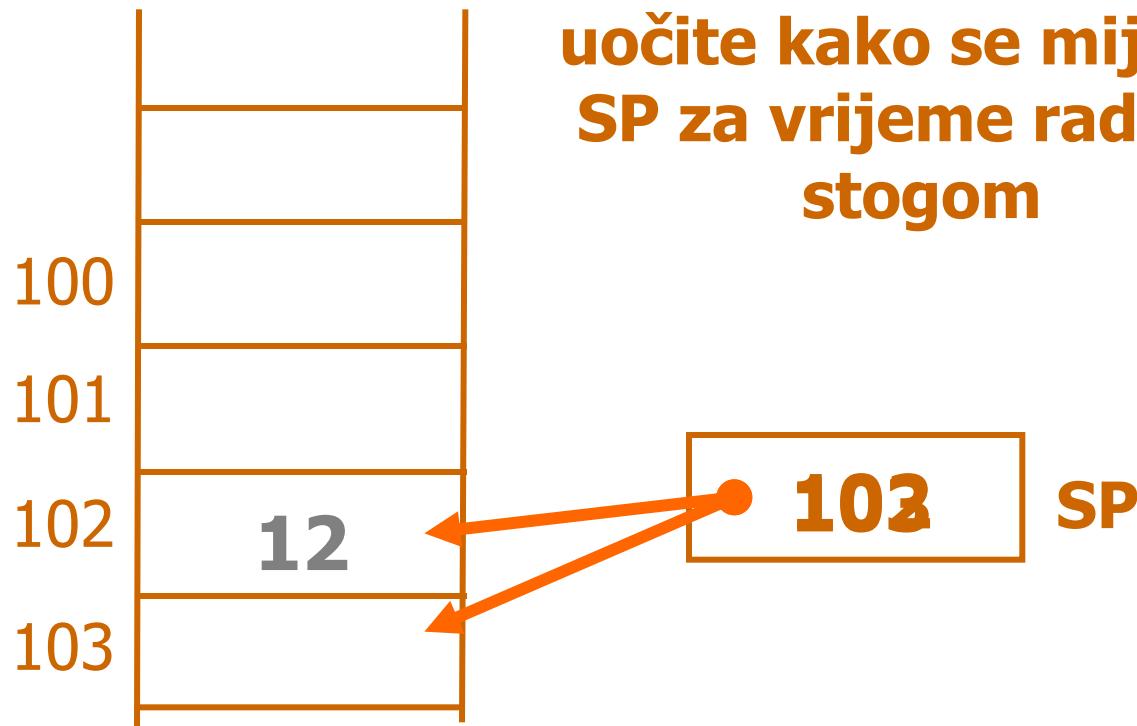
>>>



Stog

- Kako izgleda stog u memoriji računala?
- Određene memorijske lokacije koriste se kao područje za stog
- Registar SP se obično nalazi u procesoru

STAVI



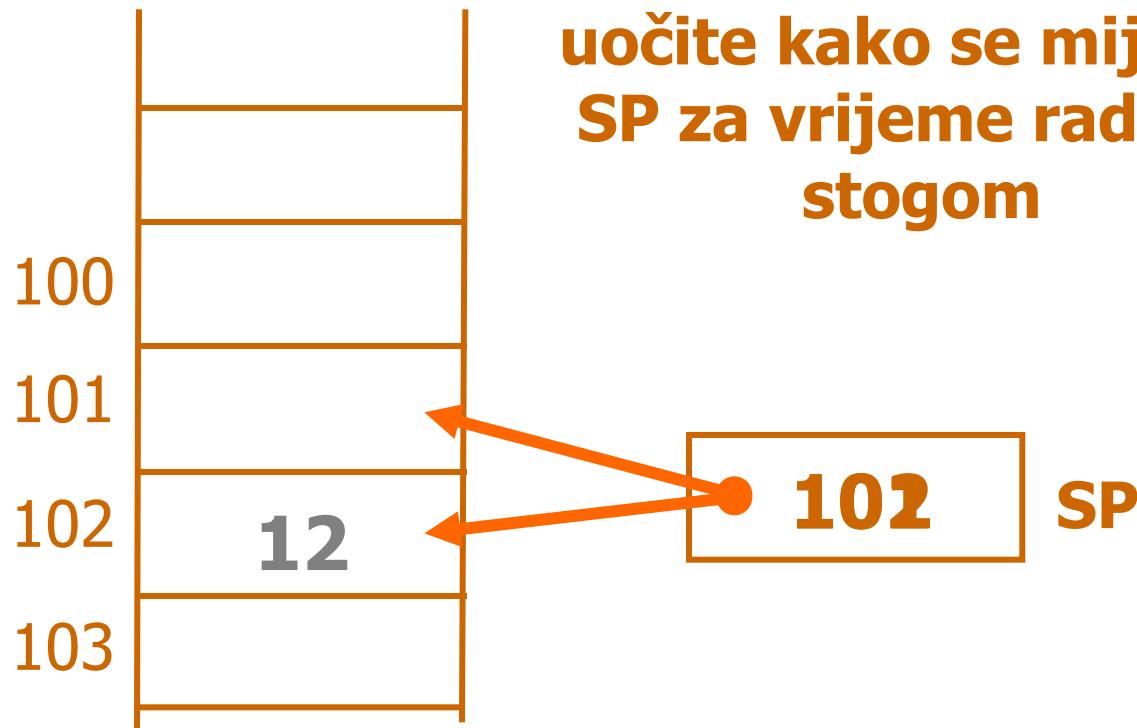
Stog

- Kako izgleda stog u memoriji računala?
- Određene memorijske lokacije koriste se kao područje za stog
- Registar SP se obično nalazi u procesoru

STAVI

**Prvo se SP
umanjuje za 1**

**uočite kako se mijenja
SP za vrijeme rada sa
stogom**

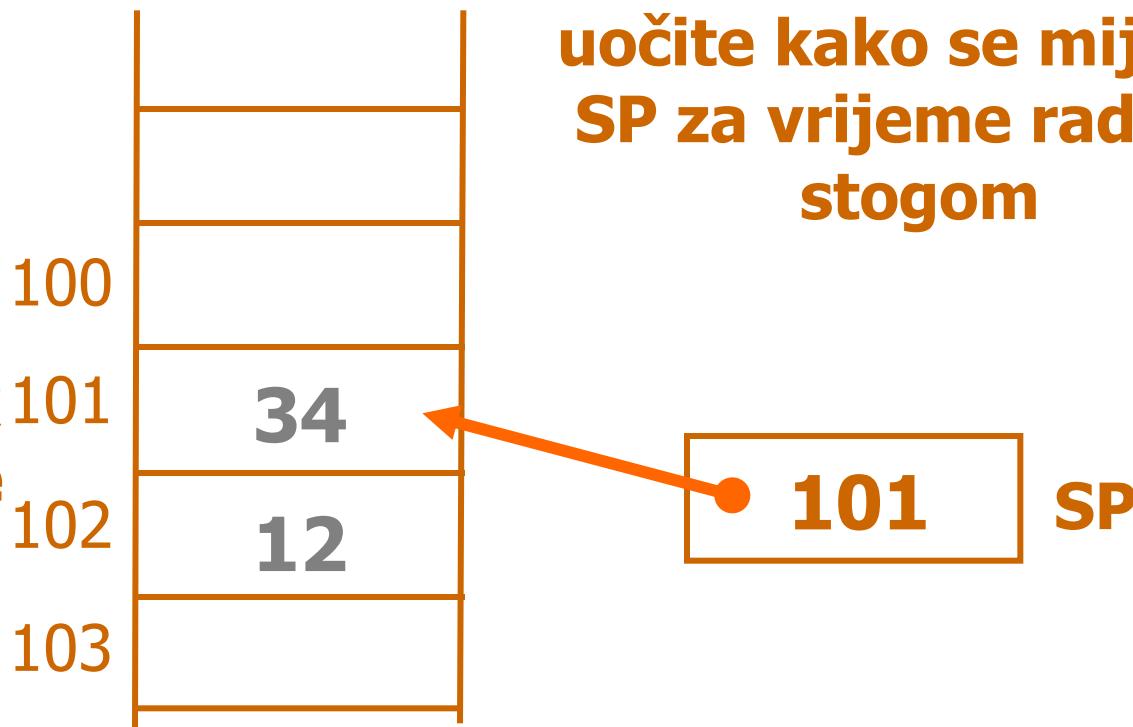


Stog

- Kako izgleda stog u memoriji računala?
- Određene memorijske lokacije koriste se kao područje za stog
- Registar SP se obično nalazi u procesoru

STAVI

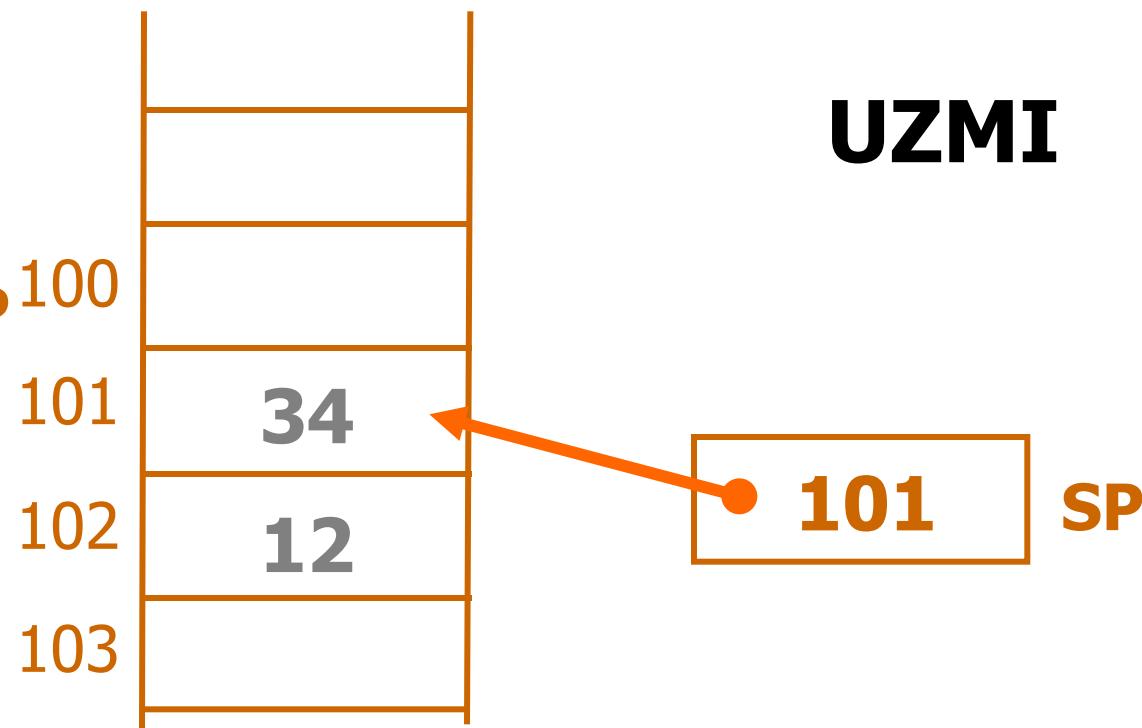
Zatim se podatak
stavlja tamo gdje
pokazuje SP



Stog

- Kako izgleda stog u memoriji računala?
- Određene memorijske lokacije koriste se kao područje za stog
- Registar SP se obično nalazi u procesoru

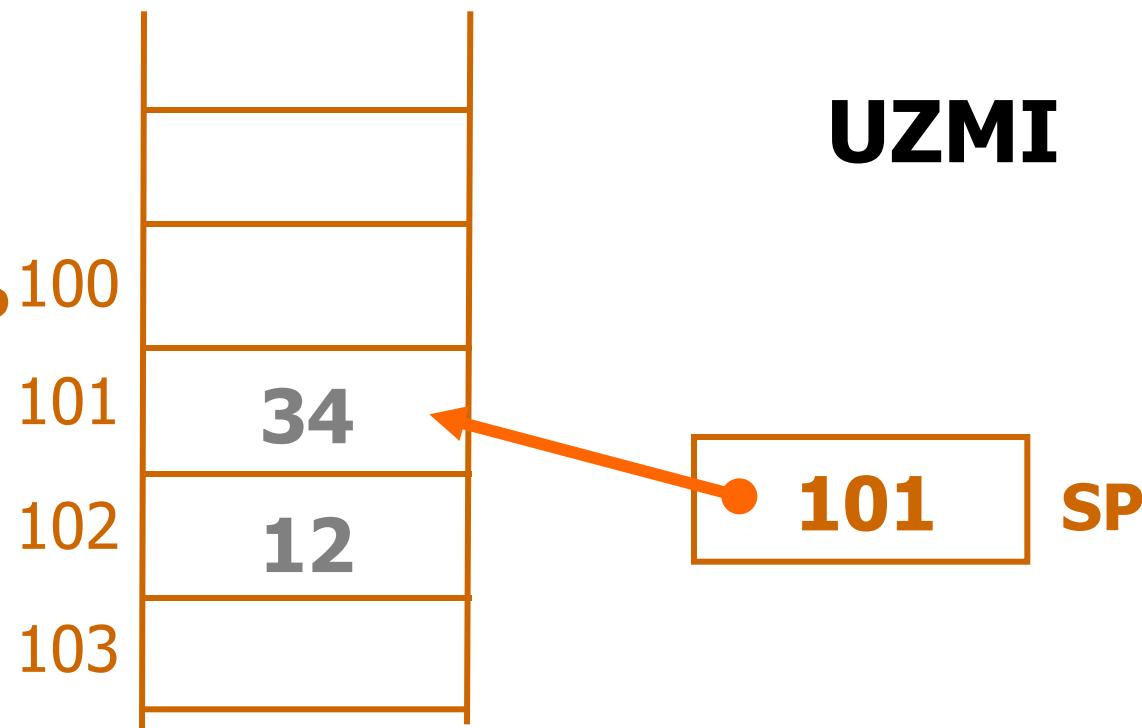
Kod uzimanja se prvo podatak uzima s mesta koje pokazuje SP



Stog

- Kako izgleda stog u memoriji računala?
- Određene memorijske lokacije koriste se kao područje za stog
- Registar SP se obično nalazi u procesoru

Kod uzimanja se prvo podatak uzima s mesta koje pokazuje SP

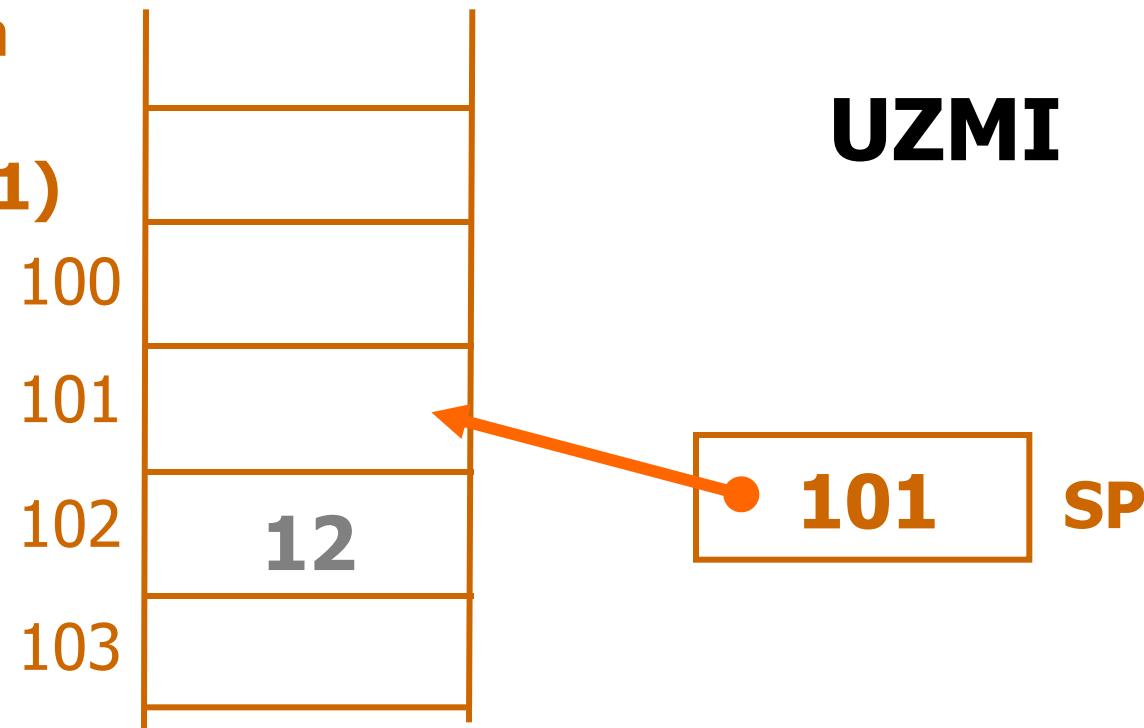


Stog

- Kako izgleda stog u memoriji računala?
- Određene memorijske lokacije koriste se kao područje za stog
- Registar SP se obično nalazi u procesoru

**Tek nakon toga
pomiče se SP
(povećava se za 1)**

UZMI

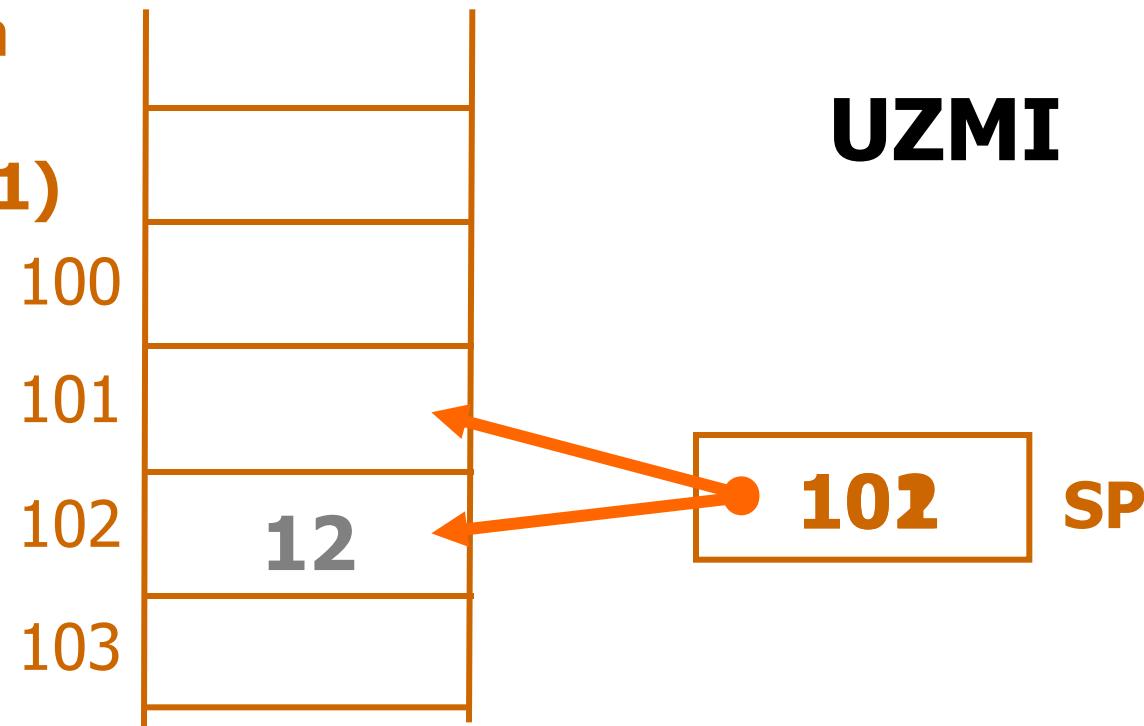


Stog

- Kako izgleda stog u memoriji računala?
- Određene memorijske lokacije koriste se kao područje za stog
- Registar SP se obično nalazi u procesoru

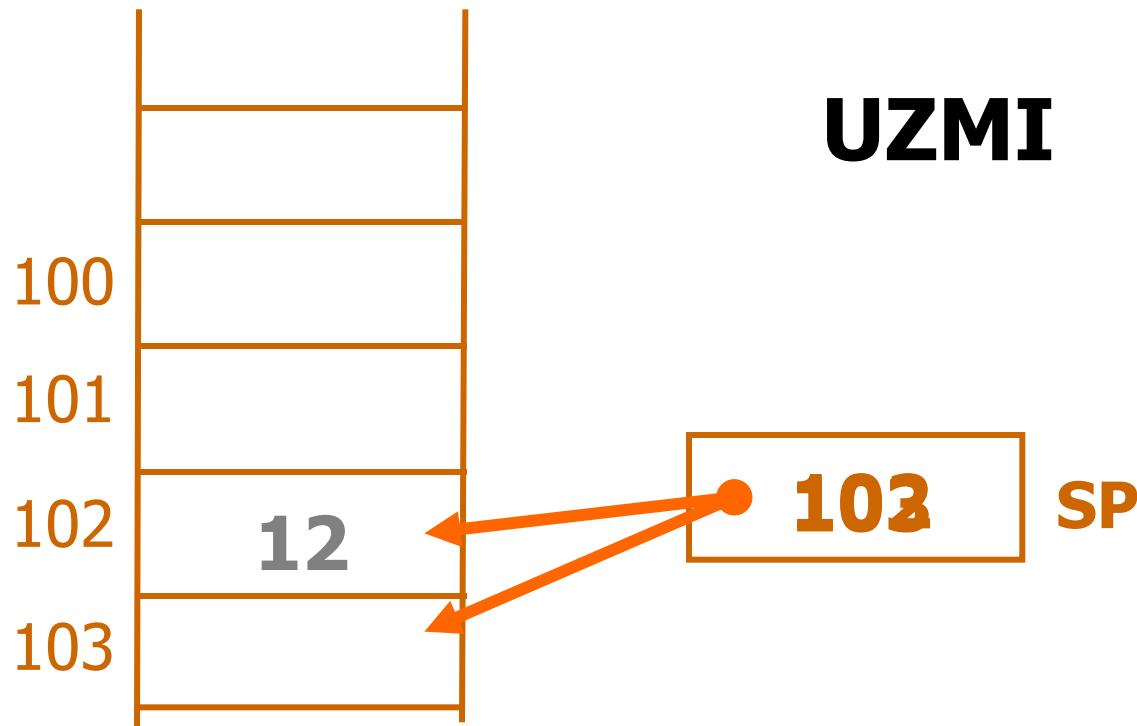
**Tek nakon toga
pomiče se SP
(povećava se za 1)**

UZMI



Stog

- Kako izgleda stog u memoriji računala?
- Određene memorijske lokacije koriste se kao područje za stog
- Registar SP se obično nalazi u procesoru



Stog

- Upravo smo vidjeli da se pokazivač stoga SP mijenja prilikom operacija sa stogom:
 - kod stavljanja podatka na stog → SP se smanjuje
 - kod uzimanja podatka sa stoga → SP se povećava *
- Vidjeli smo i redoslijed koraka prilikom operacija sa stogom:
 - stavljanje: umanji SP, zatim stavi podatak
 - uzimanje: uzmi podatak, zatim uvećaj SP **

* moguć je i obrnut smjer "rasta/pada" stoga, ali se on u praksi rijede koristi

** moguća je i drugačija realizacija stoga - više o tome kasnije

Stog

- Kod stavljanja na stog, treba u naredbi PUSH reći što se stavlja na stog (npr. neki broj ili sadržaj nekog registra)
- Kod uzimanja sa stoga, treba u naredbi POP reći gdje se upisuje pročitani podatak (npr. u neki registar)
- Važno je još napomenuti:
 - Pri uzimanju, tj. čitanju podatka, taj se podatak ne "uzima" doslovno sa stoga, tj. nakon čitanja se podatak ne briše sa stoga.
 - U stvarnosti, podatak koji je "uzet" ostaje zapisan na stogu, a samo je registar SP promijenio svoju vrijednost tako da pokazuje na prethodni podatak!!!

Arhitekture s obzirom na dohvat operanada

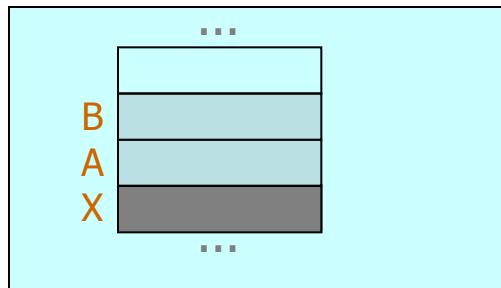
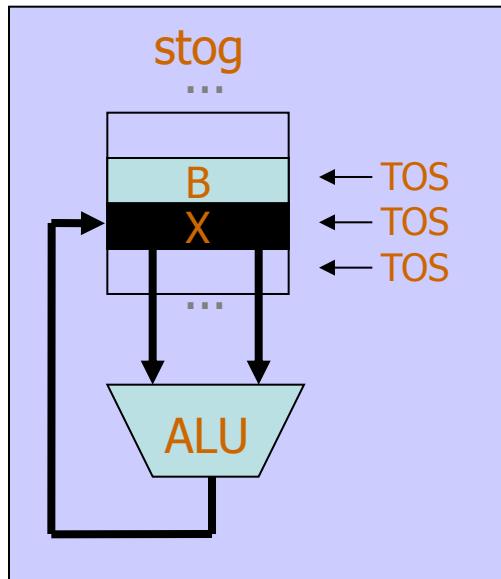
- Stogovne arhitekture
- Akumulatorske arhitekture
- Arhitekture register-memorija
- Arhitekture register-register (tzv. load-store)

Stogovna arhitektura

- Upotrebljava se stog za spremanje podataka koji se obrađuju (da bi se riješio ranije spomenuti problem dohvata podataka iz memorije)
- Ovaj stog **nalazi se u procesoru**, kao i pokazivač stoga!!!
- **Operandi su implicitno na vrhu stoga, gdje se smješta i rezultat**
- U naredbama za obradu podataka ne zadaje se eksplisitno gdje se nalaze operandi niti gdje se spremi rezultat
- Naredba za obradu podataka pristupa samo internom stogu

Stogovna arhitektura

Procesor



Memorija

- Primjer: računanje funkcije $x=a+b$

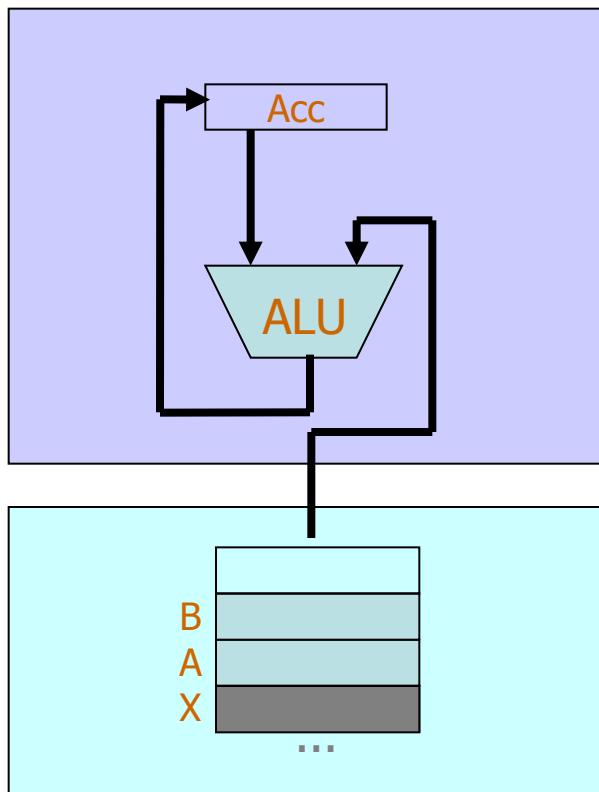
```
push A ; operand A iz memorije se stavlja na vrh stoga  
push B ; operand B iz memorije se stavlja na vrh stoga  
add      ; zbrajaju se operandi sa vrha stoga  
          ; i rezultat se stavlja na vrh stoga  
pop X   ; rezultat se sa vrha stoga sprema u memoriju
```

Stogovna arhitektura

- Stogovna arhitektura koristila se kod prvih procesora i danas se u svom osnovnom obliku više ne koristi
- Dobre strane stogovne arhitekture:
 - Naredbe su jednostavne i bez puno opcija što ih čini brzima za izvođenje
 - Izvedba upravljačke jedinice je jednostavna
 - Prevoditelji su jednostavnii
- Loše strane stogovne arhitekture:
 - Međurezultati se teško koriste
 - Prevođenje nije efikasno (jednostavna naredba višeg jezika se prevodi u dugačak niz naredaba strojnog jezika)
 - Veliki broj pristupa memoriji !!!

Akumulatorska arhitektura

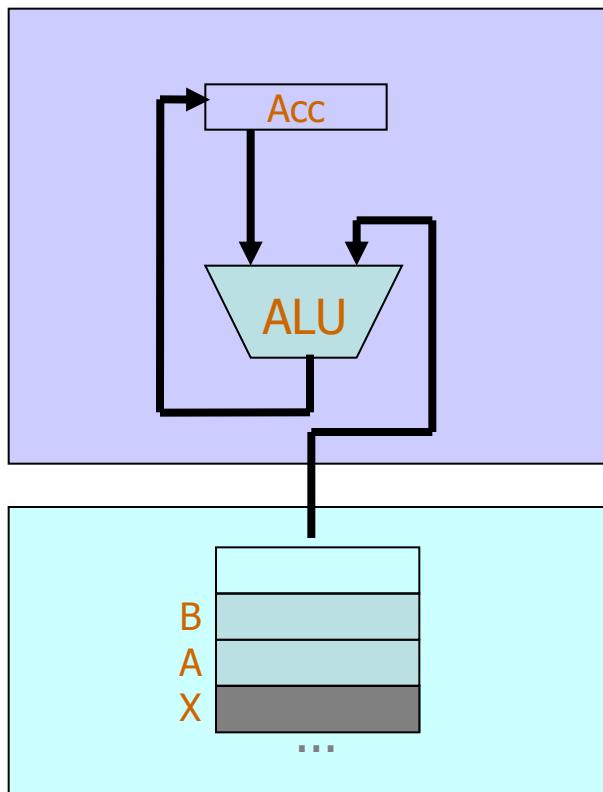
- Jedan operand je u vijek u posebnom registru koji se naziva Akumulator (Acc)



- Ako postoji, drugi operand se čita iz memorije
- Rezultat se spremi u Acc
- Naredba za obradu podataka pristupa memoriji

Akumulatorska arhitektura

- Primjer: računanje izraza
 $x=a+b$



load A ; operand A se iz memorije
; stavlja u Acc

add B ; zbraja se Acc sa operandom
; B iz memorije i rezultat
; se stavlja u Acc

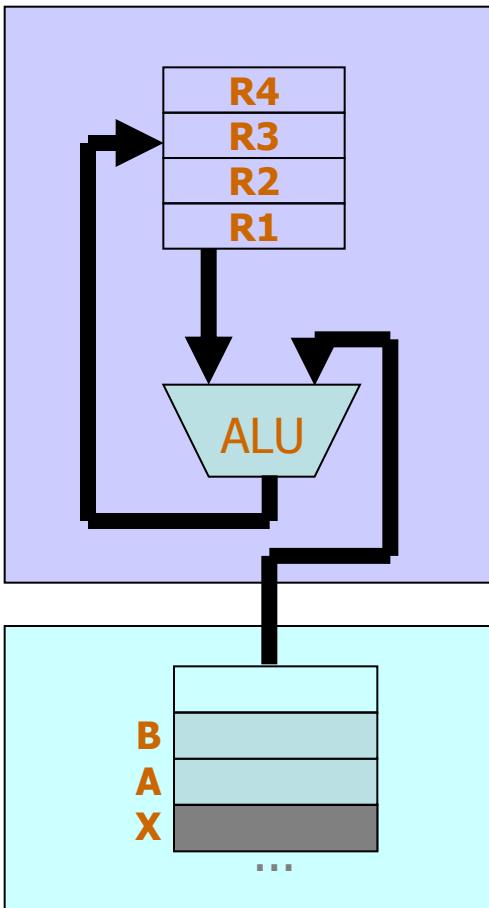
store X ; rezultat se iz Acc
; sprema u memoriju

Akumulatorska arhitektura

- Vrlo česta arhitektura prvih procesora, a danas se još može naći kod nekih jednostavnih mikrokontrolera
- Dobre strane ove arhitekture:
 - Jednostavnija za izvedbu od stogovne (Acc umjesto stoga)
 - Naredbe su jednostavne i bez puno opcija
 - Jedan operand je u memoriji (ne mora ga se dohvaćati dodatnom naredbom)
 - Prevoditelji su jednostavnii
- Loše strane ove arhitekture:
 - Međurezultati (osim zadnjeg) se ne mogu koristiti već sve mora biti pohranjeno u memoriju
 - Jako velik broj pristupa memoriji !!!
 - Prevođenje nije efikasno

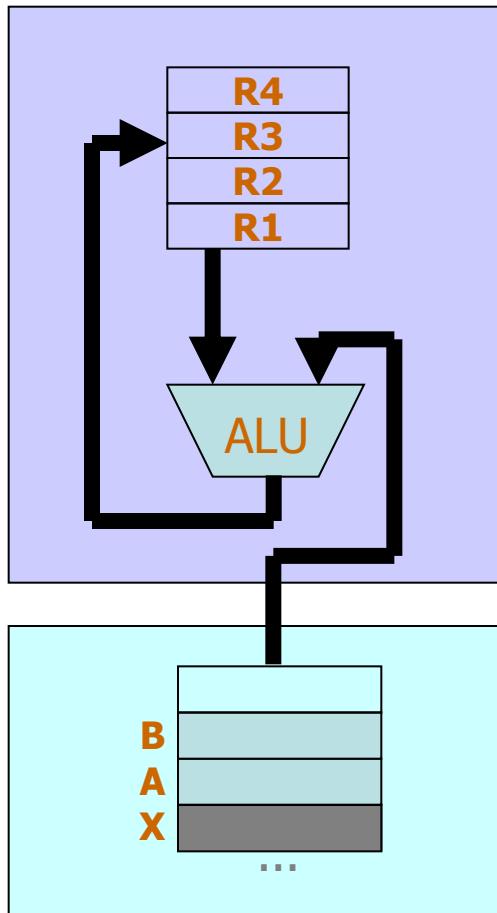
Arhitektura registar-memorija

- **Jedan operand se nalazi u skupu registara opće namjene** (registri koji se slobodno koriste i nemaju posebnu funkciju)
 - Kao i kod drugih arhitektura, uvijek može postojati jedan ili više registara specifične namjene, ali se oni ne ubrajaju u općenamjenske registre
- **Drugi operand se čita iz memorije**
- **Rezultat se sprema u neki od registara opće namjene**
- Naredba za obradu podataka pristupa memoriji



Arhitektura registar-memorija

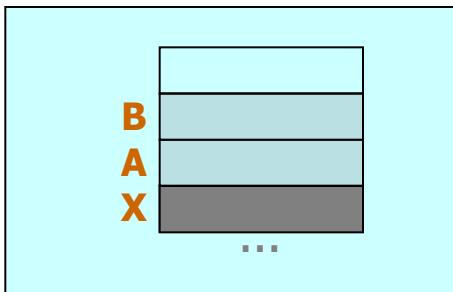
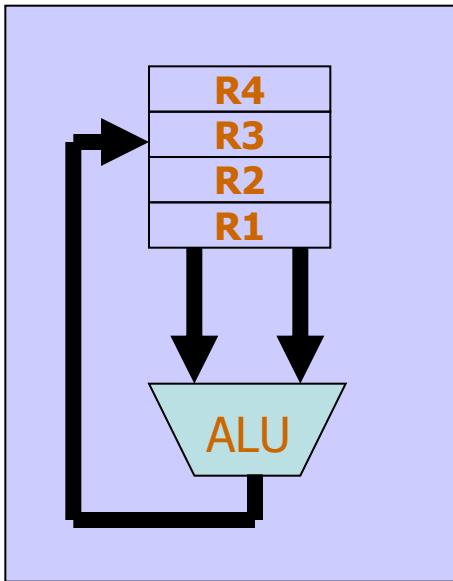
Primjer: računanje izraza $x=a+b$



```
load R1,A      ; operand A se iz memorije  
                ; stavlja u R1  
  
add R3,R1,B    ; zbraja se R1 sa operandom B iz  
                ; memorije i rezultat se stavlja  
                ; u R3  
  
store R3,X     ; rezultat se iz R3 sprema u  
                ; memoriju
```

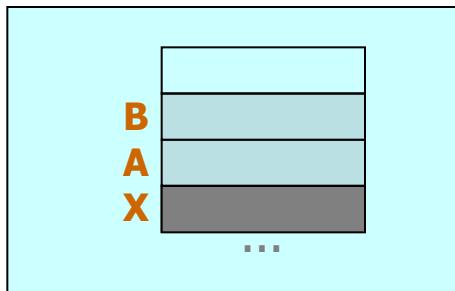
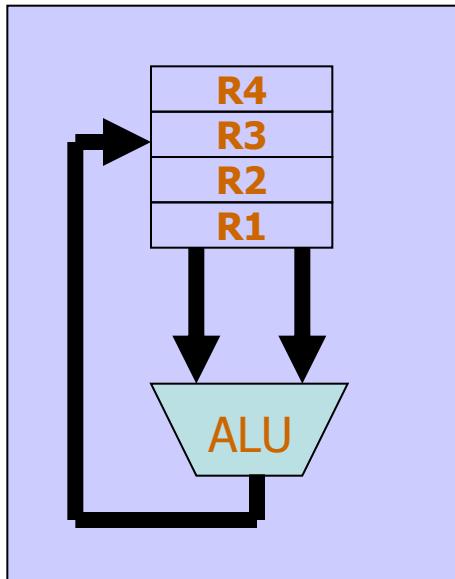
Arhitektura registar-registar (load-store)

- Oba operanda su u registrima opće namjene
- Rezultat se spremi u neki od registara opće namjene
- Naredba za obradu podataka pristupa isključivo općim registrima
- Podatci se mogu čitati iz memorije ili pisati u nju isključivo pomoću naredba LOAD i STORE



Arhitektura registar-registar (load-store)

- Primjer: računanje izraza $x=a+b$



load R1,A ; operand A se iz memorije
; stavlja u R1

load R2,B ; operand B se iz memorije
; stavlja u R2

add R3,R1,R2 ; zbraja se R1 sa R2
; i rezultat se stavlja u R3

store R3,X ; rezultat se iz R3 sprema u
; memoriju

Usporedba prethodnih arhitektura

- Na temelju prethodnih objašnjenja vidljive su neke prednosti i nedostaci pojedinih arhitektura:
- Stogovna i akumulatorska arhitektura bile su često korištene u prvim procesorima dok se danas gotovo ne koriste
- Neke ideje revitalizacije stogovne arhitekture postoje kod procesora koji izvode Java bytecode
 - npr. SUN picoJavaII: kombinacija dobrih osobina stogovne arhitekture (cirkularni stog) i regstarskih arhitektura (operacije s podatcima koji su bilo gdje na stogu)
- Osim iznimaka, većina današnjih procesora ima regstarsku arhitekturu (reg-mem ili reg-reg) među kojima su više zastupljene reg-reg (load-store) arhitekture

Usporedba prethodnih arhitektura

- Karakteristike regalarskih arhitektura su:
 - Brži pristup operandima
 - Varijable se mogu čuvati u registrima opće namjene (što je više registara to je manje potrebno komunicirati s memorijom)
 - Prevoditelji efikasnije prevode programe korištenjem registara
 - Naredbe veće i sporije

Reg-mem arhitekture

- + Jednostavan pristup podatcima u memoriji
- Dodatan pristup memoriji pri izvođenju naredaba, vrijeme izvođenja varira

Load-store arhitekture

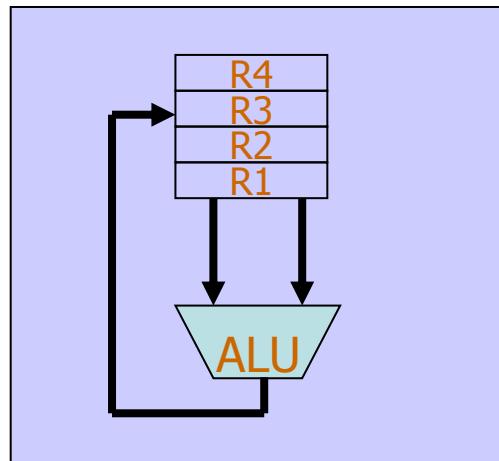
- + Brzo izvođenje, jednostavan format naredaba, jednostavno generiranje kôda, jednostavnost protočne strukture, uniformno vrijeme izvođenja
- Veći broj naredaba u programu zbog zasebnih učitavanja podataka iz mem.

Primjeri navedenih arhitektura

- Stogovna:
 - WISC CPU/16, MISC M17, picoJava
- Akumulatorska
 - EDSAC
- Registarsko-memorijska
 - Motorola 68000 i Intel 80x86 (imaju karakteristike reg-mem i reg-reg)
- Registarsko-registerska (load-store)
 - ARM, MIPS, SPARC

Odluka: Arhitektura smještaja operanada

- Od nabrojenih arhitektura trenutno je najefikasnija i u svijetu dominantna tzv. LOAD-STORE arhitektura (u području ugradbenih računala)
- Dodatna pogodnost je da se ta arhitektura može i najjednostavnije projektirati te sklopopovski izvesti
- Iz tog razloga za naš procesor izabiremo LOAD-STORE arhitekturu



Arhitekture s obzirom na skup naredaba

- S obzirom na skup naredaba procesora razvijene su dvije arhitekture:
 - CISC (Complex Instruction Set Computer)
 - RISC (Reduced Instruction Set Computer) >>>
- U današnje vrijeme komercijalni procesori nemaju više čistu arhitekturu CISC ili RISC:
 - obično u određenom procesoru prevladava jedna arhitektura, ali...
 - često se uključuju pojedina svojstva druge arhitekture
- Pogledajmo karakteristike CISC i RISC arhitektura...

- U samom početku procesori su bili vrlo jednostavni, ali su tehnološki vrlo brzo napredovali...
- Uskoro je glavni trend u oblikovanju arhitekture procesora bilo uvođenje procesorskih naredaba bliskih naredbama viših programskega jezika, npr.:
 - umanji registar za 1 i skoči na početak petlje ako je registar veći od nule
 - pomnoži matrice u memoriji
 - pronađi određeni podatak u bloku memorije
- Prednosti takvih procesorskih naredaba bile su:
 - Jednostavnije prevođenje programa iz viših programskega jezika
 - Ušteda memorije zbog manjeg broja naredaba (važno u to doba!!!)
 - Ubrzanje rada zbog manjeg broja dohvata naredaba iz memorije
- Procesori s takvim skupom naredaba nazivaju se CISC procesori

- Karakteristike CISC procesora
 - Velik broj naredaba i njihovih inačica
 - Velik broj načina adresiranja (više o tome kasnije)
 - Većinom su se naredbe unutar procesora izvodile korištenjem načela mikroprograma, tj. kompleksne naredbe izvodile su se u nizu ciklusa tijekom kojih je procesor izvodio niz jednostavnijih operacija (više o tome ćemo govoriti kasnije)
 - Registri imaju posebne namjene (brojači za petlje, za adresiranje, za podatke itd.)
 - Problem kompleksnih naredaba rješava se "unutar procesora" (možemo reći "sklo povski")
 - Skupo projektiranje i visoka cijena
- Primjeri CISC procesora: Intel 80x86, Motorola 68000

- 70tih i početkom 80tih dominaciju na tržištu imali su 8-bitni CISC procesori
- Međutim, kompleksne naredbe zahtjevaju kompleksnu logiku za dekodiranje (sporo dekodiranje i izvođenje) i izuzetno skup i dugotrajan postupak projektiranja takvih procesora
- Početkom 80-tih, u okviru tri gotovo usporedna istraživačka projekta (IBM 801, Berkeley RISC i Stanford MIPS) razvijena je potpuno nova arhitektura procesora zasnovana na jednostavnim instrukcijama koje se mogu izvoditi velikom brzinom
- Procesori s takvim skupom naredaba nazivaju se RISC (Reduced Instruction Set Computer)
- RISC procesor razvijen na sveučilištu Berkeley imao je izuzetne performanse u usporedbi s komercijalnim CISC-procesorima uz znatno jednostavniju i jeftiniju sklopošku izvedbu.

- Primjeri jednostavnih "RISC-naredaba"
 - učitaj operand iz memorije u registar
 - zbroji dva podatka iz registra
 - spremi sadržaj registra u memoriju
 - umanji sadržaj registra za 1
 - skoči na neku naredbu (npr.) početak petlje ako je zastavica ZERO=1
- Umjesto jedne kompleksne "CISC-naredbe" može se napisati niz jednostavnijih "RISC-naredaba"
 - jednostavnije naredbe se puno brže izvode pa je rezultat ubrzanje izvođenja programa bez obzira na veći broj naredaba

- Karakteristike RISC procesora
 - Relativno malen skup jednostavnih naredaba, manji broj inačica svake naredbe
 - Mali broj načina adresiranja
 - Pojedina naredba brzo se izvodi
 - Velik broj ravnopravnih registara opće namjene unutar procesora
 - Problem kompleksnih naredaba rješava se izvan procesora (možemo reći "programski")
 - Korištenje protočne strukture za ubrzanje rada (više o tome kasnije)
 - Relativno jeftino projektiranje i niska cijena
- Primjeri RISC procesora: MIPS, ARM, SPARC

Izbor RISC-CISC

- Zbog očitih prednosti na cjelokupnom tržištu ugradbenih uređaja koji u sebi sadrže procesor, RISC procesori danas dominiraju
- Mi ćemo zbog toga, a i zbog jednostavnosti arhitekture koju ćemo opisivati u okviru ovih predavanja, također odabrati da naš procesor bude tipa RISC
- **Prema ovoj vrsti arhitekture dat ćemo i ime našem procesoru. Procesor ćemo zvati FRISC što je kratica od FER RISC**

Odluka: Broj registara

- S obzirom da smo izabrali load-store arhitekturu moramo definirati koliko registara želimo u našem procesoru
- S obzirom da će se izbor pojedinog регистра obavljati postavljanjem određenih bitova unutar naredbe onda je efikasno da broj registara bude potencija broja 2
- Većina današnjih procesora ima 8 ili 16 registara opće namjene

Odluka: Broj registara

- Radi jednostavnosti arhitekture i što jednostavnijeg oblika naredbe izabiremo da naš procesor ima 8 registara opće namjene
- Nazovimo registre: R0, R1, R2, R3, R4, R5, R6 i R7

Odluka: Širina registara (i sab. pod)

- Širinu registara (u bitovima) određuje statistika najčešće korištenih podataka u programima koje želimo izvoditi na procesoru. Na primjer:
 - Byte 8b
 - Short 16b
 - Int 32b
 - Long 64b
 - Float 32b
 - Double 64b
- Daleko najčešće korišteni tipovi podataka u općim primjenama su 32-bitni int i float
- Na temelju toga možemo izabrati **32b kao optimalnu širinu registara**
- S obzirom da smo za registre opće namjene izabrali širinu od 32b, tada smo implicitno definirali i širinu interne sabirnice podataka kao i širinu ulaza i izlaza aritmetičko-logičke jedinice

Odluka: Širina sabirnica

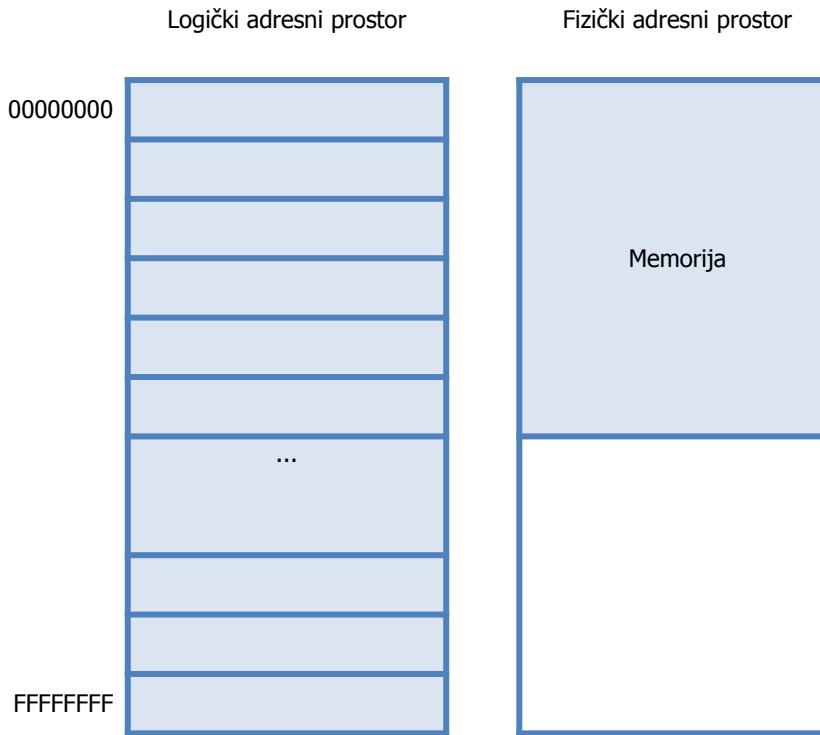
- S obzirom da smo izabrali 32b širinu registara opće namjene, tada je normalno da je i širina **sabirnice podataka 32b ***
- Širinu memorijske riječi odabiremo da bude jedan bajt (8b), ali zbog veće efikasnosti ćemo moći pročitati odjednom 4 bajta (32b) što nam dozvoljava širina podatkovne sabirnice

* Postoje procesori koji imaju različitu širinu unutarnjih i vanjskih sabirnica, no u to nećemo ulaziti u okviru ovog predmeta (razlog može biti npr. želja za što manjim brojem priključaka-pinova na procesoru).

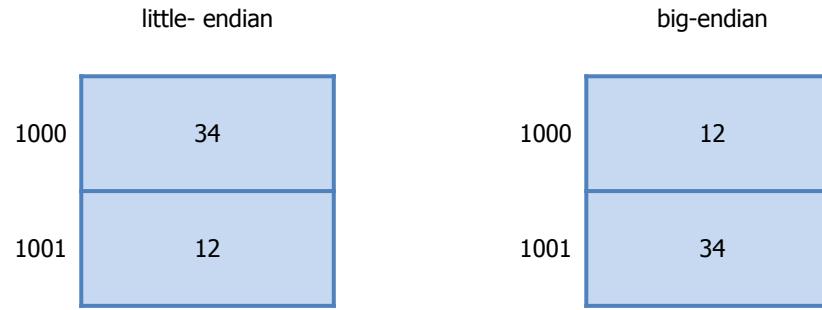
Odluka: Širina sabirnice

- Širina adresne sabirnice određena je maksimalnim željenim prostorom memorije
- Iako će naši programi biti vrlo kratki, radi jednostavnosti i uniformnosti arhitekture definirat ćemo da je i **adresna sabirnica 32b**
- **Svaka adresa odgovara jednoj memorijskoj riječi, tj. jednom bajtu**

Adresni prostor

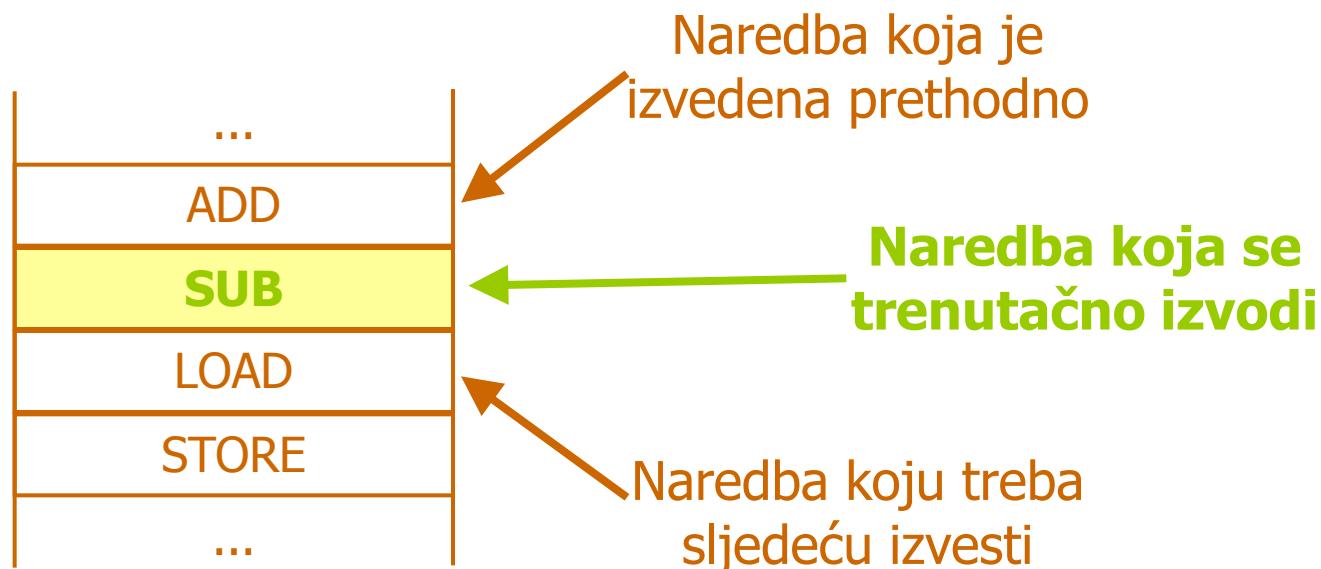


Redoslijed zapisa podataka u mem.



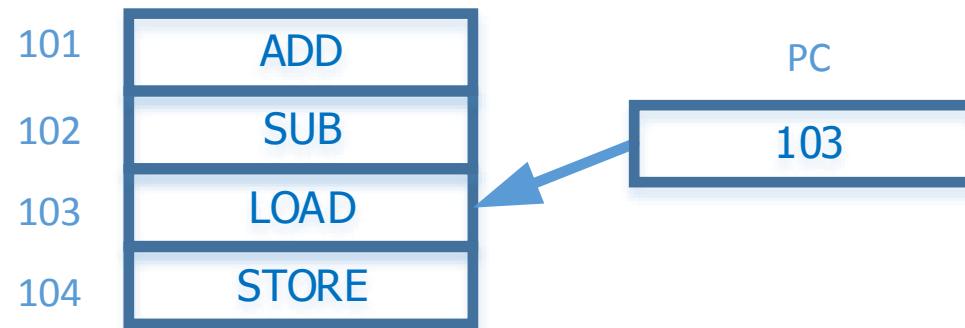
Programsko brojilo

- Spomenuli smo da procesor dohvaća naredbe iz memorije i izvodi ih
- Naredbe su u memoriji smještene slijedno jedna iza druge i tim redoslijedom se dohvaćaju i izvode (izuzetak su naredbe skoka)



Programsko brojilo

- To znači da procesor u svakom trenutku mora "zнати" adresu naredbe koju treba dohvatiti i izvesti
 - Kako procesor to "zna"?
 - Jednostavno: procesor ima jedan registar koji služi samo toj svrsi: PC (program counter) ili programsko brojilo (iako se tu zapravo ništa ne prebraja)



Programsko brojilo

- Naš register PC bit će širok 32 bita, jer smo za adresnu sabirnicu odabrali istu širinu, a logično je da se registrom PC može adresirati cijeli memoriski prostor
- Register PC se automatski uvećava (nakon dohvata svake naredbe) tako da pokazuje na sljedeću naredbu u memoriji
 - za tu svrhu, PC ima posebne sklopove za uvećavanje

Odluke: Rekapitulacija građe

- Osam 32-bitnih registara R0, R1, R2, R3, R4, R5, R6, R7
- 32-bitno programsko brojilo - registar PC
- Širine internih sabirnica i ALU su 32 bita
- Širina podatkovne sabirnice je 32 bita
- Širina adresne sabirnice je 32 bita
- Širina memorijske lokacije je 8 bita, ali se može odjednom čitati/pisati 32-bitni podatak

Odabir skupa naredaba

Strojni kod naredbe

- Pomoću tog kôda procesor razlikuje naredbe
- RISC – strojni kôd uobičajeno je širine riječi

Polje operacijskog koda	Polje prvog operanda	Polje drugog operanda	Polje trećeg operanda
-------------------------	----------------------	-----------------------	-----------------------

Polje operacijskog koda	Polje adrese (1.dio)	Polje prvog operanda	Polje adrese (2.dio)
-------------------------	----------------------	----------------------	----------------------

Širina strojnog koda naredbe - CISC

ADD R0,R1,R2

operacijski kôd {ADD}	1. operand {R0}	2. operand {R1}	3. operand {R2}
--------------------------	--------------------	--------------------	--------------------

JP_uvjet 200

operacijski kôd {naredba skoka}	Polje uvjeta {uvjet}	...	Proširenje op.koda {JP}
Adresa {200}			

ADD R0, (200),(R2+6)

operacijski kôd {ALU naredba}	1. operand {R0}	Način adresiranja {apsolutno i reg.ind. s pomakom}
Adresa {200}		
Proširenje op.koda {ADD}	3. operand {R2}	Adresni pomak {6}

Strojni kôd

Treba strogo razlikovati:

- **naredbu zapisanu tekstom** kao npr. "ADD R1,R2,R3" što je samo način kako programer piše naredbe u nekom programu za upis teksta prilikom programiranja
- **strojni kôd naredbe** što je zapis naredbe u obliku niza nula i jedinica u memoriji računala

Odabir skupa naredaba

- Jedna od najvažnijih odluka u projektiranju procesora je odabir skupa naredaba (instruction set)
- Postoji više vrsta naredaba, ovisno o procesoru, na primjer:
 - Aritmetičko-logičke naredbe obavljaju AL operacije
 - Registarske naredbe premještaju podatke između registara
 - Memorejske naredbe čitaju i spremaju podatke u/iz memorije
 - Naredbe za premještanje podataka mogu premještati podatak između registara, memorijskih lokacija i/ili puniti brojeve u registre i memorejske lokacije
 - Upravljačke naredbe omogućuju programske skokove

>>>

Odabir skupa naredaba

<<< (nastavak)

- Ulazno-izlazne naredbe služe za rad s ulazno-izlaznim jedinicama
- Naredbe za rad s bitovima omogućuju ispitivanje i mijenjanje pojedinih bitova u podatku
- Naredbe za rad s blokovima podataka omogućuju pretraživanje, čitanje i pisanje većeg broja podataka (tj. bloka) koji se nalazi u memoriji
- Specijalne naredbe s posebnom namjenom (npr. odabir načina prekidnog rada, dozvoljavanje ili zabranjivanje prekida, programsko izazivanje iznimaka, naredbe za atomarno ispitivanje i postavljanje memorijskih lokacija, rad s koprocesorima itd.)
- itd. ...

Uglavnom: postoji puno vrsta naredaba

Odabir skupa naredaba

Aritmetičko-logičke naredbe

Aritmetičko-logičke naredbe

- Naredbe koje postoje u svim procesorima su aritmetičko-logičke naredbe pa će ih i naš procesor imati
- U prethodnim razmatranjima vidjeli smo da ALU može izvoditi zbrajanje. To naravno nije dovoljno za izvođenje bilo kakvog ozbiljnijeg programa pa moramo vidjeti koje bi nam još operacije trebale
- Tipične operacije koje su često potrebne su:
 - zbrajanje - ADD
 - oduzimanje - SUB
 - logički I na bitovima - AND
 - logički ILI na bitovima - OR
 - logički EKSCLUZIVNI ILI na bitovima - XOR

Aritmetičko-logičke naredbe

- Za svaku od ovih operacija imat ćemo jednu naredbu, a za svaku naredbu moramo zadati operande
 - Budući da smo odabrali arhitekturu load-store (tj. registarsko-registarsku arhitekturu), **svi operandi i rezultat nalazit će se u općim registrima**
- Opći oblik naredaba izgledat će ovako:
naredba operand_1, operand_2, operand_3
- "izvedi operaciju između **prva dva operandi** i stavi rezultat u **treći operand**"
- Sada možemo napisati prvi program u kojem izračunavamo aritmetički izraz...

Aritmetičko-logičke naredbe - primjer

S podatcima iz registara izračunati sljedeće: $R0 := (R1+R2) - (R3+R4)$

Rješenje:

```
ADD R1, R2, R5 ; prvi dio izraza  
ADD R3, R4, R6 ; drugi dio izraza  
SUB R5, R6, R0 ; razliku spremi u R0
```

Rješenje bez promjene registara R5 i R6:

```
ADD R1, R2, R0 ; R1+R2  
SUB R0, R3, R0 ; R1+R2-R3  
SUB R0, R4, R0 ; R1+R2-R3-R4
```

Odabir skupa naredaba

Memorijske naredbe

Memorijske naredbe

- AL-naredbe mogu raditi samo s podatcima u registrima
 - Što ako bi u prethodnom primjeru bilo zadano da se podatci nalaze u memoriji? Kako ih dohvatiti? Što ako bi rezultat trebalo spremiti u memoriju?
- S obzirom da je broj registara ograničen, u praksi se (skoro) svi podatci čuvaju u memoriji. Zato trebamo naredbe pomoću kojih bi mogli pročitati podatak iz memorije ili upisati podatak u memoriju
- To će obavljati **memorijske naredbe**, a kako smo izabrali load-store arhitekturu, onda su **to jedine naredbe koje omogućuju razmjenu podataka između memorije i registara**

Memorijske naredbe

- Potrebne su nam dvije glavne operacije:
 - LOAD - čitanje 32-bitnog podatka iz memorije u registar
 - STORE - pisanje 32-bitnog podatka iz registra u memoriju
- Definirajmo operande za svaku naredbu:
LOAD register, (adresa)
STORE register, (adresa)
 - Naredba LOAD "puni register", tj. čita sadržaj četiriju memorijskih lokacija počevši od zadane *adrese* i stavlja ih u *register*
 - Naredba STORE "sprema register", tj. čita sadržaj zadanog *registra* i upisuje ga u četiri memorijske lokacije počevši od zadane *adrese*

Memorijske naredbe

- *Adresu* zadajemo kao običan broj (pozitivni cijeli broj), što znači da moramo poznavati položaj memorijske lokacije kojoj želimo pristupati
- Iako naredbe LOAD i STORE pristupaju četirima memorijskim lokacijama (jednobajtnim) odjednom, nećemo to posebno naglašavati
 - Podrazumijeva se da se zadana adresa odnosi na prvu lokaciju od potrebne četiri

Memorijske naredbe - primjeri

Primjer: izračunavanje izraza čiji operandi su u memoriji

Zbrojiti podatke iz memorijskih lokacija s adresama 1000 i 1004, a rezultat staviti na adresu 1008.

Rješenje:

```
LOAD  R0, (1000) ; dohvati 1. broj iz memorije
LOAD  R1, (1004) ; dohvati 2. broj iz memorije

ADD   R0, R1, R2 ; zbroji R0+R1 i spremi u R2

STORE R2, (1008) ; spremi rezultat u memoriju
```

Memorijske naredbe - primjeri

- Treba izračunati $R0 := (R1+55) - (R2 \text{ xor } 4ABC)$.
Pretpostavite da su konstante 55 i 4ABC spremljene u memoriji.

```
LOAD  R0, (BROJ1)      ; učitaj broj 55
ADD   R1, R0, R0        ; prvi dio izraza
LOAD  R3, (BROJ2)      ; učitaj broj 4ABC
XOR   R2, R3, R3        ; drugi dio izraza
SUB   R0, R3, R0        ; razliku spremi u R0
```

BROJ1 DW 55

BROJ2 DW 4ABC

Memorijske naredbe - primjeri

- U prethodnom primjeru vidjeli smo kako se određeni podatak može upisati u memoriju:

BROJ1 DW 55

BROJ2 DW 4ABC

- Kasnije ćemo detaljnije objasniti način pisanja i značenje ovih redaka, a sada možemo dati intuitivno objašnjenje:
 - BROJ1 i BROJ2 su labele ili nazivi memorijskih lokacija, koje možemo pisati umjesto stvarnih adresa tih memorijskih lokacija
 - Pomoću DW (*define word*) definiramo da se u memoriju upisuje određeni broj, u ovom slučaju to su brojevi 55 i 4ABC
 - Ove dvije memorijske lokacije možemo neformalno promatrati kao dvije globalne varijable s imenima BROJ1 i BROJ2 kojima smo dodijelili početne vrijednosti 55 i 4ABC

Memorijske naredbe - važna napomena

- U primjerima smo čitali ili pisali podatak s određene memorijske lokacije (npr. s adrese 1000)
 - Na toj lokaciji se mora nalaziti potrebni podatak u slučaju čitanja
 - Na toj lokaciji se mora nalaziti "slobodno" mjesto u slučaju pisanja
 - Na toj lokaciji ne smije se nalaziti neka druga naredba programa

Odabir skupa naredaba

Upravljačke naredbe

Upravljačke naredbe

- Za sada imamo:
 - Aritmetičko-logičke naredbe (ADD, SUB, AND, OR, XOR)
 - Memorejske naredbe (LOAD, STORE)
- Što **možemo** napraviti s ovim naredbama?
 - Ne puno, ali možemo ostvariti jednostavna izračunavanja pri čemu su podatci i rezultati u memoriji ili u registrima. Na primjer:
 - Izračunavanje aritmetičkih izraza: $a+3-4+b+(c-12)+d$
 - Izračunavanje operacija s bitovima:
(a OR 00001111) XOR (b AND 00111100)
 - Izračunavanje logičkih izraza: a AND b XOR true
 - Sve kombinacije gore navedenih izraza gdje se naredbe izvode **slijedno** jedna iza druge

Upravljačke naredbe

- **Ne možemo** ostvariti petlju do-while:

*do
Naredba_1
While (Uvjet)*

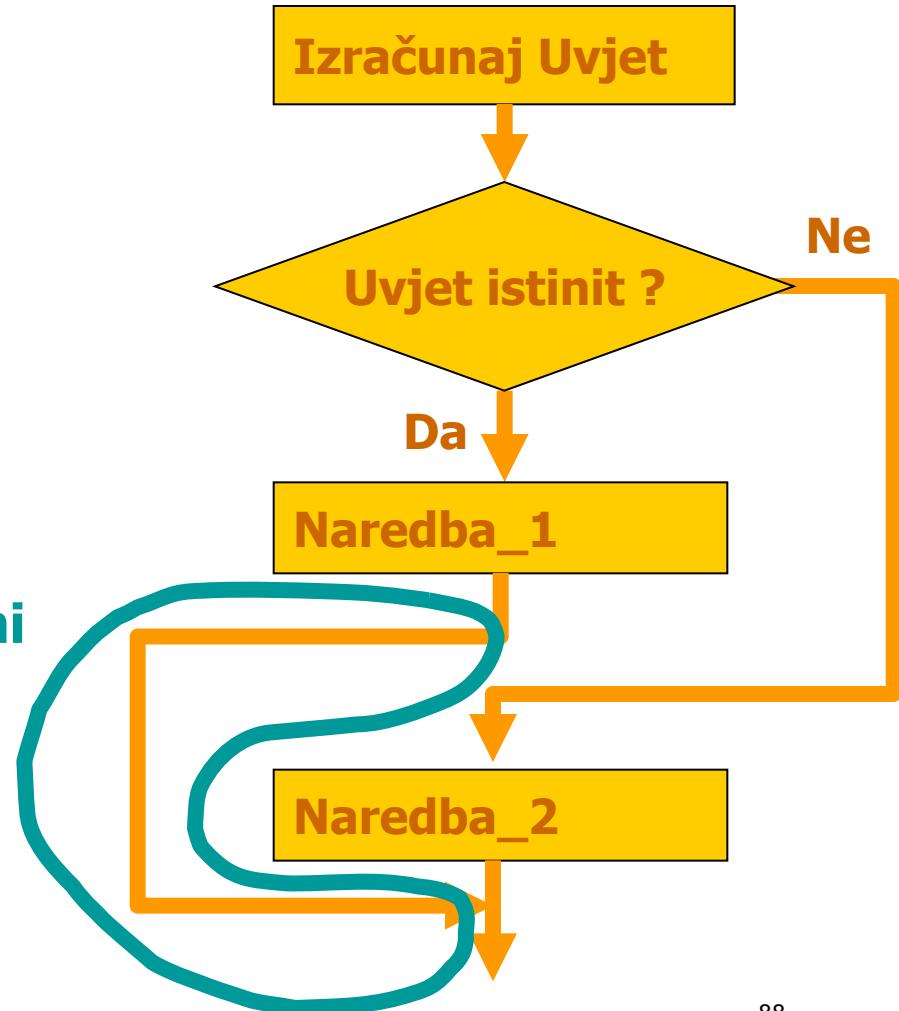


Upravljačke naredbe

- **Ne možemo** ostvariti uvjetno grananje:

```
if (Uvjet) then  
    Naredba_1  
else  
    Naredba_2  
endif
```

kako
ostvariti
bezuvjetni
skok
?



Upravljačke naredbe

- Problem je što se AL-naredbe i memorijske naredbe izvode **isključivo slijedno**, tj. jedna iza druge - onim redoslijedom kojim su napisane
- **Zaključak:** nedostaje nam mogućnost mijenjanja redoslijeda normalnog slijednog izvođenja, tj. treba nam **naredba skoka**
- Naredbe skokova svrstavaju se u upravljačke (kontrolne) naredbe jer one upravljaju tijekom izvođenja programa.

Upravljačke naredbe

- Prema prethodnim dijagramima toka, sigurno će nam trebati dvije vrste naredbe skoka i to:
 - **Naredba bezuvjetnog skoka** (promjena redoslijeda izvođenja)
 - **Naredba uvjetnog skoka** (grananje na jednu od dvije naredbe u ovisnosti o uvjetu)
- Za obje naredbe, moramo imati operand kojim zadajemo **odredište skoka**, tj. adresu naredbe na koju želimo skočiti

>>>

Upravljačke naredbe

- Nazovimo naredbu **bezuvjetnog skoka** JP (od JUMP)
- Definirajmo način pisanja i operand naredbe JP:

JP adresa

- Naredba JP bezuvjetno skače na naredbu sa zadanom *adresom*
- *Adresa* je zadana običnim brojem kao kod memorijskih naredaba (vidjet ćemo kasnije da se adresa također odnosi na četiri memorijske lokacije, što nećemo posebno naglašavati)
- Uočite da je moguć skok unaprijed ili unazad
- Kao i kod memorijskih naredaba, za *adresu* je umjesto broja moguće pisati labelu

Upravljačke naredbe

- Za **uvjetni skok** naredba se izvodi na dva moguća načina
 - Ako je uvjet ispunjen => skok se ostvaruje
 - Ako uvjet nije ispunjen => skok se ne ostvaruje, tj. izvodi se sljedeća naredba (ona koja je "ispod" naredbe skoka)
- Za naredbu uvjetnog skoka upotrijebimo isti naziv JP, ali ćemo mu dometnuti sufiks kojim ćemo označiti **uvjet**. Definirajmo način pisanja i operand naredbe JP:

JP _ uvjet adresa

- Naredba *JP_uvjet* skače na naredbu sa zadanom *adresom* samo ako je *uvjet* istinit, a inače nastavlja s izvođenjem sljedeće naredbe

Upravljačke naredbe

- Promotrimo li malo bolje, vidimo da je naredba bezuvjetnog skoka samo **specijalni slučaj** uvjetnog skoka pri čemu je uvjet uvijek istinit
- Zato će postojati **samo jedna naredba za skok JP** koju pišemo na dva načina:
`JP 100` // bezuvjetno skoči na naredbu na adresi 100
`JP _uvjet 100` // skoči na 100 ako je *uvjet* istinit
- Kako se piše **uvjet** ? >>>

Upravljačke naredbe - uvjeti

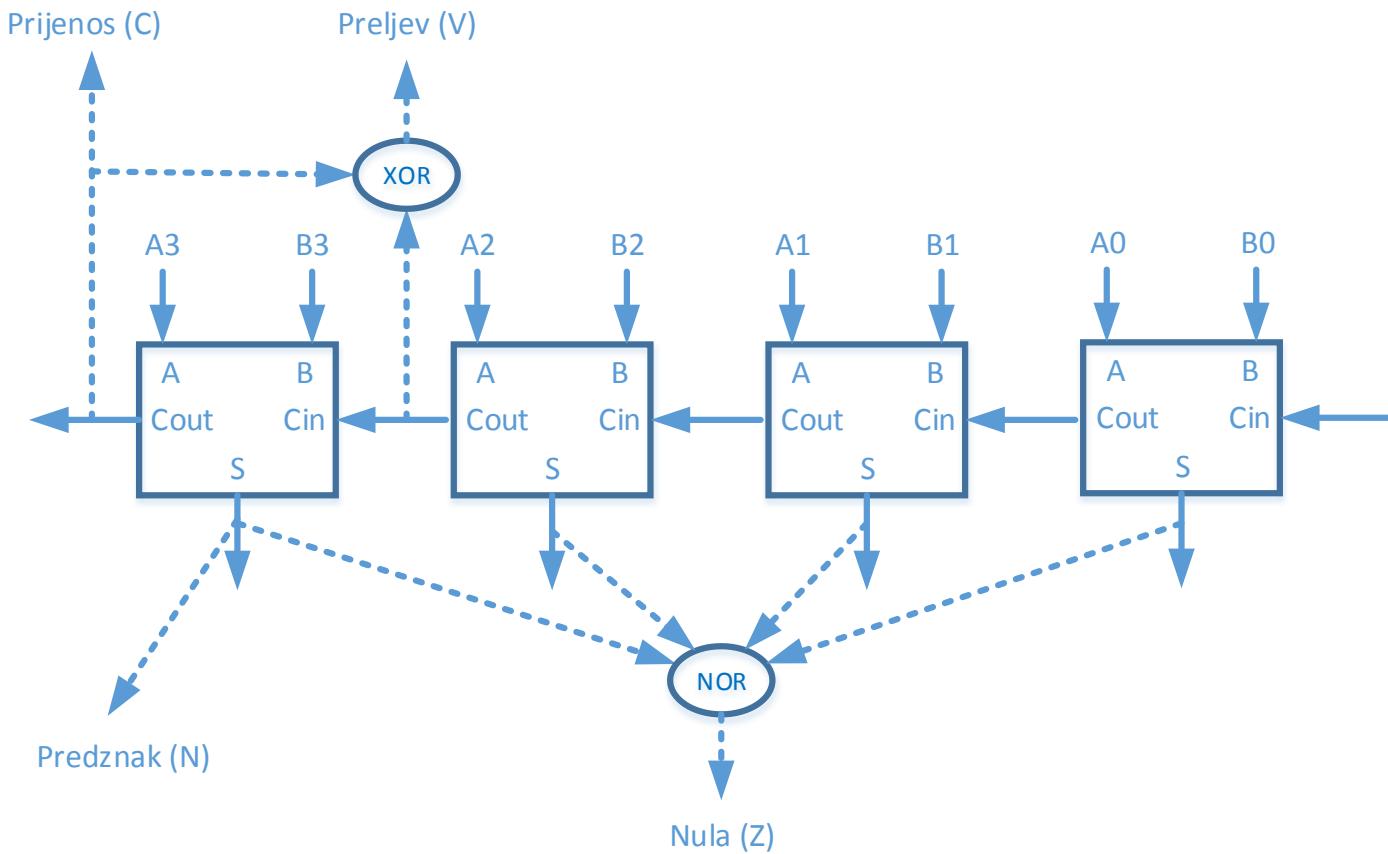
- Na primjer, uvjeti mogu biti sljedeći:
 - Jesu li dvije vrijednosti jednake?
 - Je li prva vrijednost veća od druge?
 - Je li prva vrijednost manja ili jednaka od druge?
 - itd. (općenito se uspoređuju dvije numeričke ili logičke vrijednosti)
- U uvodnom poglavlju već smo vidjeli kako se mogu usporediti dvije vrijednosti:
 1. prvo se izvede AL-operacija (najčešće je to oduzimanje)
 2. nakon toga se ispitaju zastavice (Podsjetnik: zastavice su bistabili koji se postavljaju na temelju ALU-operacije)

Upravljačke naredbe - zastavice

- C – prijenos
- V - preljev (od engl. overflow, jer je slovo O previše slično znamenki 0)
- Z - nula
- N - predznak (od engl. negative)

- Ove zastavice se postavljaju pri izvođenju aritmetičko-logičkih naredaba

Zastavice



Upravljačke naredbe - registar SR

- Zastavice se nikada ne nalaze u procesoru kao zasebni bistabili, nego su uvijek unutar registra koji čuva i druge zastavice (npr. prekidne zastavice, zastavice koje označuju stanje procesora i sl. - više o tome kasnije)
- Zato ćemo i mi staviti zastavice u jedan registar koji ćemo nazvati registrom stanja SR (engl. status register). Kasnije ćemo mu dodati i druge zastavice koje nam budu trebale:



Upravljačke naredbe - zastavice

- Utjecaj aritmetičko-logičkih naredaba na zastavice:
 - ADD, SUB: C=prijenos, V=preljev, Z=nula, N=predznak
 - AND,OR,XOR: C=0, V=0, Z=nula, N=predznak
- Zastavice Z i N postavljaju se na temelju rezultata AL-naredbe
- Zastavice C i V se u logičkim naredbama brišu jer za logičke operacije prijenos i preljev nemaju smisla

Upravljačke naredbe - uvjeti

- Napravimo popis svih uvjeta koji bi nam mogli trebati u naredbi skoka JP:
 - Uvjeti koji izravno ispituju zastavice
 - Je li zastavica postavljena (set), tj. je li jednaka jedinici
 - Je li zastavica obrisana (clear, reset), tj. je li jednaka nuli
 - Uvjeti koji služe za usporedbu brojeva
 - Usporedba NBC-brojeva
 - Usporedba 2'k brojeva

>>>

Uvjeti

- Prva skupina - izravno ispitivanje zastavica

Zapis	Značenje (Ispitivani uvjet)
C	$C=1$
NC	$C=0$
V	$V=1$
NV	$V=0$
Z	$Z=1$
NZ	$Z=0$
N	$N=1$
NN	$N=0$

Uvjeti

- Druga skupina - usporedbu brojeva (zasebno za NBC i 2'k brojeve)

Zapis	Značenje (engl.)		Ispitivani uvjet
ULE	Unsigned Less or Equal	\leq	$C=0$ or $Z=1$
UGT	Unsigned Greater Than	$>$	$C=1$ and $Z=0$
ULT	Unsigned Less Than	$<$	$C=0$
UGE	Unsigned Greater or Equal	\geq	$C=1$
SLE	Signed Less or Equal	\leq	$(N \text{ xor } V)=1$ or $Z=1$
SGT	Signed Greater Than	$>$	$(N \text{ xor } V)=0$ and $Z=0$
SLT	Signed Less Than	$<$	$(N \text{ xor } V)=1$
SGE	Signed Greater or Equal	\geq	$(N \text{ xor } V)=0$

Uvjeti

- Još neki praktični uvjeti

Zapis	Značenje (engl.)	Ispitivani uvjet
EQ	Equal	$Z=1$
NE	Not Equal	$Z=0$
M	Minus	$N=1$
P	Plus (Positive)	$N=0$

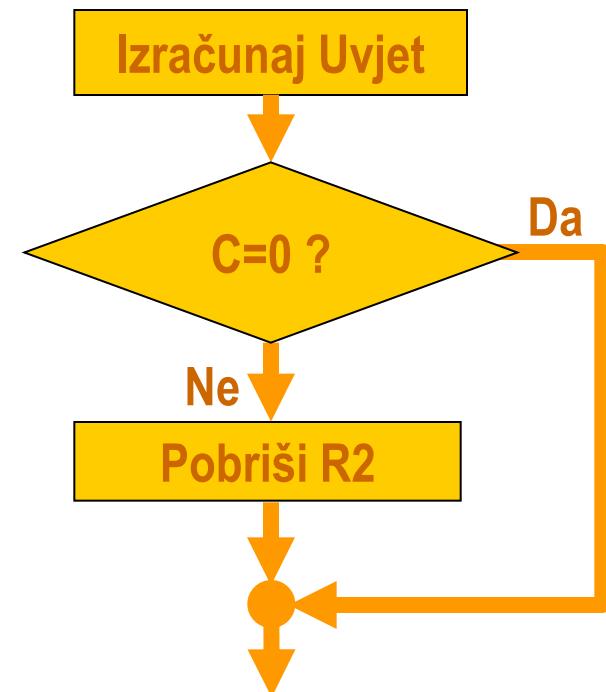
Upravljačke naredbe - primjeri

Primjer uvjetnog grananja - naredba if:

Zbrojiti registre R0 i R1 i rezultat staviti u R2. Ako nema prijenosa, onda ne treba napraviti ništa, a ako ima, onda treba pobrisati R2.

Rješenje:

```
ADD R0,R1,R2 ; AL-operacija  
  
JP_NC DALJE ; ispitivanje  
; zastavica  
; i skok  
  
SUB R2,R2,R2 ; briši R2  
  
DALJE ... ; nastavak  
; programa
```



Upravljačke naredbe - primjeri

ADD R0 ,R1 ,R2

JP_NC DALJE

SUB R2 ,R2 ,R2

DALJE . . .

- Svaki uvjet može se napisati i na "obrnut" način. U praksi uvjet "okrećemo" tako da da dobijemo što kraći i razumljiviji program
- Prethodni program napisan s "obrnutim" uvjetom izgledao bi ovako:

ADD R0 ,R1 ,R2

JP_C BRISI

JP DALJE

BRISI SUB R2 ,R2 ,R2

DALJE . . .

Upravljačke naredbe - primjeri

- Primjer uvjetnog i bezuvjetnog grananja - naredba if-else:
- Zbrojiti registre R0 i R1 i rezultat staviti u R2. Ako dođe do prijenosa treba obrisati memoriju REZ, a inače u nju treba upisati R2.
- Rješenje:

```
ADD R0 ,R1 ,R2
```

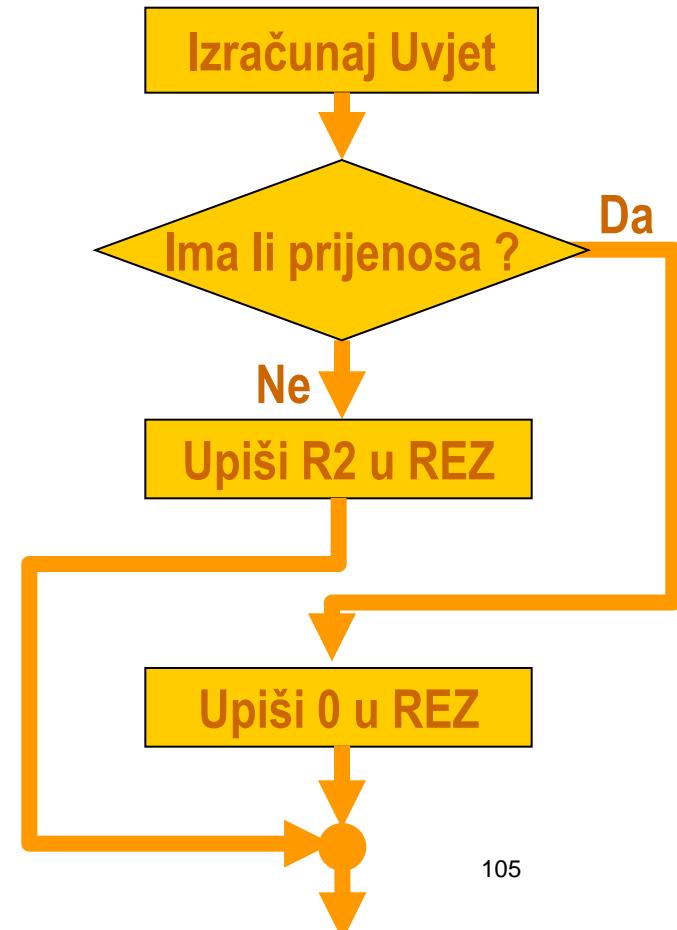
```
JP_C BRISI
```

```
PISI STORE R2 , (REZ) ; upiši R2  
; u REZ
```

```
JP DALJE
```

```
BRISI SUB R3 ,R3 ,R3 ; obriši  
STORE R3 , (REZ) ; REZ
```

```
DALJE ...
```



Upravljačke naredbe - primjeri

Primjer petlje u postupku množenja:

Treba pomnožiti dva NBC broja (označimo to kao $A*B$) koji su smješteni u memoriji na adresama 100 i 200. Rezultat množenja treba spremiti u memoriju na adresu 300.

Rješenje:

Program ćemo temeljiti na dijagramu toka. U programu ćemo vidjeti većinu onoga što smo do sada naučili:

- AL-naredbe,
- memorejske naredbe,
- rad s konstantama,
- naredbu uvjetnog i bezuvjetnog skoka.

Dodatno ćemo vidjeti i petlju s brojačem.

```

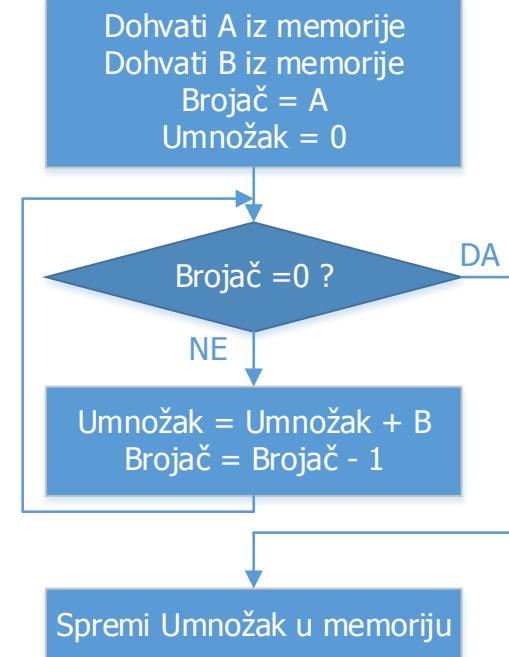
LOAD R0, (100)      ; R0 ≡ A
LOAD R1, (200)      ; R1 ≡ B
OR   R0, R0, R2     ; R2 ≡ Brojač := A
LOAD R3, (400)      ; R3 ≡ Umnožak := 0
LOAD R4, (400)      ; R4 ≡ 0
LOAD R5, (410)      ; R5 ≡ 1

PETLJA SUB   R2, R4, R2
JP_EQ KRAJ

ADD   R3, R1, R3
SUB   R2, R5, R2
JP    PETLJA

KRAJ  STORE R3, (300)
...
400   DW    0
410   DW    1

```



Strojni kôd naredaba

Općenito

Strojni kôd naredaba

- Za sada smo definirali 8 naredaba i opisali što te naredbe rade
- Ponovimo:
 - procesor svaku naredbu mora dohvatiti, dekodirati i izvesti
 - dekodiranje je, zapravo, raspoznavanje naredbe kako bi se znalo što točno treba izvesti
 - svaka naredba je zapisana u memoriji (kao niz nula i jedinica) i ima svoj posebni oblik po kojem procesor prepoznaje tu naredbu
- Dakle, svaka naredba mora imati **jedinstveni** zapis, koji će je **razlikovati** od svih drugih naredaba

Strojni kôd naredaba - FRISC

- Za sve naredbe koje smo do sada odabrali moramo definirati strojne kôdove (vodeći računa o budućim proširenjima procesora)
- Pokazat ćemo kako su zamišljeni strojni kôdovi, s time da razlozi za definiciju pojedinih bitova neće biti jasni odmah, nego tek kad budemo imali proširenu verziju procesora
- Zbog jednostavnosti ćemo objašnjenja ovakvih slučajeva odgoditi za kasnije i nećemo sada definirati strojni kôd da bude efikasan za osam dosadašnjih naredaba, jer bi onda kasnije morali mijenjati tako definirane strojne kôdove

Strojni kôd naredaba - FRISC

- Mi projektiramo ugradbeni procesor koji:
 - treba biti što jednostavniji
 - treba imati mali broj naredaba
 - ima mali broj registara
 - treba imati mali broj načina adresiranja
- Zato, odabiremo da širina strojnog kôda bude procesorska riječ, odnosno 32 bita za sve naredbe
- Također ćemo se truditi da polja budu raspoređena što pravilnije kako bi se pojednostavnilo i ubrzalo dekodiranje naredaba

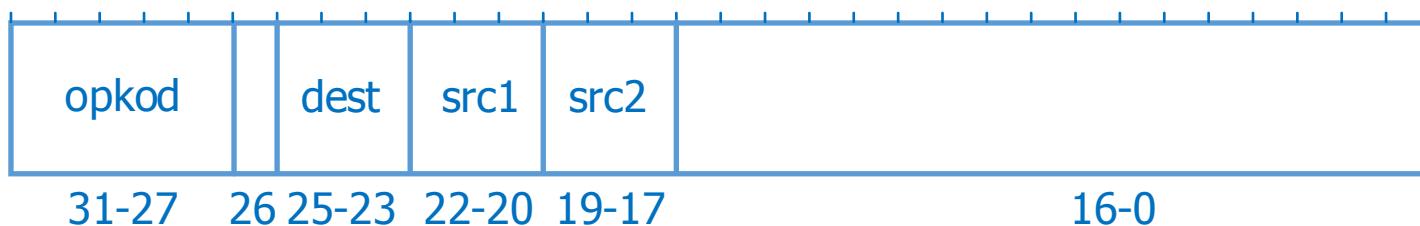
Strojni kôd naredaba

- **Polje operacijskog kôda** jednoznačno definira o kojoj se naredbi radi
- Za operacijski kôd moramo uzeti dovoljno bitova da možemo razlikovati sve potrebne naredbe (ili grupe naredaba). Odabiremo da će polje operacijskog kôda (opkod) zauzimati 5 bitova što nam daje ukupno $2^5 = 32$ kombinacija čime možemo izravno razlikovati 32 naredbe (za sada imamo samo 8 naredaba, ali ćemo ih kasnije proširivati).



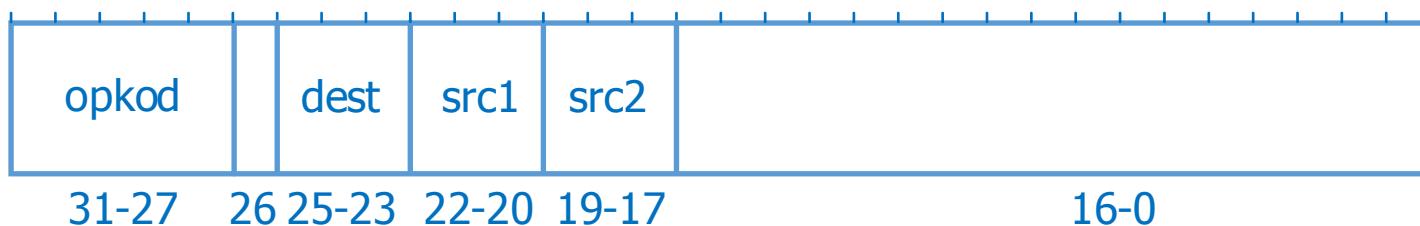
Strojni kôd naredaba

- **Polja operanada** jednoznačno definiraju s kojim operandima naredba obavlja svoju zadaću
- Na primjer, za naredbu ADD:
 - Koji registar će biti prvi operand zbrajanja
 - Koji registar će biti drugi operand zbrajanja
 - U koji registar se spremi rezultat
- Za FRISC je definirano da AL naredbe imaju tri operanda i svi su registri. Kako smo definirali da FRISC ima 8 registara opće namjene svaki operand kôdiramo sa po tri bita.



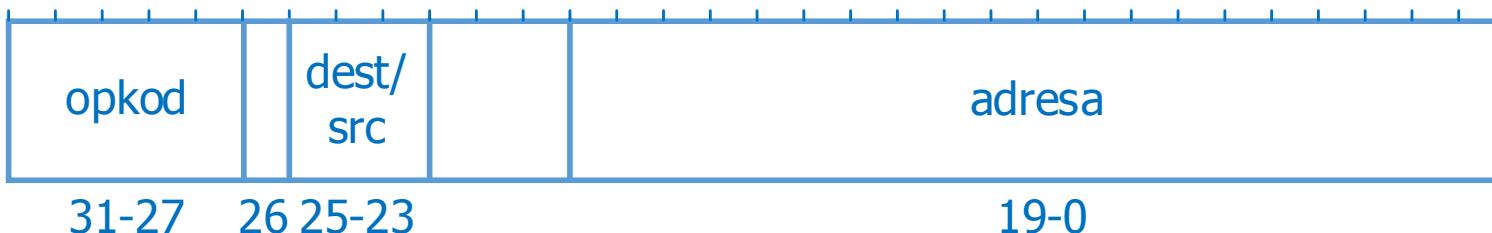
Strojni kôd naredaba

- Iz ovako definiranog strojnog kôda vidimo da imamo neiskorišteno ukupno 18 bitova
- Njih ćemo kasnije upotrijebiti za daljnja proširenja i poboljšanja AL-naredaba



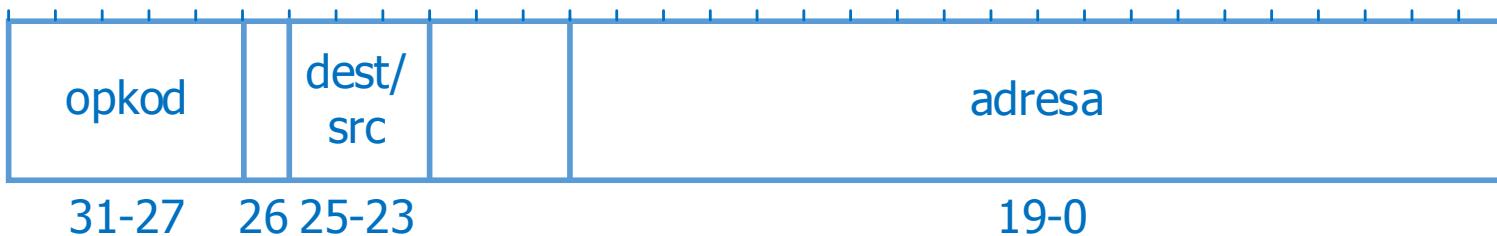
Strojni kôd - memorejske naredbe

- Memorijske naredbe imaju dva operanda:
 - prvi operand je registar čije značenje ovisi o naredbi:
 - za LOAD znači odredišni registar u koji se puni vrijednost (dest)
 - za STORE znači izvorišni registar iz kojeg se čita vrijednost (src)
 - drugi operand je adresa memorijske lokacije s koje se čita/piše podatak
- Nakon operacijskog kôda, kodiramo redom:
 - jedan bit (26) je neiskorišten
 - tri bita (23 do 25) za prvi operand, tj. za registar (dest/src)
 - tri bita (20 do 22) se ne koriste
 - preostalih 20 bitova (0 do 19) sadrže drugi operand, tj. adresu



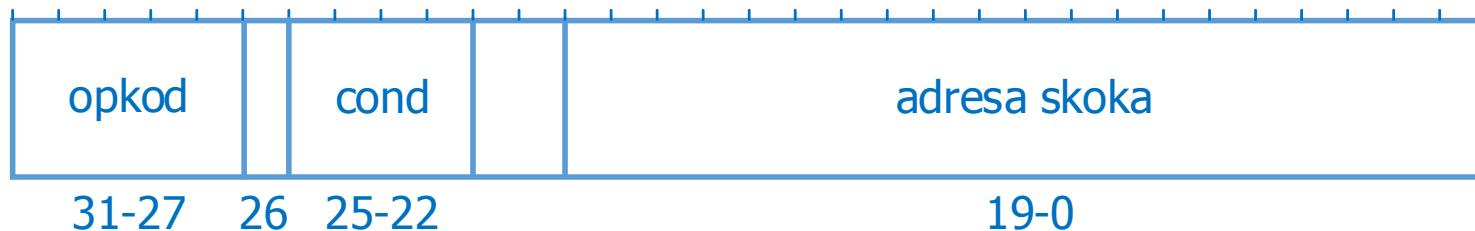
Strojni kôd - memorejske naredbe

- Vidimo da je adresa ograničena na samo 20 bitova, iako smo definirali da će adresna sabirnica i adresa imati 32 bita
- Na ovom primjeru vidimo kako odluka da svaka naredba ima samo 32 bita ograničava podatke i adrese koje moramo kodirati u naredbi
- Posljedice ovog ograničenja objasnit ćemo kasnije



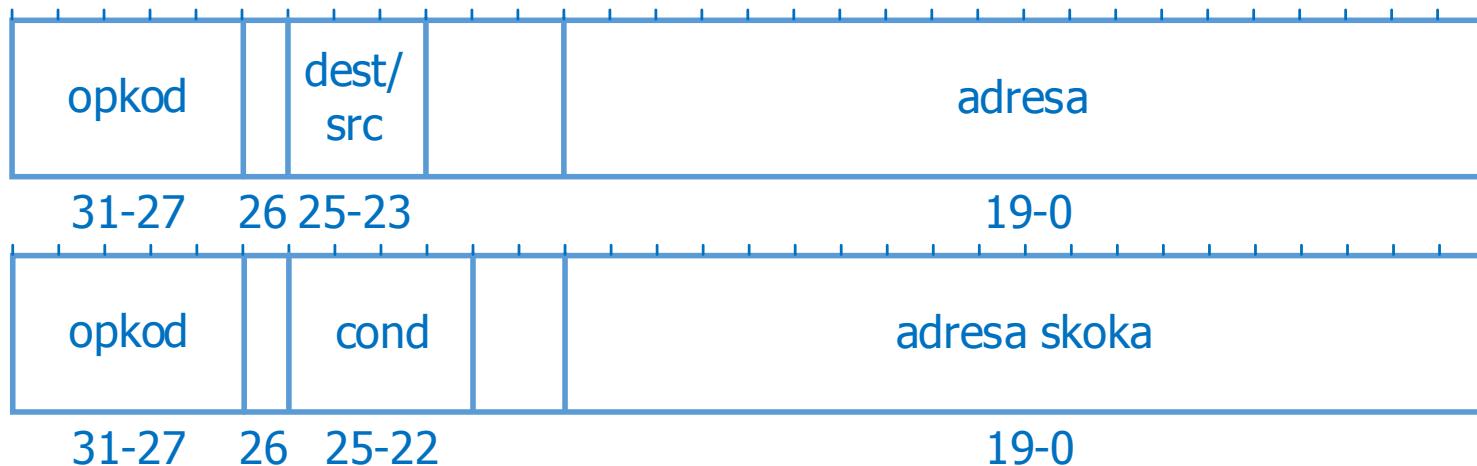
Strojni kôd - upravljačke naredbe

- Upravljačke naredbe imaju uvjet i jedan operand:
 - uvjet se piše kao dio naredbe (sufiks)
 - operand je adresa skoka (20 bita)
- Nakon operacijskog kôda, kodiramo redom:
 - jedan bit (26) je neiskorišten
 - četiri bita (22 do 25) za uvjet (cond)
 - dva bita (20 do 21) se ne koriste
 - preostalih 20 bitova (0 do 19) sadrže operand, tj. adresu skoka



Strojni kôd - upravljačke naredbe

- Usporedimo polje adrese s istim poljem u memorijskim naredbama:
 - i ovdje adresa ima samo 20 bitova od potrebnih 32
 - razlika u značenju adrese je da ovdje adresa predstavlja odredište skoka, a ne položaj podatka za čitanje/pisanje
 - budući da obje naredbe imaju jednako polje za adresu, strojni kôd je pravilniji pa će i dekodiranje i izvođenje naredaba biti jednostavnije i brže (barem tako očekujemo)



Strojni kôd - upravljačke naredbe

- Pogledajmo još je li za kodiranje uvjeta dovoljno 4 bita kojima se može kodirati 16 različitih uvjeta?
- Iz popisa uvjeta imamo sljedeće različite uvjete:
 - 8 uvjeta za izravno ispitivanje zastavica
 - 8 uvjeta za usporedbu brojeva
 - 2 uvjeta za ispitivanje jednakosti brojeva: EQ, NE
 - 2 uvjeta za ispitivanje predznaka: M i P
 - 1 uvjet koji je uvijek istinit (za bezuvjetni JP)
- Ukupno imamo 21 različitih uvjeta što je više od 16 mogućih pa na prvi pogled izgleda da imamo premalo bitova u polju

>>>

Strojni kôd - upravljačke naredbe

<<<

- Ali, ovdje se radi samo o različitim načinima **pisanja** uvjeta
- Bitno je koliko postoji različitih načina **ispitivanja zastavica**, tj. koliko postoji različitih načina izvođenja naredbe, jer se samo to mora razlikovati u strojnom kôdu
- Iz tablica uvjeta vidimo da dio uvjeta ispituje zastavice na jednak način i da postoji samo **15 različitih** ispitivanja zastavica
- Dakle: 4 bita su dovoljna



Osnovna inačica procesora

**Pregled arhitekture -
Rekapitulacija**

OSNOVNA INAČICA PROCESORA

- Do sada donesenim odlukama o izboru dijelova arhitekture i do sada opisanim naredbama, definirali smo osnovnu inačicu arhitekture našeg jednostavnog procesora:
 - Von Neumannova arhitektura
 - Load-store arhitektura
 - Osam 32-bitnih registara opće namjene (R0 do R7)
 - 32-bitni registar PC
 - Registar stanja SR (za sada ima definirana 4 bita)
 - Širina podataka unutar procesora je 32 bita
 - Adresna i podatkovna sabirnica su širine 32 bita
 - Memorija s adresiranjem okteta
 - RISC arhitektura
 - Početni skup od osam naredaba

OSNOVNA INAČICA PROCESORA

- Do sada smo se uglavnom koncentrirali na arhitekturu skupa naredaba (ISA), a manje na mikroarhitekturu
- U dalnjim predavanjima ćemo definirati poboljšanja i proširenja funkcionalnosti procesora pri čemu ćemo:
 - posvetiti više pažnje mikroarhitekturi
 - proširiti skup naredaba s novim mogućnostima koje omogućuju lakše i efikasnije programiranje

Poboljšana inačica procesora

**Detaljniji pogled na arhitekturu
skupa naredaba i mikroarhitekturu**

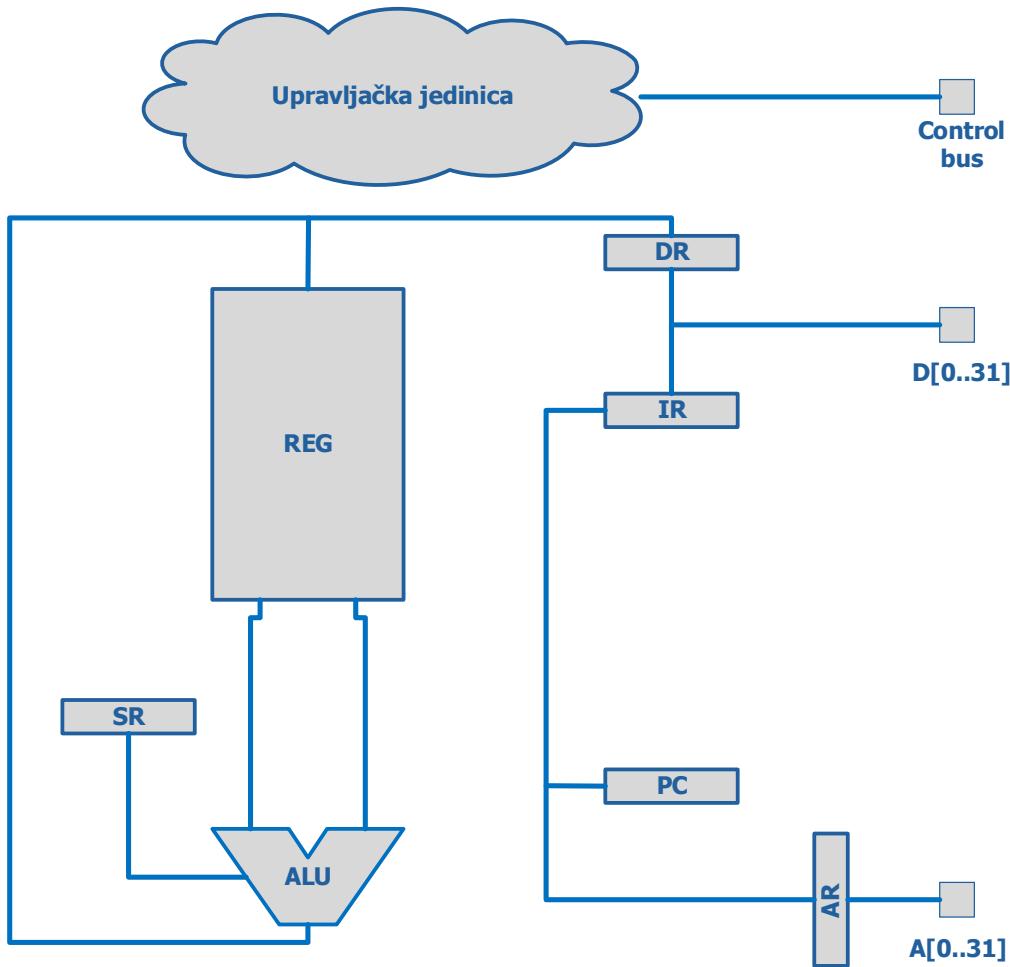
Poboljšana inačica procesora

- Za detaljnije definiranje naredaba moramo vidjeti:
 - Potrebne/praktične promjene i proširenja u skupu naredaba
 - Adresiranja
 - Strojne kôdove konačnog skupa naredaba
 - Način izvođenja naredaba i protočnu strukturu
- Za detaljnije definiranje mikroarhitekture moramo vidjeti:
 - Dijelove procesora i način njihovog spajanja
 - Put podataka i upravljačku jedinicu
 - Priklučke procesora
 - Spajanje procesora s memorijom i vanjskim jedinicama
 - Način komunikacije procesora s memorijom i vanjskim jedinicama

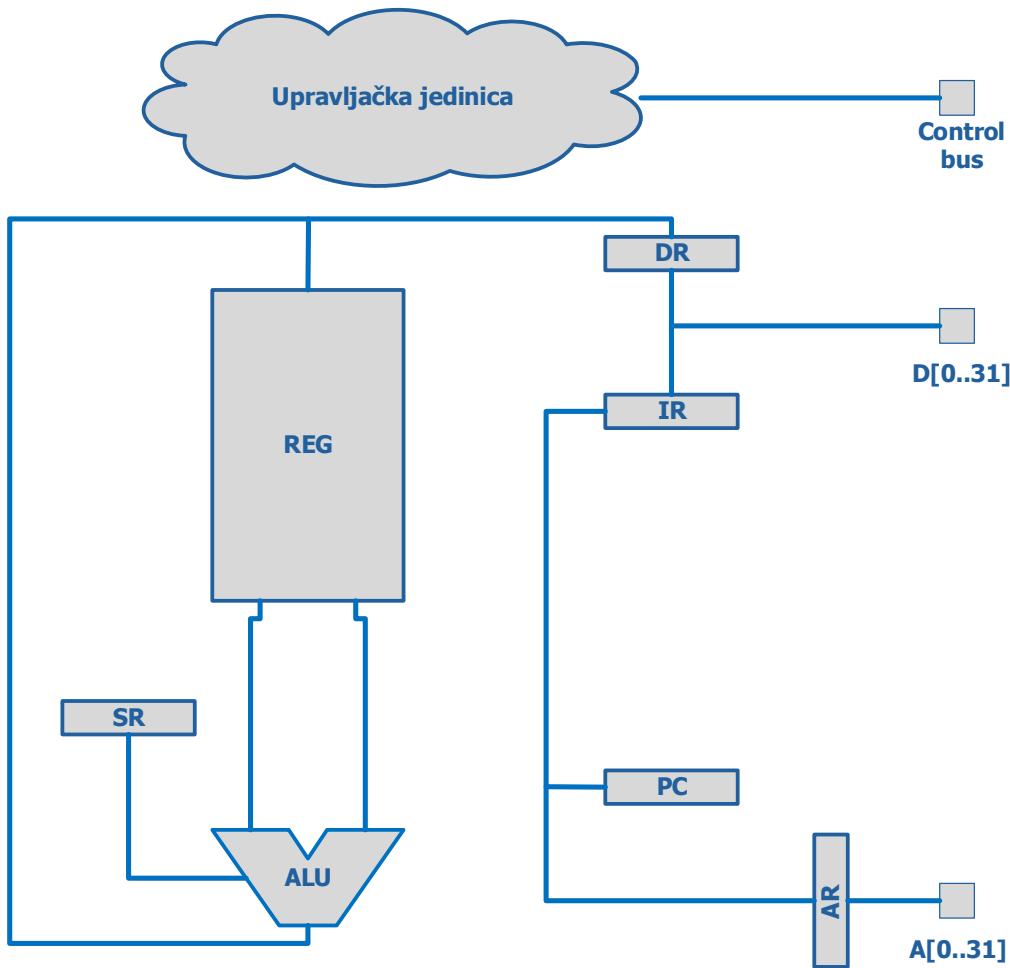
Načelna mikroarhitektura

- Pogledajmo kako načelno izgleda mikroarhitektura našeg procesora:
 - koje dijelove sadrži
 - kako su ti dijelovi povezani
- Slika će za početak biti samo shematska i pojednostavljena, a kasnije ćemo je prikazivati sve detaljnije

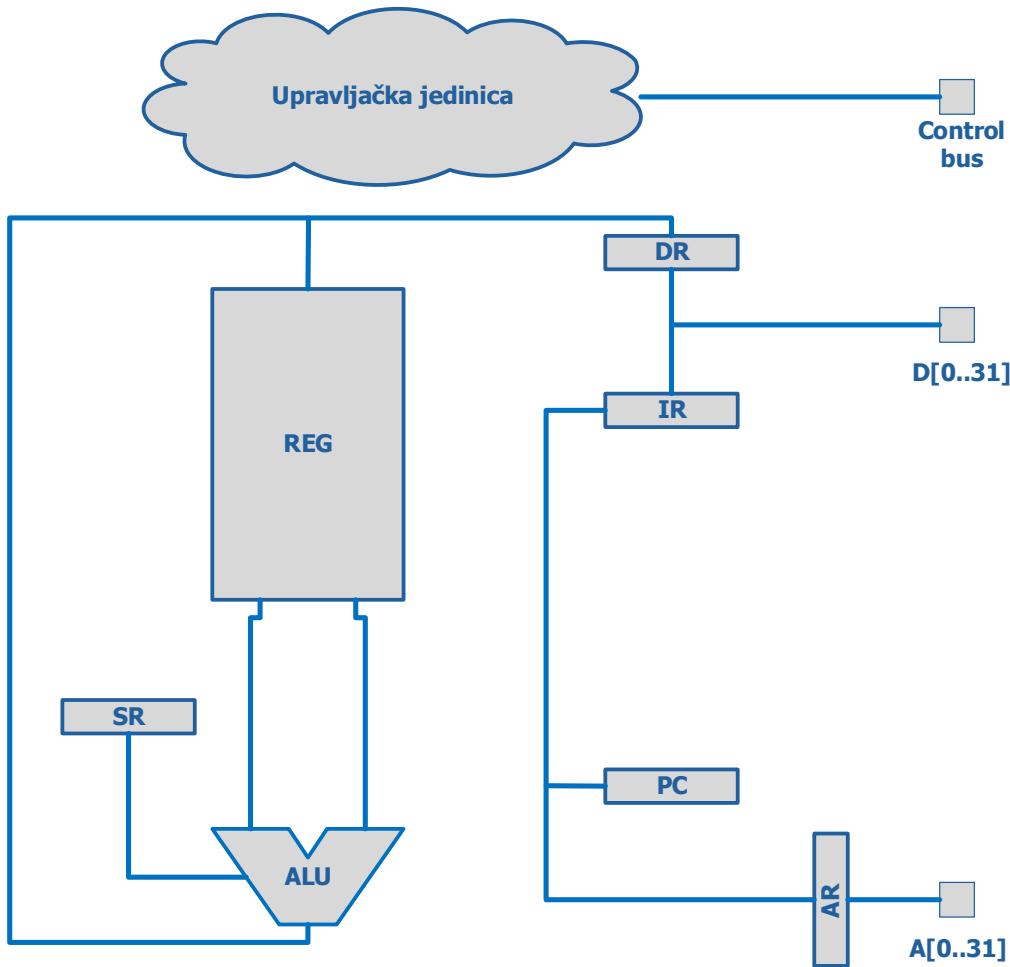
>>>



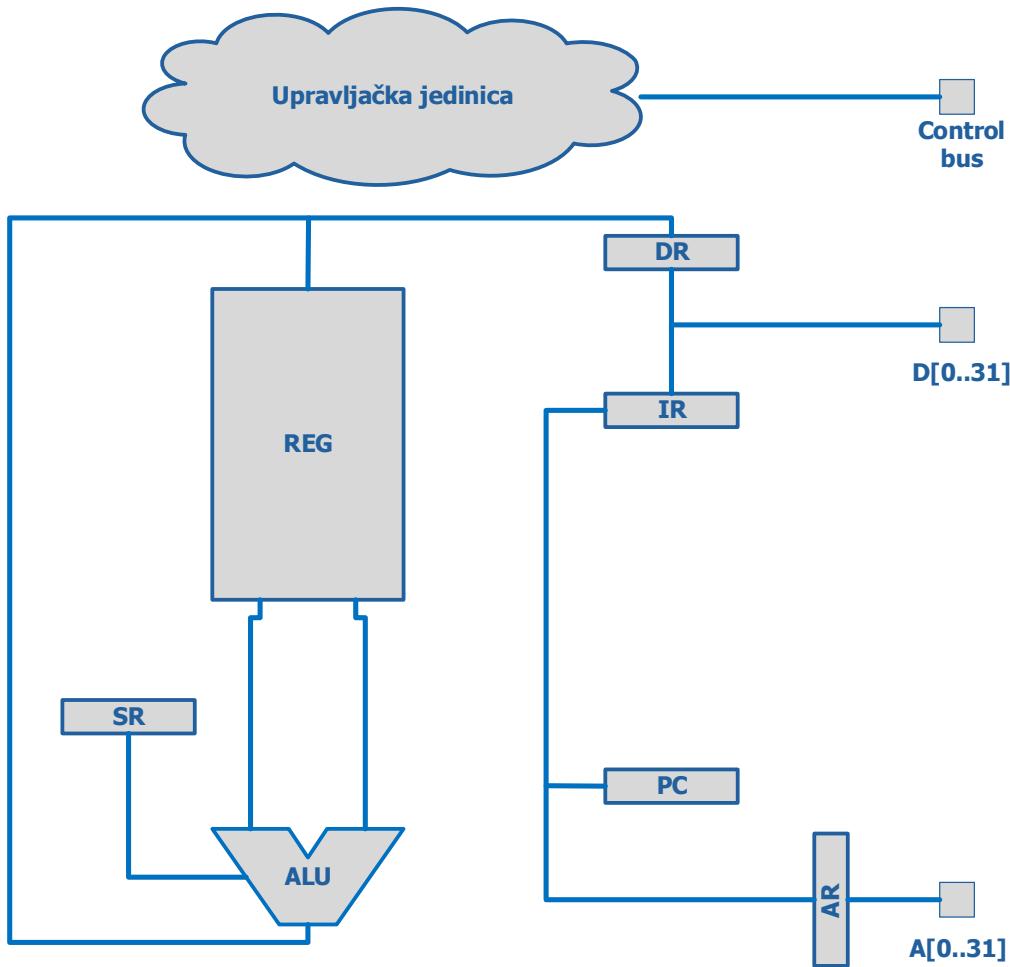
Na slici vidimo skup registara opće namjene (REG) i ostale registre procesora kao i način na koji su načelno spojeni >>>



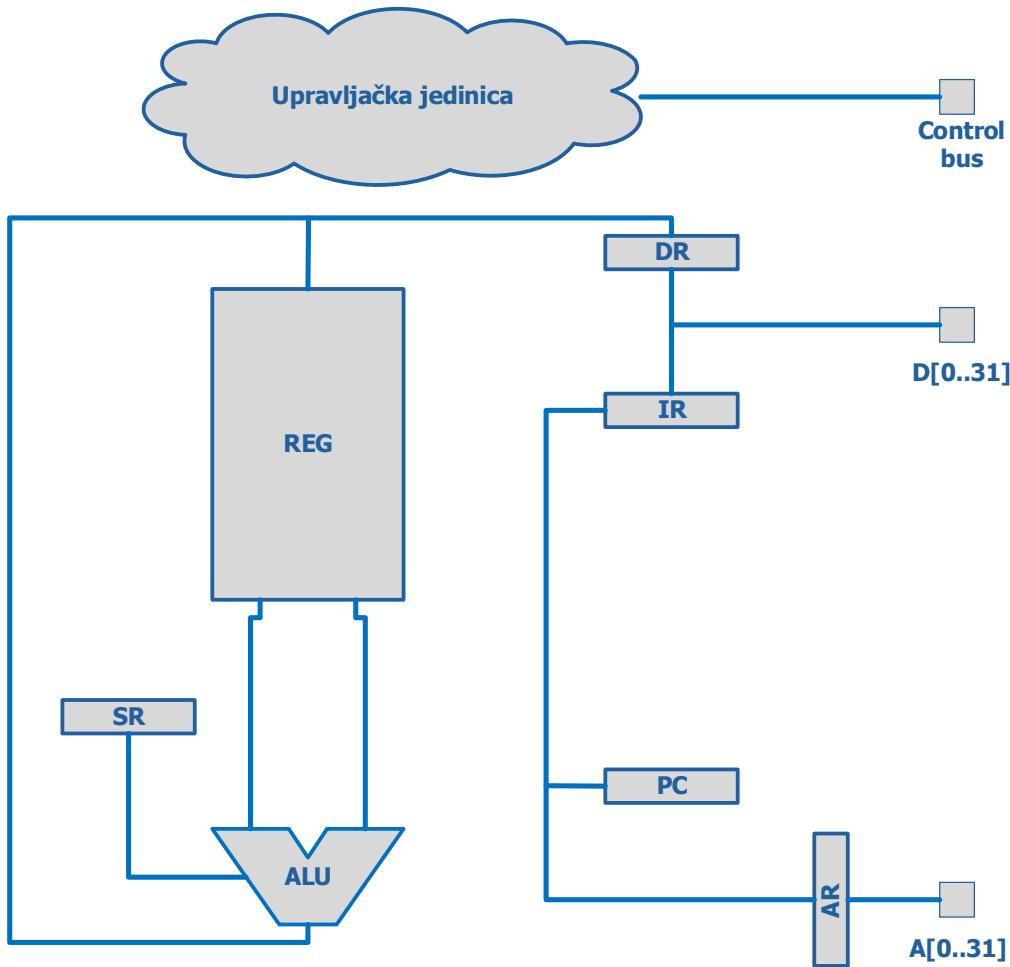
Iz registara opće namjene postoje dva spojna puta do ALU za slanje operanada u AL-naredbama



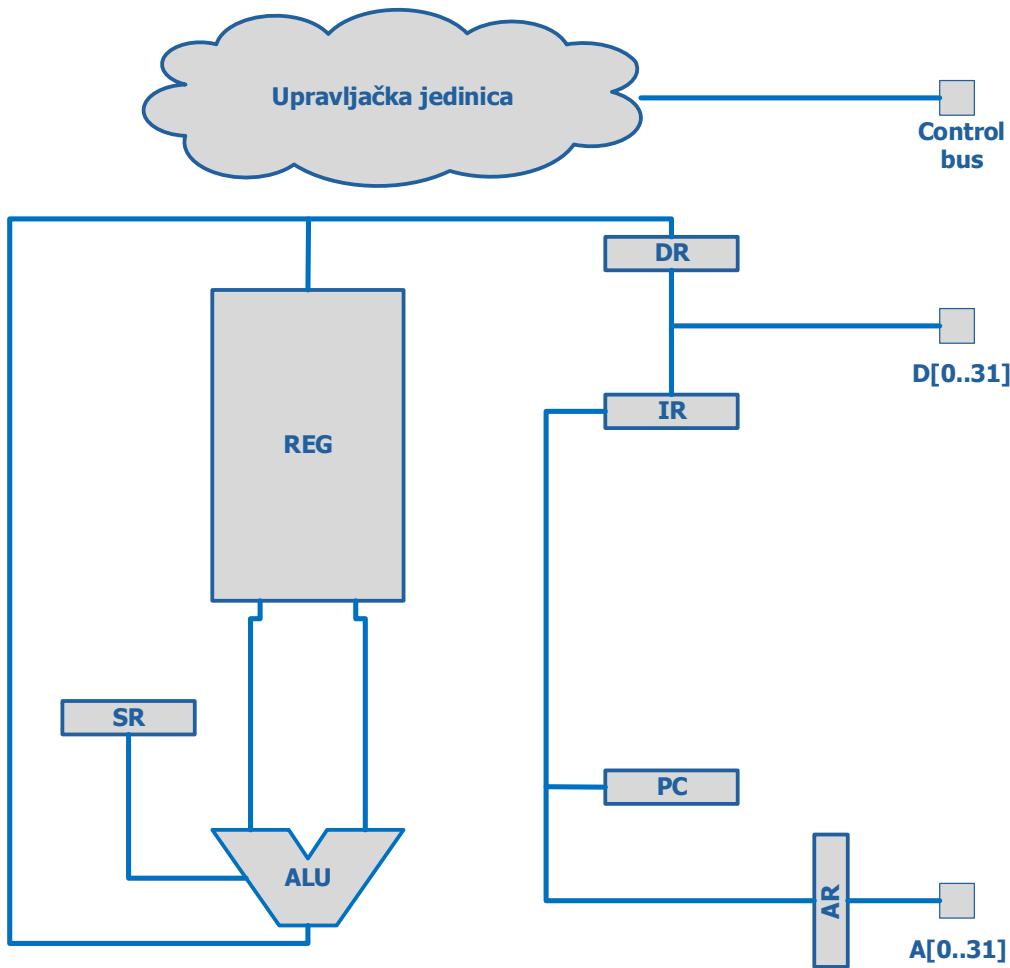
Iz ALU postoji veza do registra stanja SR, kako bi AL-operacija mogla utjecati na stanje zastavica



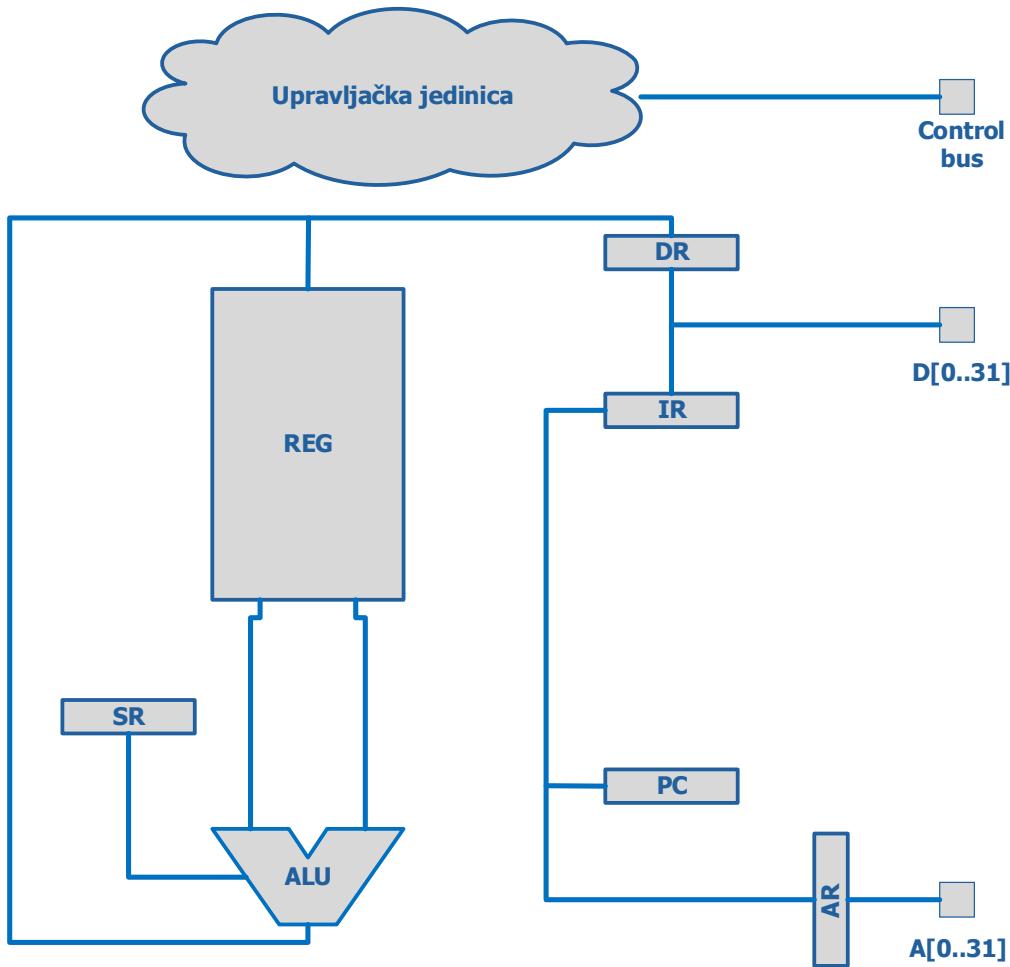
Postoji spojni put od izlaza ALU do skupa općih registara, kako bi u ALU-naredbama rezultat mogao biti zapisan u jedan od općih registara



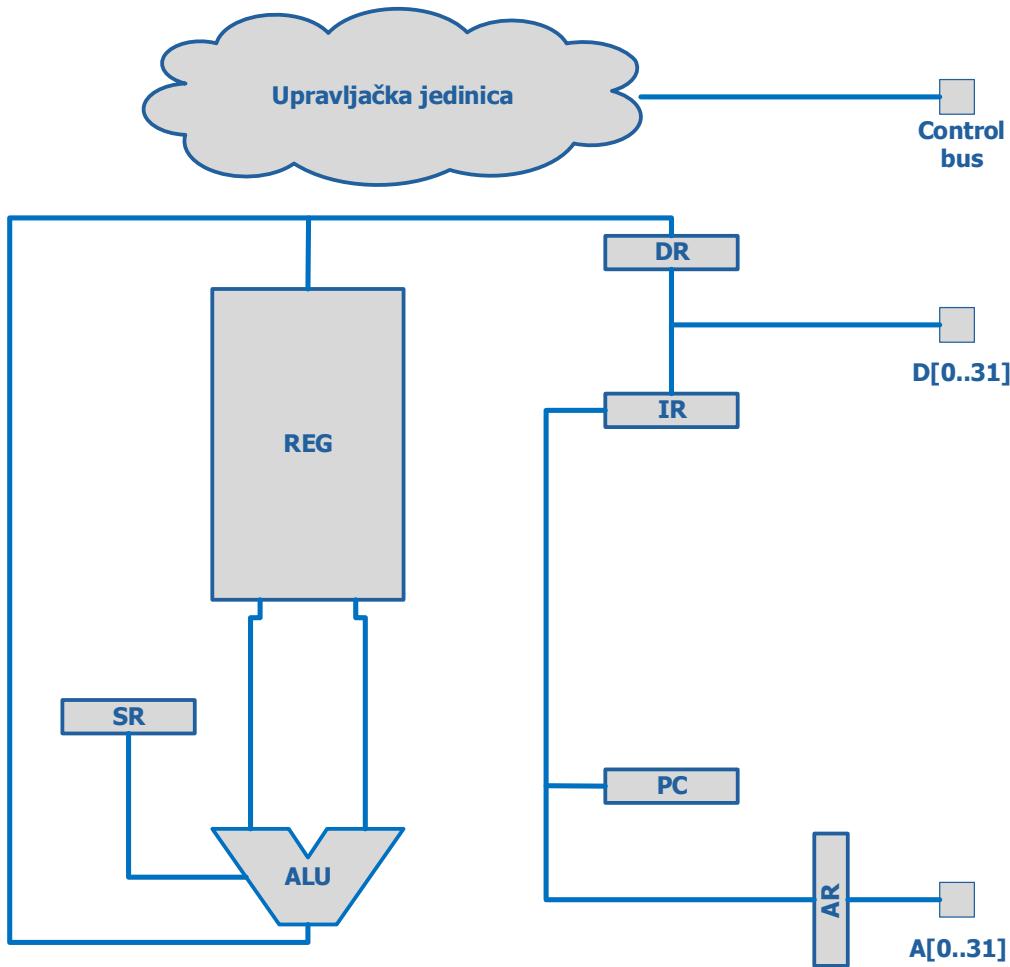
Registrar podataka DR (Data Register) služi kao međuspremnik između unutrašnjosti procesora i sabirnice podataka, koja je spojena na procesor preko podatkovnih priključaka



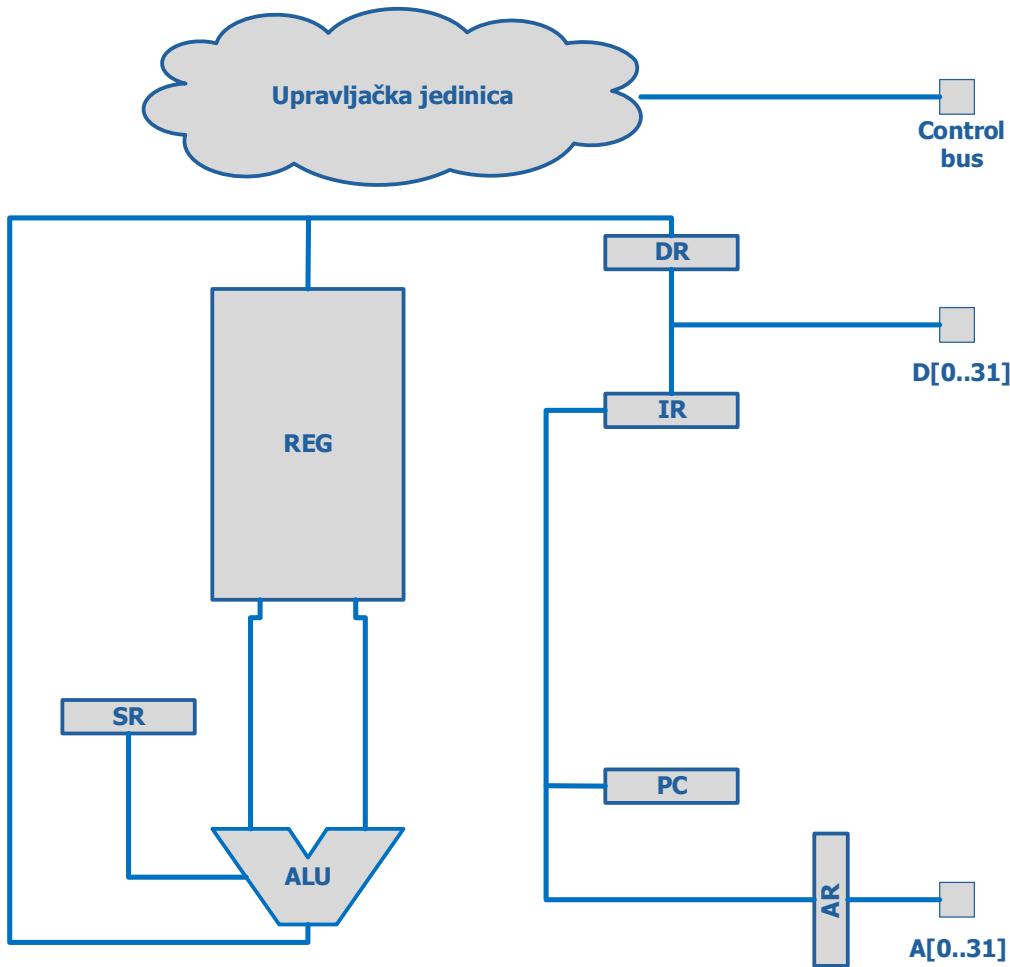
Podatci koji se čitaju i pišu preko sabirnice podataka uvijek prolaze kroz registar DR (uz jednu iznimku)



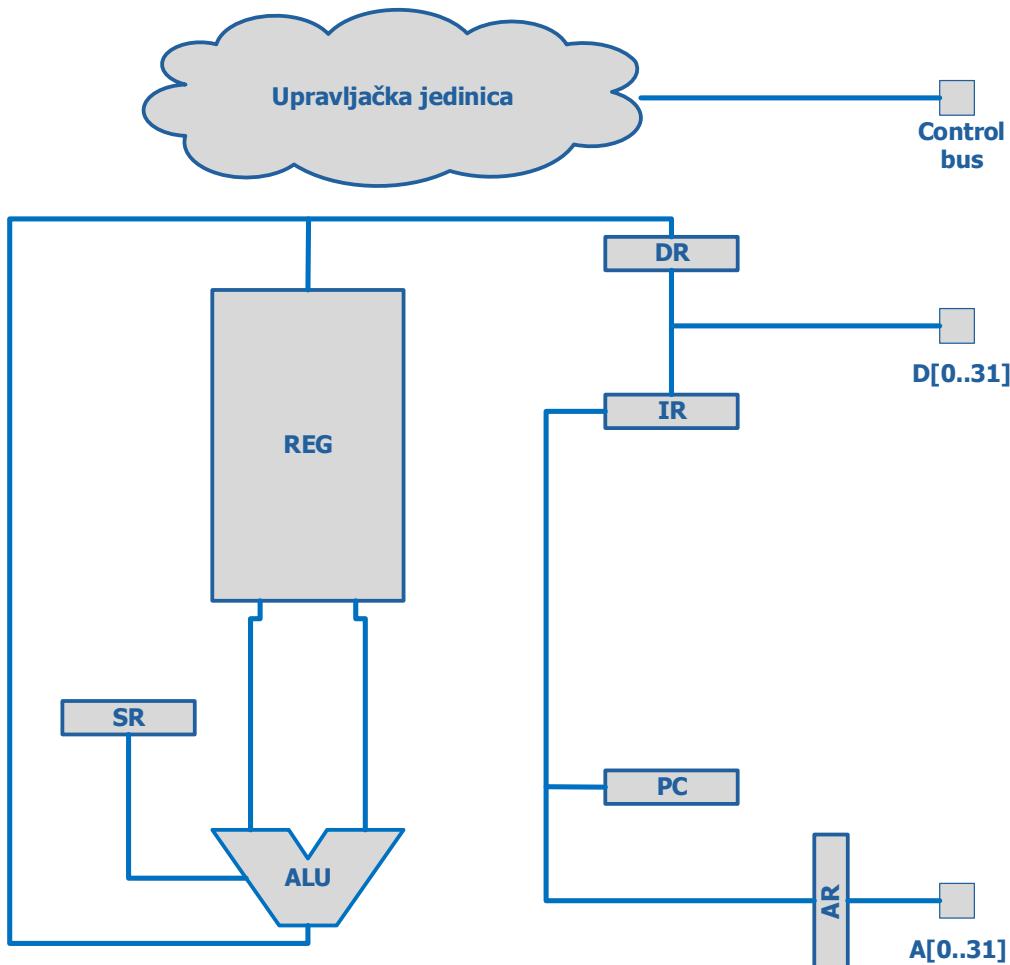
Ta iznimka je dohvaćanje naredbe iz memorije. Tada se ona ne spremi u DR, nego izravno u naredbeni registar IR (Instruction Register)



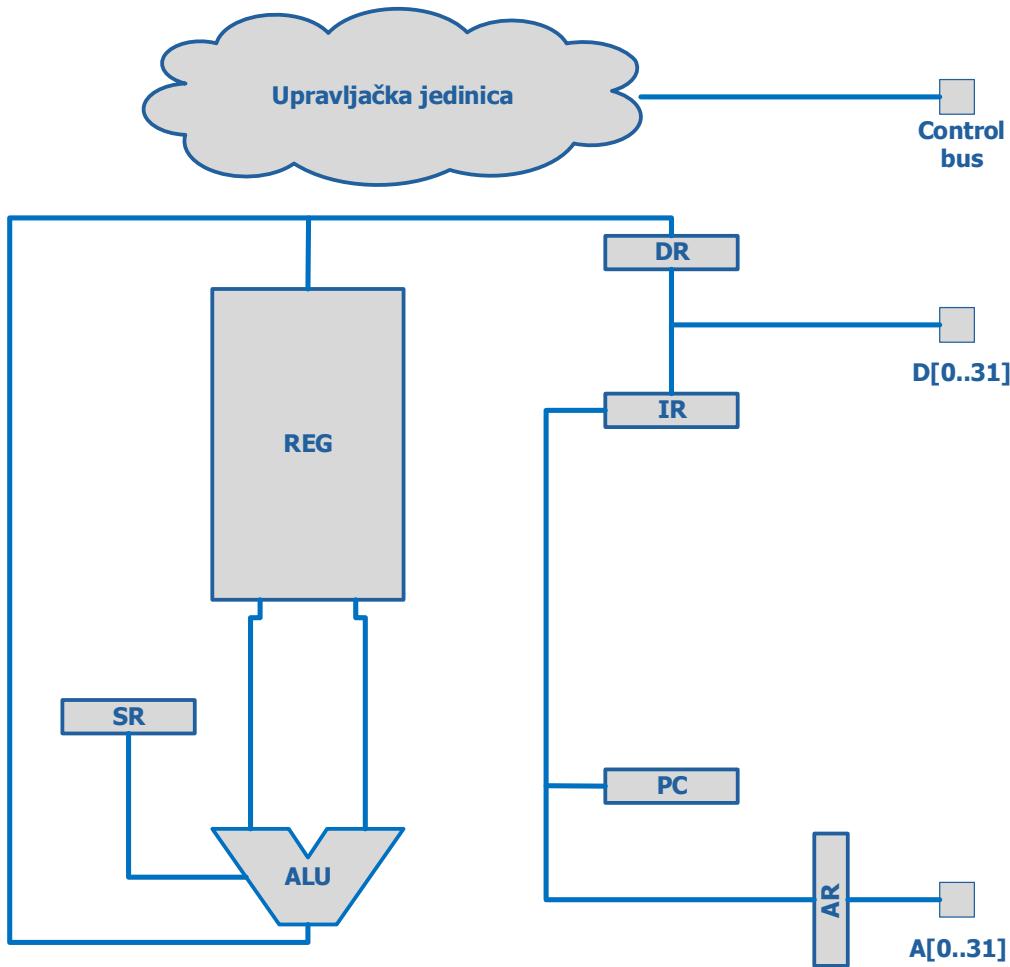
Za vrijeme dekodiranja naredbe njezin strojni kôd je spremljen u registru IR i čita ga upravljačka jedinica (veza od IR do upravljačke jedinice nije prikazana na slici).



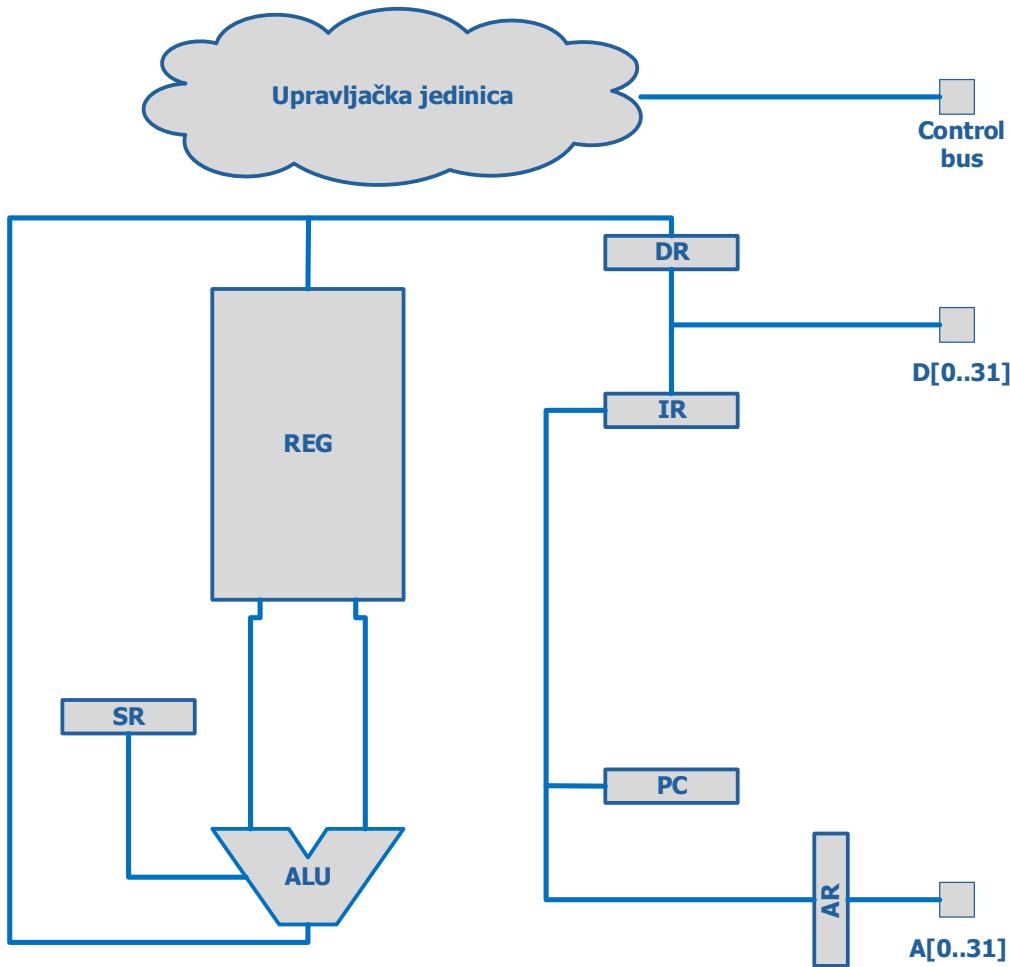
Adresni registar AR (Address Register) služi kao međuspremnik između unutrašnjosti procesora i adresne sabirnice, koja je spojena na procesor preko adresnih priključaka.



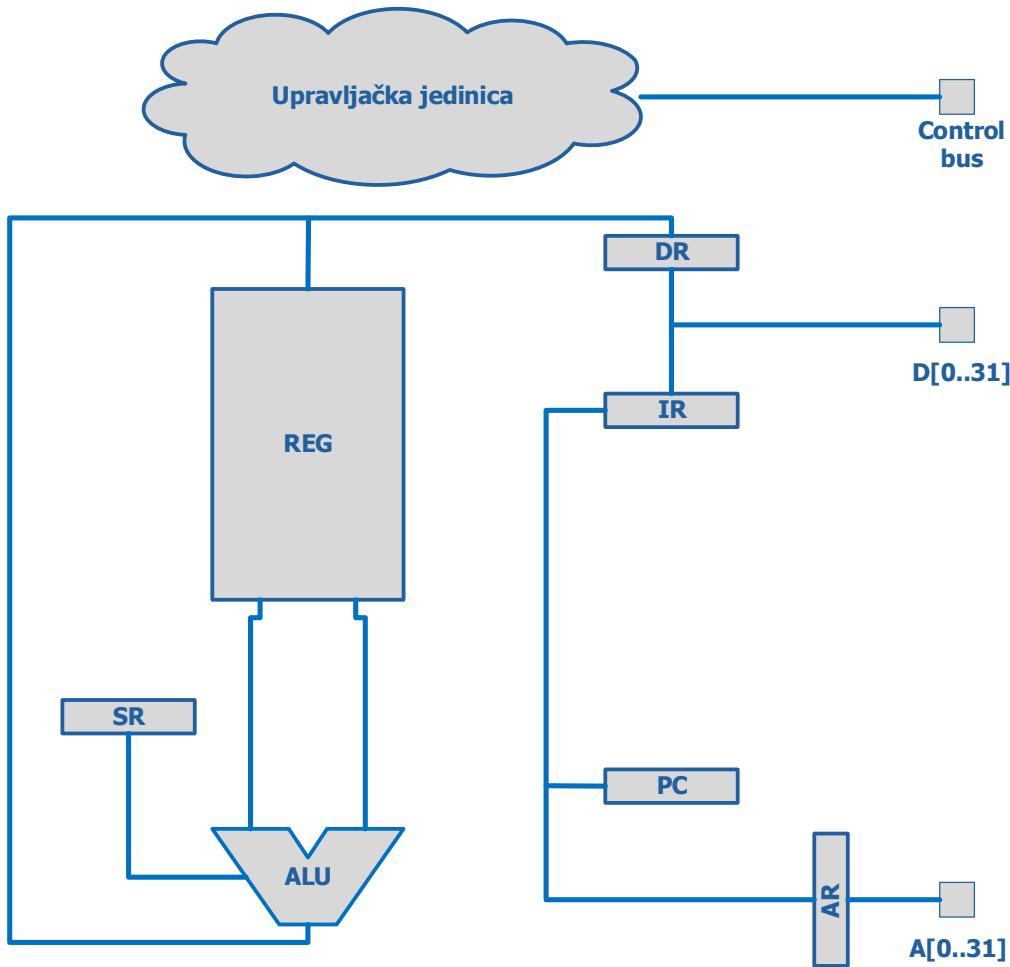
Registar AR uvijek se koristi kad procesor pristupa memoriji. U AR se spremi adresa memorijske lokacije koju procesor želi čitati ili pisati.



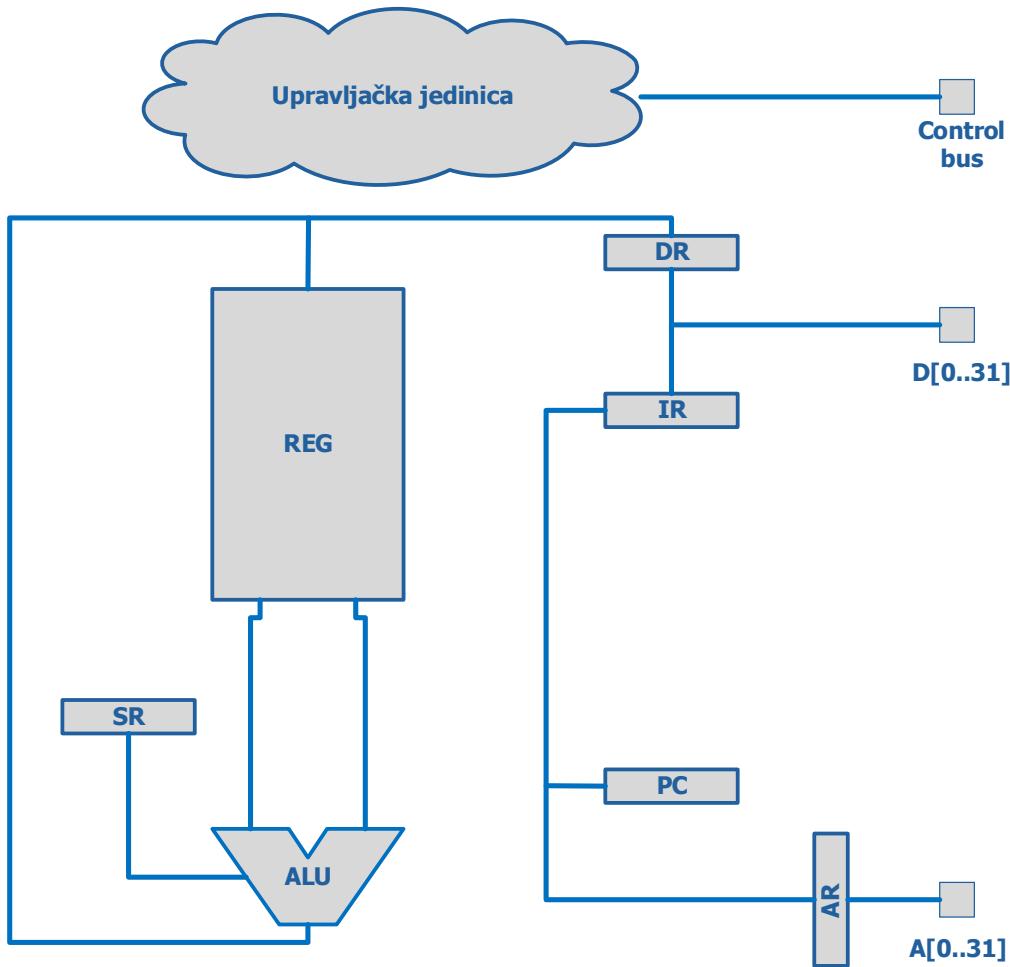
Postoji spojni put između registra DR i općih registara jer tim putom prolaze podatci prilikom izvođenja naredaba LOAD i STORE



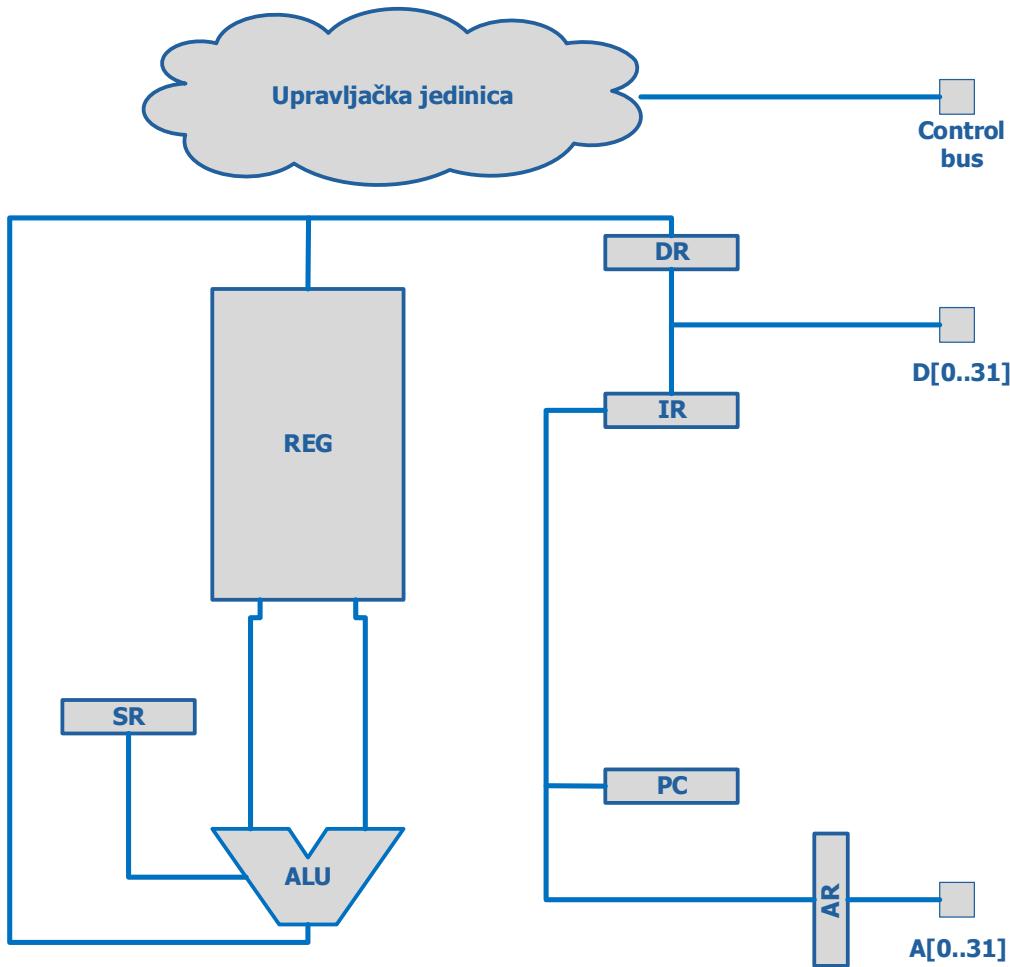
Registar IR u sebi čuva strojni kôd u kojem može biti adresa za naredbu skoka JP. Ta adresa šalje se u PC.



Postoji spojni put između registra PC i AR jer se tim putom šalje adresa sljedeće naredbe koju želimo dohvatiti iz memorije



Registar IR u sebi čuva strojni kôd u kojem može biti adresa za naredbe LOAD/STORE. Ta adresa šalje se u adresni registar AR.



Veze od upravljačke jedinice do ostalih dijelova procesora nisu prikazane jer ih ima previše - svaki dio upravljan je s jednim ili više signala koji dolaze iz upravljačke jedinice

Načelna mikroarhitektura

- Za sada će nam ovo objašnjenje mikroarhitekture biti dovoljno
- Novi važni dijelovi procesora koje smo upravo uveli su:
 - **DR** - podatkovni registar koji služi kao međusklop između unutrašnjosti procesora i podatkovne sabirnice
 - **AR** - adresni registar koji služi kao međusklop između unutrašnjosti procesora i adresne sabirnice
 - **IR** - naredbeni registar koji čuva strojni kôd naredbe za vrijeme njenog dekodiranja
 - Ovi registri su interni, u smislu da programer nema izravnog pristupa do njih

Detaljnija arhitektura i proširenja skupa naredaba

Proširenja skupa naredaba

- Do sada uvedene naredbe omogućuju izvođenje većine zadaća koje nam mogu zatrebatи
- Ipak, da bi programiranje bilo lakše, uvest ćemo još nekoliko naredaba i načina adresiranja
 - način adresiranja = način na koji se pristupa podatku
- Pri tome moramo voditi računa o strojnim kôdovima, jer oni također ograničavaju: broj naredaba, broj načina adresiranja, vrste operanada, širine podataka, širine adresa itd.
- Pogledajmo neke praktične zadaće koje bi mogli lakše riješiti s drugačijim ili novim naredbama ...

Proširenja aritmetičko-logičkih naredaba

Proširenja AL-naredaba

- Prepostavimo da treba registar R0 uvećati za 20. Sada je moguće ovakvo rješenje:

```
LOAD  R1, (100)
ADD   R0, R1, R0
      . . .
```

100 DW 20 ; na adresi 100 nalazi se broj 20

- Loše strane ovog rješenja:
 - Potrebne su dvije naredbe (brzina i zauzeće memorije)
 - Potrebna je dodatna memorijska lokacija (s brojem 20)
 - Treba koristiti dodatni registar (npr. R1). Moguće je da on nije slobodan, pa će ga trebati spremiti i kasnije obnoviti (za što trebaju još dvije dodatne naredbe i još jedna dodatna memorijska lokacija)

Proširenja AL-naredaba

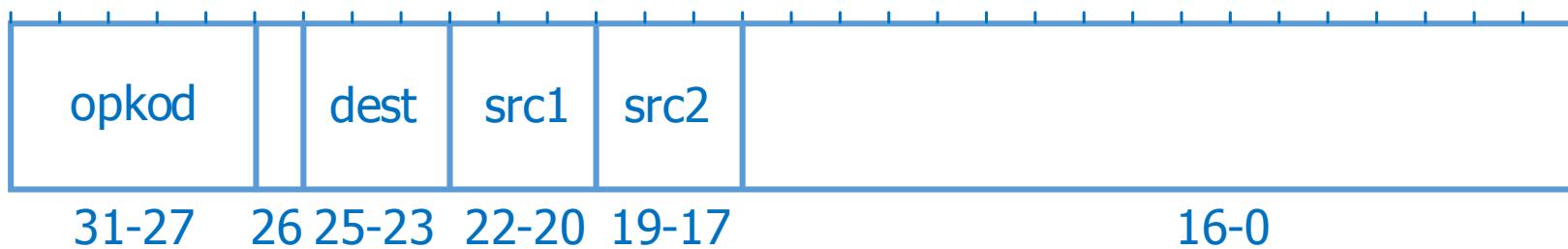
- Bolje rješenje bi moglo izgledati ovako:

ADD R0 , 20 , R0

- Dobre strane ovog rješenja:
 - Potrebna je samo jedna naredba
 - Ne trebaju dodatne memorijske lokacije
 - Ne treba koristiti dodatne registre
- Ali, naredba ADD se komplikira:
 - Operandi više nisu samo registri, nego mogu biti i brojevi
- Jedno od pravila pri oblikovanju procesora:
 - Ubrzati rad onoga što se često koristi

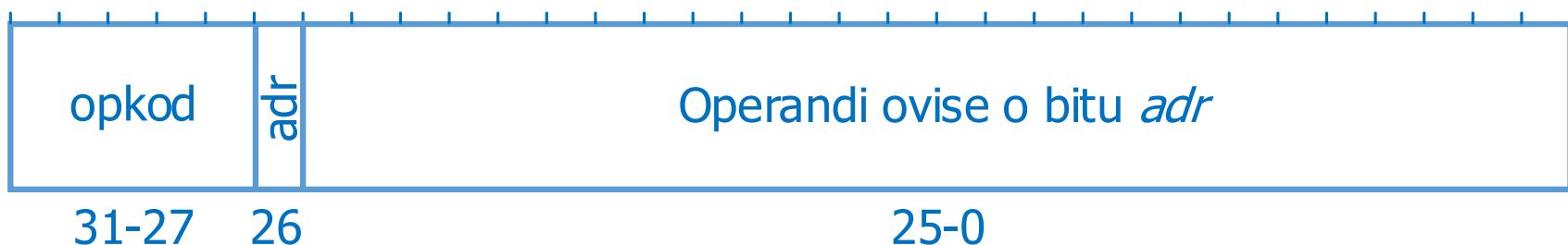
Proširenja AL-naredaba

- Proširimo naredbu ADD tako da kao drugi operand može imati:
 - registar (kao i do sada)
 - broj (novo)
- Moramo vidjeti ima li mjesta za ovakvo proširenje u strojnom kôdu i kako će on sada izgledati
- Do sada smo imali ovakav strojni kôd:



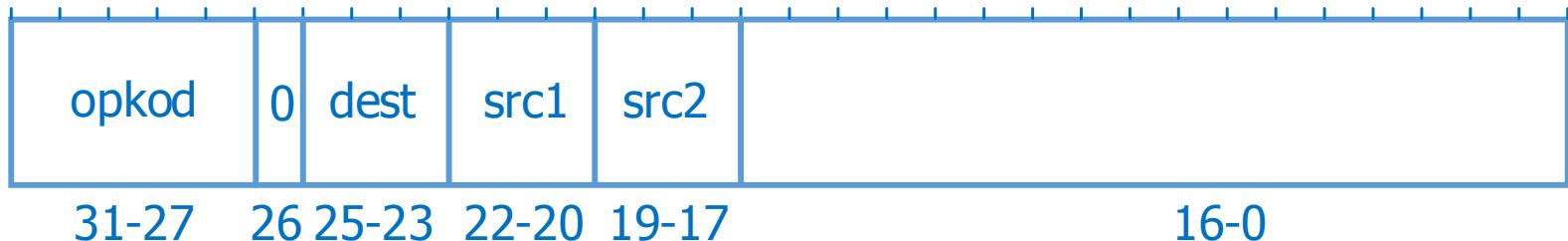
Proširenja AL-naredaba

- Moramo imati različit strojni kôd za različite vrste operanada:
 - Neiskorišteni **bit 26 (adr)** će označavati o kojem se obliku naredbe radi (to je dodatno 1-bitno polje koje određuje **način adresiranja**):
 - bit 26 u 0 (nuli) označava da su oba operanda registri
 - bit 26 u 1 (jedinici) označava da je drugi operand broj

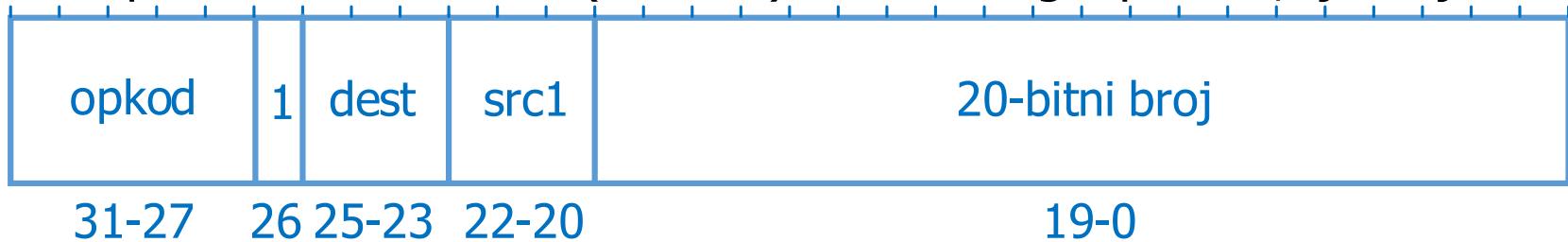


Proširenja AL-naredaba

- Kad su oba operanda registri, strojni kôd ostaje kao prije s razlikom da je bit adr (26) u nuli:



- Kad je drugi operand broj, bit adr (26) je u jedinici, a zatim redom kodiramo
 - tri bita (23 do 25) za prvi odredište (dest)
 - tri bita (20 do 22) za prvi operand (src1)
 - preostalih 20 bitova (0 do 19) sadrže drugi operand, tj. broj



Proširenja AL-naredaba

- Teorijski bi mogli imati i oblike:
 - **ADD 20, R1, R2** - prvi operand je broj, a drugi je registar
 - **ADD 20, 30, R4** - oba operanda su brojevi
- Međutim, to ima nedostataka:
 - Osim bita 26, trebao bi dodatni bit za odabir vrste prvog operanda
 - Ne dobiva se na fleksibilnosti naredaba
 - Oblik s dva podatka je besmislen jer troši vrijeme na izračunavanje konstante vrijednosti, a u strojnom kôdu i nema mesta za dva broja
- Zato, ostajemo na samo dva predložena oblika, ali će oni **vrijediti za sve aritmetičko-logičke naredbe** (radi pravilnosti i jednostavnosti arhitekture)

Proširenja AL-naredaba - primjer

- Izračunati izraz $R0 := (R1+55)-(R2 \text{ xor } 4ABC)$.
Usporediti rješenje bez novih naredaba i rješenje korištenjem novouvedenog zadavanja podatka u AL-naredbama.

Bez novih naredaba

```
LOAD  R0, (KONST1)
ADD   R1, R0, R0
LOAD  R3, (KONST2)
XOR   R2, R3, R3
SUB   R0, R3, R0
```

; Podaci u memoriji

KONST1 DW 55

KONST2 DW 4ABC

S novim naredbama

ADD R1, 55, R0

XOR R2, 4ABC, R3

SUB R0, R3, R0

Prednosti u odnosu na prethodni program su očite:

- kraći program (3 naredbe prema 5)
- nema dodatnih podataka u memoriji (2 podatka)

Proširenja AL-naredaba - proširenje broja

- Aritmetičko-logička jedinica je 32-bitna pa, prema tome, očekuje 32-bitne operande
- U okviru strojnog kôda za kodiranje broja imamo na raspolaganju samo 20 bitova
- Što s gornjih 12 bitova koji nedostaju? Ove bitove treba nekako definirati.
- Procesor će 20-bitni broj (iz strojnog kôda) prije slanja u ALU **predznačno proširiti** na 32 bita

Proširenja AL-naredaba - proširenje broja

- DZ: proučiti dokument „Proširivanje 20 na 32 bita“

Sažetak:

- **Proširenje nulama čuva iznos NBC brojeva**
(ali ne čuva iznos 2'k brojeva)
- **Predznačno proširenje čuva iznos 2'k brojeva**
(ali ne čuva iznos NBC brojeva)
- U praksi je u aritmetičkim operacijama puno potrebnije imati i pozitivne i negativne brojeve, nego puni opseg NBC-a.
 - Zato naš procesor koristiti predznačno proširenje

Proširenja AL-naredaba - pomaci i rotacije

- U asemblerском програмирању често треба обавити разлиčite vrste pomaka i rotacija podataka:
 1. logički pomak ulijevo i udesno
 2. aritmetički pomak udesno
 3. rotacija ulijevo i udesno
 4. rotacija ulijevo i udesno kroz zastavicu
- Mogu se dozvoliti pomaci i rotacije само за један бит или за жељени број битова
- Теоријски би могли одабрати само две операције помака/ротирања, а остале остварити програмски.
 - То би било у складу с идејом "једнотавног процесора"

Proširenja AL-naredaba - pomaci i rotacije

- Međutim, da bi pojednostavnili programiranje i ubrzali izvođenje, odabrat ćemo nešto veći broj naredaba za operacije:
 1. logičkog pomaka ulijevo i udesno
 2. aritmetičkog pomaka udesno
 3. rotacije ulijevo i udesno
- Također ćemo definirati da se izlazni bit spremi u zastavicu C kako bi ga mogli jednostavno ispitati nakon naredbe

Proširenja AL-naredaba - pomaci i rotacije

- Definiramo pisanje i operande naredaba pomaka i rotacije:

SHL src1 , src2 , dest	logički pomak u lijevo (SHift Left)
SHR src1 , src2 , dest	logički pomak u desno (SHift Right)
ASHR src1 , src2 , dest	aritmetički pomak u desno(Arithmetic SHR)
ROTL src1 , src2 , dest	rotacija u lijevo (ROTate Left)
ROTR src1 , src2 , dest	rotacija u desno (ROTate Right)

- Podatak koji se pomiče/rotira uzima se iz prvog operanda (src1)
- Broj pomaka/rotacija zadan je drugim operandom (src2)
- Rezultat pomaka/rotacije stavlja se u treći operand (dest)

"pomakni podatak iz **src1 za **src2** bitova i spremi rezultat u **dest**"**

Proširenja AL-nar. za višestruku preciznost

- Dvije AL-naredbe koje služe za rad s podatcima u višestrukoj preciznosti su naredbe zbrajanja i oduzimanja s prijenosom:
 - ADC (add with carry)
 - SBC (subtract with carry)
- Kasnije ćemo vidjeti kako se ove naredbe koriste i kako točno rade
- Ove naredbe imaju jednake operande kao i obično zbrajanje i oduzimanje. Pišu se ovako:

ADC src1, src2, dest
SBC src1, src2, dest

Proširenja AL-naredaba - naredba usporedbe

- Zadnja AL naredba koju ćemo uvesti, a koja postoji u većini procesora je naredba za usporedbu dvaju brojeva CMP (compare)
- Ova naredba slična je naredbi za oduzimanje SUB, s razlikom da se rezultat oduzimanja zanemaruje i ne upisuje u jedan od općih registara (CMP nema treći operand)
- Ovo je ujedno i prednost naredbe CMP, jer su u većini slučajeva registri zauzeti s različitim podatcima i međurezultatima
- Naredbu CMP zapravo pozivamo zato da postavi zastavice koje ćemo nakon toga ispitati naredbom uvjetnog skoka
- Naredba se piše ovako (drugi operand je registar ili broj):

CMP src1, src2 ; src1-src2

Rekapitulacija: proširenja AL-naredaba

- Ovime smo kompletirali skup aritmetičko-logičkih naredaba, kojih sada ima 13:
 - **ADD , SUB , ADC , SBC**
 - **CMP**
 - **AND , OR , XOR**
 - **SHL , SHR , ASHR , ROTL , ROTR**
- Osim toga, kao drugi operand sada osim registra smijemo pisati i 20-bitni broj



Proširenja AL-naredaba - zastavice

- Većina novih naredaba ima isti utjecaj na postavljanje zastavica kao i ranije definirane naredbe:

ADD, ADC, SUB, SBC i CMP: C=prijenos, V=preljev, Z=nula, N=predznak

AND, OR i XOR: C=0, V=0, Z=nula, N=predznak

SHL, SHR, ASHR, ROTL, ROTR: C=izlazni bit, V=0, Z=nula, N=predznak

- Jedina novost kod postavljanja zastavica je u naredbama pomaka i rotacije gdje se u zastavicu C upisuje izlazni bit od zadnjeg koraka pomaka/rotacije.

Primjer

U registru R0 nalaze se dvije NBC poluriječi. Treba usporediti te poluriječi. Ako je viša poluriječ veća od niže, onda treba skočiti na adresu 100, a ako nije, onda treba skočiti na adresu 200.



Postupak: da bismo mogli obaviti usporedbu, moramo poluriječi imati u zasebnim registrima, npr. R0 i R1.



Proširenja AL-naredaba - primjer

Rješenje:

```
ROTR  R0, 10, R1      ; viša poluriječ u niži dio R1
AND   R0,0FFF,R0      ; više bitove R0 stavljamo u 0
AND   R1,0FFF,R1      ; više bitove R1 stavljamo u 0

CMP   R1, R0          ; usporedba

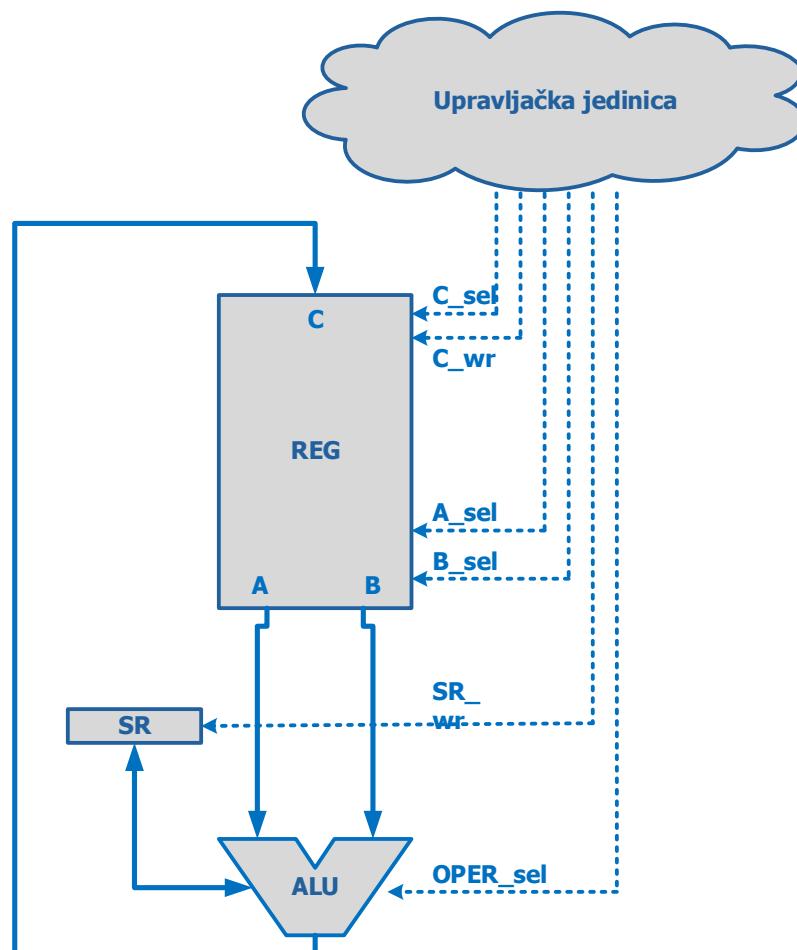
JP_UGT 100            ; viša p.r. > niža p.r. ⇒ "goto 100"
JP 200                ; viša p.r. <= niža p.r. ⇒ "goto 200"
```

Proširenja AL-naredaba - put podataka

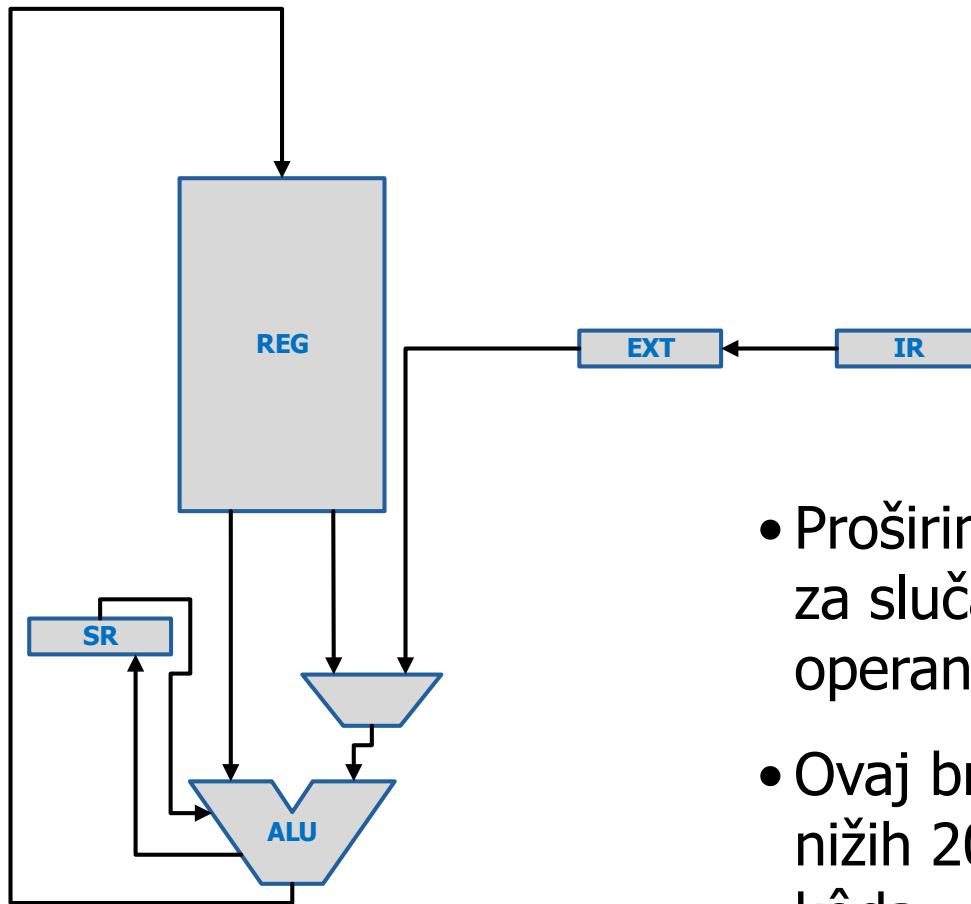
- Na kraju prethodnog poglavlja prikazali smo shematski mikroarhitekturu FRISC-a, bez ulazeњa u detalje
- U ovom poglavlju ćemo postupno uvesti detaljniji prikaz arhitekture koji se u literaturi obično naziva **put podataka** (engl. datapath)

>>>

Put podataka za AL naredbe



Proširenja AL naredaba



- Proširimo put podataka i za slučaj kad je drugi operand 20-bitni broj
- Ovaj broj je zapisan u nižih 20 bitova strojnog kôda

Uvođenje registrarskih naredaba

Uvođenje registrarskih naredaba

- Dosta česta zadaća koju treba obaviti je punjenje broja (konstante) u registar, na primjer:

- pri inicializaciji brojača za petlju
 - pri incijalizaciji varijable, itd.

- Sada to možemo riješiti pomoću naredbe LOAD:

```
LOAD    R1 , (BROJ)
BROJ    DW      32
```

- Loše strane ovog rješenja su:

- Potrebna je memorijska naredba (vidjet ćemo kasnije da se memorijske naredbe izvode sporije od npr. AL-naredaba)
 - Potrebna je dodatna memorijska lokacija (s brojem)

Uvođenje registrarskih naredaba

- Koji puta (ali ne previše često) treba vrijednost iz jednog registra upisati u drugi registar
 - Sada to možemo riješiti pomoću AL-naredaba, npr. naredbom OR, AND, ADD, SUB ili ROTL/ROTR. Pokažimo kako bi upisali vrijednost iz R1 u R2:

OR	R1 ,R1 ,	R2
AND	R1 ,R1 ,	R2
ADD	R1 ,0 ,	R2
SUB	R1 ,0 ,	R2
ROTL	R1 ,0 ,	R2

- Nije estetski, ali uglavnom funkcionira ☺ ...

Uvođenje registrarskih naredaba

- Međutim, AL-naredbe mijenjaju zastavice u registru SR.
To koji puta može biti nepoželjno.
- Ne znamo kako spremiti i obnoviti vrijednost registra SR



Uvođenje registrarskih naredaba

- Zbog svega navedenog, uvodimo novu naredbu MOVE
- Ona po svojim značajkama ne pripada ni jednoj postojećoj skupini naredaba (AL, memorijske, upravljačke)
 - svrstat ćemo je u zasebnu skupinu registrarskih naredaba (jer ona prvenstveno radi s registrima)
- Definiramo pisanje i operande naredbe MOVE:

MOVE src , dest

- src - izvor podatka; može biti: SR, R0-R7 ili 20-bitni podatak koji se predznačno proširuje na 32 bita
- dest - odredište podatka; može biti: SR ili R0-R7

Uvođenje registrarskih naredaba

- Registrar SR je 5-bitni (to još ne znamo, ali tako će ispasti ☺) pa moramo definirati kako MOVE barata s podatcima različitih širina (Ri označuje opći register R0-R7):
 - **MOVE podatak, SR** → nakon predznačnog proširenja podatka na 32 bita, odbacuje se gornjih 24 bita i u SR se puni samo najnižih 5 bita podatka
 - **MOVE Ri, SR** → u SR se puni najnižih 5-bitu iz registra Ri
 - **MOVE SR, Ri** → SR se puni u najniže bitove od Ri, a viši bitovi se pune nulama

Uvođenje registrarskih naredaba - primjer

Primjer učitavanja brojeva u registre:

U registar R0 treba upisati broj 12345, u registar R1 broj FFFF1234, u R2 broj -100A, u R3 broj 12345678, a u registar R4 broj 9ABCDEF0.

- ; Prvi način je izravno korištenje naredbe MOVE.
- ; Ovo je pogodno za brojeve malog absolutnog iznosa.
- ; "Mali brojevi" su oni koji "stanu" u strojni kod.

MOVE 12345, R0	; 12345 stane u 17+1 bit
MOVE 0FFF1234, R1	; isto kao -EDCC, a ; EDCC stane u 16+1 bit
MOVE -100A, R2	; 100A stane u 13+1 bit

>>>

Uvođenje registrarskih naredaba - primjer

- ; Drugi način je pogodan za "velike brojeve" koji ne
- ; "stanu" u strojni kod. Broj se "puni" u registar u
- ; više koraka: Na primjer, kombinacijom MOVE i ADD

```
MOVE 1234, R3          ; 12345678 stane u 29+1 bit
ROTL R3, %D 16, R3
ADD  R3, 5678, R3
```

- ; Treći način za "velike brojeve" znamo od prije:
- ; upotreba naredbe LOAD i dodatne memorijske lokacije

```
LOAD R4, (BROJ)
BROJ DW    9ABCDEF0
```

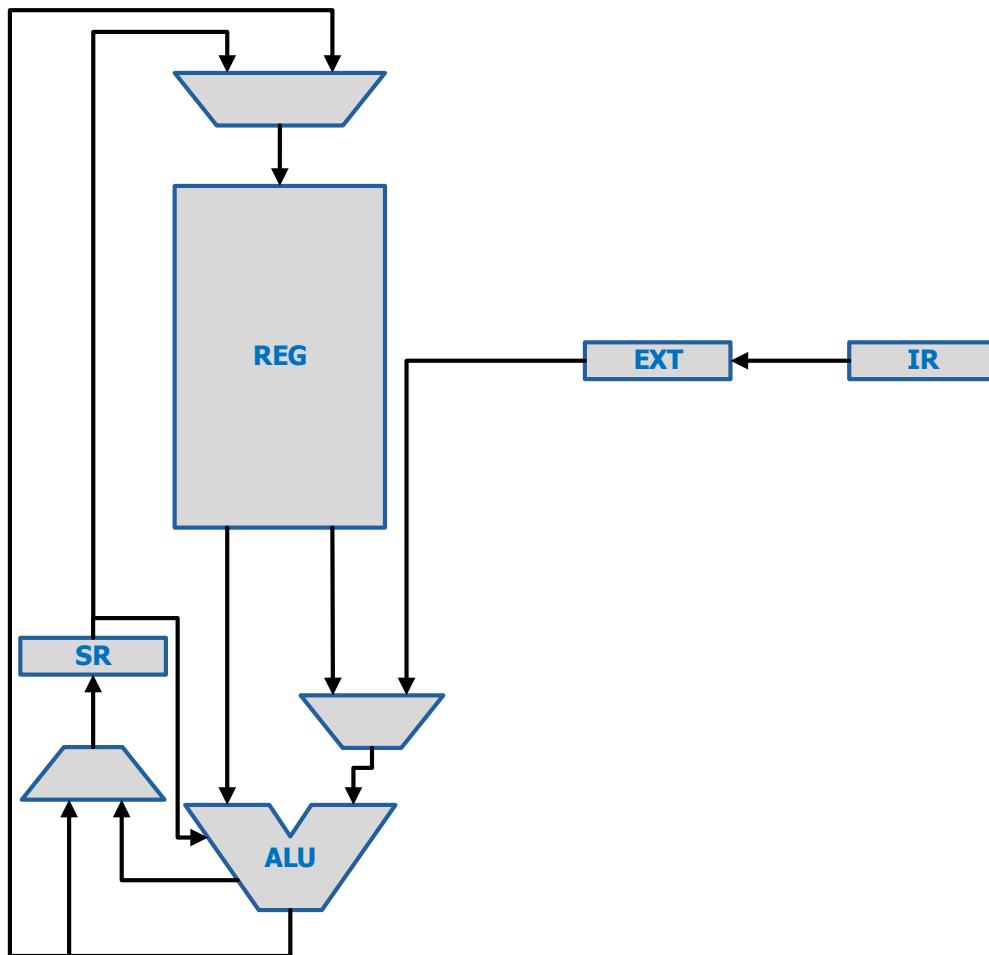
Digresija: 20-bitni brojevi u naredbama

- 20-bitni broj iz strojnog kôda se prilikom izvođenja predznačno proširuje na 32 bita
 - Dakle, dobiveni 32-bitni broj će sigurno u svih gornjih 13 bitova imati ili sve nule ili sve jedinice (ovisno o najvišem bitu 20-bitnog broja)
- Prilikom pisanja programa mogu se pisati pozitivni i negativni brojevi: 123, -2, FFFF5678 itd.
 - Asemblerski prevoditelj svaki ovaj broj prvo pretvori u 32-bitni zapis:
 - 32-bitni NBC ako je broj pozitivan
 - 32-bitni 2'k zapis ako je broj negativan
 - Ako su u dobivenom 32-bitnom zapisu gornjih 13 bitova isti, onda je sve u redu i u strojni kôd se upisuje najnižih 20 bitova 32-bitnog zapisa
 - Ako gornjih 13 bitova nisu isti, onda je to pogrešno napisana naredba (poruka: wrong number)

Uvođenje registrarskih nar. - strojni kôd

- Proučite u knjizi objašnjenje načina formiranja strojnog koda za prethodne naredbe

Registarske naredbe – put podataka



Proširenje memorijskih naredaba

Proširenje memorijskih naredaba

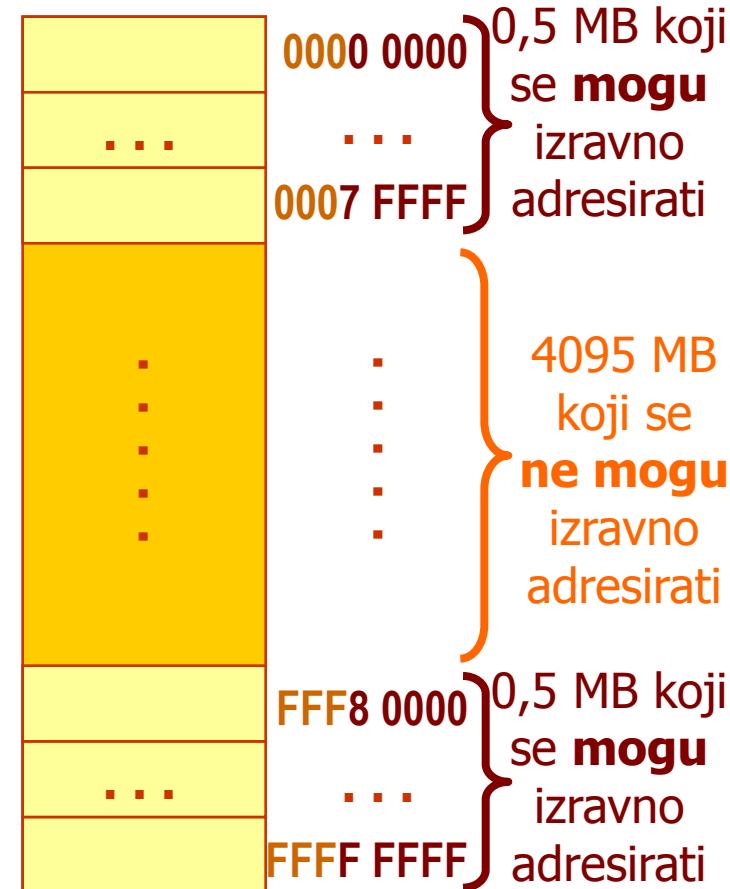
- Podsjetimo se da su memorijske naredbe LOAD i STORE imale zadavanje adrese **brojem**
- U strojnom kôdu je za adresu bilo na raspolaganju samo 20 bitova
- Budući da smo za adresnu sabirnicu odabrali širinu od 32 bita, **moramo** točno definirati vrijednost gornjih 12 bitova

Proširenje memorijskih naredaba

- U strojnom kôdu ostalih naredaba postoji pravilnost:
 - AL-naredbe mogu imati u nižih 20 bitova zadan 20-bitni broj koji će biti drugi operand. Ovaj broj se predznačno proširuje prije dovođenja u ALU
 - Registarske naredbe mogu imati u nižih 20 bitova zadan broj koji se predznačno proširuje i stavlja u odredišni registar
 - Zbog pravilnosti arhitekture definirat ćemo da se i u naredbama LOAD i STORE **konačna 32-bitna adresa dobiva predznačnim proširivanjem 20-bitne adrese iz strojnog kôda**

Proširenje memorijskih naredaba - adrese

- Adresama 0000 0000 do 0007 FFFF adresiramo najnižih 219 lokacija memorije (najnižih pola MB)
- Adresama FFF8 0000 do FFFF FFFF adresiramo najviših 219 lokacija memorije (najviših pola MB), što ukupno daje 220 lokacija (tj. jedan MB)
- Na ovaj način ne možemo adresirati svih 232 lokacija (tj. 4 GB ili 4096 MB)



Proširenje memorijskih naredaba - adrese

- Adresni prostor od 1 megabajta je dovoljan za potrebe ugradbenog računala, ali ne i za opću upotrebu procesora
- Trebamo zadavanje bilo koje 32-bitne adrese
- Moguća rješenja
 - Proširiti strojni kôd (nekih) naredaba tako da zauzimaju dvije riječi i u drugu riječ staviti 32-bitnu adresu
 - Upotrijebiti jedan od općih registara za adresiranje
- Budući da smo odlučili da sve naredbe budu jednake širine, odabiremo drugo rješenje

Proširenje memorijskih naredaba

- Sada ćemo moći naredbe LOAD i STORE pisati na primjer ovako:

```
LOAD R0 , (R5)
STORE R1 , (R4)
STORE R2 , (R3)
```

DZ:

Proučite u knjizi način formiranja strojnih kodova memorijskih naredaba

Primjer rada s adresama većim od 20 bita

Zamijeniti vrijednosti memorijskih lokacija s adresa 200 i 300000:

```
LOAD R2, (200)          ; Dohvati prvi podatak.  
LOAD R0, (A)            ; Stavi broj 300000 u R0 i s  
LOAD R3, (R0)           ; njim adresiraj drugi podatak.  
  
STORE R3, (200)  
STORE R2, (R0)  
  
; U memoriji na adresi A mora biti upisan  
; broj 300000 koji ćemo koristiti kao adresu:  
  
A      DW 300000 ; Služi kao adresa za drugi podatak  
...  
200    DW 1234   ; Prvi podatak.  
...  
300000 DW 2468   ; Drugi podatak.
```

Proširenje memorijskih nar. - odmak

- U srojnom kodu naredbe kod dresiranja regsitrom nam ostaje neiskorištenih 20 najnižih bitova



- Naredbu možemo učiniti još fleksibilnijom i praktičnijom za upotrebu ako **uvedemo 20-bitni odmak** (engl. offset)
- Dobivamo konačni drugi oblik memorijskih naredaba u kojem kodiramo adresni register (adrreg), ali i dodatni 20-bitni odmak

Proširenje memorijskih naredaba - odmak

- Konačno, drugi oblik memorijskih naredaba izgleda ovako:

LOAD R1 , (R4+20)

STORE R2 , (R5-10)

STORE R3 , (R6) odmak=0 , ne mora se pisati

- Adresa se tijekom izvođenja formira na sljedeći način:

1) 20-bitni odmak se predznačno proširi do 32 bita,

2) Vrijednost adresnog registra zbroji se s 32-bitnim odmakom,

3) Ovaj zbroj je konačna 32-bitna adresa za memoriju.

Proširenje memorijskih nar. - širina podatka

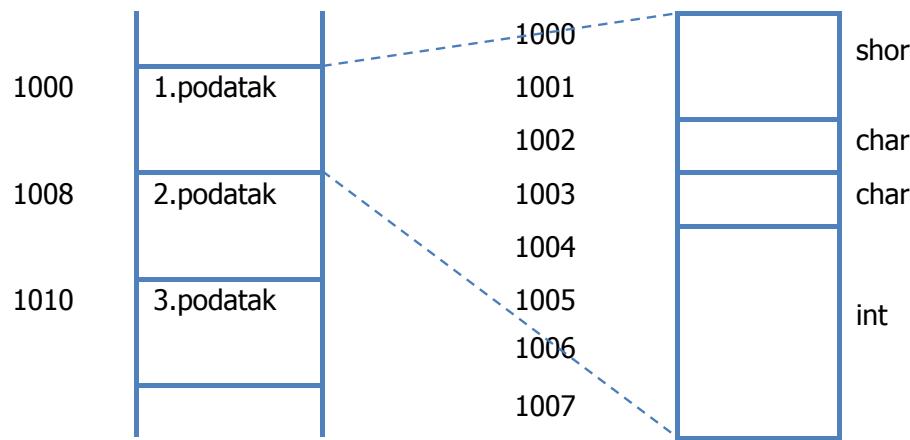
- Naredbe LOAD i STORE rade s 32-bitnim podatcima koji su u memoriji zapisani u četiri uzastopne memorijske lokacije
- Da bi mogli pristupati pojedinim memorijskim lokacijama, tj. bajtovima, **dodat ćemo još i naredbe LOADB i STOREB** (load byte i store byte) koje imaju jednake operande kao i LOAD i STORE
- također **dodajemo naredbe LOADH i STOREH** (load halfword i store halfword) koje imaju jednake operande kao i LOAD/STORE te LOADB/STOREB

Proširenje memorijskih nar. - širina podatka

- U pristupanju bajtovima, riječima i poluriječima obično postoji ograničenja pa ih i mi uvodimo:
 - Riječi imaju adrese djeljive s 4
 - Poluriječi imaju adrese djeljive s 2
 - Bajtovi nemaju ograničenja na adresu
- U nekim procesorima će u slučaju zadavanja pogrešne adrese doći do tzv. iznimke. Budući da želimo imati jednostavan procesor, mi ćemo izbjegći ovakvu mogućnost na sljedeći način:
 - FRISC automatski stavlja nule u dva najniža bita adrese prilikom izvođenja naredbe LOAD/STORE
 - FRISC automatski stavlja nulu u najniži bit adrese prilikom izvođenja naredbe LOADH/STOREH
 - FRISC ne mijenja adresu prilikom izvođenja naredbe LOADB/STOREB

Primjer

- U memoriji se od adrese 1000 nalazi niz 64-bitnih složenih podataka kao na slici (npr. C struktura sa short-om, dva char-a i int-om)



- Treba kopirati 16 i 32-bitni dio iz prvog podatka u nizu u treći podatak u nizu, a 8-bitne dijelove treba kopirati iz trećeg podatka u prvi.

; registri za adresiranje:

MOVE 1000, R1 ; adresa prvog podatka

MOVE 1010, R3 ; adresa trećeg podatka

; kopiraj polurićeč iz 1. u 3. strukturu

LOADH R5, (R1)

STOREH R5, (R3)

; kopiraj oktete iz 3. u 1. strukturu

LOADB R5, (R3+2)

STOREB R5, (R1+2)

LOADB R5, (R3+3)

STOREB R5, (R1+3)

; kopiraj riječeč iz 1. u 3. strukturu

LOAD R5, (R1+4)

STORE R5, (R3+4)

Proširenje memorijskih naredaba - stog

- Većina procesora koristi stog za spremanje podataka i povratnih adresa iz potprograma i prekidnih potprograma
- Već smo vidjeli što je stog i dvije osnovne operacije za stavljanje i uzimanje podataka sa stoga - PUSH i POP
- Da bi omogućili rad sa stogom, **uvest ćemo naredbe PUSH i POP** koje će:
 - stavljati na stog podatak iz jednog od općih registara (PUSH)
 - uzimati podatak sa stoga i stavljati ga u jedan od općih registara (POP)
- Budući da se stog nalazi u memoriji, PUSH i POP ćemo svrstati u skupinu memorijskih naredaba

Proširenje memorijskih naredaba - stog

- Podsjetnik: stog se nalazi u memoriji, a u svakom trenutku moramo znati adresu vrha stoga
 - Trebamo pokazivač stoga SP u kojem će se pamtitи adresa vrha stoga
- Mogli bi uvesti posebni 32-bitni registar za tu namjenu
 - Tada bi morali imati i posebne naredbe koje bi mogle upisivati vrijednost u SP i čitati vrijednost registra SP
 - Time bi donekle zakomplicirali arhitekturu i proširili skup naredaba
- Zato ćemo "žrtvovati" jedan od općih registara i dodijeliti mu posebnu ulogu pokazivača stoga - to će biti R7:
 - R7 će se moći nazivati alternativnim imenom SP
 - Inače se R7 može ravnopravno koristiti u ostalim naredbama kao i preostali registri R0 do R6

Proširenje memorijskih naredaba - stog

- SP pokazuje na zadnji podatak na stogu
- Definiramo pisanje i operande naredaba PUSH i POP:

PUSH src

POP dest

- src - opći register iz kojeg se uzima podatak koji se stavlja na stog
- dest - opći register u kojem se stavlja podatak sa stoga

PUSH src:

R7-4 → R7

src → (R7)

POP dest:

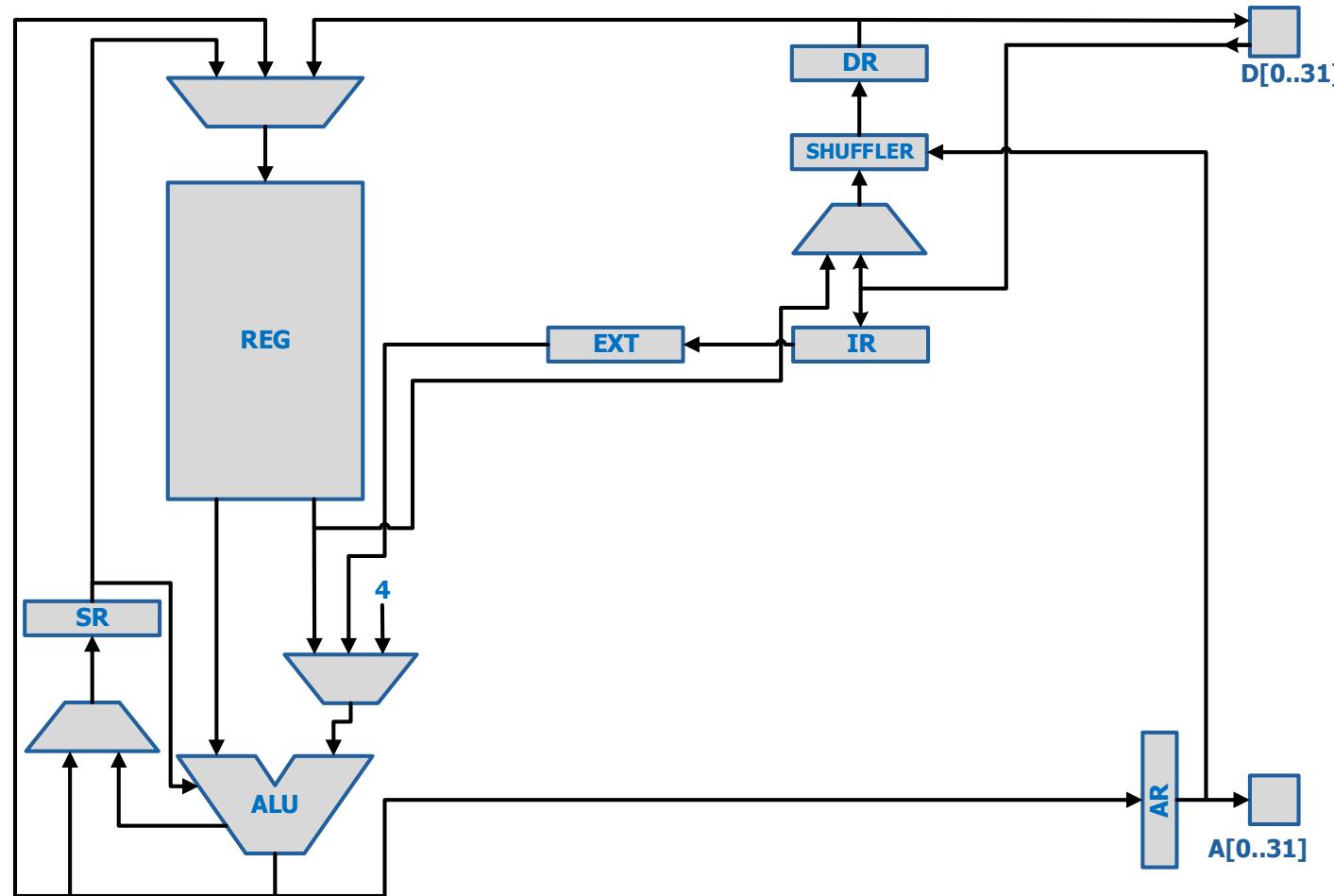
(R7) → dest

R7+4 → R7

Proširenje memorijskih naredaba - stog

- Uočite da se R7 smanjuje i povećava za četiri, jer se na stog **uvijek stavlja i uzima 32-bitni podatak**
- Slično naredbama LOAD i STORE, da bi se izbjegla greška u naredbama PUSH i POP, automatski se stavljaju nule na najniža dva bita adrese
- Proučite u knjizi strojne kodove za prethodne naredbe

Put podataka...



Proširenje upravljačkih naredaba

Proširenje upravljačkih naredaba

- Za sada imamo samo naredbu skoka JP u kojoj adresu skoka zadajemo 20-bitnim brojem
- Budući da adresa mora biti 32-bitna, ponovno moramo definirati što će biti u viših 12 bitova
- Zbog pravilnosti i jednostavnosti arhitekture, i ovdje će se 20-bitna adresa iz strojnog kôda predznačno proširiti da bi se dobila konačna 32-bitna adresa
 - ovo postavlja ista ograničenja kao i u naredbama LOAD i STORE
 - izravno se može adresirati samo najnižih i najviših 0,5 megabajta memorije, što znači da se samo na tim lokacijama može nalaziti odredište skoka

Proširenje upravljačkih naredaba

- Da bi se moglo skočiti na bilo koju memorijsku adresu, upotrijebit ćemo istu ideju kao kod naredba LOAD/STORE:
 - Jedan registar upotrijebit ćemo kao adresni register koji će svojim 32-bitnim sadržajem zadati adresu skoka na bilo kojoj memorijskoj lokaciji u prostoru od 4 GB

Proširenje upravljačkih naredaba - primjeri

Primjer skoka na 32-bitnu adresu:

Treba skočiti na adresu 200000.

```
LOAD R0, (A_200000)    ;;; Priprema adrese u R0
JP  (R0)                ;;; Skok "na R0"
...
```

;; Na adresi A_200000 nalazi se adresa skoka 200000
A_200000 DW 200000

;; Na adresi 200000 nalazi se
;; dio programa na koji skačemo...

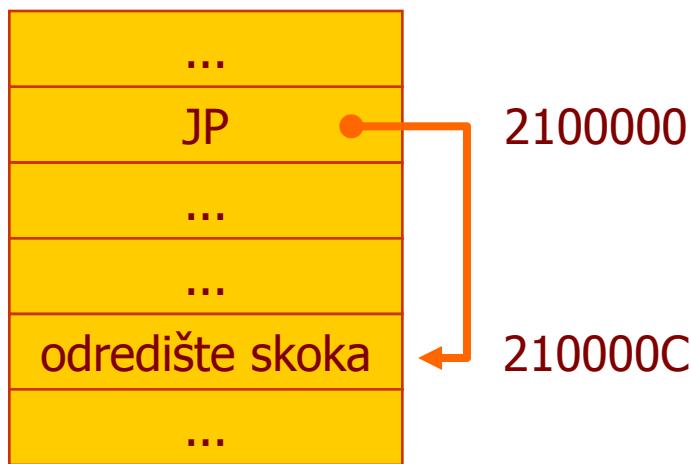
```
200000      ADD R2, R3, R4      ; neka naredba
...
...
```

Proširenje upravljačkih nar. - relativni skok

- Vidimo da način skakanja nije najpraktičniji, ali funkcioniра:
 - možemo s bilo kojeg mesta u memoriji skočiti na bilo koje drugo mjesto
- Nepraktično je:
 - za skok trebaju dvije naredbe i jedan slobodni registar
 - za skok treba dodatna memorijska lokacija u kojoj je adresa skoka (praktično je da je ova lokacija bude negdje unutar memorije koja se može adresirati s 20 bitova, jer inače i nju moramo dohvaćati indirektno preko nekog drugog регистра)
- Pogledajmo kada je ovakav način skakanja naročito nepraktičan i koje je moguće rješenje...

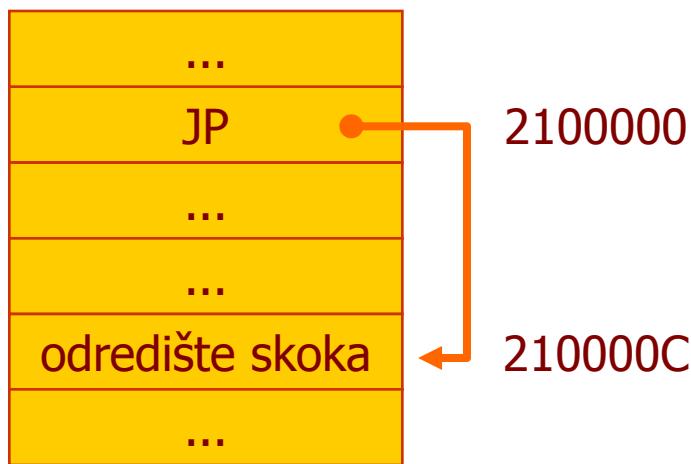
Proširenje upravljačkih nar. - relativni skok

- Iako korištenjem registra možemo skočiti bilo kuda, postoje slučajevi kad je to naročito nepraktično:



- Prepostavimo da na procesor spojimo npr. 256MB memorije
- Prepostavimo da na adresi 2100000 (u "sredini memorije") imamo neki dio programa u kojem se nalazi npr. petlja sa dvadesetak naredaba ili npr. naredba skoka kojom želimo preskočiti 2 naredbe (kao na slici)
- Ovakvi **kratki skokovi** su najčešći u programima

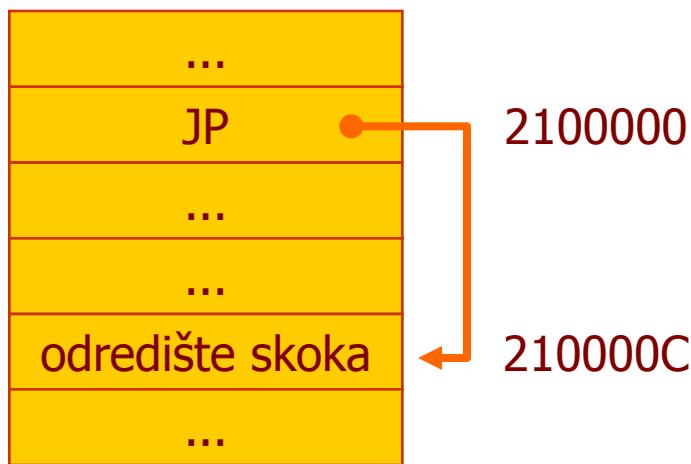
Proširenje upravljačkih nar. - relativni skok



- Budući da ne možemo zadati adresu sa 20-bitnim brojem, morali bi koristiti registar i dodatnu memorijsku lokaciju
- Budući da je program prepun ovakvih kratkih skokova, ovo bi bilo vrlo nepraktično i neefikasno

.....

Proširenje upravljačkih nar. - relativni skok



- Dodatna nepraktičnost: ne možemo koristiti labelu za skok (labele povećavaju čitljivost)
- Također treba znati adresu na koju želimo skočiti (Znamo li je? Teorijski znamo, ali u praksi NE - zato što obično ne vodimo računa na kojim adresama se nalaze naredbe)
- **Postoji li bolje rješenje?**

Proširenje upravljačkih nar. - relativni skok



- **Bolje rješenje je relativni skok**

- Relativni skok je takav skok kod kojeg znamo početni položaj naredbe skoka i "udaljenost" odredišta skoka
- Početni položaj zapisan je u registru PC (koji pokazuje jednu naredbu dalje od naredbe JP)

Proširenje upravljačkih nar. - relativni skok



- U naredbi relativnog skoka treba zadati samo **udaljenost** odredišta skoka
- S obzirom da su skokovi većinom kratki, onda nam je npr. 20-bitna udaljenost i više nego dovoljna

Proširenje upravljačkih nar. - relativni skok

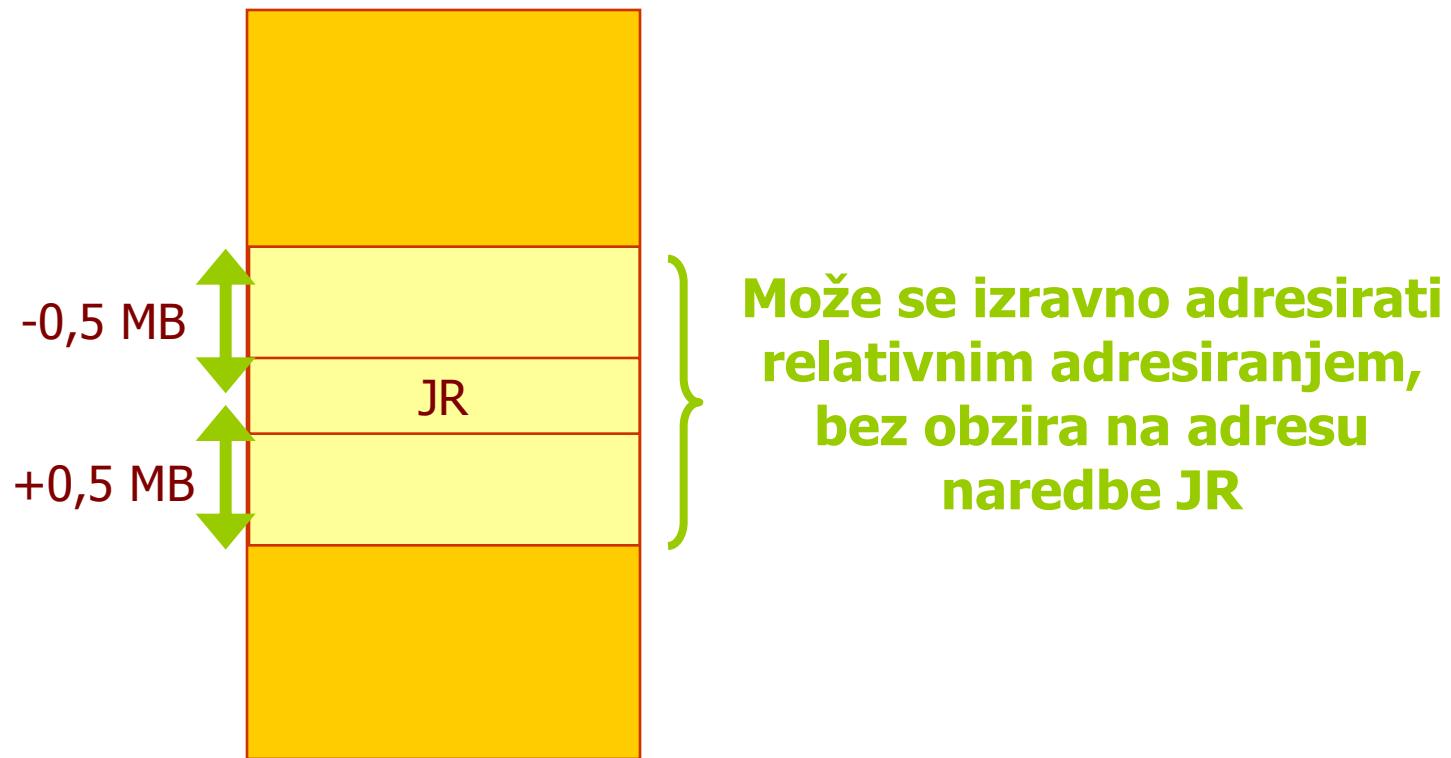
- **Uvodimo naredbu JR** (jump relative) kojoj se odredište skoka ne zadaje apsolutnom adresom, nego odmakom od same naredbe JR
- Odmak je 20-bitni i predznačno se proširuje čime se dobivaju skokovi "unaprijed" i "unatrag"
- Programer ne mora izračunavati ovaj odmak (udaljenost, engl. offset), nego se za to brine asemblerski prevoditelj, a programer piše adresu skoka (obično kao labelu)
- Naredba može koristiti uvjet i piše se ovako:

JR adresa

JR uvjet adresa

Proširenje upravljačkih nar. - relativni skok

- Ovako definirana naredba daje mogućnost adresiranja 1 MB memorijskog prostora **u okolini trenutačne naredbe JR**:
- Moguć je skok unaprijed za otprilike 0,5 MB i unatrag za također 0,5 MB:



Proširenje upravljačkih naredaba - primjer

Primjer dugačkih i relativnih skokova:

Glavni program je smješten na "niskim adresama", a programski odsječak na "visokoj adresi" 300000.

Iz glavnog programa treba **skočiti na odsječak na adresu 300000**. Odsječak mora zbrojiti pet 32-bitnih podataka (**petlja**) počevši od adrese 1000 i staviti rezultat u R0. Nakon toga, odsječak **skače na labelu NASTAVI** koja se nalazi u glavnom programu.

Glavni program ("niske adrese")

Programski odsječak (na adresi 300000)

0,5 MB

4095 MB

0,5 MB



; Glavni program na nižim adresama memorije:

```
GLAVNI LOAD R1, (A_ODSJECAK)
          JP (R1) ←
          ...
          . . .
```

absolutno adresiranje je nemoguće
jer je adresa 300000 šira od 20
bitova, a relativno je nemoguće jer
je skok preduz

```
NASTAVI . . . ; neke naredbe
```

```
A_ODSJECAK DW 300000 ; adresa za skok na odsječak
1000        DW 12F,5B54C367,23A9,87DDB000,33578
```

; Na adresi 300000 nalaze se naredbe odsječka:

```
300000 MOVE 5, R5           ; brojač za petlju
          MOVE 0, R0           ; suma
          MOVE 1000, R1         ; adresa podataka      >>>
          .
          .
          .
PETLJA   LOAD  R2, (R1)
          ADD   R0, R2, R0
          ADD   R1, 4, R1
          SUB   R5, 1, R5
          JR_NZ PETLJA
```

```
          JP NASTAVI
```

; Glavni program na nižim adresama memorije:

```
GLAVNI LOAD R1, (A_ODSJECAK)
    JP (R1)
```

...

NASTAVI ... ; neke naredbe

```
A_ODSJECAK DW 300000 ; adresa za skok na odsječak
1000          DW 12F,5B54C367,23A9,87DDB000,33578
```

; Na adresi 300000 nalaze se naredbe odsječka:

```
300000 MOVE 5, R5           ; brojač za petlju
        MOVE 0, R0           ; suma
        MOVE 1000, R1         ; adresa podataka      >>>
```

```
PETLJA LOAD R2, (R1)
        ADD R0, R2, R0
        ADD R1, 4, R1
        SUB R5, 1, R5
        JR_NZ PETLJA
```

praktično je koristiti relativno adresiranje i naredbu JR, jer je skok kratak, a odredište skoka ima adresu širu od 20 bita

JP NASTAVI

; Glavni program na nižim adresama memorije:

```
GLAVNI LOAD R1, (A_ODSJECAK)
    JP (R1)
```

...

NASTAVI ... ; neke naredbe

```
A_ODSJECAK DW 300000 ; adresa za skok na odsječak
1000          DW 12F,5B54C367,23A9,87DDB000,33578
```

; Na adresi 300000 nalaze se naredbe odsječka:

```
300000 MOVE 5, R5           ; brojač za petlju
          MOVE 0, R0           ; suma
          MOVE 1000, R1          ; adresa podataka
```

```
PETLJA LOAD R2, (R1)
        ADD  R0, R2, R0
        ADD  R1, 4, R1
        SUB  R5, 1, R5
        JR_NZ PETLJA
        JP NASTAVI
```

JR se ne može koristiti zbog prevelike udaljenosti odredišta skoka, a absolutna adresa se može koristiti jer je odredišna adresa malena

Proširenje upravljačkih nar. - potprogrami

- S običnim skokovima i ostalim naredbama možemo isprogramirati svaki algoritam
- Međutim, da bi mogli programirati, u praksi nam treba mogućnost korištenja potprograma (bolja struktura i modularnost programa)
- Za potprograme nam trebaju posebne naredbe skoka:
 - **naredba za poziv potprograma CALL**
 - **naredba za povratak iz potprograma RET**
- Zbog pravilnosti i ove naredbe izvode se uvjetno

Proširenje upravljačkih nar. - potprogrami

- **Naredba CALL** se uvijek piše s adresom potprograma koji pozivamo (kao što se i naredba skoka JP piše s adresom na koju treba skočiti). Ova adresa se u praksi obično piše kao labela, koja se koristi kao ime potprograma. Definiramo pisanje naredbe CALL:

```
CALL adr  
CALL (adrreg)  
CALL_uvjet adr  
CALL_uvjet (adrreg)
```

- adr - 20-bitna adresa skoka koja se predznačno proširuje
- adrreg - opći register u kojem je adresa skoka
- Koriste se isti načini zadavanja adrese skoka kao u naredbi JP

Proširenje upravljačkih nar. - potprogrami

- Naredba CALL **mora omogućiti povratak** na naredbu koja se nalazi neposredno iza nje, **tj. mora spremiti povratnu adresu**
- Kako naredba CALL "zna" adresu sljedeće naredbe?
 - Jednostavno: adresa se nalazi u registru PC
 - Podsjetnik: u PC-u se nalazi adresa sljedeće naredbe koju treba izvesti, a to je upravo naredba koja se nalazi neposredno iza naredbe CALL
- Nakon spremanja povratne adrese, CALL može skočiti u potprogram tako da u PC stavi adresu potprograma (adresa je zapisana unutar strojnog kôda ili unutar jednog od općih registara)

>>>

Proširenje upravljačkih nar. - potprogrami

<<<

- **Gdje se spremi povratna adresa? Na stog***
- Dakle, naredba "CALL ADRESA" radi sljedeće:
 - spremi PC na stog
 - skoči na adresu potprograma (tj. na ADRESA)
- Ovako to radi "interno":
 - $SP - 4 \rightarrow SP$
 - $PC \rightarrow (SP)$
 - $ADRESA \rightarrow PC$

* To nije jedina mogućnost spremanja povratne adrese,
ali je najčešće korištena

Proširenje upravljačkih nar. - potprogrami

- **Naredba za povratak iz potprograma RET** piše se bez operanada (to je jedina naredba skoka u kojoj se ne zadaje adresa odredišta skoka)
- Naredbi RET adresa skoka nije potrebna, jer ona podrazumijeva da se adresa skoka nalazi na vrhu stoga
- Naravno, pretpostavka je da je negdje ranije izveden CALL koji je povratnu adresu stavio na stog, jer naredba RET ne može "znati" što je zaista na stogu
- Definiramo pisanje naredbe RET:
 - RET
 - RET_uvjet

Proširenje upravljačkih nar. - potprogrami

<<<

- Dakle, naredba "RET" radi sljedeće:
 - uzme povratnu adresu sa stoga i skoči na nju
- Ovako to radi "interno":
 - $(SP) \rightarrow PC$
 - $SP + 4 \rightarrow SP$

Proširenje upravljačkih nar. - potprogrami

- Zašto se povratna adresa spremi na stog?
- Zato jer je stog promjenjive veličine, a to omogućava jednostavno grijanje potprograma do potrebne dubine, kao i rekursivno pozivanje potprograma
- Spremanje na fiksnu memoriju lokaciju (npr. u varijablu) ne bi bilo praktično, jer bi već drugi ugniježđeni poziv potprograma uništio prvu spremljenu povratnu adresu
- Kod spremanja na stog, novi podatak uvijek se spremi na novo mjesto, ovisno o trenutačnoj popunjenoosti stoga

Proširenje upravljačkih nar. - potprogrami

- Definirajmo još dvije varijante naredbe RET
- One će nam trebati kod prekidnog U-I prijenosa, pa ih nećemo sada objašnjavati
- Definirat ćemo samo način njihovog pisanja:

RETI

RETI_uvjet

RETN

RETN_uvjet

Proširenje upravljačkih naredaba

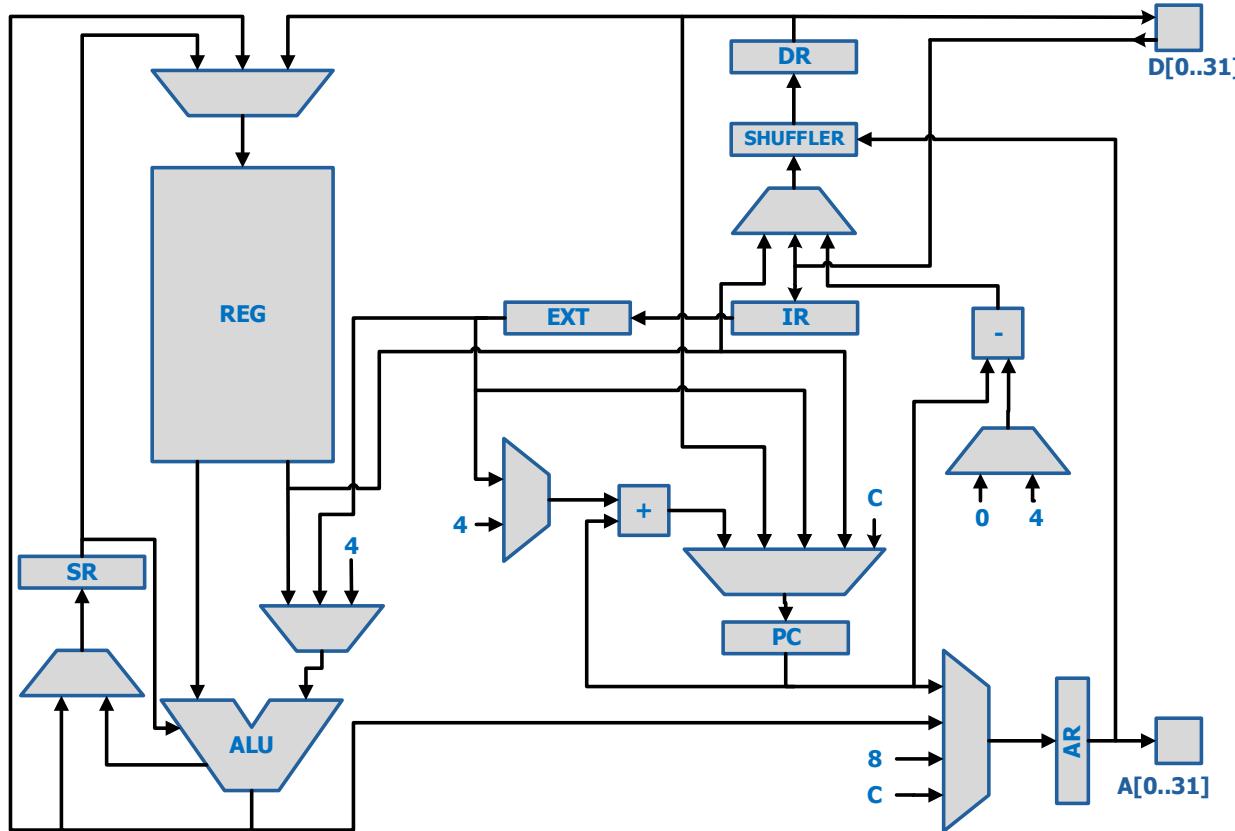
- Na kraju, dodajmo još jednu naredbu koja postoji u mnogim procesorima
- To je naredba za **zaustavljanje rada procesora** i naziva se **HALT**
- Budući da ova naredba prekida normalni slijed izvođenja, svrstat ćemo je u upravljačke (iako nije naredba skoka)
- Zbog pravilnosti će se i naredba HALT moći izvoditi uvjetno

HALT_uvjet
HALT

Proširenje upravljačkih nar. - put podataka

- Za izvođenje upravljačkih naredaba, put podataka je kompliciraniji nego kod ostalih naredaba

Put podataka...



JP adr
JP (reg)
JR
CALL
RET
PC+4

Načini adresiranja

Načini adresiranja

- Do sada smo već spominjali načine adresiranja i vidjeli ih u pojedinim naredbama. Pogledajmo sada i definirajmo načine adresiranja procesora FRISC.
- Pod načinima adresiranja (addressing modes) u širem smislu misli se na načine zadavanja operanada, rezultata ili odredišta skoka u naredbama
- U užem smislu, adresiranje se odnosi samo na načine adresiranja podataka ili odredišta skoka u memoriji, ali ovdje ćemo koristiti taj pojam u širem smislu
- Ovdje se radi o **procesorskim načinima adresiranja**

Načini adresiranja

- Treba razlikovati procesorska adresiranja od asemblerskih adresiranja
- Procesorska adresiranja:
 - odnose se na različite načine na koje procesor adresira podatke prilikom izvođenja naredaba
 - međusobno se raspoznaju po strojnom kôdu naredbe
- Asemblerska adresiranja (detaljniji opis kasnije):
 - različiti načini pisanja adresa koje dozvoljava asemblerski prevoditelj
 - svako od asemblerskih adresiranja će asemblerski prevoditelj prevesti u jedno od procesorskih adresiranja
 - različita asemblerska adresiranja daju isti strojni kôd

Načini adresiranja

- Iako je naš procesor RISC arhitekture, ipak ima relativno velik broj adresiranja. To su:
- **Registarsko adresiranje**
- **Neposredno adresiranje**
- **Apsolutno adresiranje**
- **Relativno adresiranje**
- **Registarsko indirektno adresiranje**
- **Registarsko indirektno adresiranje s odmakom**
- **Implicitno adresiranje**

Procesorski načini adresiranja

Procesorski načini adresiranja

- Registarsko adresiranje (register addressing)
 - Podatak ili rezultat nalazi se u jednom od registara procesora
 - ADD R0, 12, R2
- Neposredno adresiranje (immediate addressing)
 - Podatak se zadaje neposredno u naredbi kao broj. Nakon prevodenja, taj podatak je zapisan u strojnom kôdu.
 - Podatak predstavlja broj s kojim se izvodi operacija, a ne adresu u memoriji
 - MOVE 200, R3

Procesorski načini adresiranja

- Apsolutno adresiranje (absolute addressing)
 - broj predstavlja adresu podatka ili skoka u memoriji
 - LOAD R0, (1200)
- Relativno adresiranje (relative addressing)
 - Koristi se samo u naredbi relativnog skoka JR
 - JR 1200
- Registarsko indirektno adresiranje (register indirect addressing)
 - U upravljačkim naredbama
 - U registru se nalazi adresa odredišta skoka
 - CALL (R6)

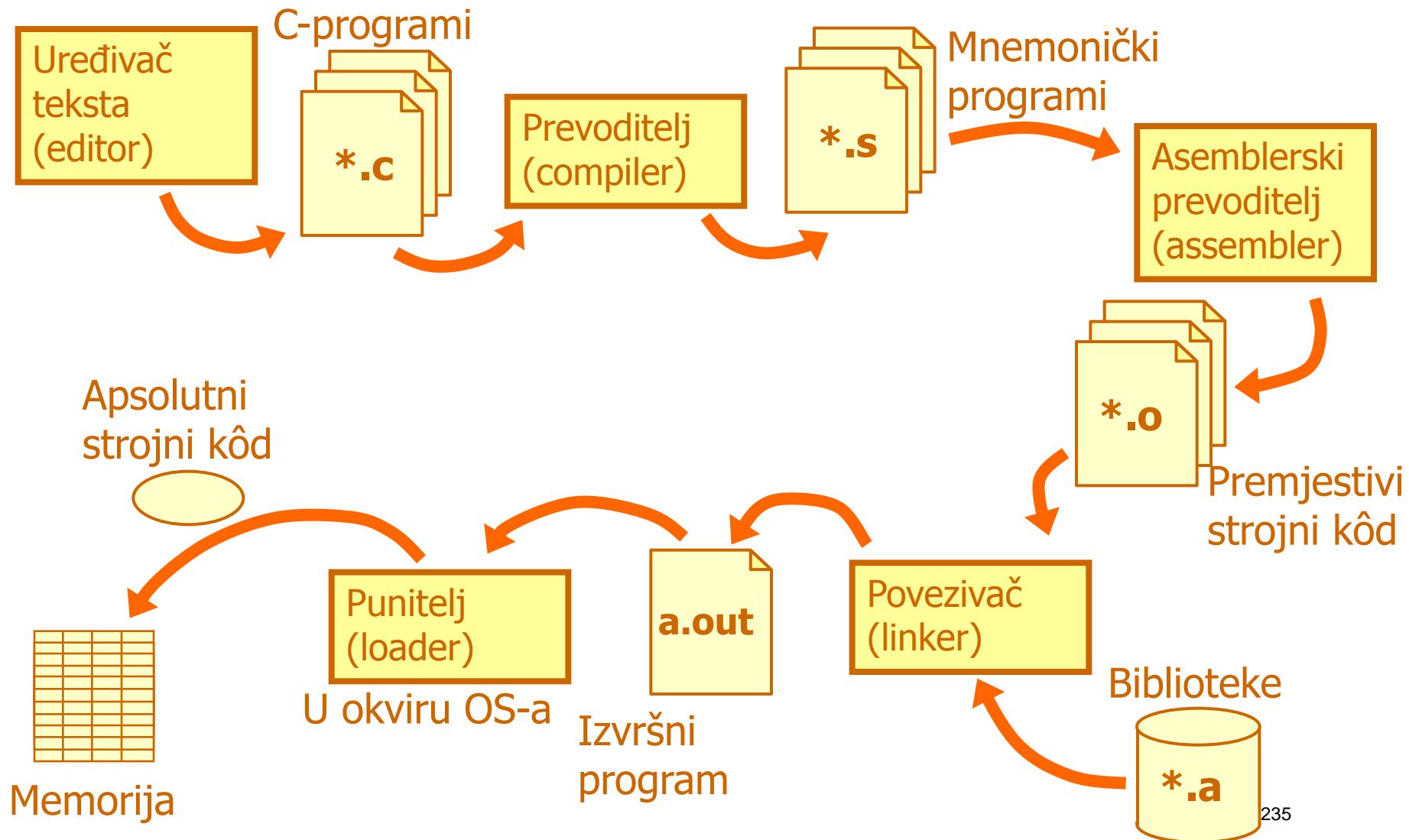
Procesorski načini adresiranja

- Registarsko indirektno adresiranje s odmakom (register indirect addressing with displacement; base plus offset addressing)
 - U memorijskim naredbama
 - LOAD R5, (R0+4)
- Implicitno adresiranje (implied addressing)
 - iz same naredbe se zna gdje se nalazi podatak ili odredište skoka i sl.
 - PUSH R5

Asembleri

Asembleri - Uvod

- Tipičan tijek pri prevodenju viših programskih jezika (UNIX):

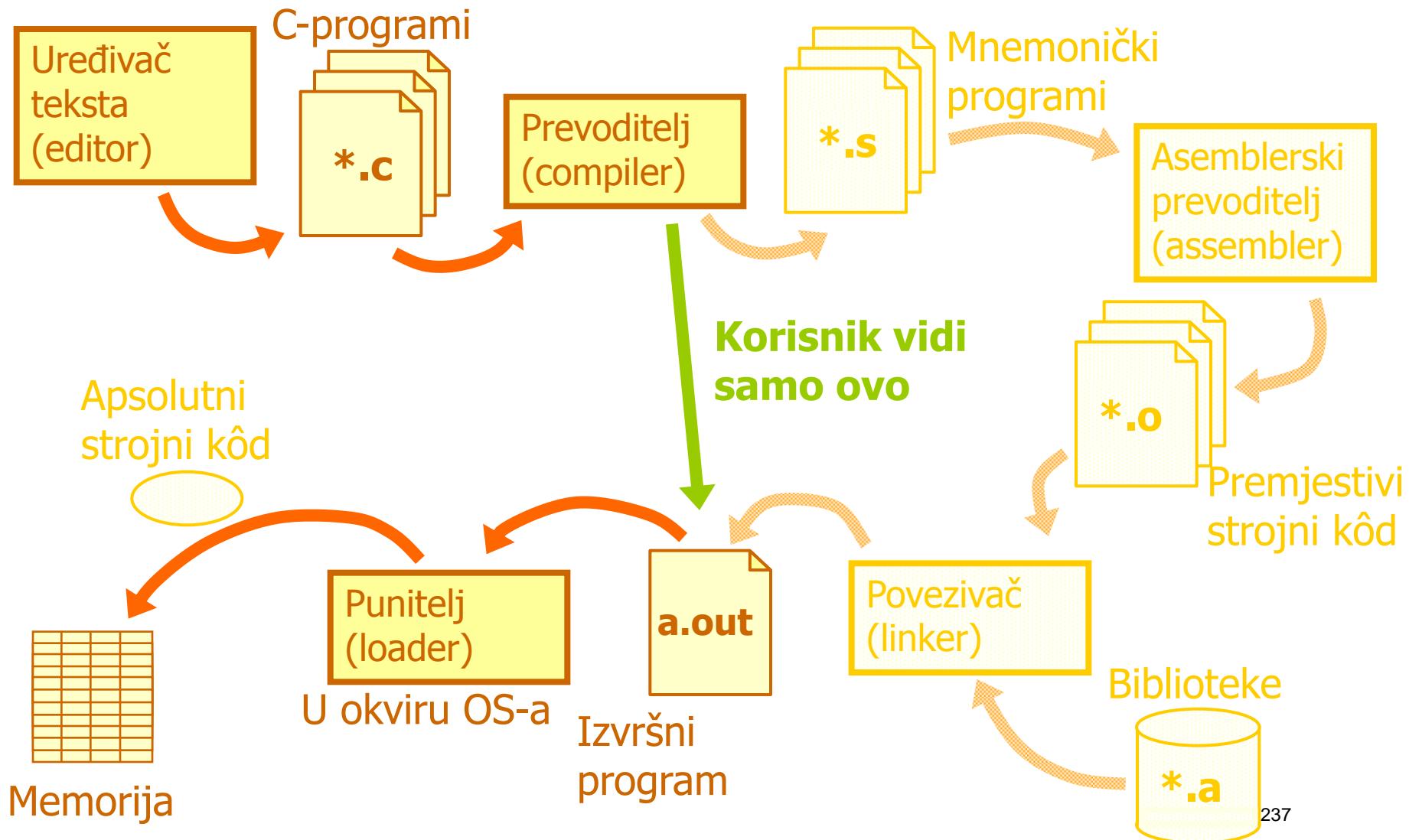


Asembleri - Uvod

- Podjela posla s prethodne slike može biti i drugačija:
 - Povezivanje sa statickim bibliotekama može obavljati povezivač, a povezivanje sa dinamičkim bibliotekama može obavljati punitelj
 - Punjenje i povezivanje su zadaće koje može obavljati jedan program.
 - Mnemonički program se stvara samo kao privremena datoteka koja se odmah dalje prevodi asemblerским prevoditeljem, a ne kao datoteka koja će ostati zapisana na disku (ovo je za korisnika nevidljivo)
 - Izvršni program može biti u absolutnom ili premjestivom obliku

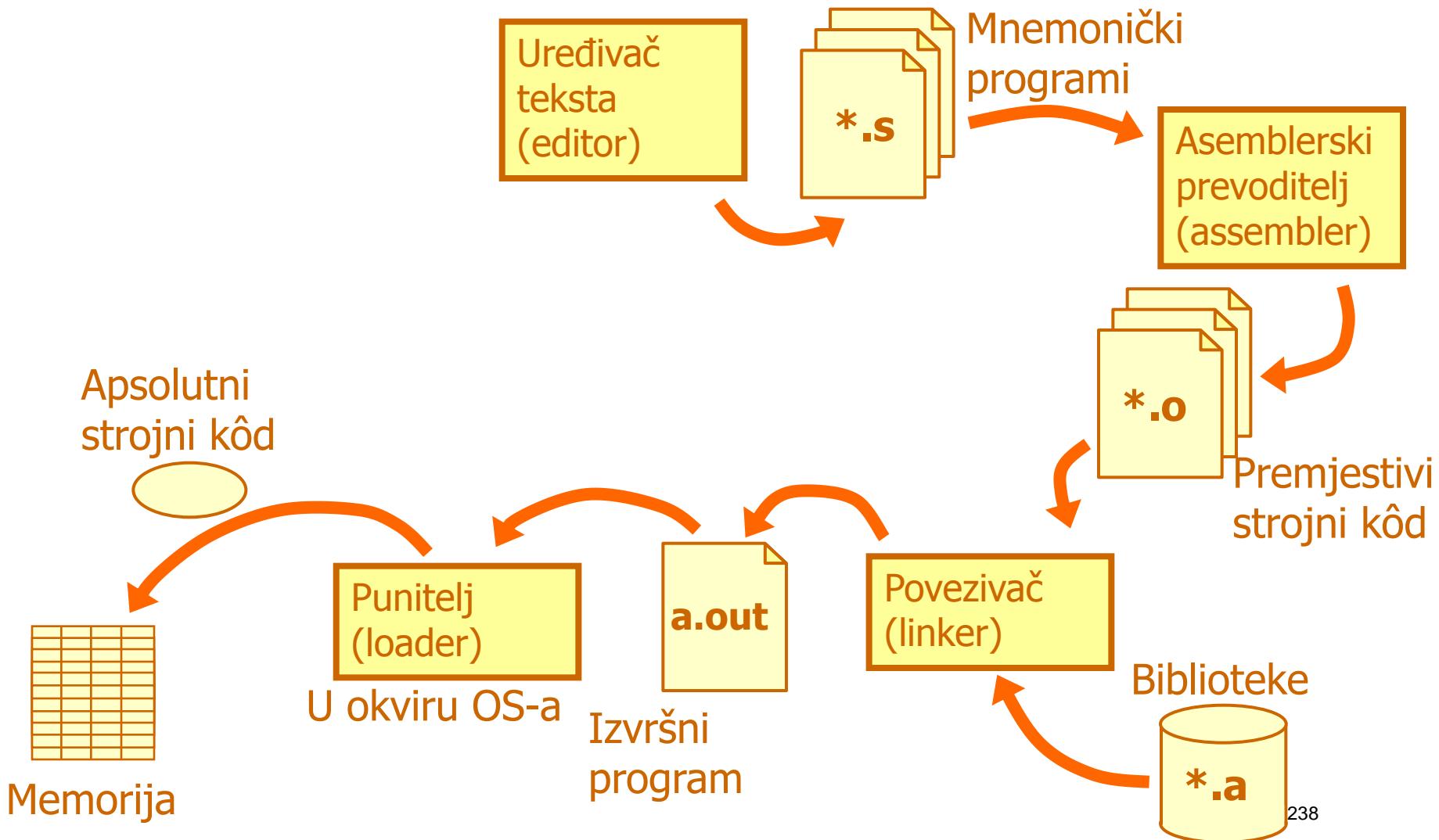
Asembleri - Uvod

- Za korisnika je većina ovog nevidljiva:



Asembleri - Uvod

- Tipičan tijek pri prevodenju mnemoničkih programa:



Asembleri - Uvod

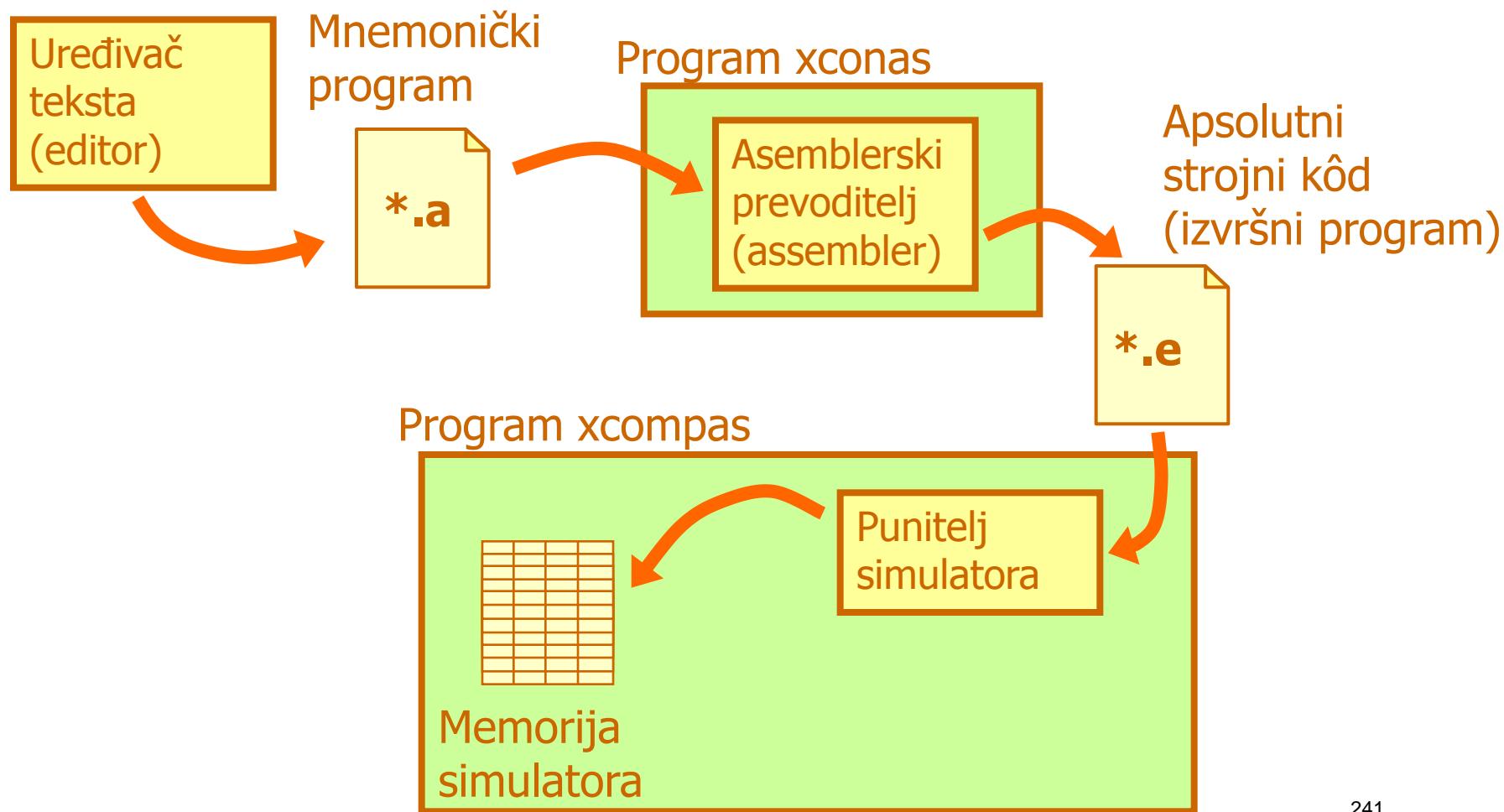
- **Asemblerski prevoditelji**, ili kraće asembleri, su programi koji prevode programe pisane u mnemoničkom jeziku u strojni kôd određenog procesora
 - postupak prevodenja nazivamo asembliranje
 - asembleri su prevoditelji, ali znatno jednostavniji od prevoditelja za više programske jezike
- **Mnemonički jezik** je jezik niske razine i prilagođen je pojedinom procesoru
- Svaki **strojni kôd** ima odgovarajući mnemonik s kojim je u odnosu "jedan na jedan"

Asembleri - Uvod

- Nakon povezivanja može se dobiti program koji je još uvijek premjestiv ili je u absolutnom obliku
 - Program u absolutnom obliku ima određene sve adrese podatka i potprograma i spremam je za izravno punjenje u memoriju računala i izvođenje
 - Za program u premjestivom obliku mora se prilikom punjenja odrediti početna adresa i na temelju toga preračunati sve adrese koje se u programu koriste.
- Nakon punjenja u memoriju računala, program se pokreće
 - Punjenje se tipično odvija pod upravljanjem operacijskog sustava (OS-a)
 - Punjenje se ne zadaje izravno, nego se podrazumijeva kad pokrenemo neki program
 - Korisnik pokreće program pomoću ljudske (npr. tcsh/bash na UNIX-u) ili grafičkog sučelja (Microsoft Windows, X Window na UNIX-u)

Asembleri - Uvod

- Prevodenje mnemoničkih programa u ATLAS-u:



Asembleri - Uvod

- U ATLAS-u se može koristiti samo jedna datoteka s mnemoničkim programom
- Ona se odmah prevodi u izvršnu datoteku u absolutnom obliku, tj. sadrži strojni kôd koji ima zadanu adresu punjenja u memoriju
- ATLAS je simulator računala na niskoj razini i u njemu ne postoji operacijski sustav - njegove najosnovnije zadaće preuzima korisničko sučelje simulatora u kojem se programi mogu puniti i izvoditi

Asembleri - Mnemonički jezik

- U ovom poglavlju naučit ćemo programirati procesor u mnemoničkom ili asemblerском jeziku* (assembly language)
 - Mnemonički jezik ovisi o procesoru za kojega je namijenjen, za razliku od viših programskega jezika koji ne ovise o računalu i/ili operacijskom sustavu na kojem će se izvoditi
 - Proizvođač procesora propisuje simbolička imena (mnemonike) za naredbe svog procesora
 - Dok smo objašnjavali arhitekturu i naredbe FRISC-a, već smo vidjeli primjere manjih programa i djelomično pravila pisanja programa u mnemoničkom jeziku

Asembleri - Mnemonički jezik

- Osim propisanih mnemonika, asemblerski prevoditelji dodaju svoja pravila pisanja, ograničenja ili dopunske mogućnosti
- Datoteke u mnemoničkom jeziku su obične tekstovne datoteke pisane prema pravilima pojedinog procesora i asemblerskog prevoditelja*
- Ovdje ćemo koristiti pravila za program CONAS (CONfigurable ASsembler), koji je asemblerski prevoditelj programskog sustava ATLAS

*** Napomena:** i asemblerski jezik i asemblerski prevoditelj često se nazivaju skraćeno *asembler*

Asembleri - Pravila pisanja

- Mnemoničke datoteke nemaju slobodan format pisanja kao viši programski jezici, nego su **retkovno orijentirane**:
 - Naredba se **ne može** protezati kroz više redaka
 - U jednom retku može biti **najviše jedna** naredba
 - Smije se pisati prazan redak (zbog bolje čitljivosti)
- Svaki redak sastoji se od sljedećih polja:

POLJE_LABELE

POLJE_NAREDBE

POLJE_KOMENTARA

Asembleri - Pravila pisanja

- Polja imaju sljedeća značenja i pravila pisanja:
- **Polje labele:**
 - Obavezno počinje od prvog stupca datoteke, ali se smije ispuštiti
 - Labela je simboličko ime za adresu
 - Labela se sastoji od niza slova i znamenaka te znaka podvlake, a prvi znak mora biti slovo
 - Duljina labele nije ograničena, ali se razlikuje samo prvih deset znakova

Asembleri - Pravila pisanja

- Polja imaju sljedeća značenja i pravila pisanja:
- **Polje naredbe:**
 - Polje naredbe ispred sebe obavezno mora imati prazninu (znak razmaka ili tabulatora), bez obzira stoji li ispred labela ili ne
 - Polje naredbe se smije ispustiti (tada naravno nije potrebno stavljati praznine)
 - Naredba se piše prema pravilima definiranim za pojedini procesor
 - U polju naredbe umjesto naredbe smije stajati i pseudonaredba (bit će objašnjene kasnije)

Asembleri - Pravila pisanja

- Polja imaju sljedeća značenja i pravila pisanja:
- **Polje komentara:**
 - Polje komentara počinje znakom komentara i proteže se do kraja tekućeg retka
 - Znak komentara ovisi o procesoru
 - za FRISC, ARM to je znak točka-zarez ;
 - Polje komentara se također može ispuštiti
 - Polje komentara se zanemaruje prilikom prevodenja

Asembleri - Pravila pisanja

- Primjeri:

POLJE_LABELE	POLJE_NAREDBE	POLJE_KOMENTARA
--------------	---------------	-----------------

PETLJA	ADD R0, R1, R2 ;naredba ADD SUB R3, R2, R3	
PODATCI	ORG 200 ;pseudonaredba `ORG	
LABELA_3	;	labela smije stajati bez naredbe
	;	komentar smije početi od prvog stupca
	;	ovaj bi red bio prazan da nema komentar :)

Asembleri - Vrste asemblera

- Asembleri se mogu podijeliti po broju prolaza na:
 - jednoprozne ili apsolutne asemblere
 - dvoprolazne ili simboličke asemblere
 - troprolazne asemblere
 - četveroprolazne asemblere
- Ovisno o broju prolaza, asembleri imaju različite mogućnosti - što više prolaza, to više mogućnosti

} tzv. makroasembleri

Objašnjenje načina prevodenja proučite za domaću zadaću (pogledajte datoteku nazvanu "03 Način asempliranja DZ")

Asembleri - Labele

- U asembleru se **labele** također koriste kao **odredište skoka**
 - Za razliku od viših programskih jezika, u asembleru je korištenje labela i naredbe skoka (npr. JUMP) jedini način za upravljanje tokom programa
 - Kao odredište skoka može se pomoći broj zadati i stvarna adresa skoka, ali tada moramo **točno znati na koju adresu želimo skočiti**, tj. moramo tu adresu "ručno izračunati".



Asembleri - Labele

- Labele su simbolički nazivi za adrese, a glavne prednosti su:
 - jednostavnije i brže programiranje
 - bolja čitljivost i lakše održavanje programa
 - izračunavanje adresa obavlja asemblerski prevoditelj što ujedno smanjuje mogućnost pogreške
- Asembler "izračunava" stvarne vrijednosti labela (tj. adrese) točno onako kako ih i mi "ručno izračunavamo"
- Korištenje labela naziva se **simboličko adresiranje**, a korištenje stvarnih adresa zadanih brojem naziva se **apsolutno adresiranje**
- **POZOR:** ovo su **asemblerска adresiranja** i ne treba ih miješati s **procesorskim adresiranjima**, naročito ne s istoimenim **apsolutnim procesorskim adresiranjem** o čemu će više biti riječi kasnije

Asembleri - Pseudonaredbe

- Pseudonaredbe:
 - nemaju veze s procesorom
 - to su naredbe za asemblerski prevoditelj: one upravljaju njegovim radom govoreći mu pobliže kako treba obavljati prevođenje
 - "izvodi" ih asemblerski prevoditelj tijekom prevođenja
- U okviru ARH1 koristiti će se pseudonaredbe ORG, EQU, DW, DH, DB, DS, MACRO i ENDMACRO.

xconas - Pseudonaredba ORG

- Pseudonaredba ORG (origin) zadaje asembleru adresu punjenja strojnog kôda i piše se ovako:

ORG adresa

- Adresa mora biti zadana brojem, a ne labelom
 - podaci će biti smješteni od zadane adrese
 - Strojni kôdovi dobiveni prevodenjem sljedećih redaka datoteke smjestit će se u memoriji od zadane adrese (ako je djeljiva s 4) ili prve sljedeće adrese koja je djeljiva s 4
- ATLAS-ov asembler daje absolutni strojni kôd pa se mora znati početna adresa punjenja programa:
 - zadaje se pomoću ORG u prvom retku datoteke
 - ako se ORG ispusti, onda se pretpostavlja početna adresa 0

xconas - Pseudonaredba ORG

- Moguće je u datoteci navesti više pseudonaredba ORG čije adrese:
 - moraju biti u rastućem redoslijedu
 - ne smiju biti manje od adresе zadnjeg prevedenog strojnog kôda

program:

```
ORG 0
ADD ...
STORE ...

ORG 14
LOAD ...
HALT
```

memorija:

adresa	sadržaj
0 :	ADD
4 :	STORE
8 :	0
C:	0
10:	0
14:	LOAD
18:	HALT

zapravo ADD
zauzima adrese 0-3, STORE 4-7, itd.

} "preskočeno"
} do adrese 14

brojevi su heksadekadski

VAŽNO: Poravnanje naredaba za FRISC

• **Poravnanje naredaba za FRISC**

- Memoriske lokacije široke su jedan bajt (tj. najmanja količina memorije koja se može adresirati je jedan bajt)
- Podatkovna sabirnica je širine 32 bita, što znači da se može odjednom pročitati sadržaj 4 memoriske lokacije
- Naredbe su široke 32 bita pa su u memoriji uvijek **spremljene na adresama djeljivima s 4**
 - Kažemo da su naredbe poravnate na adresu dijeljivu s 4 (memory aligned).

VAŽNO: Poravnanje naredaba i ORG

- Čak ako sa ORG zadamo adresu koja nije djeljiva s 4, prevoditelj će poravnati naredbe:

program:

```
ORG 0
ADD ...
STORE ...
...
ORG 25
LOAD ...
HALT
```

memorija:

adresa	sadržaj
0-3:	ADD
4-7:	STORE
...	...
25:	0
26:	0
27:	0
28-2B:	LOAD
2C-2F:	HALT

asemblerски prevoditelj automatski
"poravnava" naredbe na adresu djeljivu s 4

xconas- Pseudonaredba ORG

- Primjer:
program:

```
ORG 20  
ADD ...  
STORE ...  
LOAD ...
```

```
ORG 0  
XOR ...  
HALT
```

memorija:

adresa	sadržaj
20:	ADD
24:	STORE
28:	LOAD

greška: 0 je manje od adrese prethodnog ORG-a
(iako bi na adresama 0 i 4 bilo mesta za strojni kod naredaba XOR i HALT)

xconas - Pseudonaredba ORG

- Primjer:
program:

```
ORG 0  
ADD ...  
STORE ...  
LOAD ...
```

```
ORG 4  
XOR ...  
HALT
```

memorija:

adresa	sadržaj
0 :	ADD
4 :	STORE
8 :	LOAD

greška: 4 je manje od adrese prethodne naredbe LOAD
(veći je od prethodnog ORG-a,
ali ne "dovoljno")

xconas - Pseudonaredba EQU

- Pseudonaredba EQU (equal) služi za "ručno" definiranje vrijednosti labele (kao da definiramo imenovanu konstantu):

LABELA EQU podatak

- Labela i podatak se obavezno pišu, pri čemu podatak mora biti zadan numerički, a ne nekom drugom labelom
- Inače, kad nema pseudonaredbe EQU, asemblerski prevoditelj samostalno određuje vrijednost labele na temelju trenutačne adrese (spremljene u lokacijskom brojilu) i ubacuje labelu u tablicu labela
- Pri nailasku na pseudonaredbu EQU, prevoditelj će zanemariti vrijednost lokacijskog brojila. Umjesto toga jednostavno će uzeti labelu i podatak i staviti ih zajedno u tablicu labela

xconas - Pseudonaredba DW

- Pseudonaredba DW (define word) služi za izravan upis riječi (4 bajta) u memoriju (bez prevodenja):

DW podatci

- Podatci moraju biti zadani numerički, a ne labelom
- Ispred pseudonaredbe DW može stajati labela
- Prevoditelj jednostavno uzima podatke i stavlja ih od sljedeće memorijske riječi na dalje, čime se zauzima i inicijalizira memorija

xconas - Pseudonaredba DH

- Pseudonaredba DH (define half-word) služi za izravan upis poluriječi (2 bajta) u memoriju (bez prevodenja):

DH podatci

- Podatci moraju biti zadani numerički, a ne labelom
- Ispred pseudonaredbe DH može stajati labela
- Prevoditelj jednostavno uzima podatke i stavlja ih od sljedeće memorijske riječi na dalje, čime se zauzima i inicijalizira memorija

xconas - Pseudonaredba DB

- Pseudonaredba DB (define byte) služi za izravan upis bajta u memoriju (bez prevođenja):

DB podatci

- Podatci moraju biti zadani numerički, a ne labelom
- Ispred pseudonaredbe DB može stajati labela
- Prevoditelj jednostavno uzima podatke i stavlja ih od sljedeće memorijske riječi na dalje, čime se zauzima i inicijalizira memorija

xconas - Pseudonaredba DS

- Pseudonaredba DS (define space) služi za zauzimanje većeg broja memorijskih lokacija (bajtova) i njihovu inicijalizaciju u nulu:

LABELA DS podatak

- Labela se može ispustiti, a podatak se obavezno piše te mora biti zadan numerički, a ne labelom
- Podatak zadaje koliko memorijskih lokacija treba zauzeti
- Labela će biti adresa prve lokacije u nizu koji je zauzela pseudonaredba DS

xconas - Pisanje brojeva

- Brojevi se pišu u podrazumijevanoj bazi koja je heksadekadska
- To se odnosi na sve brojeve koji se pišu u pseudonaredbama i naredbama bez obzira predstavljaju li adresu, podatak, broj podataka ili bilo što drugo
- za brojeve u drugim bazama, svaki se broj pojedinačno može napisati u željenoj bazi ako se napiše sa jednim od prefiksa %B za binarnu bazu, %D za dekadsku i %H za heksadekadsku.
- Kako bi asembler razlikovao brojeve od labela, brojevi će uvijek počinjati znamenkom a labele slovom: svi heksadekadski brojevi koji počinju slovom na početku imaju dodanu nulu (npr. 0A38C)

Primjeri programa

Uspoređivanje brojeva

- FRISC ima sve potrebne uvjete u upravljačkim naredbama za jednostavno uspoređivanje brojeva (u formatima NBC i 2'k), tako da ne treba "ručno" ispitivati pojedine zastavice
- Usporedbe se (najčešće) obavljaju na sljedeći način:
 - Prvo se dva broja oduzmu naredbom CMP (ili SUB ako je potrebna i njihova razlika)
 - Naredbom uvjetnog skoka usporedi se brojevi: ako je uvjet istinit, onda se skok izvodi, a inače se nastavlja s izvođenjem sljedeće naredbe
 - U naredbi skoka, uvjet se interpretira kao da je operator usporedbe stavljen "između" operanada koji su oduzimani

Uspoređivanje brojeva

- Na primjer, želimo li usporediti je li broj u R5 veći ili jednak od broja u R2 (uz pretpostavku da su to NBC-brojevi):
- Želimo postaviti uvjet $R5 \geq R2$ pa upotrijebimo sufiks UGE:
 - U: unsigned - jer uspoređujemo NBC-brojeve
 - G: greater - jer ispitujemo je li R5 veći od R2
 - E: equal - jer ispitujemo je li R5 jednak R2
- U naredbi CMP pišemo brojeve u istom redoslijedu kao u uvjetu $R5 \geq R2$:

CMP R5 , R2
JR _UGE UVJET_ISTINIT

>>>

Pisanje uvjeta..

- Na primjer: ako je $R5 \geq R2$, onda treba uvećati $R7$ za 7, a inače ne treba napraviti ništa. Izravnim pisanjem uvjeta dobivamo:

```
CMP      R5, R2  
JR_UGE  UVECAJ_R7  
NISTA   JR      DALJE  
UVECAJ_R7 ADD     R7, 7, R7  
DALJE   ...
```

- S obrnutim uvjetom program je nešto razumljiviji, kraći i brži:

```
CMP      R5, R2  
JR_ULT  DALJE  
UVECAJ_R7 ADD     R7, 7, R7  
DALJE   ...
```

Uspoređivanje brojeva

Usporediti dva 2'k-broja spremljena u registrima R0 i R1. Manji od njih treba staviti u R2. Drugim riječima treba napraviti:

$$R2 = \min (R0, R1)$$

Rješenje:

	CMP	R0 , R1
	JR _ SLT	R0 _ MANJI
R0 _ VECI _ JEDNAK	MOVE	R1 , R2 ; R1 je manji
	JR	KRAJ
R0 _ MANJI	MOVE	R0 , R2 ; R0 je manji
KRAJ	HALT	

Uspoređivanje brojeva

Ispitati 2'k-broj u registru R6. Ako je negativan, u R0 treba staviti broj -1. Ako je jednak ništici, u R0 treba upisati 0. Ako je pozitivan, treba u R0 upisati 1. Drugim riječima treba napraviti:

$$R0 = \text{signum} (R6)$$

Rješenje:

```
OR      R6,R6,R0 ; broj ispitaj i stavi u R0
JR_Z   KRAJ        ; ako je 0, u R0 je rezultat
JR_N   NEG         ; ispitaj predznak
POZ    MOVE 1, R0
       JR  KRAJ
NEG    MOVE -1, R0
KRAJ  HALT
```

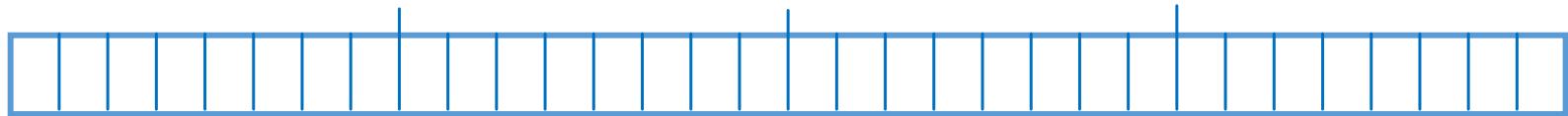
Rad s bitovima

- Čest zadatak u asemblerskom programiranju
- Potrebno je mijenjati ili ispitivati bitove u registru ili memorijskoj lokaciji
 - Rad s bitovima u registru jednostavno se ostvaruje kombinacijama aritmetičko-logičkih naredaba
 - Rad s bitovima u memorijskoj lokaciji nije moguć pa se zato podatak prvo prebaci u jedan od registara, radi se s bitovima te se podatak vrati u memorijsku lokaciju
- Osnovne operacije s bitovima:
 - postavljanje (set)
 - brisanje (reset)
 - komplementiranje (complement)
 - ispitivanje (test)

Rad s bitovima

- Nekoliko pojmljiva vezanih za bitove u podatku:

najviši bajt



viši bitovi

niži bitovi

- Paritet / Parnost

- Maska

Rad s bitovima - Postavljanje bitova

U registru R0 treba **postaviti** najniža 4 bita, a ostali se ne smiju promijeniti.

```
LOAD    R1, (MASKA)
OR      R0, R1, R0
HALT
```

```
MASKA DW      %B 1111 ; ostali bitovi su 0
```

Ili jednostavnije (i bolje):

```
OR      R0, %B 1111, R0
HALT
```

Rad s bitovima - Ispitivanje bitova

Treba **ispitati** je li broj u registru R0 paran ili neparan. Ako je paran, treba ga upisati u R2, a inače u R2 treba upisati broj 1.

Rješenje:

Dakle, zadatak se svodi na **ispitivanje stanja jednog bita** što se ostvaruje brisanjem bitova koji se ne ispituju i testiranjem zastavice Z.

>>>

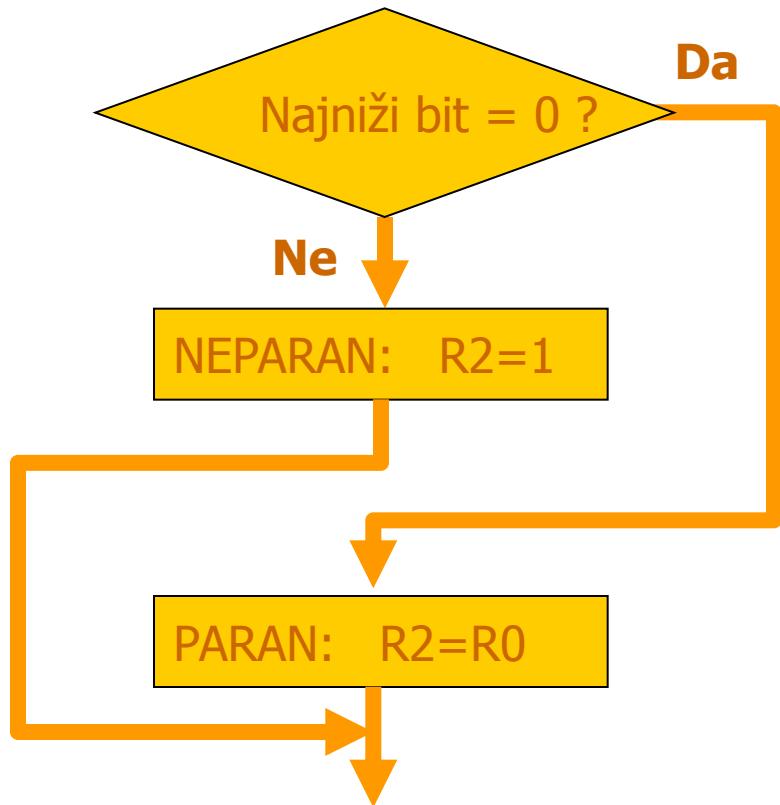
Rad s bitovima - Ispitivanje bitova

AND	R0 , 1 , R1	; ispitaj najniži bit
JR_Z	PARAN	

NEPARAN	MOVE 1 , R2	
	JR KRAJ	

PARAN	MOVE R0 , R2	
--------------	---------------------	--

KRAJ	HALT	
-------------	-------------	--



Mogući nedostatak: ispitivanje uništava sadržaj R1

Rad s bitovima - Ispitivanje bitova

- **Ispitivanje stanja više bitova**
 - jesu li svi ispitivani bitovi jednaki nulama?
- Obrisati sve bitove koje ne ispitujemo (_), a bitove koje ispitujemo (?) ostavimo nepromijenjene
- Ispitamo zastavicu Z, tj. ispitamo je li rezultat jednak ništici:
 - Ako rezultat=0, onda su svi ispitivani bitovi u nulama
 - Ako rezultat≠0, onda nisu svi ispitivani bitovi u nulama

Početni broj: ?? ? ???

?=ispitivani bit

Nakon maskiranja: 00??0000?00???

Rad s bitovima - Ispitivanje bitova

Ispitati jesu li u registru R0 u bitovima 0, 1, 30, 31 **sve nule**.
Ako jesu, treba obrisati R0, a inače ga ne treba mijenjati.

```
LOAD  R1 , (MASKA)
AND   R0 , R1 , R1
JR_NZ IMA_1
```

```
SVE_0 MOVE 0 , R0      ; u isp. bitovima su sve 0
```

```
IMA_1 HALT           ; u isp. bitovima ima 1
```

```
MASKA DW %B 110000000000000000000000000000011
```

Rad s bitovima - Ispitivanje bitova

- **Ispitivanje stanja više bitova:**
 - jesu li svi ispitivani bitovi jednaki jedinicama?
- Postaviti sve bitove koje ne ispitujemo (_), a bitove koje ispitujemo (?) ostavimo nepromijenjene
- Komplementirati sve bitove
- Ispitamo zastavicu Z, tj. ispitamo je li rezultat jednak ništici:
 - Ako rezultat=0, onda su svi ispitivani bitovi u jedinicama
 - Ako rezultat≠0, onda nisu svi ispitivani bitovi u jedinicama

Početni broj:

 ?? ? ???

Nakon maskiranja:

11??1111?11???

Nakon komplementa:

00??0000?00???

Rad s bitovima - Ispitivanje bitova

Ispitati jesu li u registru R0 u bitovima od 1 do 4 i bitovima od 12 do 17 te u bitu 30 **sve jedinice**. Ako jesu, treba obrisati R0, a inače ga ne treba mijenjati.

LOAD R1 , (MASKA)

OR R0 , R1 , R1

XOR R1 , -1 , R1 ; komplementiranje

JR_NZ IMA_0

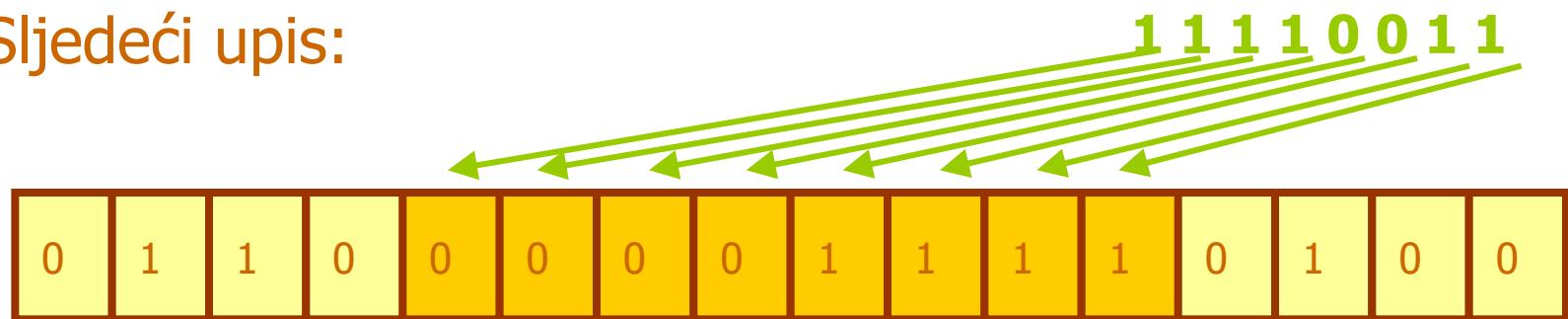
SVE_1 MOVE 0 , R0 ; u isp. bitovima su sve 1

IMA_0 HALT ; u isp. bitovima ima 0

MASKA DW %B 1011111111111000000111111100001

Rad s bitovima – Upis bitova

Sljedeći upis:



Daje rezultat:



Rad s bitovima – Upis bitova

- Postupak upisa je sljedeći:
 - Bitovi podatka koji se žele upisati (**y**) se ne mijenjaju, a ostali se brišu (_)
 - Bitovi registra koji se žele mijenjati (_) se obrišu, a ostali se ne mijenjaju (**x**)
 - "Poravna" se podatak "iznad" registra
 - Napravi se operacija OR između poravnatog registra i podatka

Podatak:	_____yyyy	Registar:	xx_____xx
Maskirani podatak:	0000yyyy	Maskirani registar:	xx0000xx
Poravni podatak:	00yyyy00		
OR:		xxyyyyxx	

Primjer

U bitove 2 do 10 registra R0 treba **upisati** bitove 20 do 28 iz registra R7 (brisanje+OR).

```
; brisanje bitova 2 do 10 u R0
LOAD  R1, (MASKA0)
AND   R0, R1, R0
; brisanje svih bitova osim 20 do 28 u R7
LOAD  R1, (MASKA7)
AND   R7, R1, R7

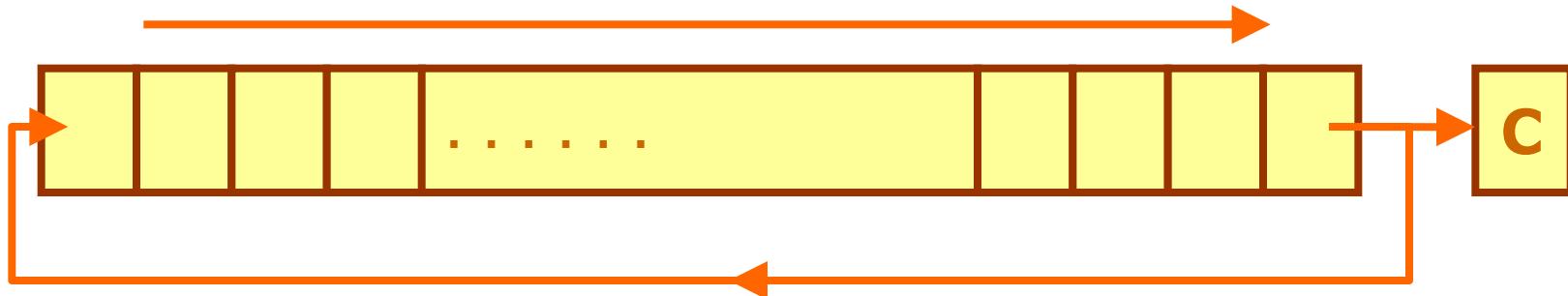
ROTR  R7, %D 18, R7      ; poravnavanje

OR    R0, R7, R0          ; upis u R0

HALT
```

Rad s bitovima – Prebrajanje bitova

- Pod prebrajanjem bitova misli se na prebrajanje nula ili jedinica u određenom nizu bitova u podatku
- Najlakše se ostvaruje naredbama rotacije (ulijevo ili udesno) i ispitivanjem zastavice C
- Rotacija radi tako da izlazni bit odlazi u zastavicu C:



Rad s bitovima – Prebrajanje bitova

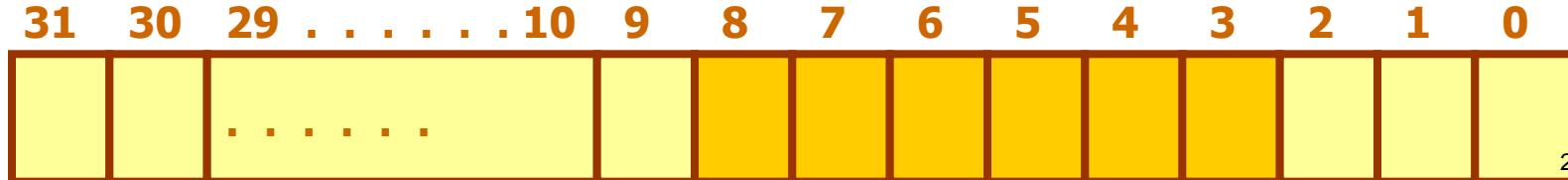
- Ako se rotacija obavlja za više bitova, onda u C odlazi samo izlazni bit od "zadnjeg koraka rotacije"
- Zato treba rotirati registar jedan po jedan bit (u petlji) i ispitivati zastavicu C*
- Prebrajanje bitova koristi se, npr. kod određivanja pariteta podatka

*Drugi način je ispitivanje zastavice N, jer se u njoj nalazi najviši bit регистра nakon rotacije (nije toliko uobičajeno rješenje)

Primjer

Koliko nula ima u bitovima 3 do 8 registra R0. Broj nula treba spremiti u memoriju lokaciju NULE.

```
MOVE 0, R1          ; R1 = brojač nula
MOVE 6, R2          ; R2 = brojač za petlju
ROTR R0, 3, R0      ; "izbaci" bitove 0 do 2
LOOP ROTR R0, 1, R0
        JR_C JEDAN
        ADD R1, 1, R1
JEDAN     SUB R2, 1, R2
        JR_NZ LOOP
        STORE R1, (NULE)
        HALT
```



Višestruka preciznost

Višestruka preciznost

- Dijelovi računala (memorijske lokacije, registri, ALU, sabirnice) su ograničeni na određen broj bita
- Npr. FRISC ima 32-bitnu arhitekturu (tj. riječ mu ima 32 bita) pa može normalno raditi s podatcima te širine
- Ako treba raditi s brojevima većeg opsega ili kakvim drugim podatcima širima nego što stanu u riječ procesora ili memorijsku lokaciju, onda koristimo **višestruku preciznost**
- Ovisno koliko procesorskih riječi se koristi za zapis podatka, govorimo o dvostrukoj, trostrukoj, itd. preciznosti

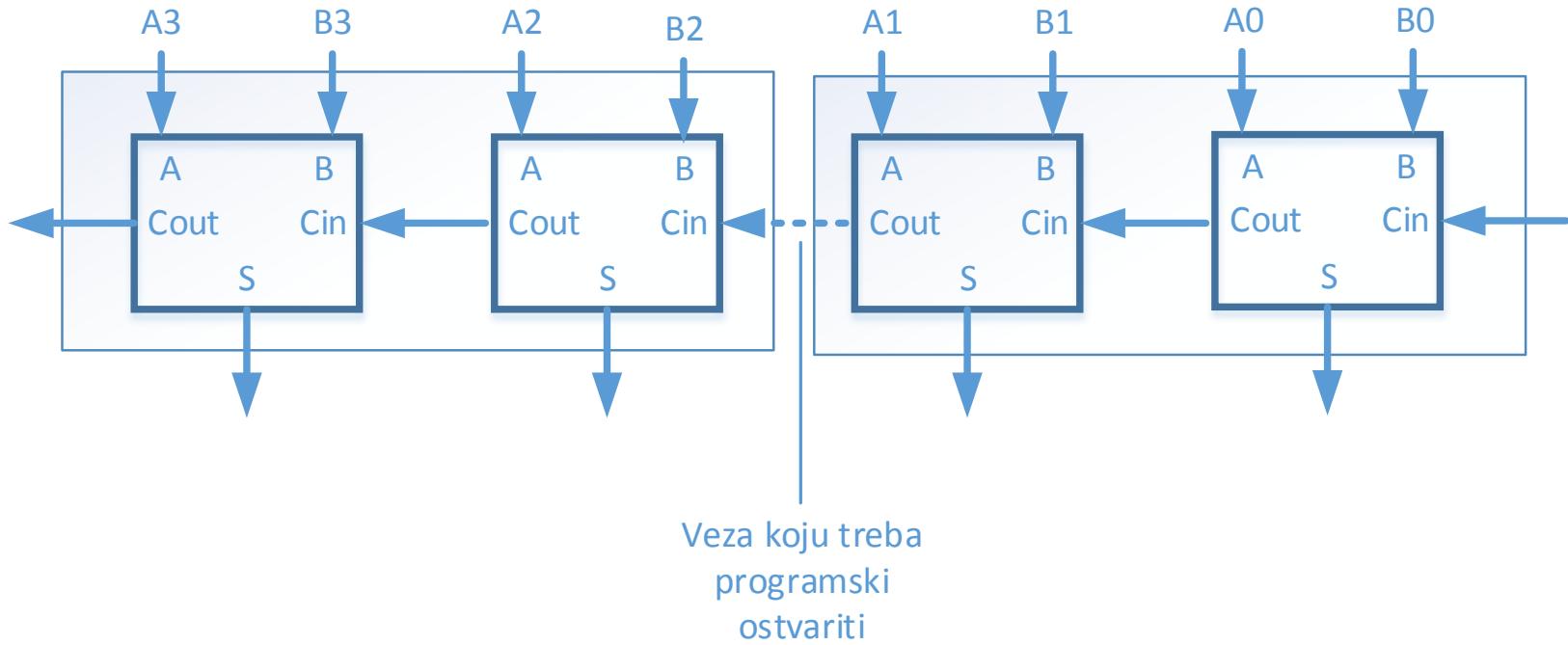
Višestruka preciznost

- **Načelno** se operacije u višestrukoj preciznosti uvijek obavljaju jednako, bez obzira koristimo li dvostruku, trostruku ili neku veću preciznost
 - Zato ćemo pokazati kako se koristi dvostruka preciznost
- Kod zapisivanja podataka u višestrukoj preciznosti u memoriji, treba podatak zapisivati u više uzastopnih lokacija
 - slično zapisivanju 32-bitnih riječi u bajtnoj memoriji, treba voditi računa o rasporedu zapisivanja pojedinih dijelova podatka unutar memorijskih lokacija
 - Budući da FRISC koristi little-endian za zapis 32-bitnih riječi unutar memorije, onda možemo koristiti isti redoslijed i kod višestruke preciznosti (iako to nije nužno)

Višestruka preciznost - Pohrana podataka

- Kod zapisivanja podatka u dvostrukoj preciznosti u registrima, također se moraju koristiti dva регистра
- Kod označavanja podataka obično se koriste sufiksi L i H koji označavaju:
 - niži dio podatka (L - low)
 - viši dio podatka (H - high)
- Npr. podatak A u dvostrukoj preciznosti označava se (po dijelovima) oznakama AL i AH

Višestruka preciznost - Zbrajanje



Višestruka preciznost - Zbrajanje

- Kako uračunati međuprijenos? Pomoću **naredbe ADC**.
 - Podsjetnik: naredba ADC, osim dva pribrojnika, pribraja i vrijednost prijenosa iz prethodne operacije zbrajanja
- Budući da je prijenos od zbrajanja spremlijen u zastavici C, onda naredba ADC zapravo radi ovako:

$$\text{ADC } X, Y, R \quad = \quad X + Y + \text{prijenos} \rightarrow R \quad = \quad X + Y + C \rightarrow R$$

- Sklopovalski se naredba ADC izvodi tako da se na ulaz C0, na najnižem potpunom zbrajalu, dovede stanje iz zastavice C (podsjetnik: kod običnog zbrajanja dovodi se 0)

Višestruka preciznost - Primjer

Zbrojiti NBC ili 2'k brojeve u dvostrukoj preciznosti. Prvi operand smješten je na memorijskim lokacijama AL (niži dio) i AH (viši dio), a drugi na lokacijama BL i BH. Rezultat se spremi na RL i RH.

; ZBROJI NIŽE DIJELOVE

LOAD R0, (AL)

LOAD R1, (BL)

ADD R0, R1, R2

STORE R2, (RL)

; ZBROJI VIŠE DIJELOVE

LOAD R0, (AH)

LOAD R1, (BH)

ADC R0, R1, R2

STORE R2, (RH)

HALT

; OPERANDI

AL DW 0A3541E21

AH DW 942F075F

BL DW 936104A7

BH DW 017F3784

; MJESTO ZA REZULTAT

RL DW 0

RH DW 0

Višestruka preciznost - Logičke operacije

- Logičke operacije AND, OR, XOR, NOT rade neovisno na pojedinim bitovima podataka
- Zato nije bitan redoslijed obavljanja operacija na pojedinim riječima podatka - jedino treba obaviti operacije na svim riječima
- Također, ne postoji nikakvi podatci, kao međuprijenosni, koje bi trebalo prenositi između viših i nižih riječi podataka

Višestruka preciznost - Pomaci i rotacije

- Sklopovali ostvareni pomaci i rotacije prenose bitove između pojedinih riječi pa to također treba napraviti i u programu
- Redoslijed operacija na pojedinim riječima podatka nije bitan, ali ovisno o operaciji može biti praktičniji jedan ili drugi redoslijed. Npr. pomak u lijevo za 1 bit:



- Prvo pomaknemo ulijevo RL. Izlazni bit je u zastavici C. Pomaknemo RH ulijevo i upišemo C u najniži bit od RH.
- Prvo pomaknemo ulijevo RH. Zatim pomaknemo ulijevo RL. Izlazni bit je u zastavici C. Upišemo C u najniži bit od RH.
- Oba redoslijeda izgledaju podjednako komplikirano...

Višestruka preciznost - Pomaci i rotacije

- Prvo pomaknemo ulijevo RL. Izlazni bit je u zastavici C. Pomaknemo RH ulijevo i upišemo C u najniži bit od RH:

	SHL	RL, 1, R1
	JR_C	JEDAN
NULA	SHL	RH, 1, RH
	JR	DALJE
JEDAN	SHL	RH, 1, RH
	OR	RH, 1, RH

- Prvo pomaknemo ulijevo RH. Zatim pomaknemo ulijevo RL. Izlazni bit je u zastavici C. Upišemo C u najniži bit od RH:

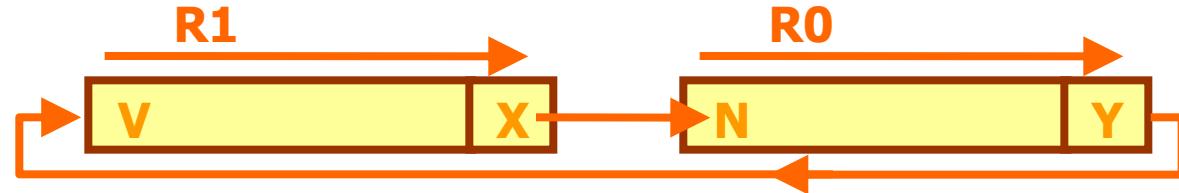
SHL	RH, 1, RH
SHL	RL, 1, RL
ADC	RH, 0, RH

Ipak je ovo
jednostavnija varijanta

Primjer:

Rotirati u desno za 1 mjesto podatak u dvostrukoj preciznosti zapisan u registrima R0 (niža riječ) i R1 (viša riječ).

Početno stanje prije rotacije i način rotacije:



Željeno stanje nakon rotacije u dvostrukoj preciznosti:



Nakon neovisnih rotacija izvedenih na R0 i R1:

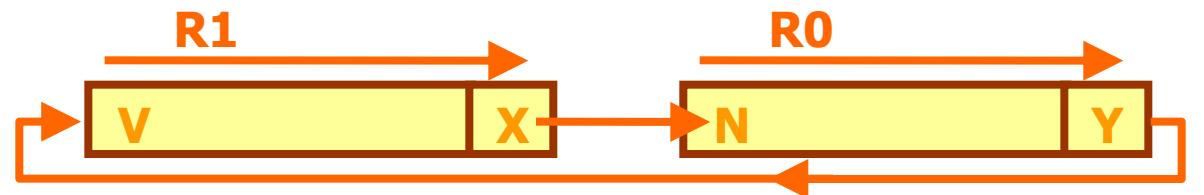


Vidimo da su bitovi V i N na ispravnom mjestu, ali bitovi X i Y moraju zamijeniti mjesta

Rješenje:

1. varijanta: napraviti dvije obične rotacije na R0 i R1, a zatim im zamijeniti vrijednosti najviših bitova X i Y.
2. varijanta: prvo zamijeniti najniže bitove X i Y, pa tek onda napraviti obje rotacije. Ovo će biti programski lakše.

Početno stanje prije rotacije i način rotacije:



Nakon zamjene bitova X i Y:



Nakon neovisnih rotacija izvedenih na R0 i R1:



Kako najjednostavnije zamijeniti bitove X i Y? Oni mogu imati sljedeće vrijednosti:

X	Y	operacija
0	0	-
0	1	treba ih zamijeniti
1	0	treba ih zamijeniti
1	1	-

zamjenu bitova koji su
različiti jednostavno
ostvarimo tako da ih
komplementiramo

- Komplementiranje znamo napraviti od prije: XOR s maskom
- Takvu masku dobivamo ako napravimo XOR između R0 i R1 i zatim obrišemo sve bitove osim bita na poziciji X i Y (najniži bit)

```
; stvaranje maske u R3 (za zamjenu X i Y)
; ako su bitovi X i Y jednaki => maska=0
; ako su bitovi X i Y različiti => maska=1

XOR    R0, R1, R3      ; Usporedi najniže bitove
AND    R3, 1, R3       ; u R0 i R1, tj. bitove X i Y.

; zamjena najnižih bitova R0 i R1 (tj. X i Y)

XOR    R3, R0, R0
XOR    R3, R1, R1

; Nezavisna rotacija R0 i R1

ROTR   R0, 1, R0
ROTR   R1, 1, R1

HALT
```

Potprogrammi

Potprogrami

- Namjena potprograma u asemblerском програмирању ista je kao i u višim programskim jezicima.
- Čemu služi potprogram znate već od prije (iz PIPI-ja):
 - bolja modularnost i organizacija programa
 - lakše programiranje
 - povećana čitljivost
 - lakše održavanje programa
 - manje zauzeće memorije

Potprogrami

- Kao i u višim programskim jezicima, moramo znati kako se koriste potprogrami, ali u asemberskom programiranju moramo dodatno znati i neke detalje "niže razine"
- Trebamo znati:
 - Kako se potprogram poziva i kako se vraćamo iz potprograma
 - Gdje se sprema povratna adresa
 - Kako se prenose parametri i vraćaju rezultati iz potprograma
 - Gdje se čuvaju lokalne varijable potprograma
 - Kako se ostvaruje da stanja registara iz glavnog programa ne budu promijenjena za vrijeme potprograma
- U višim programskim jezicima, o ovim stvarima uglavnom ne moramo voditi računa jer se o njima brine prevoditelj (koji stvara asemblersku verziju našeg programa)

Potprogrami

poziv
potprograma

POTP

mjesto
povratka iz

potprograma

```
...  
SUB R0, R1, R2  
CALL POTP  
ADD R3, R4, R5  
...
```

glavni
program

adresa (labela)
potprograma

POTP

povratak iz

potprograma

POTP

```
POTP XOR R1, R4, R3
```

...

```
LOAD R2, (R5)
```

```
RET
```

potprogram
POTP

Potprogrami - CALL i RET

- **Podsjetnik:** kako rade naredbe CALL i RET?
 - Naredba: **CALL adresa_potprograma**
 - sprema povratnu adresu iz PC-a na stog
 - to je adresa naredbe IZA naredbe CALL, jer za vrijeme izvođenja naredbe CALL, PC već pokazuje na sljedeću naredbu, tj. sadrži povratnu adresu
 - skače na zadanu adresu **adresa_potprograma**
 - Naredba: **RET**
 - uzima podatak (povratnu adresu) sa stoga
 - skače na tu povratnu adresu

Potprogrami - CALL i RET

- **Gdje se u memoriji nalazi stog ???**
- Položaj stoga u memoriji definiramo programski na početku glavnog programa gdje **MORAMO inicijalizirati pokazivač stoga** (SP)
- Ako to zaboravimo učiniti, doći će do greške u programu (u ATLAS-u će vjerojatno javiti grešku "čitanje sa sabirnice u stanju High Z")

Potprogrami - stog

Tipična organizacija memorije: 0000 0000

- Na najnižim adresama se nalazi program i variable/konstante.
- Iza toga se nalazi područje gomile (heap). Gomila dinamički raste prema najvišim adresama. (*nema veze sa strukturom podataka koja se također naziva heap - iz "Algoritama i struktura podataka"*).
- Na najvišim adresama se nalazi stog koji raste prema nižim adresama.

FFFF FFFF

Program i
variable/konst.
(fiksna veličina)

Gomila (heap):
malloc i free



Stog (stack):
push i pop

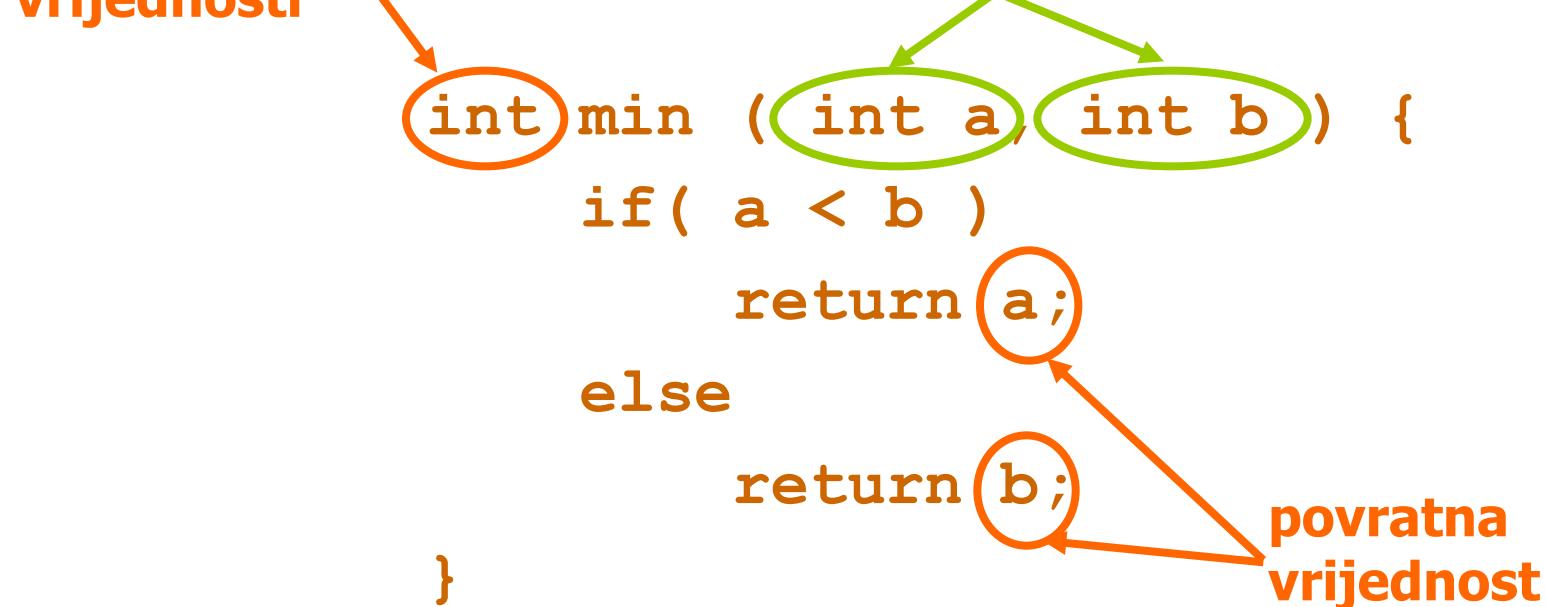
Potprogrami - stog

- Mi ćemo prepostaviti da je na FRISC spojena memorija od 64 KB. Najviša adresa (8-bitne memorijske lokacije) u memoriji od 64 KB je FFFF
- Na stog se uvijek stavljuju 32-bitni podatci pa im adrese moraju biti djeljive sa 4
- FFFC je adresa 32-bitnog podatka koji je "na zadnjoj 32-bitnoj lokaciji", tj. "na najvišim lokacijama memorije"
- Budući da SP pokazuje na zadnji podatak na stogu (podatak koji na početku još ne postoji), onda SP inicijaliziramo na jednu 32-bitnu riječ dalje: $4 + \text{FFFF} = 10000$



Potprogrami - Parametri i rezultati

tip povratne
vrijednosti



Potprogrami - Parametri i rezultati

- U višim programskim jezicima parametri i povratna vrijednost imaju tipove koje provjerava prevoditelj
- U asembleru nema nikakvih provjera ni tipova:
 - Dužnost je programera da se brine kakve podatke šalje u potprogram i kakve rezultate prima iz potprograma
 - Velika fleksibilnost u programiranju
 - Velika mogućnost greške u programu
- Terminologija:
 - parametar (formalni parametar) - "varijabla" u potprogramu preko koje potprogram prima vrijednosti od pozivatelja
 - argument (stvarni parametar) - stvarna vrijednost koju pozivatelj šalje potprogramu preko njegovih parametara

Potprogrami - Parametri i rezultati

- Tri najčešća načina slanja parametara i vraćanja povratne vrijednosti:
 - Pomoću registara
 - Pomoću fiksnih memorijskih lokacija
 - Pomoću stoga
- Teorijski je moguće slobodno kombinirati više različitih načina prijenosa u jednom potprogramu. Na primjer:
 - prvi parametar prenosi se registrom, drugi stogom, a ostali pomoću fiksnih lokacija
 - prvi rezultat se vraća stogom, a drugi fiksnom lokacijom

Prijenos registrima

Potprogrami - Prijenos registratorima

- Za svaki potprogram zasebno, propiše se preko kojih registara prima podatke i preko kojih vraća rezultate
- Pozivatelj potprograma (glavni program ili bilo koji drugi potprogram) prije poziva mora staviti argumente u zadane registre
- Potprogram polazi od prepostavke da su u zadanim registratorima argumenti i koristi ih pri izračunavanju rezultata
- Potprogram mora staviti rezultat u zadane registre
- Pozivatelj, nakon povratka iz potrograma, prepostavlja da se u zadanim registratorima nalaze rezultati i koristi ih

Potprogrami - Prijenos registrima

Primjer:

Napisati potprogram koji izračunava logičku operaciju NILI između registara R0 i R1 i vraća rezultat registrom R2. Glavni program iz memorije učitava dva podatka za koje računa NILI (pomoću potprograma) i rezultat spremi natrag u memoriju.

Rješenje:

(na sljedećem slajdu)

>>>

<<<

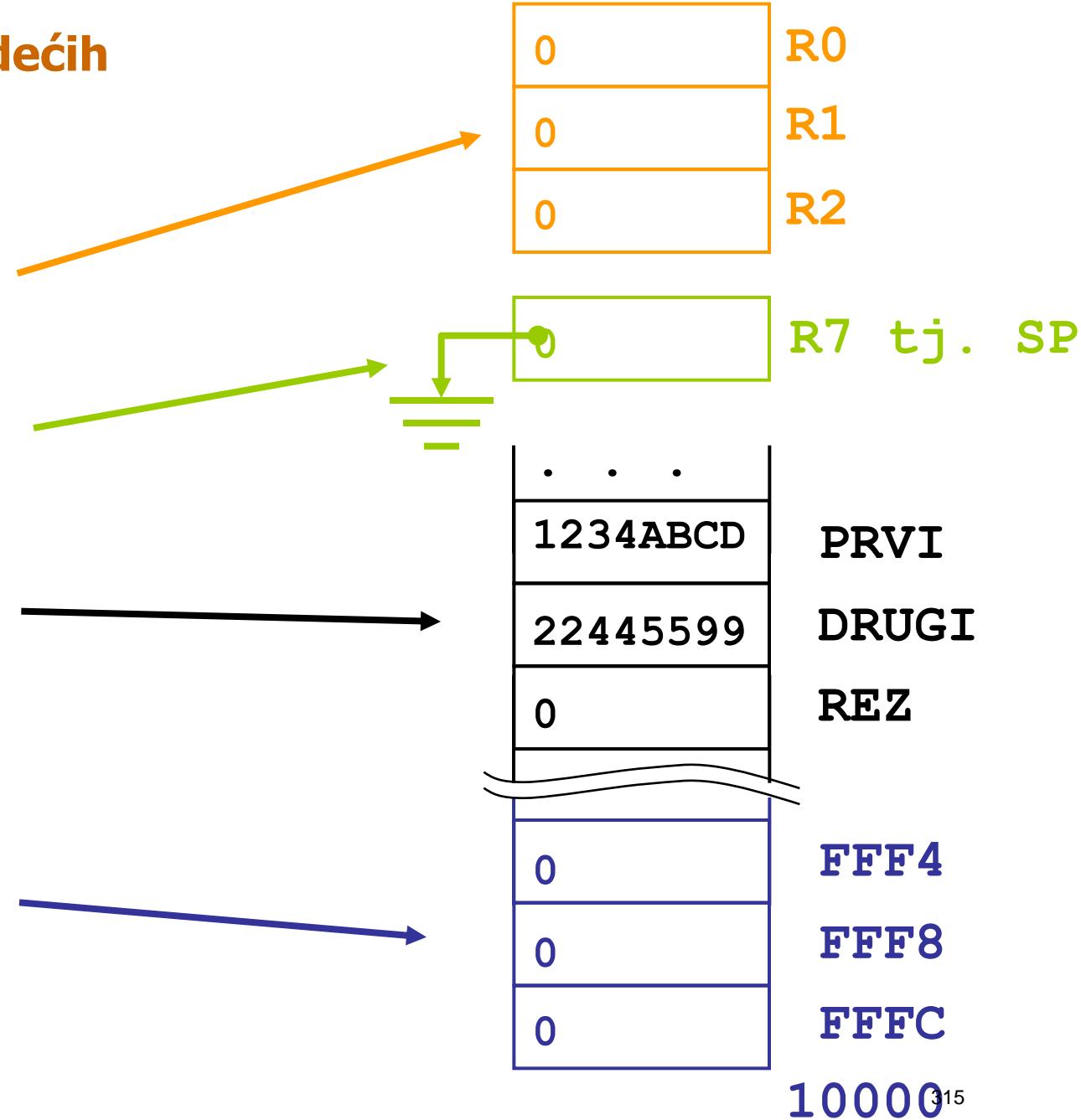
Pratimo stanje sljedećih registara/lokacija:

tri registra opće namjene koje koristimo

pokazivač stoga

dio memorije s podatcima

dio memorije u kojem je stog



<<< Izvođenje programa:

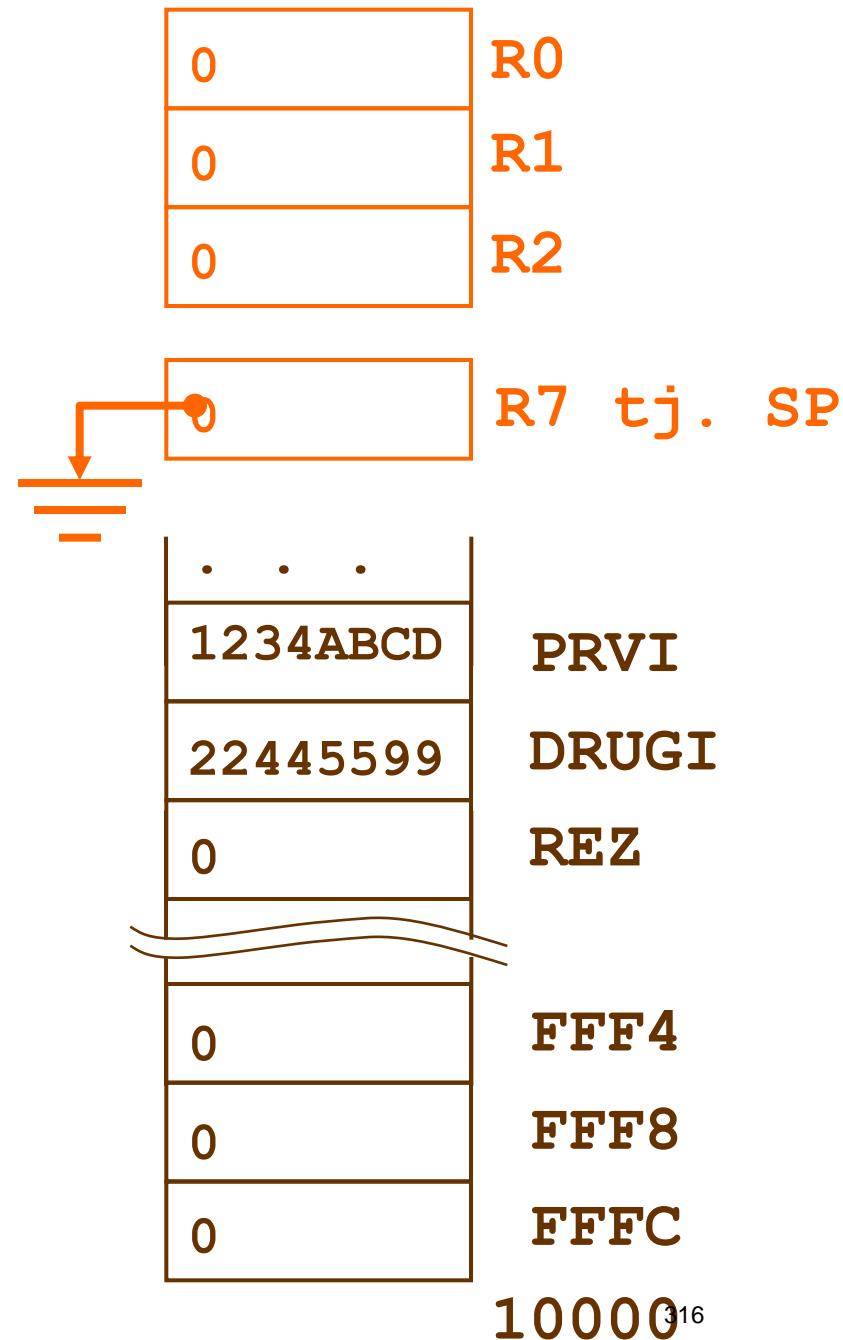
GLAVNI MOVE 10000, SP

LOAD R0, (PRVI)
LOAD R1, (DRUGI)
CALL NILI
STORE R2, (REZ)

HALT

NILI OR R0, R1, R2
XOR R2, -1, R2
RET

PRVI DW 1234ABCD
DRUGI DW 22445599
REZ DW 0



<<< Izvođenje programa:

GLAVNI MOVE 10000, SP ←

```

LOAD R0 , (PRVI)
LOAD R1 , (DRUGI)
CALL NILI
STORE R2 , (REZ)
```

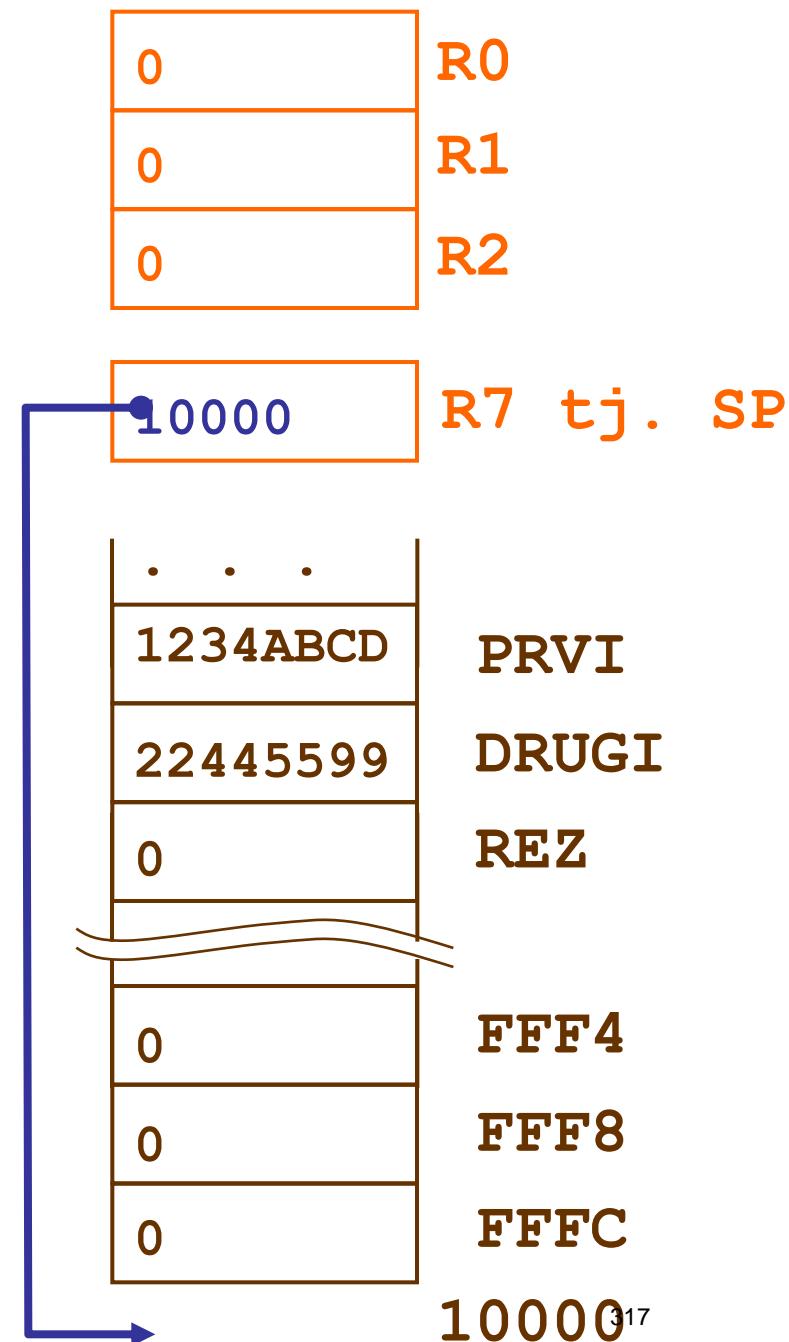
HALT

```

NILI OR R0 , R1 , R2
XOR R2 , -1 , R2
RET
```

```

PRVI DW 1234ABCD
DRUGI DW 22445599
REZ DW 0
```



<<< Izvođenje programa:

GLAVNI MOVE 10000, SP

LOAD R0, (PRVI) ←

LOAD R1, (DRUGI)

CALL NILI

STORE R2, (REZ)

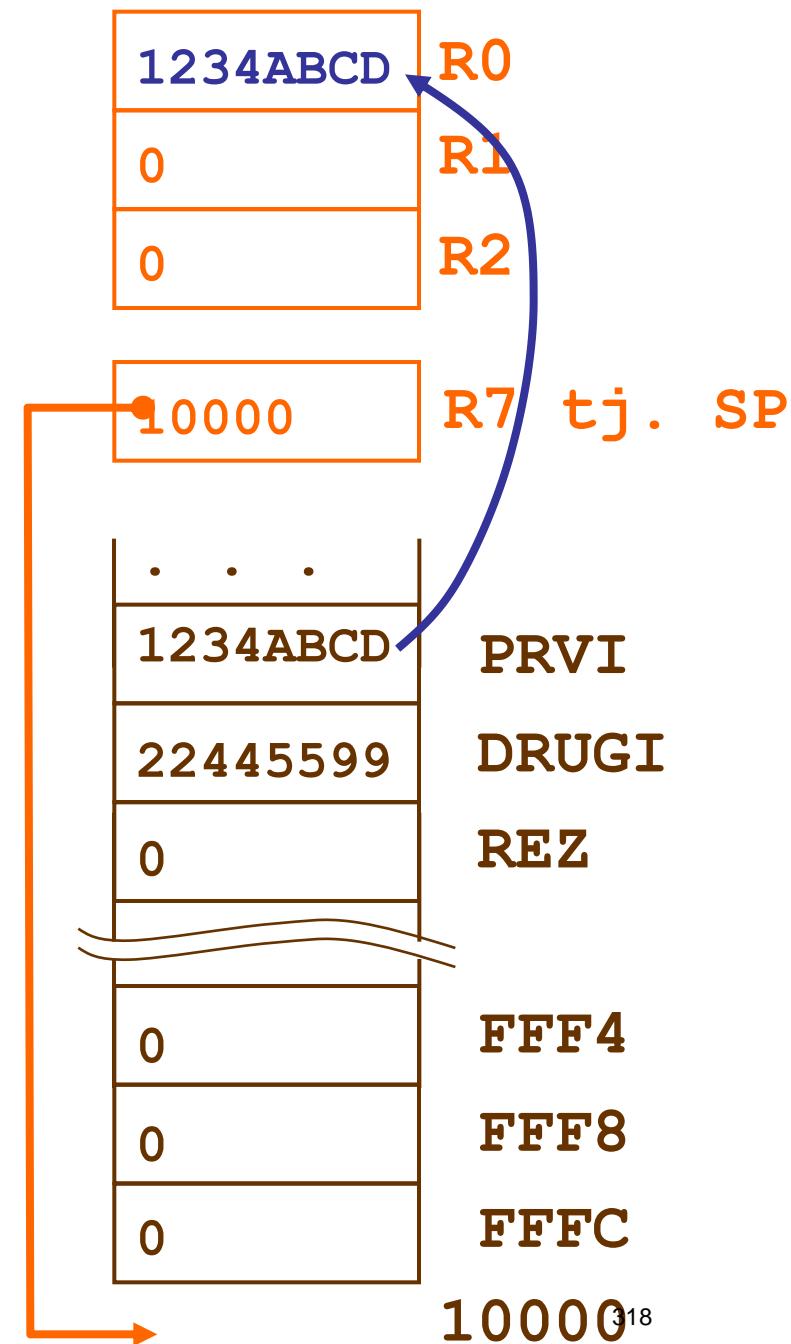
HALT

NILI OR R0, R1, R2
XOR R2, -1, R2
RET

PRVI DW 1234ABCD

DRUGI DW 22445599

REZ DW 0



<<< Izvođenje programa:

```

GLAVNI MOVE 10000, SP

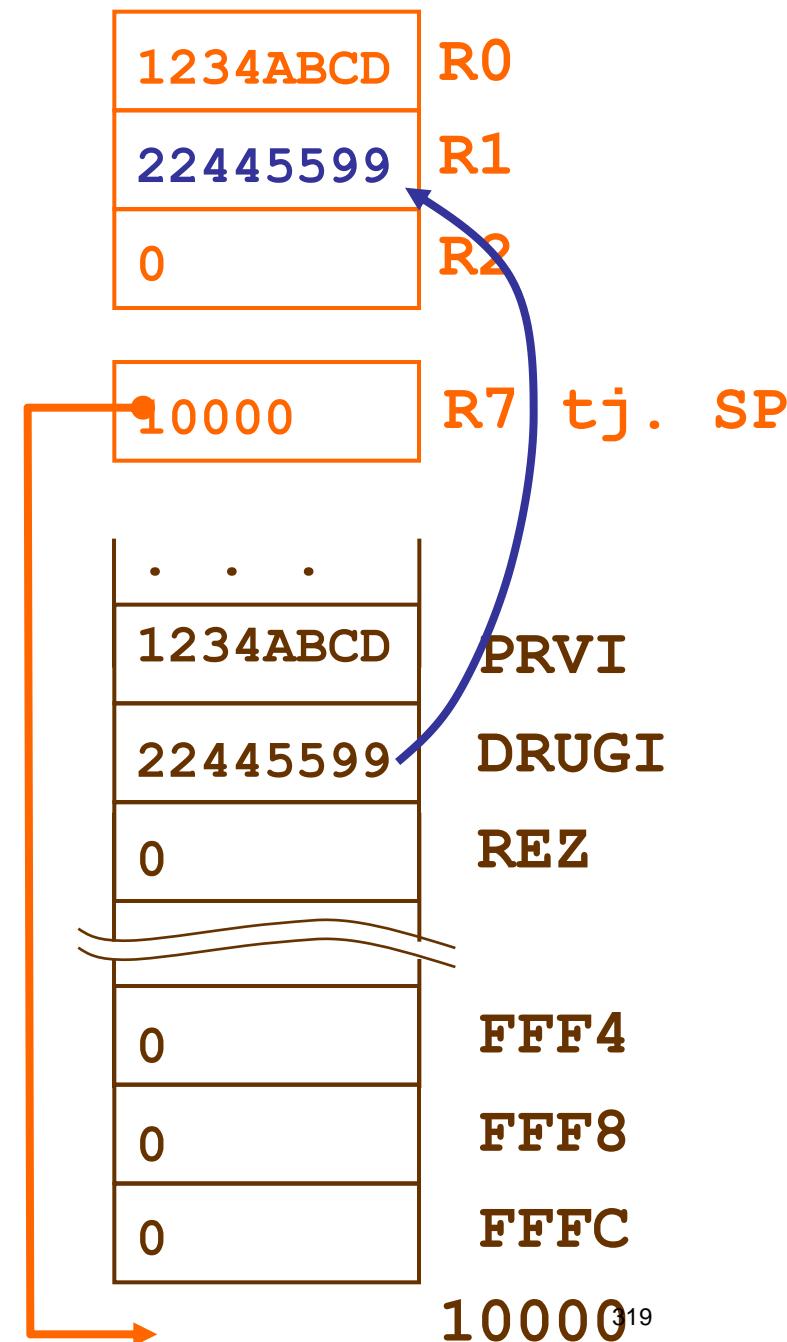
LOAD R0, (PRVI)
LOAD R1, (DRUGI) ←
CALL NILI
STORE R2, (REZ)

HALT

NILI OR R0, R1, R2
XOR R2, -1, R2
RET

PRVI DW 1234ABCD
DRUGI DW 22445599
REZ DW 0

```



<<< Izvođenje programa:

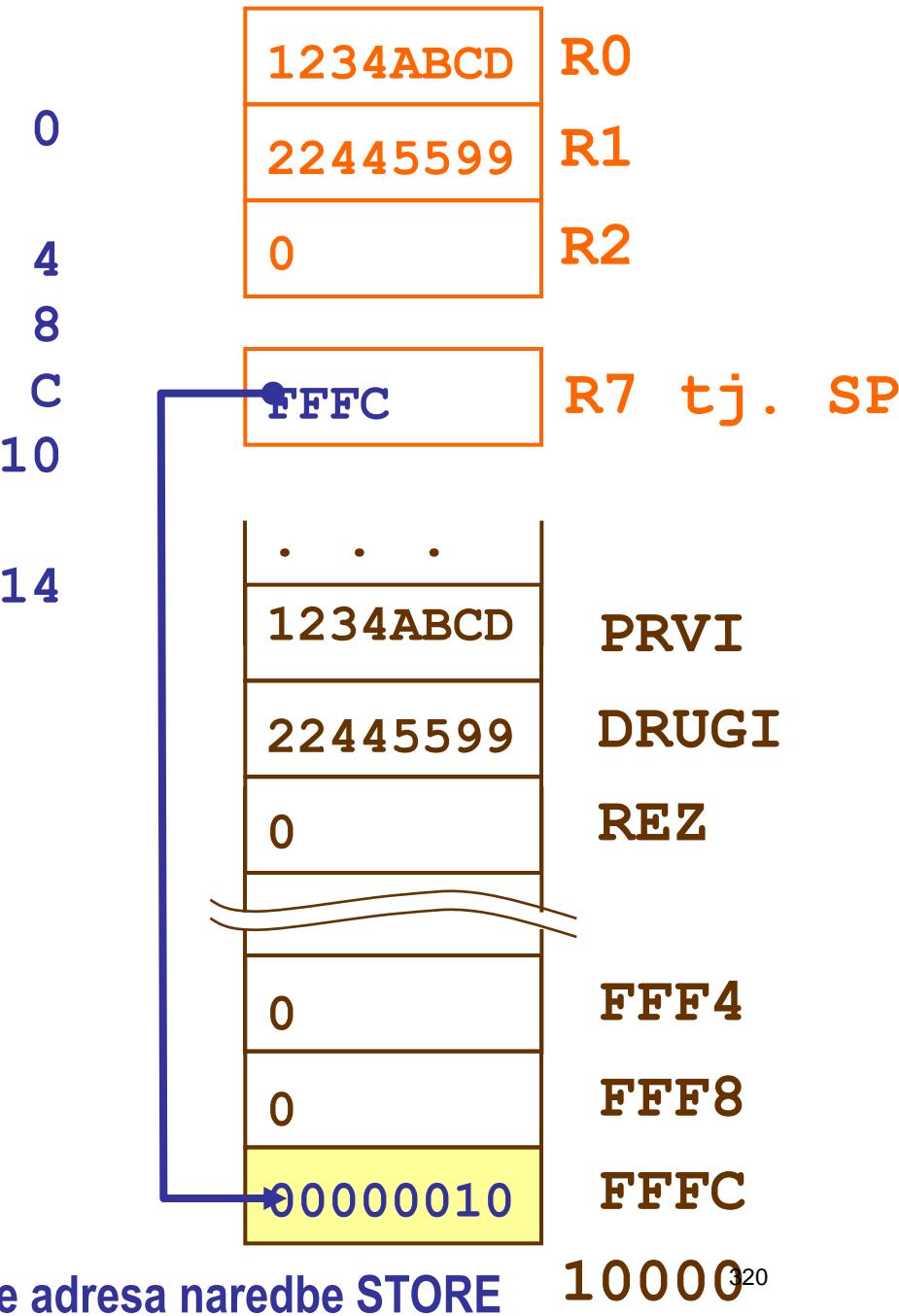
GLAVNI MOVE 10000, SP

LOAD R0, (PRVI)
LOAD R1, (DRUGI)
CALL NILI ←
STORE R2, (REZ)

HALT

NILI OR R0, R1, R2
XOR R2, -1, R2
RET

PRVI DW 1234ABCD
DRUGI DW 22445599
REZ DW 0



<<< Izvođenje programa:

GLAVNI MOVE 10000, SP

```

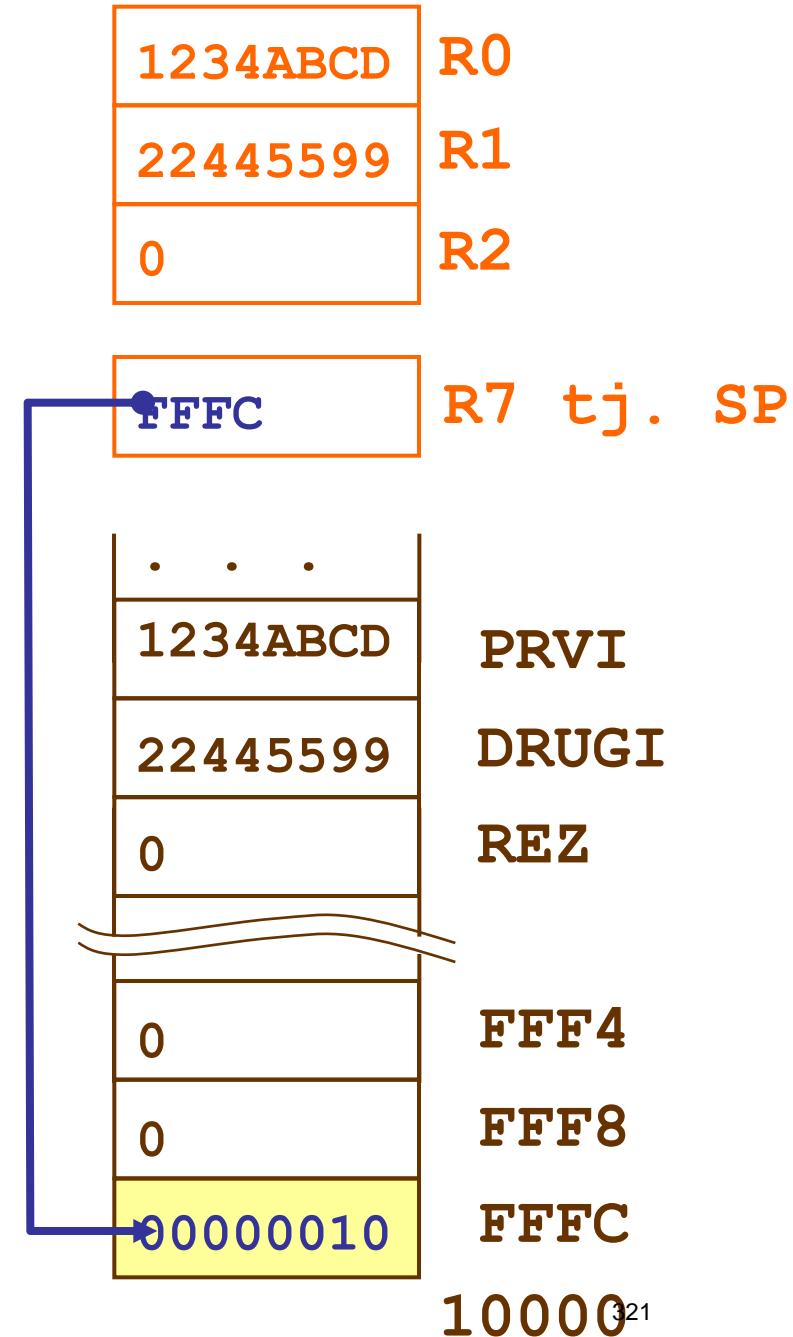
LOAD  R0 , (PRVI)
LOAD  R1 , (DRUGI)
CALL  NILI      ←
STORE R2 , (REZ)

```

HALT

NILI OR R0, R1, R2
 XOR R2, -1, R2
 RET

PRVI DW 1234ABCD
 DRUGI DW 22445599
 REZ DW 0



<<< Izvođenje programa:

GLAVNI MOVE 10000, SP

```

LOAD  R0 , (PRVI)
LOAD  R1 , (DRUGI)
CALL  NILI
STORE R2 , (REZ)
```

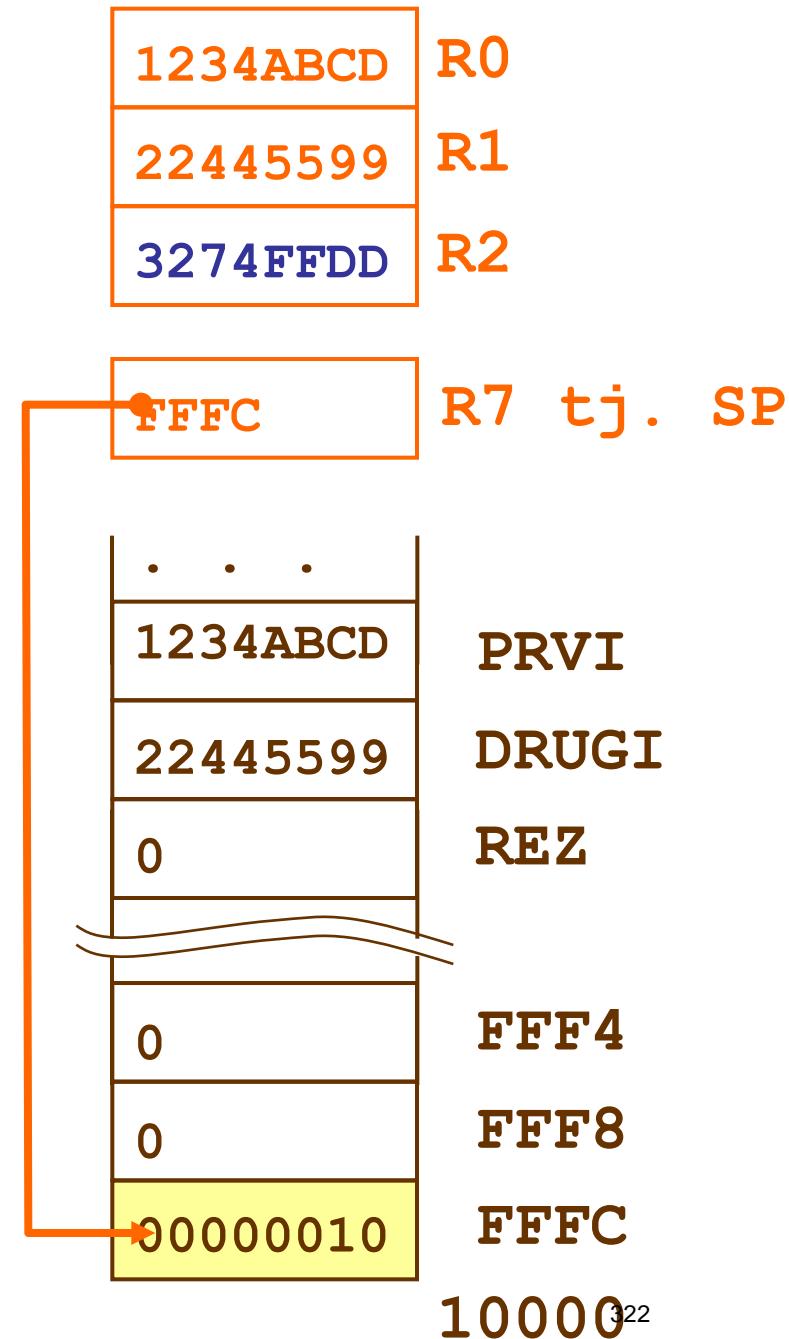
HALT

```

NILI   OR  R0 , R1 , R2 ←
      XOR R2 , -1 , R2
      RET
```

```

PRVI  DW  1234ABCD
DRUGI DW  22445599
REZ   DW  0
```



<<< Izvođenje programa:

GLAVNI MOVE 10000, SP

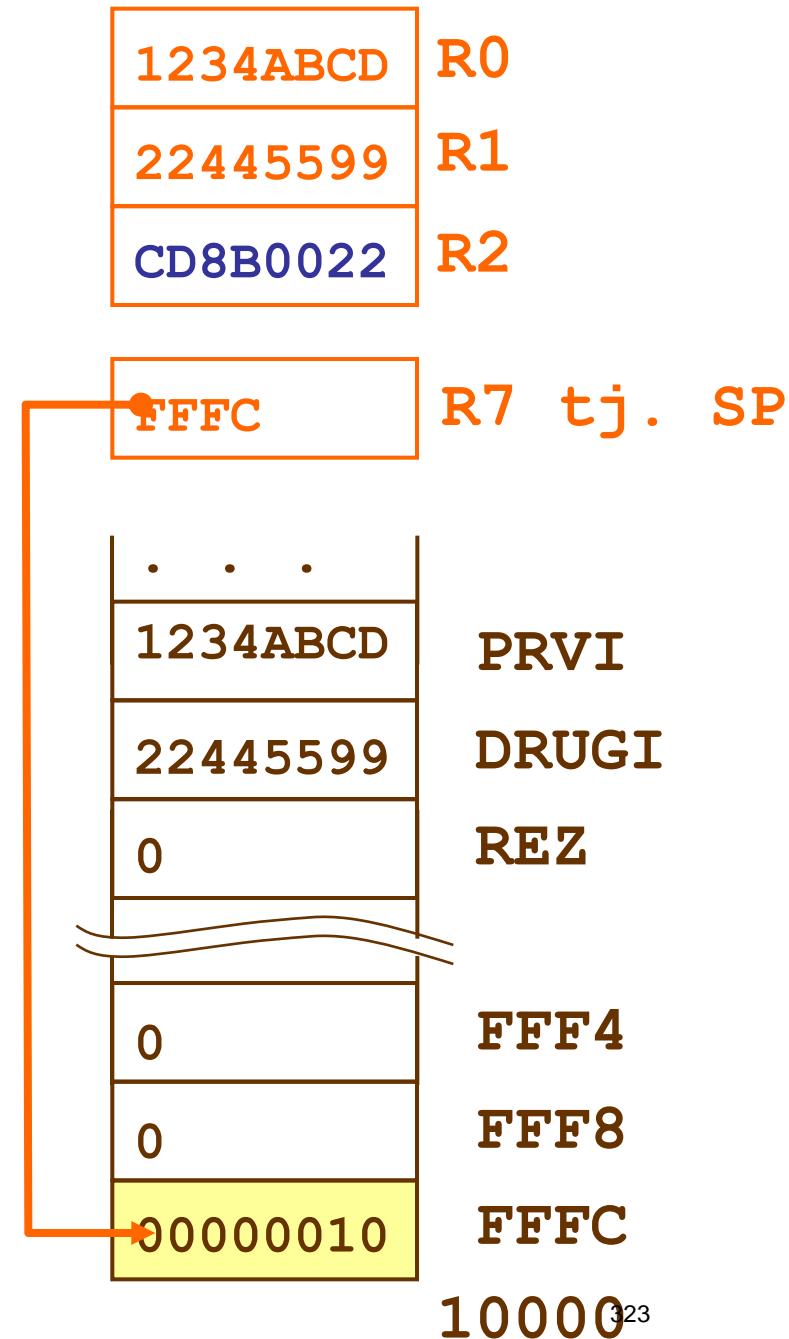
```

LOAD  R0 , (PRVI)
LOAD  R1 , (DRUGI)
CALL  NILI
STORE R2 , (REZ)
```

HALT

NILI OR R0, R1, R2
 XOR R2, -1, R2 ←
 RET

PRVI DW 1234ABCD
 DRUGI DW 22445599
 REZ DW 0



<<< Izvođenje programa:

GLAVNI MOVE 10000, SP

```

LOAD R0, (PRVI)
LOAD R1, (DRUGI)
CALL NILI
STORE R2, (REZ)
```

HALT

NILI OR R0, R1, R2
 XOR R2, -1, R2
 RET

PRVI DW 1234ABCD
 DRUGI DW 22445599
 REZ DW 0

1234ABCD
22445599
CD8B0022

R0
R1
R2

10000

R7 t.j. SP

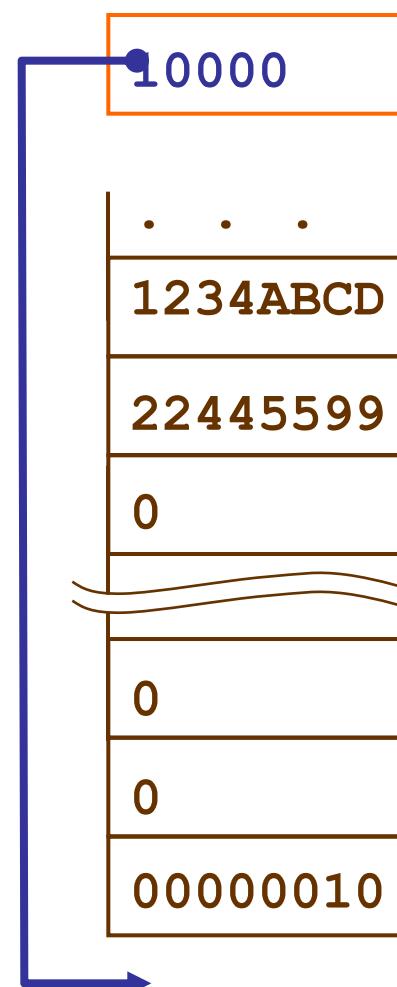
...
1234ABCD
22445599
0

PRVI
DRUGI
REZ

0
0
00000010

FFF4
FFF8
FFFC

10000³²⁴



<<< Izvođenje programa:

GLAVNI MOVE 10000, SP

```

LOAD  R0 , (PRVI)
LOAD  R1 , (DRUGI)
CALL  NILI
STORE R2 , (REZ)
```

HALT

NILI OR R0, R1, R2
 XOR R2, -1, R2
 RET

PRVI DW 1234ABCD
 DRUGI DW 22445599
 REZ DW 0

1234ABCD
22445599
CD8B0022

R0
R1
R2

10000

R7 t.j. SP

...
1234ABCD
22445599
0

PRVI
DRUGI
REZ

0
0
00000010

FFF4
FFF8
FFFC

10000



<<< Izvođenje programa:

GLAVNI MOVE 10000, SP

```

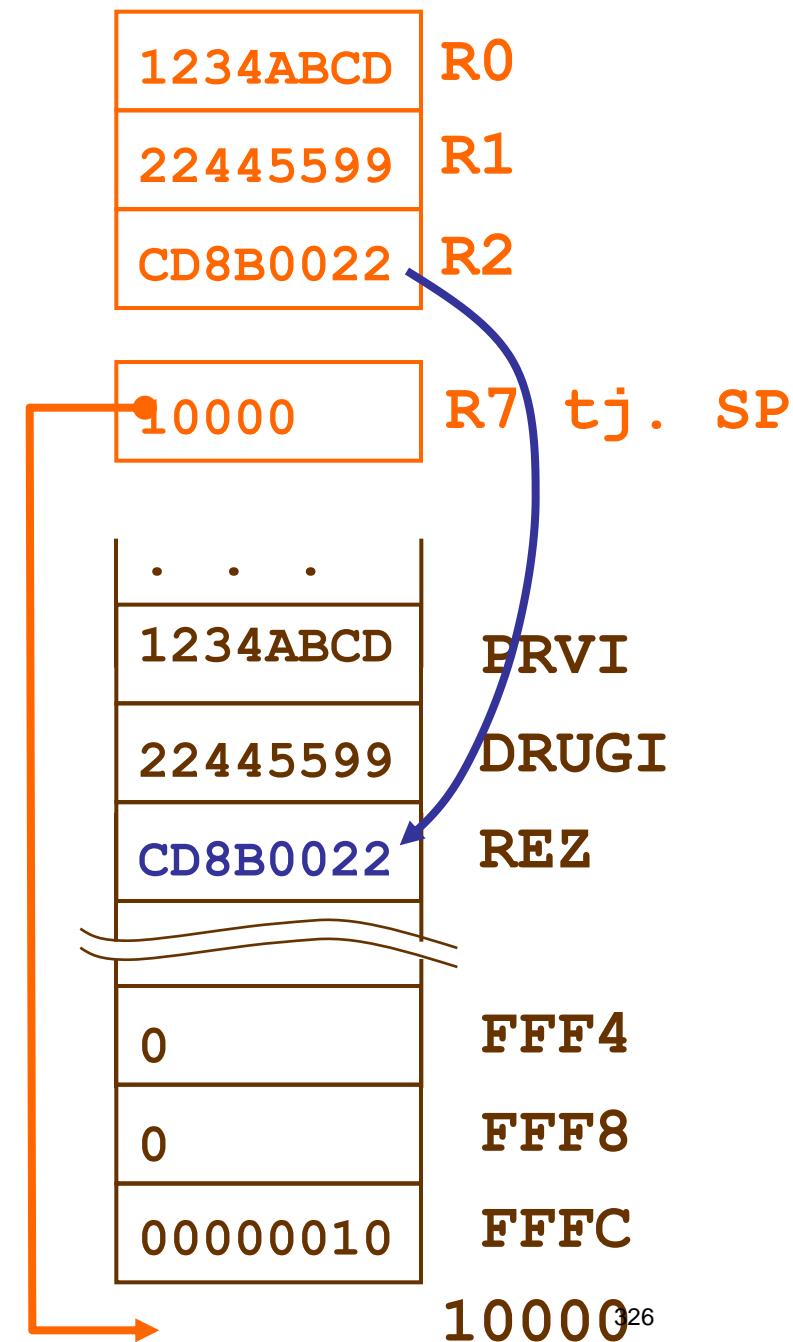
LOAD  R0 , (PRVI)
LOAD  R1 , (DRUGI)
CALL  NILI
STORE R2 , (REZ) ←

```

HALT

NILI OR R0, R1, R2
 XOR R2, -1, R2
 RET

PRVI DW 1234ABCD
 DRUGI DW 22445599
 REZ DW 0



<<< Izvođenje programa:

GLAVNI MOVE 10000, SP

```

LOAD  R0 , (PRVI)
LOAD  R1 , (DRUGI)
CALL  NILI
STORE R2 , (REZ)
```

HALT

NILI OR R0, R1, R2
 XOR R2, -1, R2
 RET

PRVI DW 1234ABCD
 DRUGI DW 22445599
 REZ DW 0

1234ABCD
22445599
CD8B0022

R0
R1
R2

10000

R7 t.j. SP

...
1234ABCD
22445599
CD8B0022

PRVI
DRUGI
REZ

0
0
00000010

FFF4
FFF8
FFFC

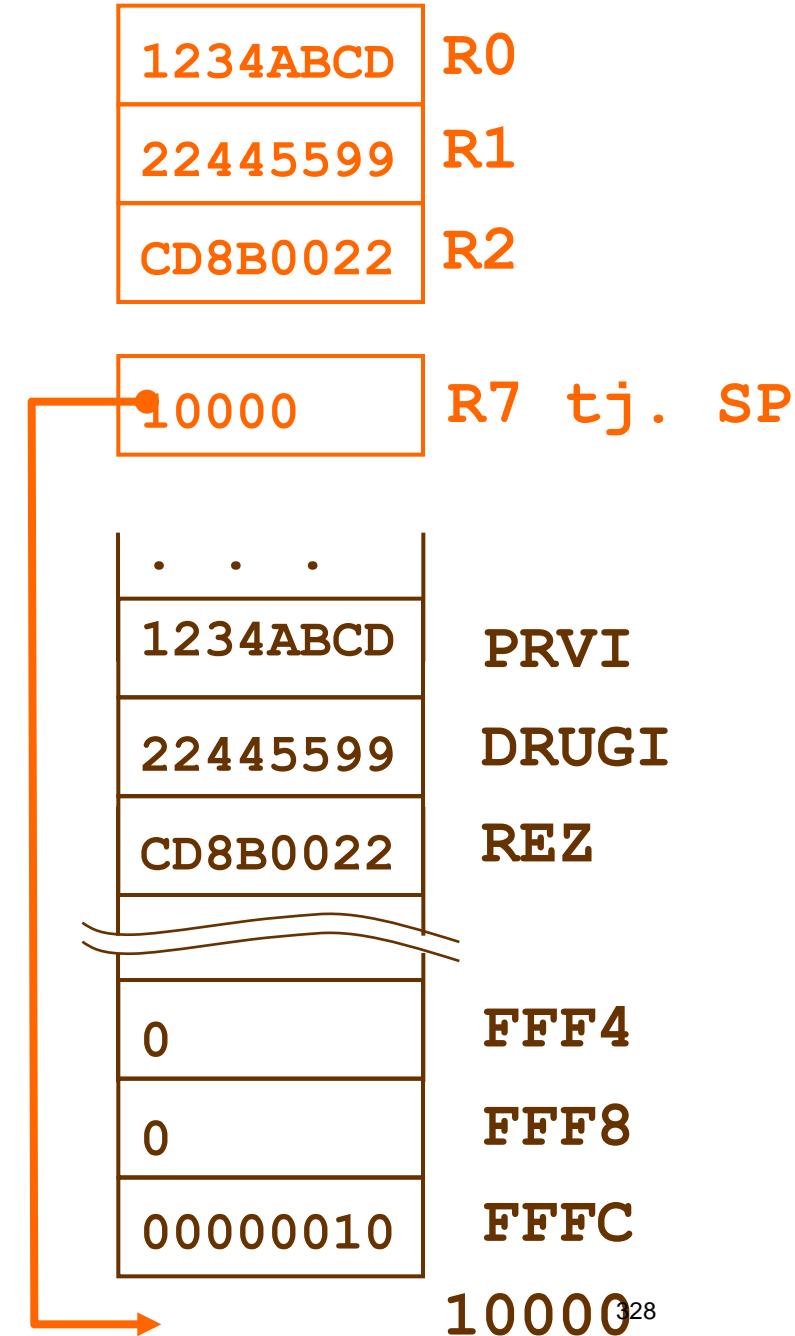
10000



Komentar:

Treba uočiti da ovaj potprogram mijenja samo sadržaj registra R2 (ne računajući R7). Međutim, R2 je register preko kojeg se vraća vrijednost, tj. glavni program očekuje da će taj register biti promijenjen nakon povratka iz potprograma. Zato glavni program prije poziva potprograma ne smije imati u R2 spremlijen nikakav podatak koji bi mu još mogao zatrebatи.

Registre R0 i R1 promijenio je glavni program, jer je preko njih slao parametre. Zato je za pretpostaviti da glavni program nije imao nikakve korisne podatke u R0 i R1 prije upisa parametara u njih.



<<< (kompletan listing programa s komentarima)

; glavni program

GLAVNI MOVE 10000, SP ;važno: inicijaliziraj SP !!!
LOAD R0, (PRVI) ;vrijednost u prvi parametar
LOAD R1, (DRUGI) ;vrijednost u drugi parametar
CALL NILI ;poziv potprograma
STORE R2, (REZ) ;spremanje rezultata
HALT

; potprogram NILI

NILI OR R0, R1, R2 ; koriste se parametri R0 i R1
XOR R2, -1, R2 ; rezultat se upisuje u R2
RET ; povratak iz potprograma

; podatci i mjesto za rezultat

PRVI DW 1234ABCD
DRUGI DW 22445599
REZ DW 0

Potprogrami - Prijenos registrima

• **Prijenos registrima**

- Prednosti:
 - Brz prijenos (rad s registrima je najbrži)
 - Jednostavan prijenos (sa stajališta programiranja)
- Nedostatci:
 - Može se prenijeti ograničen broj argumenata (najviše 7)
 - Registri obično čuvaju međurezultate i nisu slobodni za korištenje kao parametri
 - Nije moguće rekurzivno pozivanje potprograma

Prijenos fiksnim lokacijama

Potprogrami - Prijenos fiksnim lok.

- Pod fiksnim lokacijama misli se na memorijske lokacije čija adresa je unaprijed zadana i ne mijenja se tijekom programa
- Za svaki potprogram zasebno, propiše se preko kojih fiksnih memorijskih lokacija prima podatke i preko kojih vraća rezultate
- Pozivatelj prije poziva mora staviti argumente u zadane lokacije
- Potprogram polazi od pretpostavke da su u zadanim lokacijama argumenti i koristi ih pri izračunavanju rezultata
- Potprogram mora staviti rezultat u zadane lokacije
- Pozivatelj, nakon povratka iz potrograma, pretpostavlja da se u zadanim lokacijama nalaze rezultati i koristi ih

Potprogrami - Prijenos fiksnim lokacijama

Primjer:

Napisati potprogram koji izračunava logičku operaciju NILI. Parametri se nalaze na memoriskim lokacijama P1 i P2, a rezultat se vraća lokacijom R_NILI. Glavni program iz memorije učitava dva podatka za koje računa NILI (pomoću potprograma) i rezultat spremi natrag u memoriju.

Rješenje:

na sljedećem slajdu

>>>

<<< Izvođenje programa:

GLAVNI MOVE 10000, SP

LOAD R0, (PRVI)

STORE R0, (P1)

LOAD R0, (DRUGI)

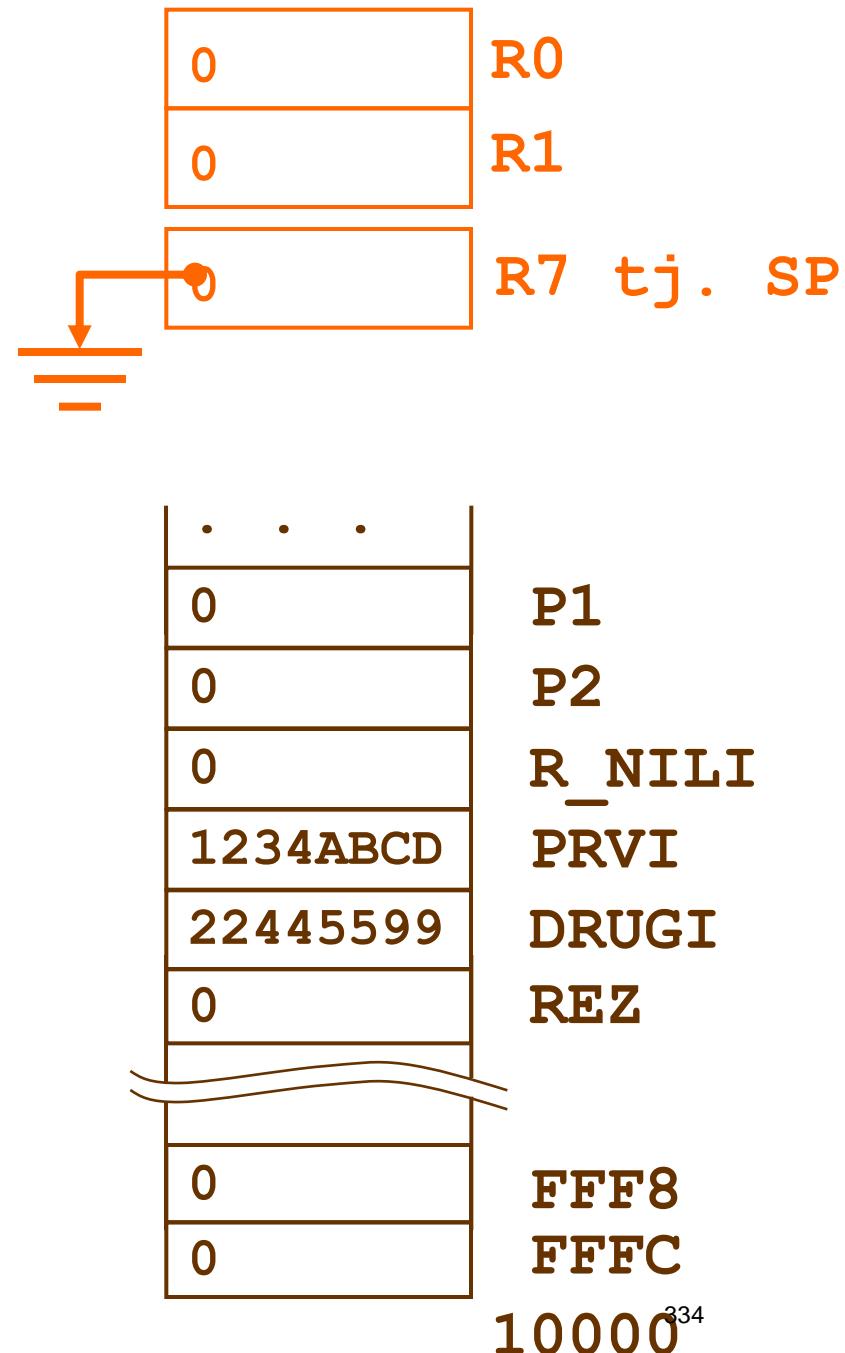
STORE R0, (P2)

CALL NILI

LOAD R0, (R_NILI)

STORE R0, (REZ)

HALT



<<< Izvođenje programa:

GLAVNI MOVE 10000, SP ←

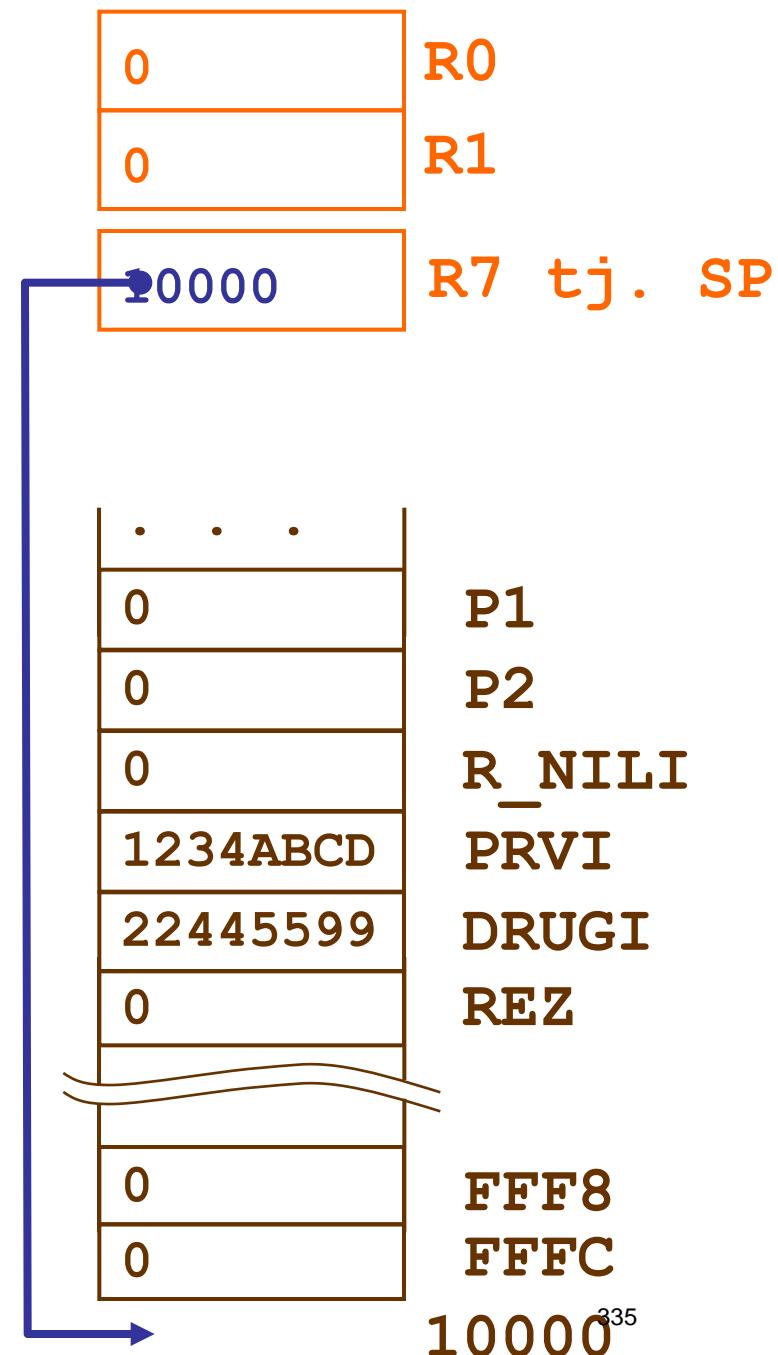
LOAD R0, (PRVI)
STORE R0, (P1)

LOAD R0, (DRUGI)
STORE R0, (P2)

CALL NILI

LOAD R0, (R_NILI)
STORE R0, (REZ)

HALT



<<< Izvođenje programa:

GLAVNI MOVE 10000, SP

LOAD R0, (PRVI) ←

STORE R0, (P1)

LOAD R0, (DRUGI)

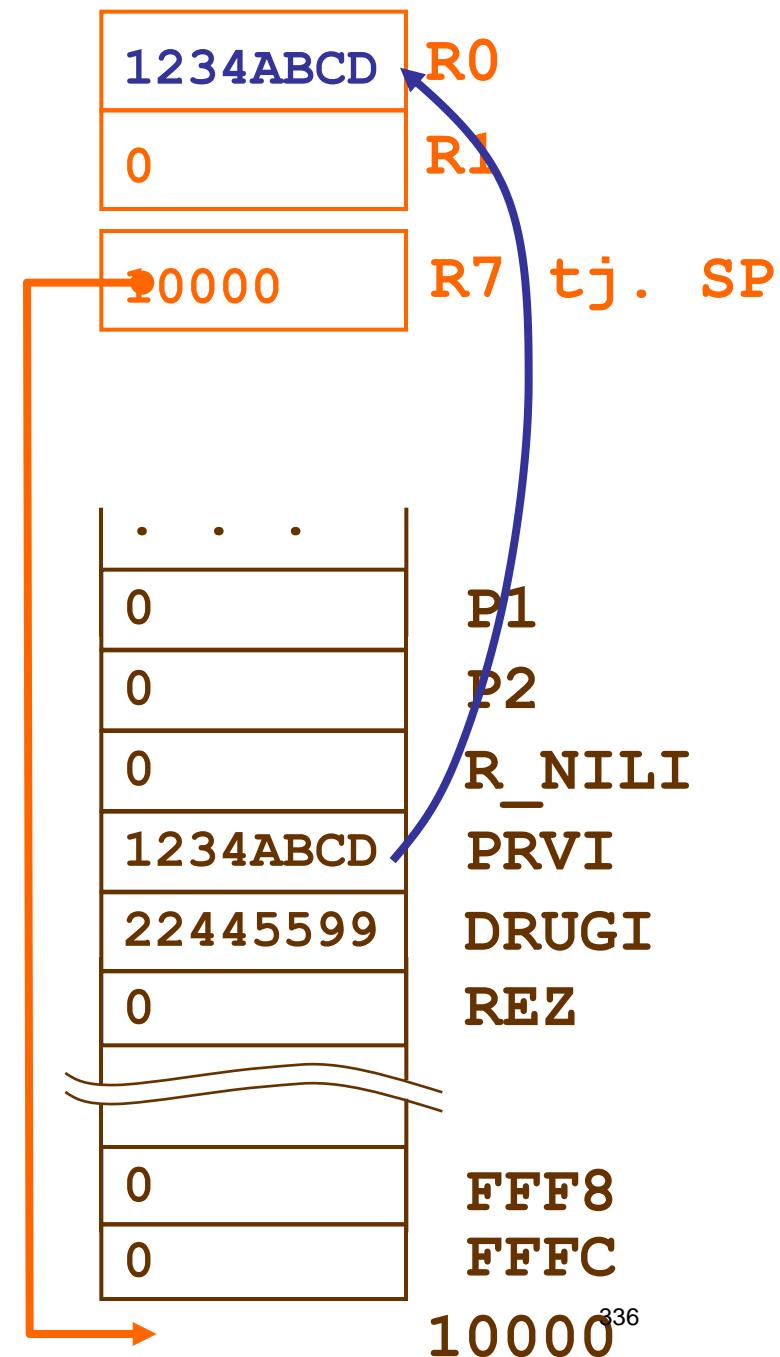
STORE R0, (P2)

CALL NILI

LOAD R0, (R_NILI)

STORE R0, (REZ)

HALT



<<< Izvođenje programa:

GLAVNI MOVE 10000, SP

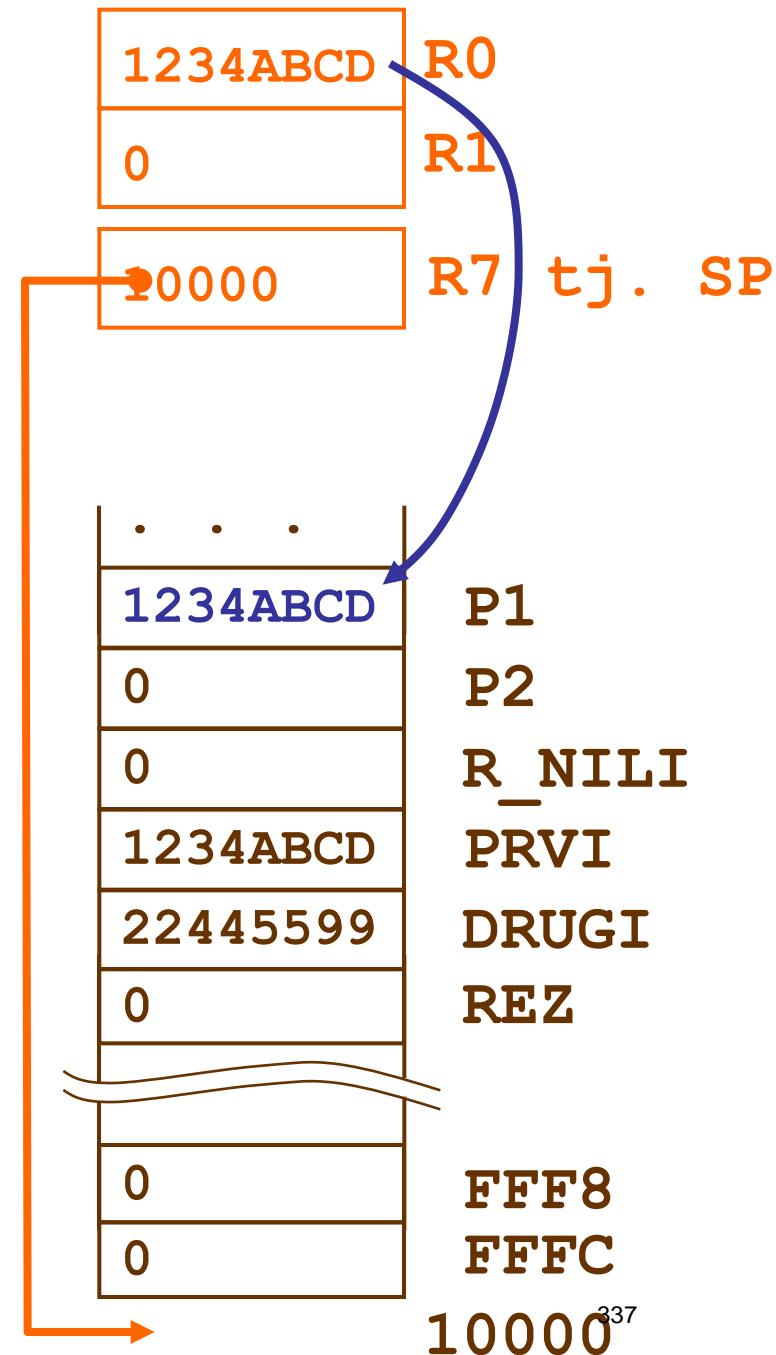
LOAD R0, (PRVI)
STORE R0, (P1) ←

LOAD R0, (DRUGI)
STORE R0, (P2)

CALL NILI

LOAD R0, (R_NILI)
STORE R0, (REZ)

HALT



<<< Izvođenje programa:

GLAVNI MOVE 10000, SP

LOAD R0, (PRVI)

STORE R0, (P1)

LOAD R0, (DRUGI) ←

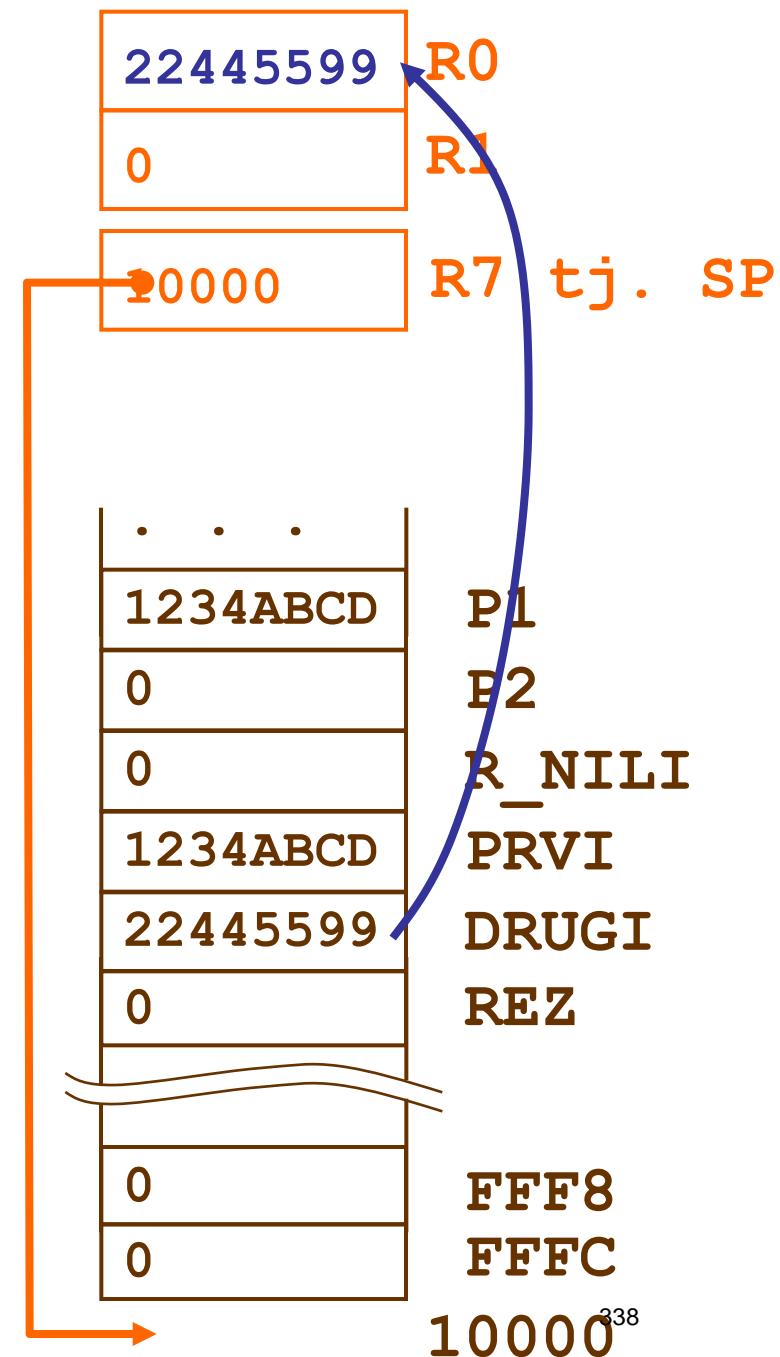
STORE R0, (P2)

CALL NILI

LOAD R0, (R_NILI)

STORE R0, (REZ)

HALT



<<< Izvođenje programa:

GLAVNI MOVE 10000, SP

LOAD R0, (PRVI)

STORE R0, (P1)

LOAD R0, (DRUGI)

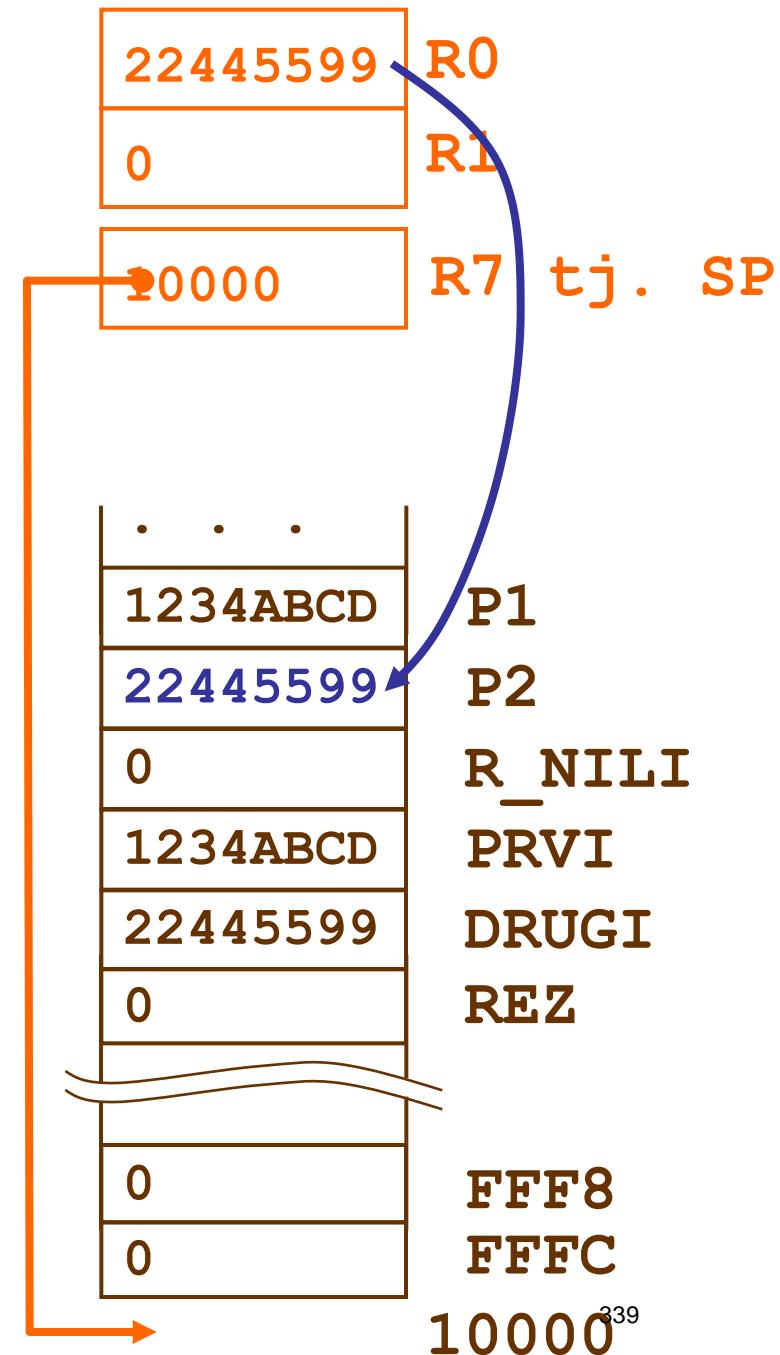
STORE R0, (P2) ←

CALL NILI

LOAD R0, (R_NILI)

STORE R0, (REZ)

HALT



<<< Izvođenje programa:

GLAVNI MOVE 10000, SP

LOAD R0, (PRVI)

STORE R0, (P1)

LOAD R0, (DRUGI)

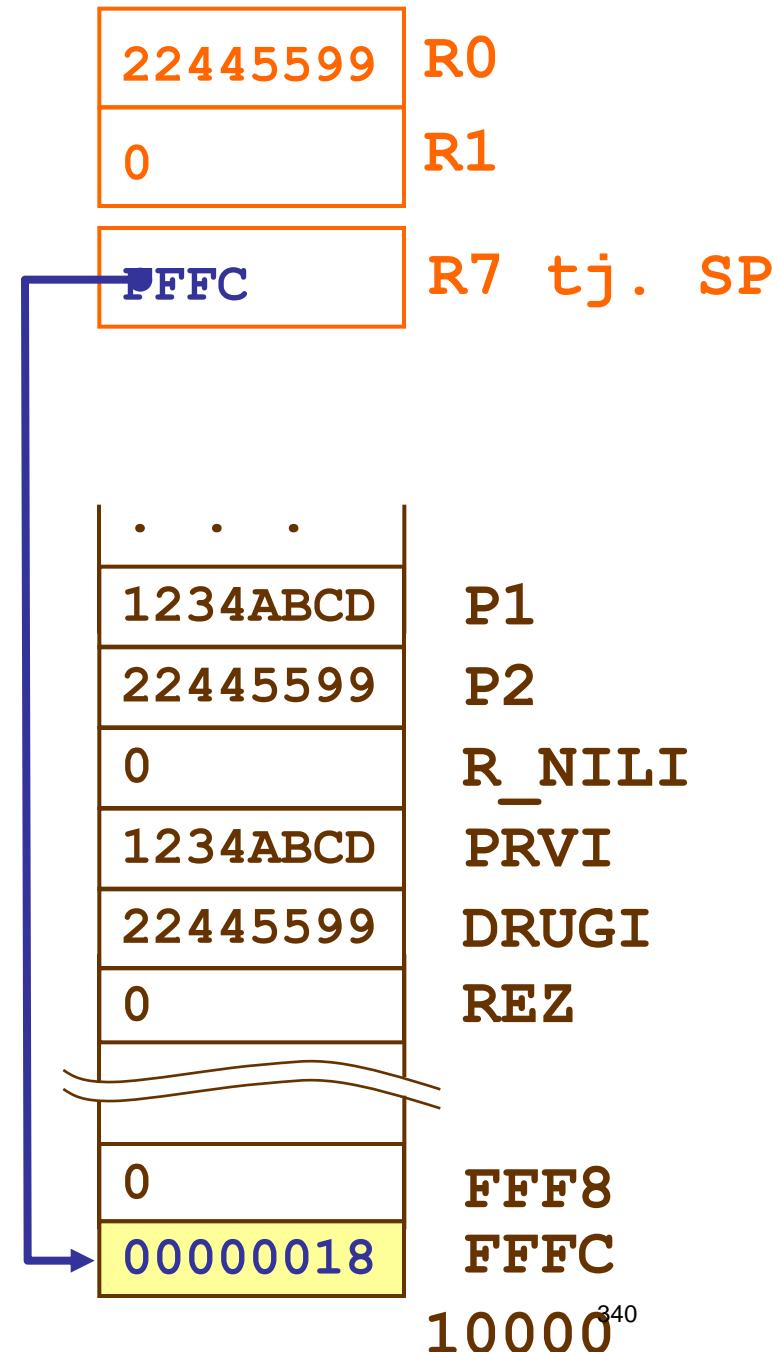
STORE R0, (P2)

CALL NILI

LOAD R0, (R_NILI)

STORE R0, (REZ)

HALT

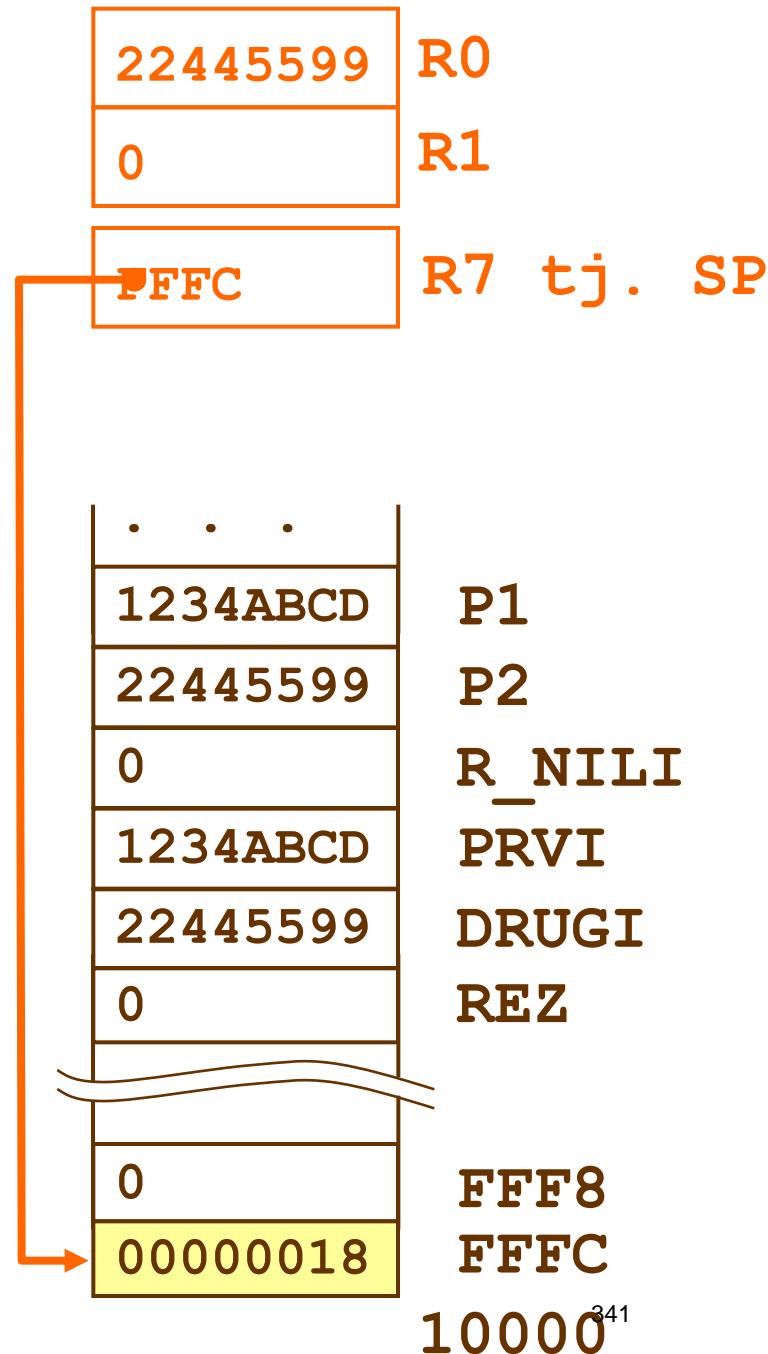


<<< Izvođenje programa:

```
NILI    LOAD   R0 , (P1)
        LOAD   R1 , (P2)

        OR      R0 , R1 , R0
        XOR   R0 , -1 , R0

        STORE  R0 , (R_NILI)
        RET
```

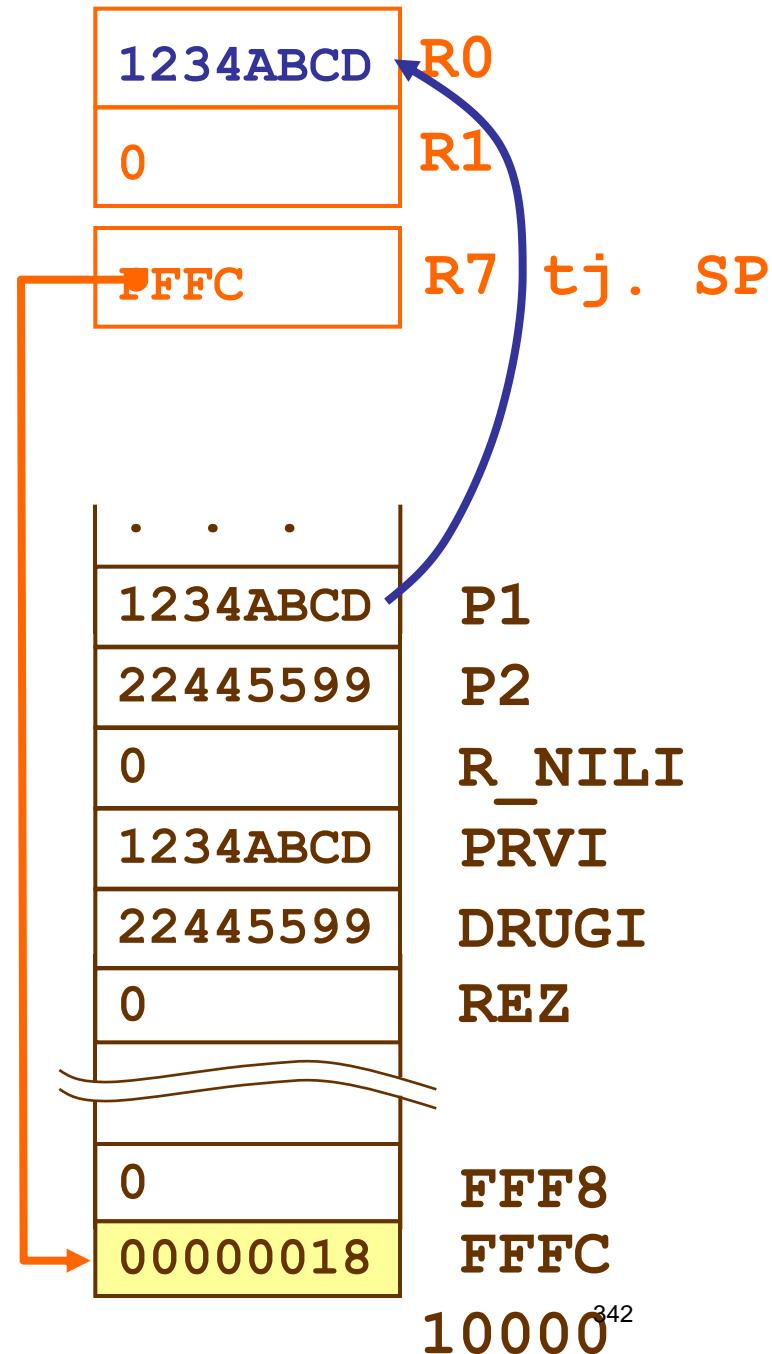


<<< Izvođenje programa:

```
NILI    LOAD   R0 , (P1)      ←
        LOAD   R1 , (P2)

        OR      R0 , R1 , R0
        XOR   R0 , -1 , R0

        STORE  R0 , (R_NILI)
        RET
```

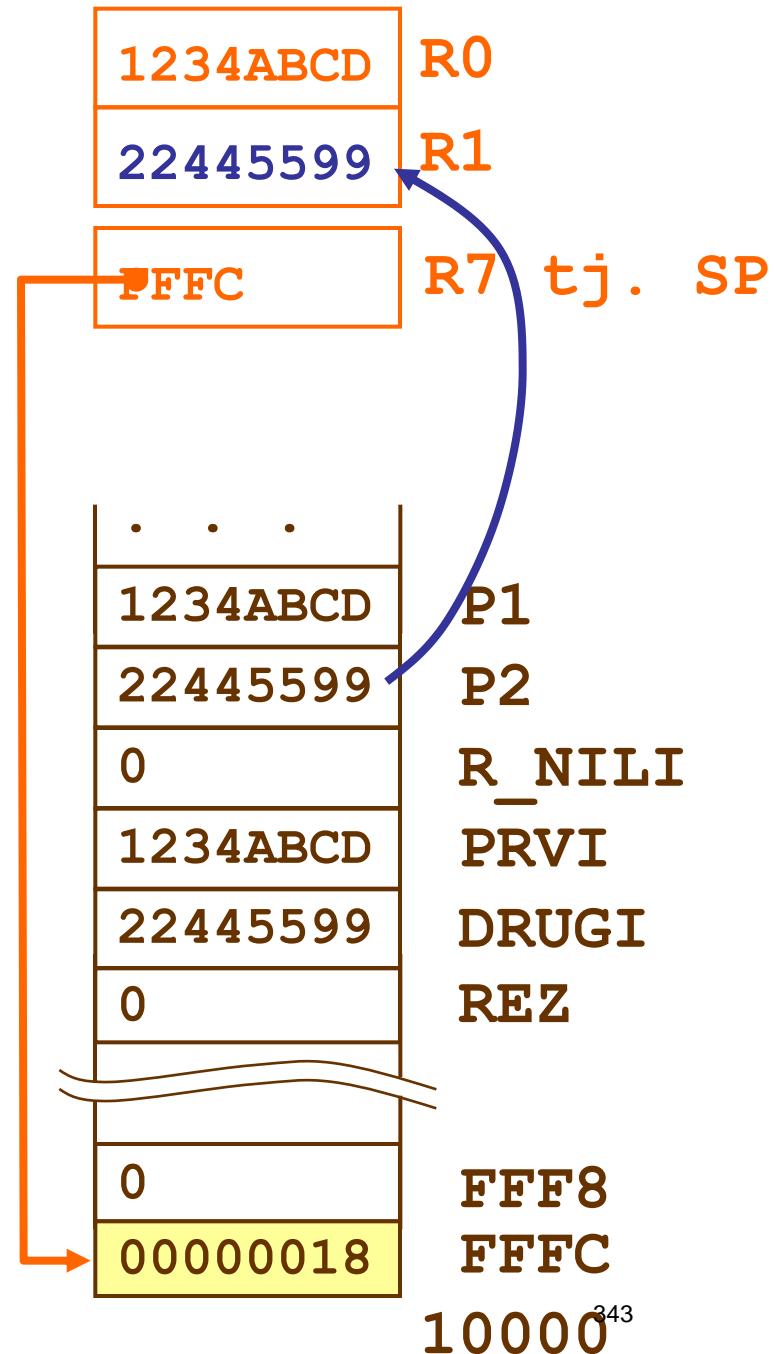


<<< Izvođenje programa:

```
NILI    LOAD   R0 , (P1)
       LOAD   R1 , (P2)
```

```
OR      R0 , R1 , R0
XOR    R0 , -1 , R0
```

```
STORE  R0 , (R_NILI)
RET
```

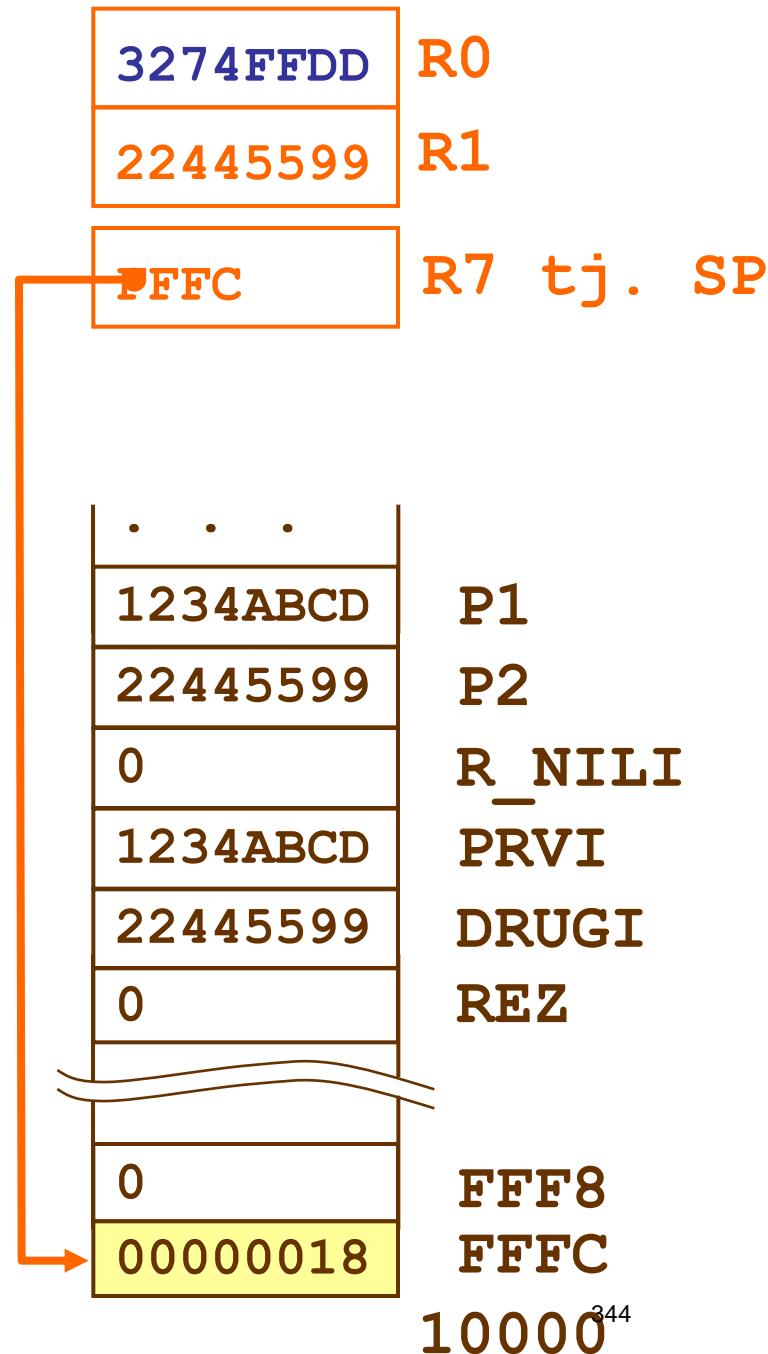


<<< Izvođenje programa:

```
NILI    LOAD   R0 , (P1)
        LOAD   R1 , (P2)

        OR      R0 , R1 , R0 ←
        XOR   R0 , -1 , R0

        STORE R0 , (R_NILI)
RET
```

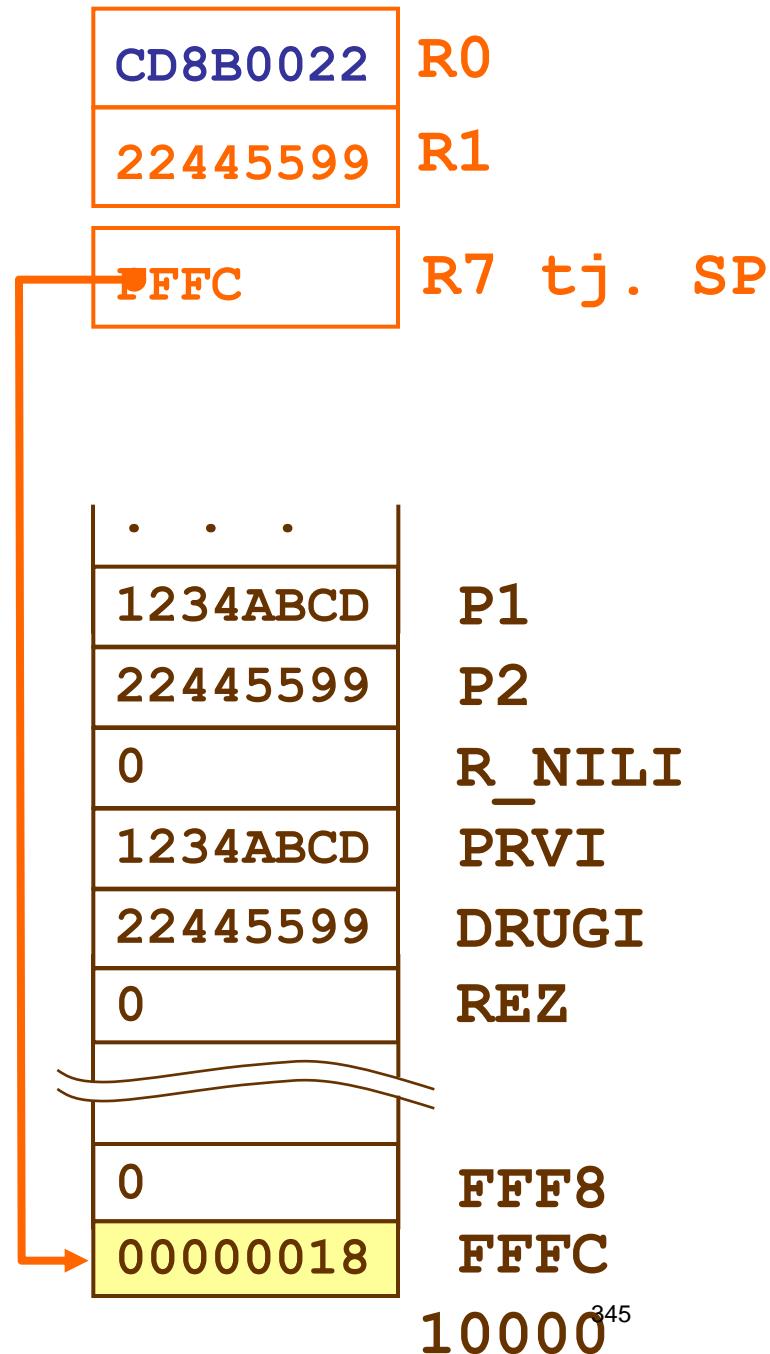


<<< Izvođenje programa:

```
NILI    LOAD   R0 , (P1)
        LOAD   R1 , (P2)

        OR      R0 , R1 , R0
        XOR   R0 , -1 , R0 ←

        STORE R0 , (R_NILI)
        RET
```

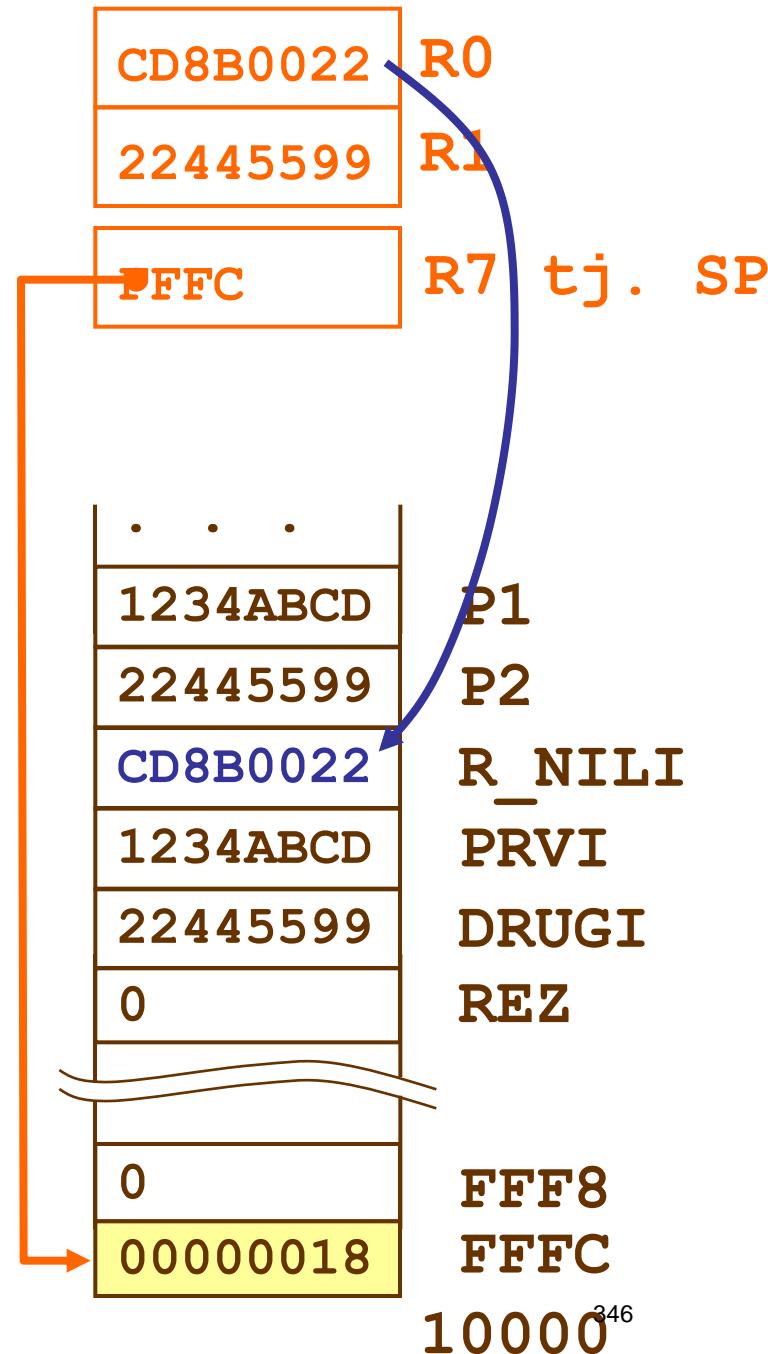


<<< Izvođenje programa:

```
NILI    LOAD   R0 , (P1)
        LOAD   R1 , (P2)

        OR      R0 , R1 , R0
        XOR   R0 , -1 , R0

        STORE  R0 , (R_NILI) ←
RET
```

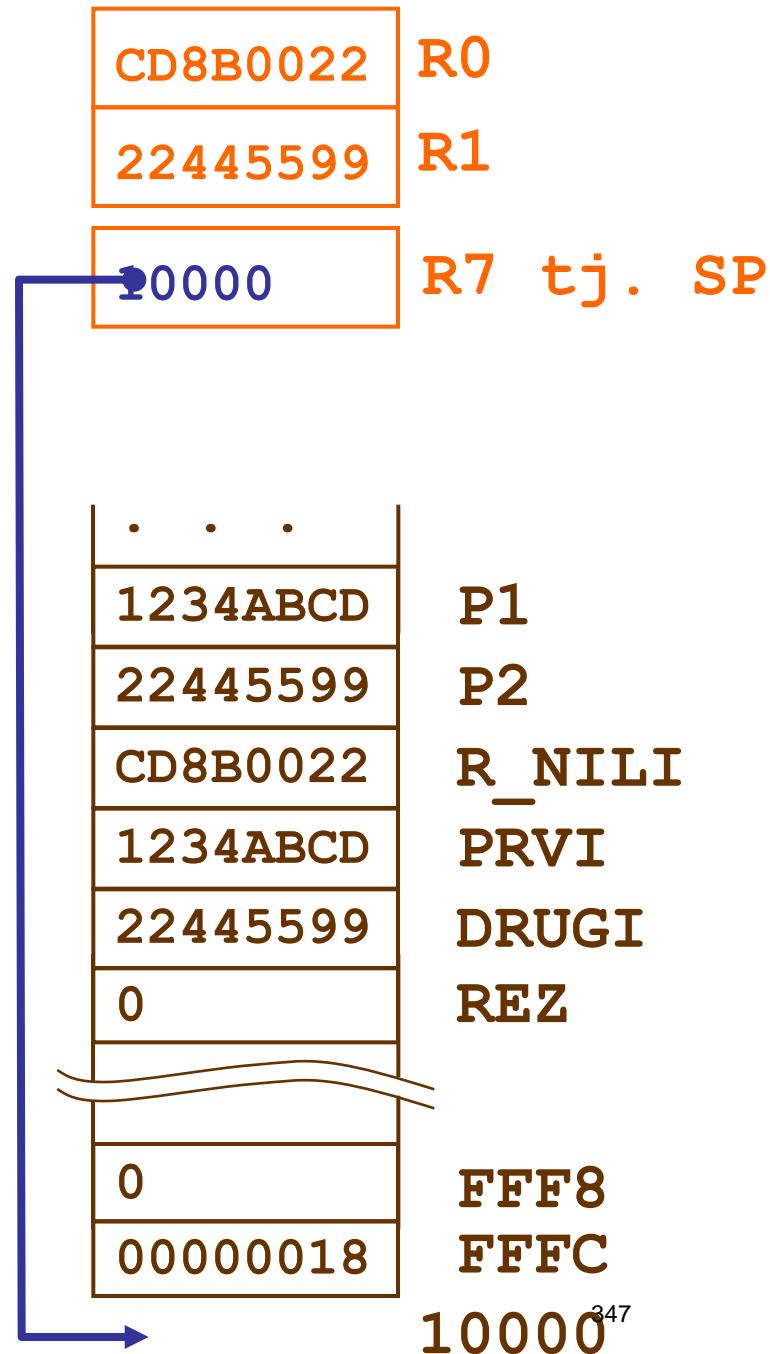


<<< Izvođenje programa:

```
NILI    LOAD   R0 , (P1)
        LOAD   R1 , (P2)

        OR      R0 , R1 , R0
        XOR   R0 , -1 , R0

        STORE  R0 , (R_NILI)
RET
```



<<< Izvođenje programa:

GLAVNI MOVE 10000, SP

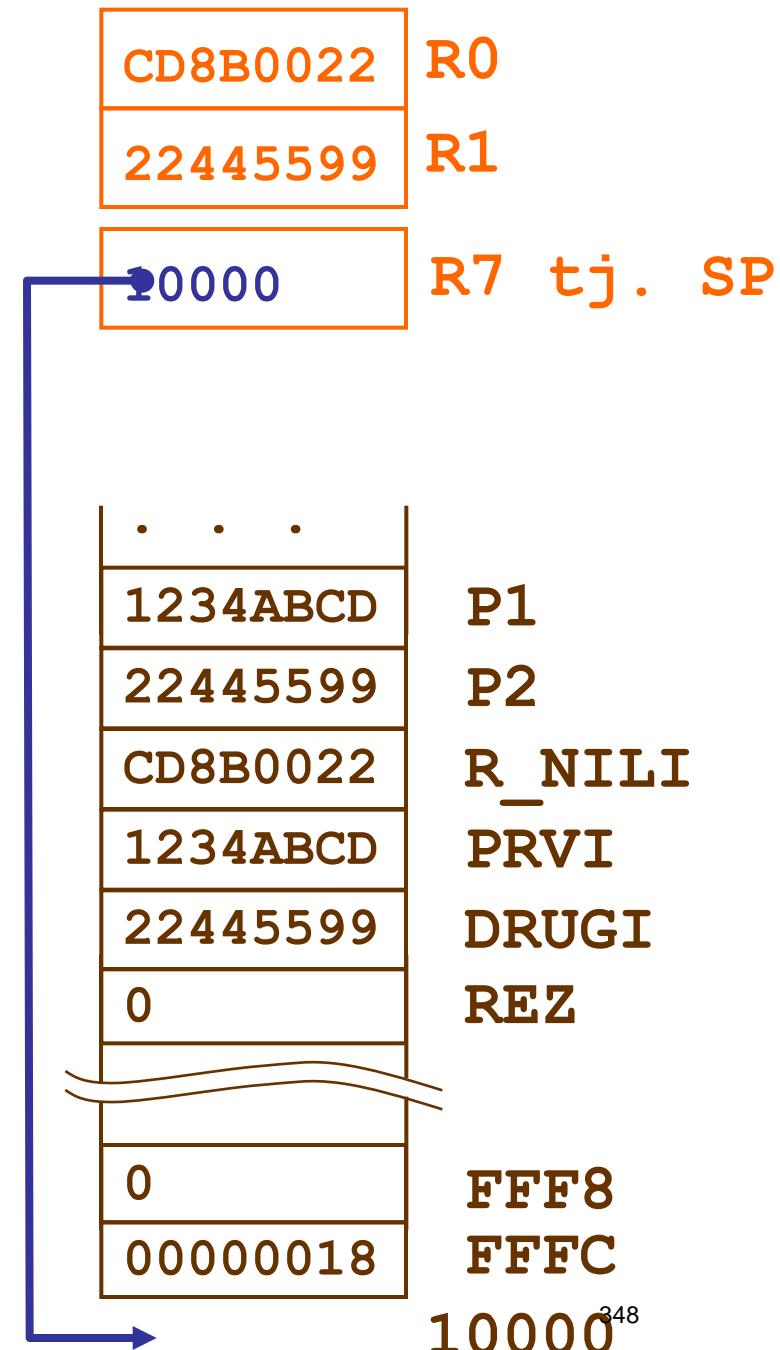
LOAD R0, (PRVI)
STORE R0, (P1)

LOAD R0, (DRUGI)
STORE R0, (P2)

CALL NILI

LOAD R0, (R_NILI)
STORE R0, (REZ)

HALT



<<< Izvođenje programa:

GLAVNI MOVE 10000, SP

LOAD R0, (PRVI)

STORE R0, (P1)

LOAD R0, (DRUGI)

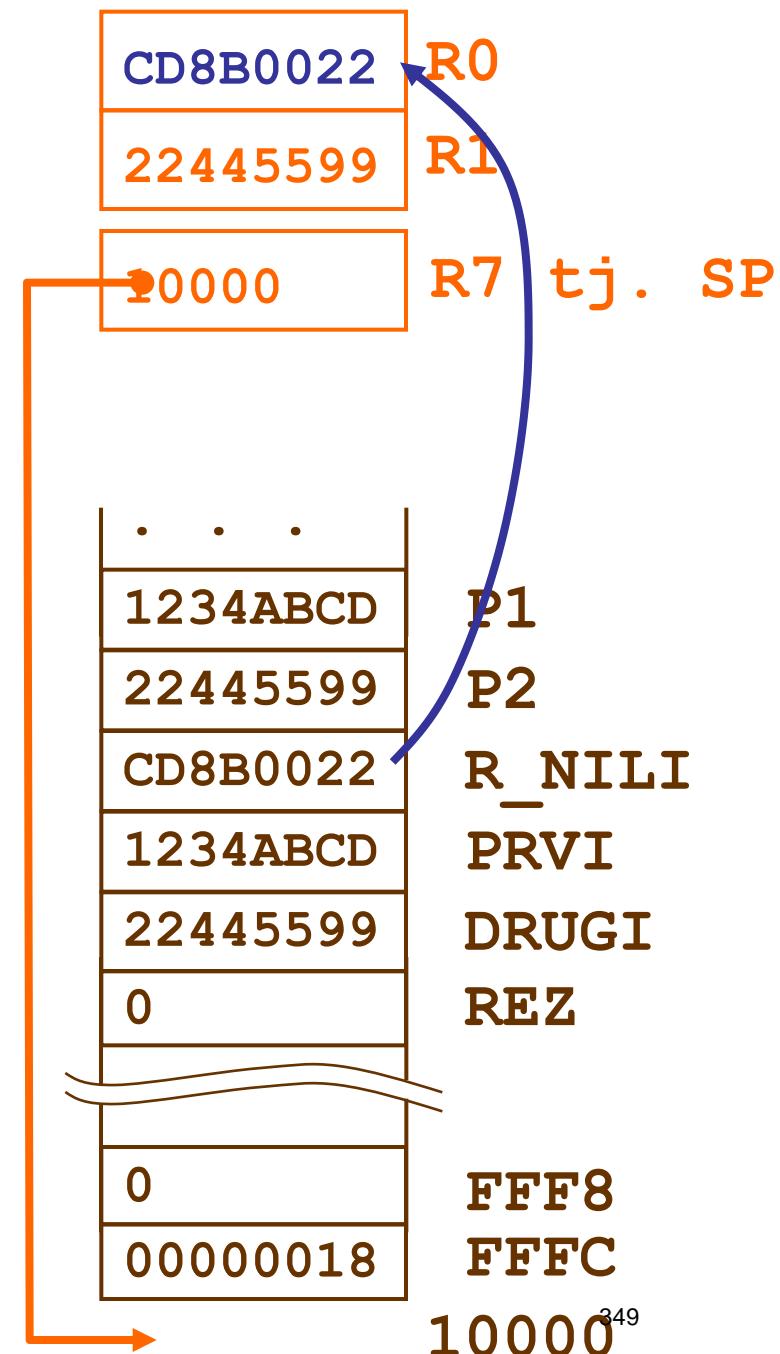
STORE R0, (P2)

CALL NILI

LOAD R0, (R_NILI) ←

STORE R0, (REZ)

HALT



<<< Izvođenje programa:

GLAVNI MOVE 10000, SP

LOAD R0, (PRVI)

STORE R0, (P1)

LOAD R0, (DRUGI)

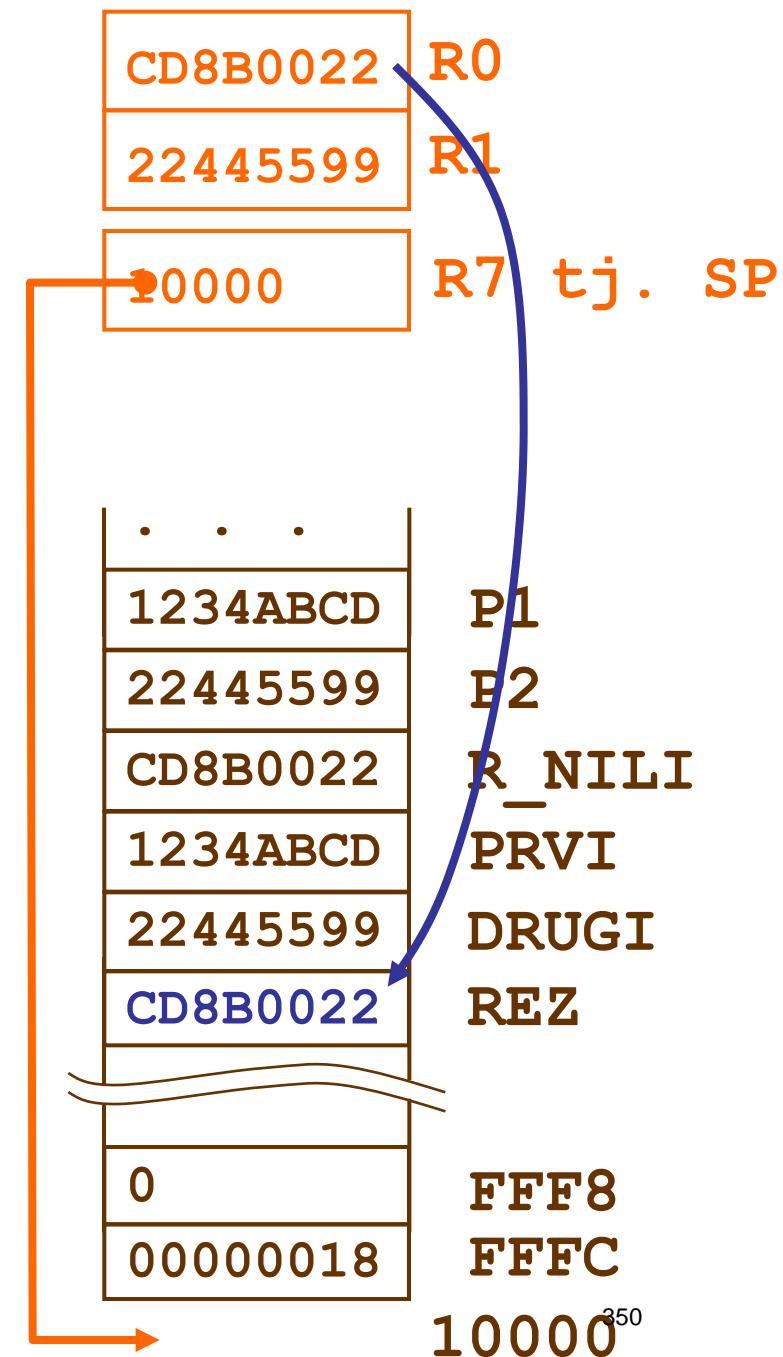
STORE R0, (P2)

CALL NILI

LOAD R0, (R_NILI)

STORE R0, (REZ)

HALT



<<< Izvođenje programa:

GLAVNI MOVE 10000, SP

LOAD R0, (PRVI)

STORE R0, (P1)

LOAD R0, (DRUGI)

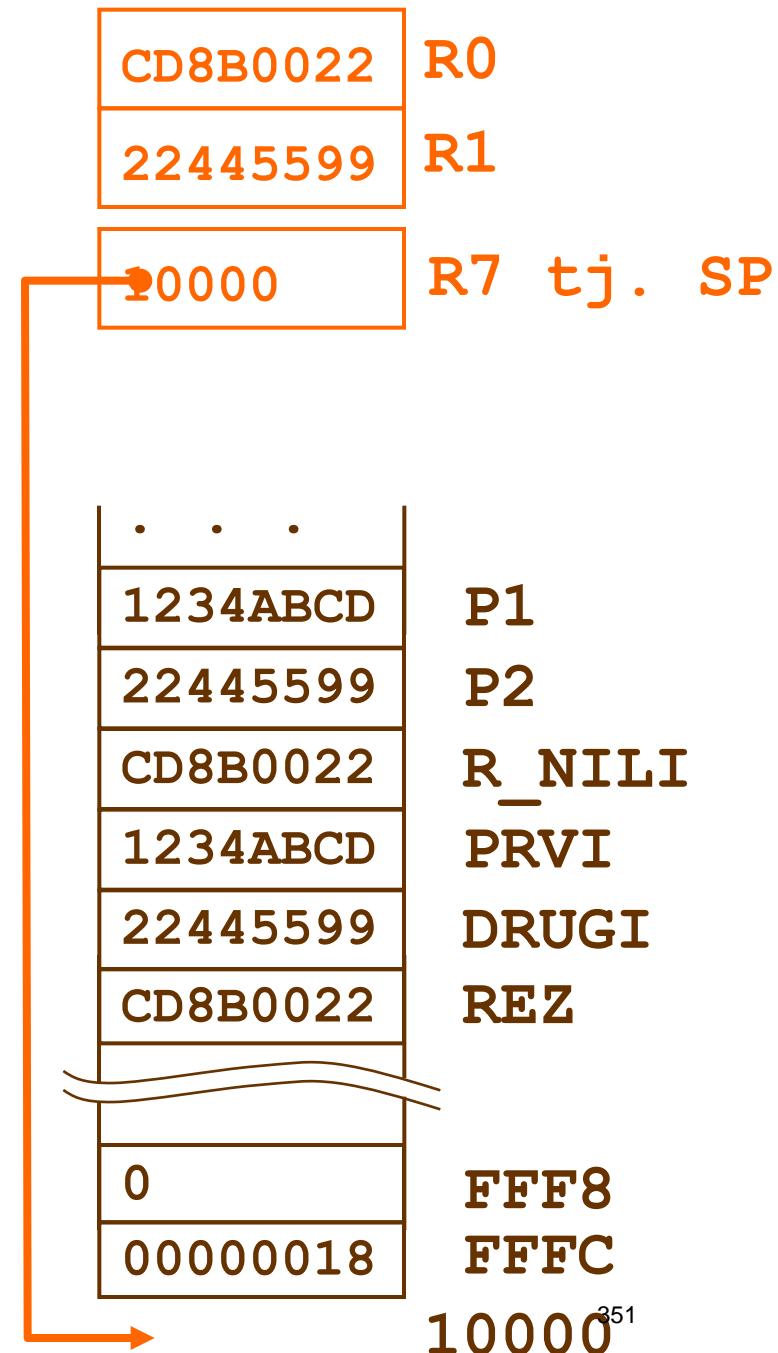
STORE R0, (P2)

CALL NILI

LOAD R0, (R_NILI)

STORE R0, (REZ)

HALT



<<< (kompletan listing s komentarima)

; glavni program

GLAVNI MOVE 10000, SP ;važno: inicijaliziraj SP !!!

; stavi vrijednost u prvi parametar

LOAD R0, (PRVI)

STORE R0, (P1)

; stavi vrijednost u drugi parametar

LOAD R0, (DRUGI)

STORE R0, (P2)

CALL NILI ;poziv potprograma

; uzmi rezultat i spremi ga

LOAD R0, (R_NILI)

STORE R0, (REZ) ;spremanje rezultata

HALT

>>>

<<<

; potprogram NILI

NILI LOAD R0, (P1) ; dohvati prvi parametar
 LOAD R1, (P2) ; dohvati drugi parametar

 OR R0, R1, R0 ; Izračunavanje
 XOR R0, -1, R0 ; rezultata.

 STORE R0, (R_NILI) ; upis rezultata u memoriju
 RET

; fiksne lokacije za parametre
; i povratnu vrijednost

P1 DW 0
P2 DW 0
R_NILI DW 0

; podatci i mjesto za rezultat

PRVI DW 1234ABCD
DRUGI DW 22445599
REZ DW 0

>>>

Komentar

Prethodni potprogram ima značajan nedostatak, a to je promjena vrijednosti u registrima R0 i R1. Ukoliko je glavni program imao u njima podatke koji će mu još trebati, oni će nakon poziva potprograma biti izgubljeni i glavni program neće raditi ispravno.

Jedna (loša) mogućnost je da se za svaki potprogram zna koje registre mijenja. Tada se pri izradi programa mora voditi računa da na mjestima pozivanja potprograma u tim registrima ne budu nikakvi korisni podatci.

Bolje rješenje je da potprogram "sačuva" sadržaje registara koje će mijenjati.

Ovo je bio naš potprogram koji mijenja registre (R0 i R1):

NILI

```
LOAD  R0,  (P1)      ; dohvati prvi parametar
LOAD  R1,  (P2)      ; dohvati drugi parametar
OR    R0,  R1,  R0    ; Izračunavanje
XOR   R0,  -1,  R0    ; rezultata.
STORE R0,  (R_NILI) ; upis rezultata u memoriju
```

RET

Ovo je potprogram koji sprema registre koje mijenja

Registri se spremaju na fiksne memorijske lokacije

```
NILI    STORE R0 , (R0_SPREM)      ; spremi R0 i R1 na
          STORE R1 , (R1_SPREM)      ; fiksne lokacije

          LOAD   R0 , (P1)           ; dohvati prvi parametar
          LOAD   R1 , (P2)           ; dohvati drugi parametar

          OR     R0 , R1 , R0         ; Izračunavanje
          XOR     R0 , -1 , R0        ; rezultata.

          STORE R0 , (R_NILI) ; upis rezultata u memoriju
          LOAD   R0 , (R0_SPREM)      ; obnovi vrijednosti
          LOAD   R1 , (R1_SPREM)      ; registara R0 i R1
          RET

R0_SPREM DW 0      ; Dodatne lokacije za
R1_SPREM DW 0      ; spremanje registara
```

Ovakvo spemanje onemogućuje rekurzivne pozive i zahtijeva definiranje posebnih memorijskih lokacija - nije idealno

Bolje rješenje je spremanje na stog:

```
NILI    PUSH   R0      ; spremi R0 i R1
        PUSH   R1      ; na stog

        LOAD   R0 , (P1)   ; dohvati prvog parametra
        LOAD   R1 , (P2)   ; dohvati drugog parametra

        OR     R0 , R1 , R0   ; Izračunavanje
        XOR     R0 , -1 , R0   ; rezultata.

        STORE R0 , (R_NILI) ; upis rezultata u memoriju
        POP    R1      ; obnovi vrijednosti registara
        POP    R0      ; R0 i R1 (OPREZ: REDOSLIJED!!!)
        RET
```

Ovakvo spremanje je bolje, omogućuje rekurzivne pozive i uobičajeno se koristi

Stog tijekom izvođenja:

GLAVNI . . .

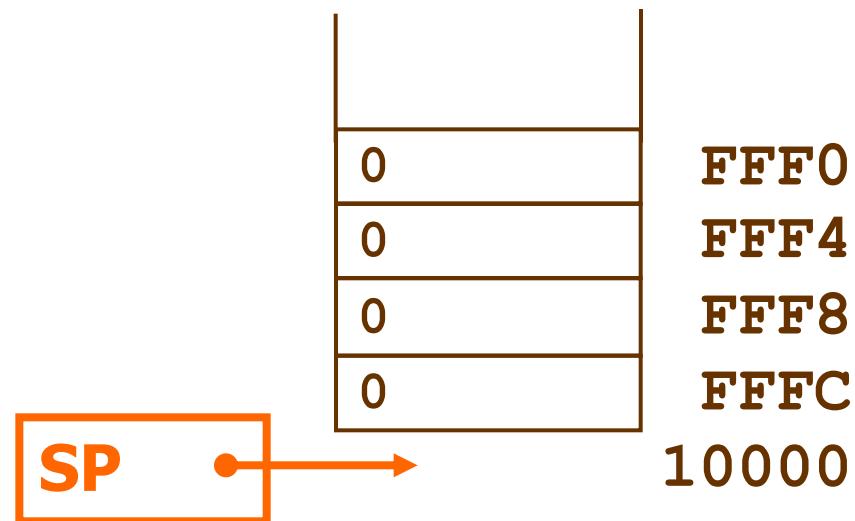
```
CALL    NILI  
LOAD    R0 ,  (R_NILI)
```

. . .

NILI PUSH R0
PUSH R1

. . . ; naredbe
. . . ; potprograma

```
POP    R1  
POP    R0  
RET
```



Stog tijekom izvođenja:

GLAVNI . . .

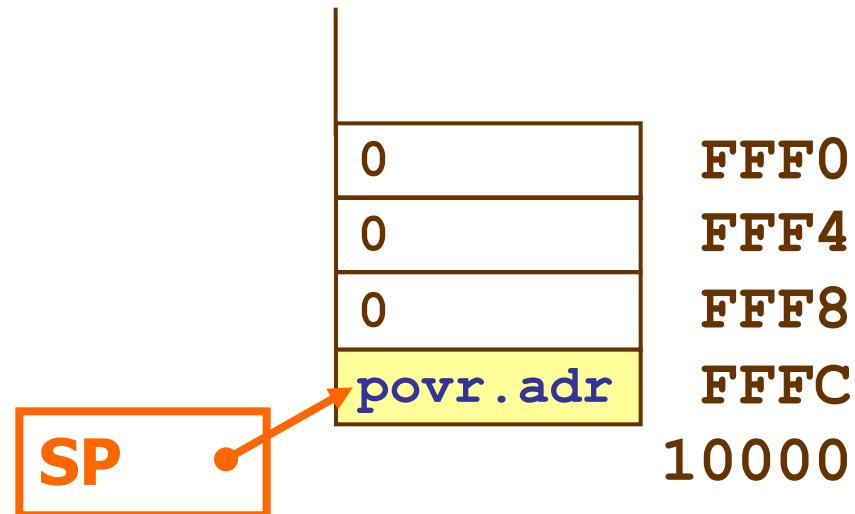
CALL NILI ←
LOAD R0 , (R_NILI)

. . .

NILI PUSH R0
PUSH R1

. . . ; naredbe
. . . ; potprograma

POP R1
POP R0
RET



Stog tijekom izvođenja:

GLAVNI . . .

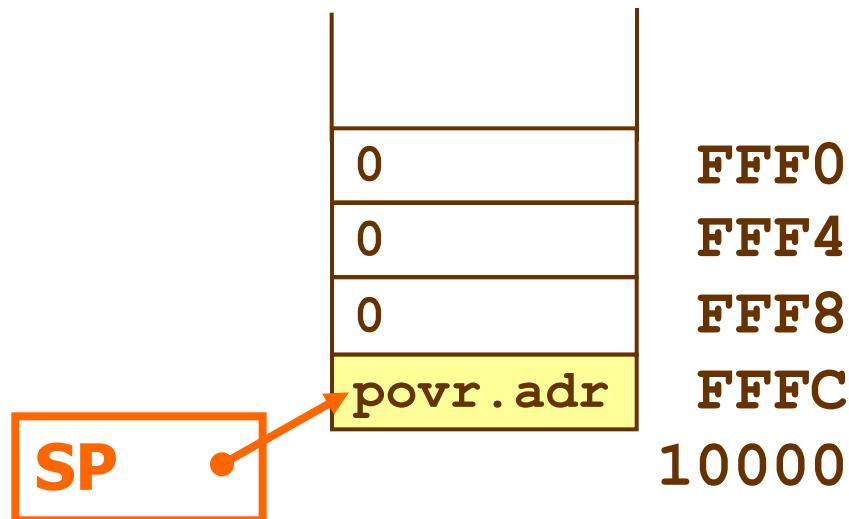
```
CALL    NILI
LOAD    R0 ,  (R_NILI)
```

. . .

NILI PUSH R0
PUSH R1

. . . ; naredbe
. . . ; potprograma

```
POP    R1
POP    R0
RET
```



Stog tijekom izvođenja:

GLAVNI . . .

```
CALL    NILI  
LOAD    R0 ,  (R_NILI)
```

. . .

NILI PUSH R0 ←

```
PUSH   R1
```

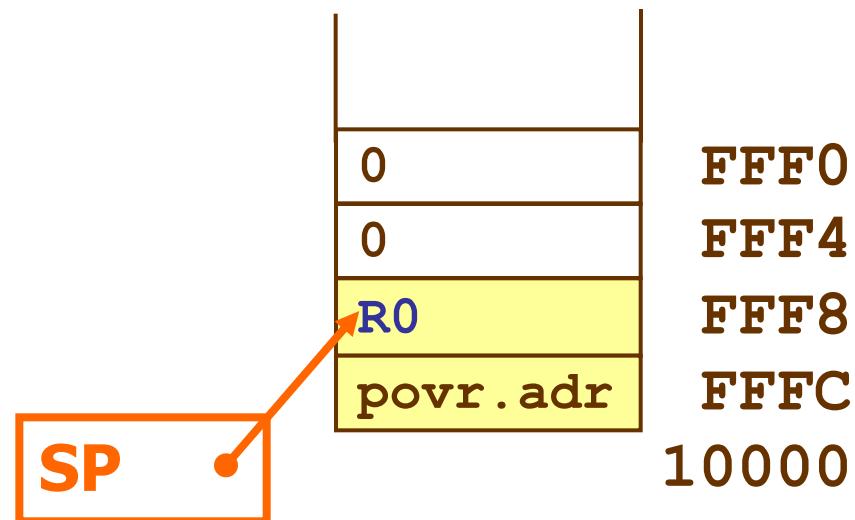
. . . ; naredbe

. . . ; potprograma

```
POP    R1
```

```
POP    R0
```

```
RET
```



Stog tijekom izvođenja:

GLAVNI . . .

```
CALL    NILI  
LOAD    R0 ,  (R_NILI)
```

. . .

NILI PUSH R0

PUSH R1 

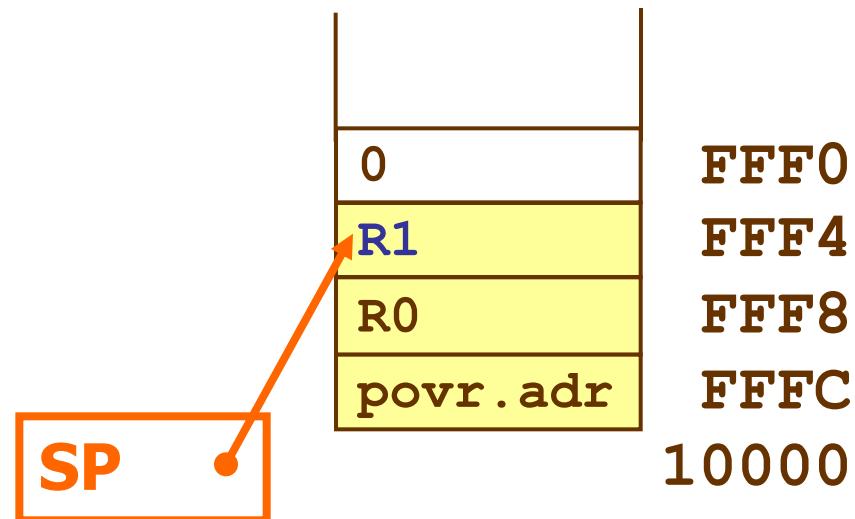
. . . ; naredbe

. . . ; potprograma

POP R1

POP R0

RET



Stog tijekom izvođenja:

GLAVNI . . .

```
CALL    NILI  
LOAD    R0 ,  (R_NILI)
```

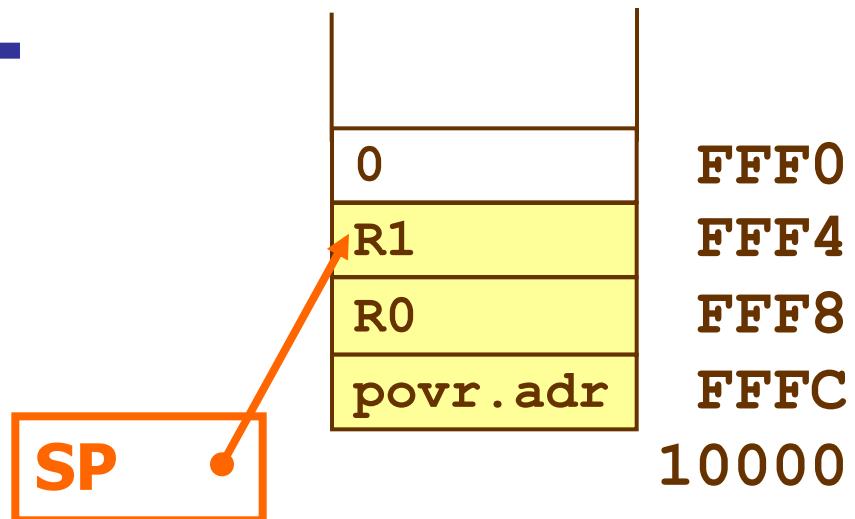
. . .

NILI PUSH R0
PUSH R1

. . . ; naredbe
. . . ; potprograma

```
POP    R1  
POP    R0  
RET
```

Registri R0 i R1 se mijenjaju.
Stanje stoga se ne mijenja
(smije se mijenjati, ako nakon
naredaba potprograma stanje
bude jednako kao prije njih)



Stog tijekom izvođenja:

GLAVNI . . .

```
CALL    NILI  
LOAD    R0 ,  (R_NILI)
```

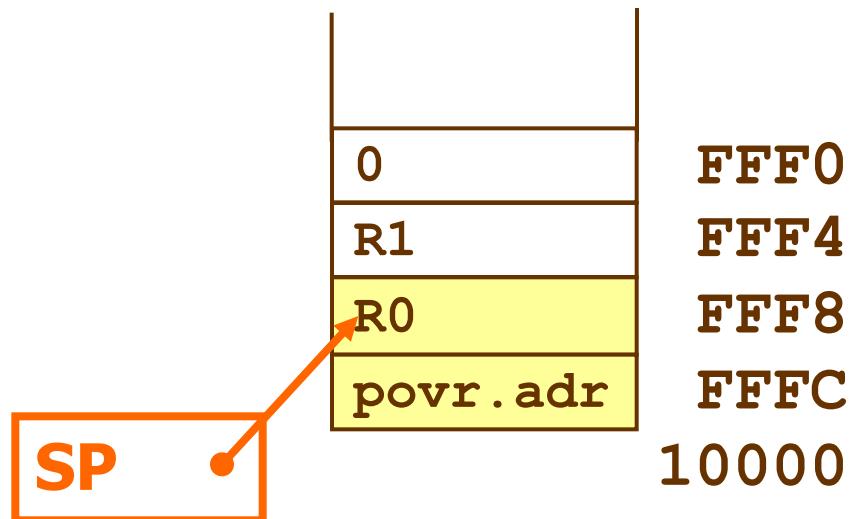
. . .

Registar R1 se obnavlja.

NILI PUSH R0
PUSH R1

. . . ; naredbe
. . . ; potprograma

```
POP    R1 ←  
POP    R0  
RET
```



Stog tijekom izvođenja:

GLAVNI . . .

```
CALL    NILI  
LOAD    R0 ,  (R_NILI)
```

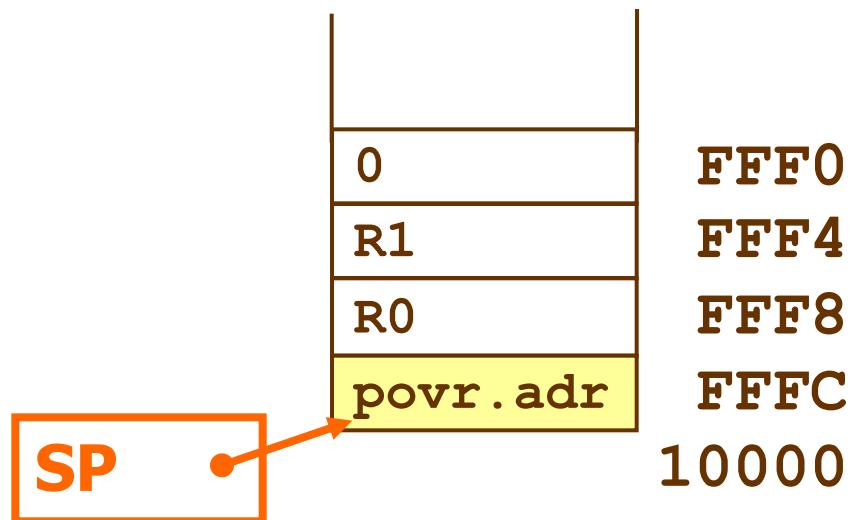
. . .

NILI PUSH R0
PUSH R1

. . . ; naredbe
. . . ; potprograma

```
POP    R1  
POP    R0 ←  
RET
```

Registar R0 se obnavlja.



Stog tijekom izvođenja:

GLAVNI . . .

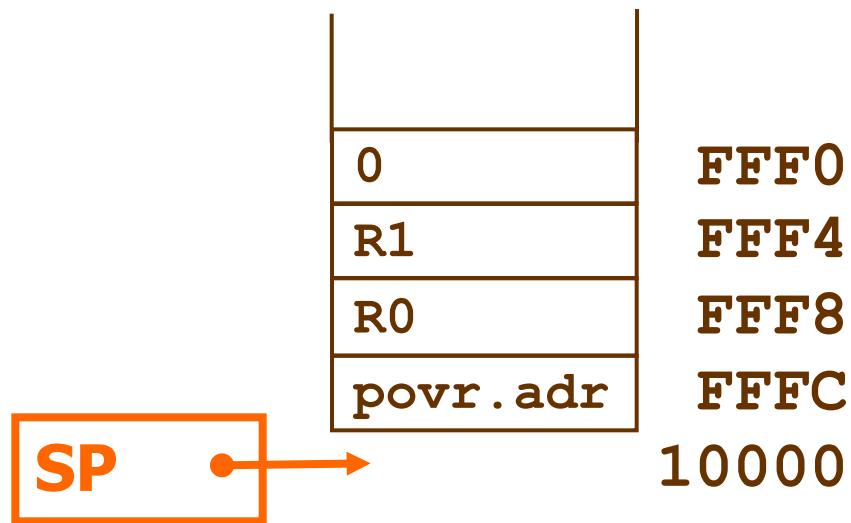
```
CALL    NILI  
LOAD    R0 ,  (R_NILI)
```

. . .

NILI PUSH R0
PUSH R1

. . . ; naredbe
. . . ; potprograma

```
POP    R1  
POP    R0  
RET
```



Stog tijekom izvođenja:

GLAVNI . . .

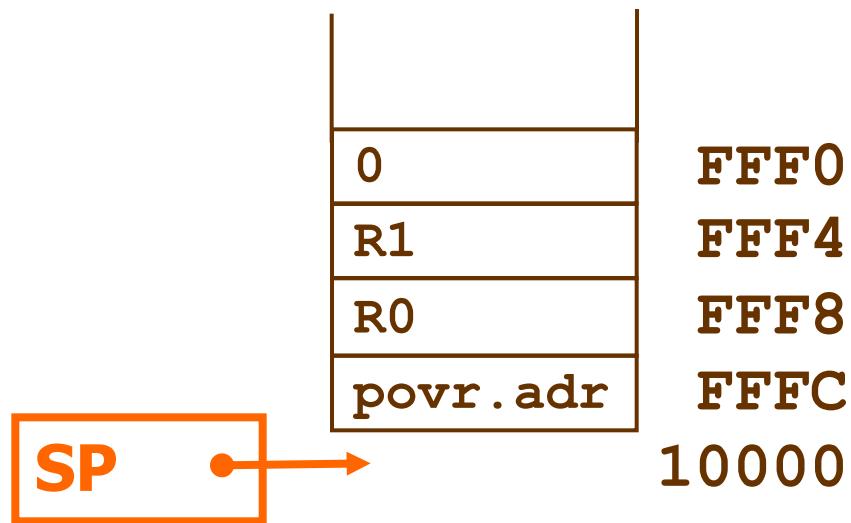
```
CALL    NILI  
LOAD    R0 ,  (R_NILI)
```

. . .

NILI PUSH R0
PUSH R1

. . . ; naredbe
. . . ; potprograma

```
POP    R1  
POP    R0  
RET
```



Prijenos stogom

Potprogrami - Prijenos stogom

- Pozivatelj prije poziva mora staviti argumente na stog
 - **Oprez: naredba CALL stavlja povratnu adresu na stog**
- Potprogram polazi od pretpostavke da su na stogu ispod povratne adrese argumenti i koristi ih pri izračunavanju rezultata **(kako?)**
- Potprogram mora staviti rezultat na stog **(kako?)**
 - **Oprez: naredba RET uzima s vrha stoga povratnu adresu**
- Pozivatelj, nakon povratka iz potprograma, pretpostavlja da se na vrhu stoga nalaze rezultati koje uzima sa stoga i koristi ih

Potprogrami - Prijenos stogom

- Problem je povratna adresa koja se automatski stavlja i uzima na stog naredbama CALL i RET.
- Ova povratna adresa:
 - smeta dohvatu argumenata sa stoga
 - smeta stavljanju rezultata na stog
- Ispravno baratanje sa stogom moguće je na više načina, ali treba poznavati kako radi stog i naredbe CALL i RET

Potprogrami - Prijenos stogom

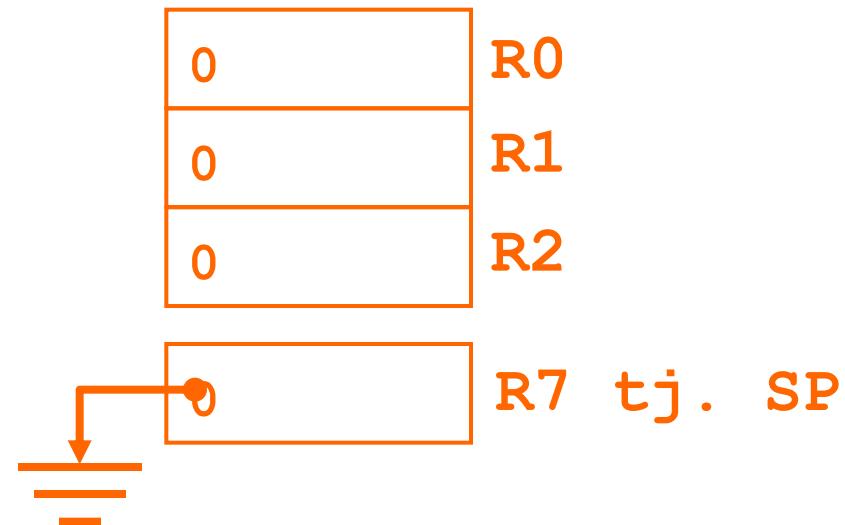
Napisati potprogram koji izračunava logičku operaciju NILI. Parametri i rezultat se prenose stogom. Glavni program iz memorije učitava dva podatka za koje računa NILI (pomoću potprograma) i rezultat spremi natrag u memoriju.

Rješenje:

- U ovom rješenju potprogram mijenja registre R0, R1 i R2, ali **ne čuva njihove vrijednosti (LOŠE!!!)**
- Kasnije ćemo pokazati bolje rješenje (ovo je samo primjer kako se može zaobići povratna adresa pri slanju parametara i povratu rezultata)

<<< Izvođenje programa:

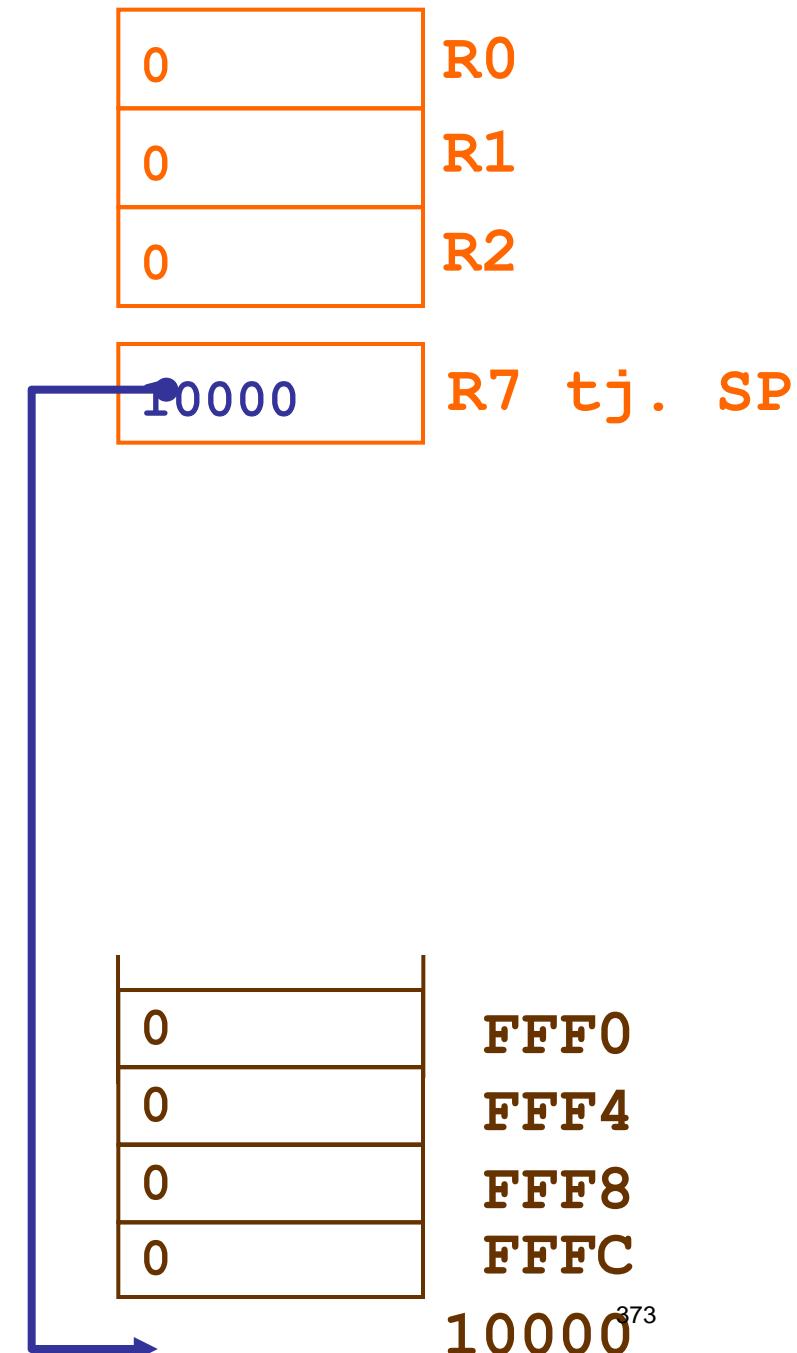
```
GLAVNI MOVE 10000, SP  
          LOAD R0, (PRVI)  
          PUSH R0  
  
          LOAD R0, (DRUGI)  
          PUSH R0  
  
          CALL NILI  
  
          POP R0  
          STORE R0, (REZ)  
  
          HALT
```



0	FFF0
0	FFF4
0	FFF8
0	FFFC

<<< Izvođenje programa:

```
GLAVNI MOVE 10000, SP ←  
LOAD R0, (PRVI)  
PUSH R0  
  
LOAD R0, (DRUGI)  
PUSH R0  
  
CALL NILI  
  
POP R0  
STORE R0, (REZ)  
  
HALT
```



<<< Izvođenje programa:

GLAVNI MOVE 10000, SP

LOAD R0, (PRVI) ←

PUSH R0

LOAD R0, (DRUGI)

PUSH R0

CALL NILI

POP R0

STORE R0, (REZ)

HALT

param.1	R0
0	R1
0	R2

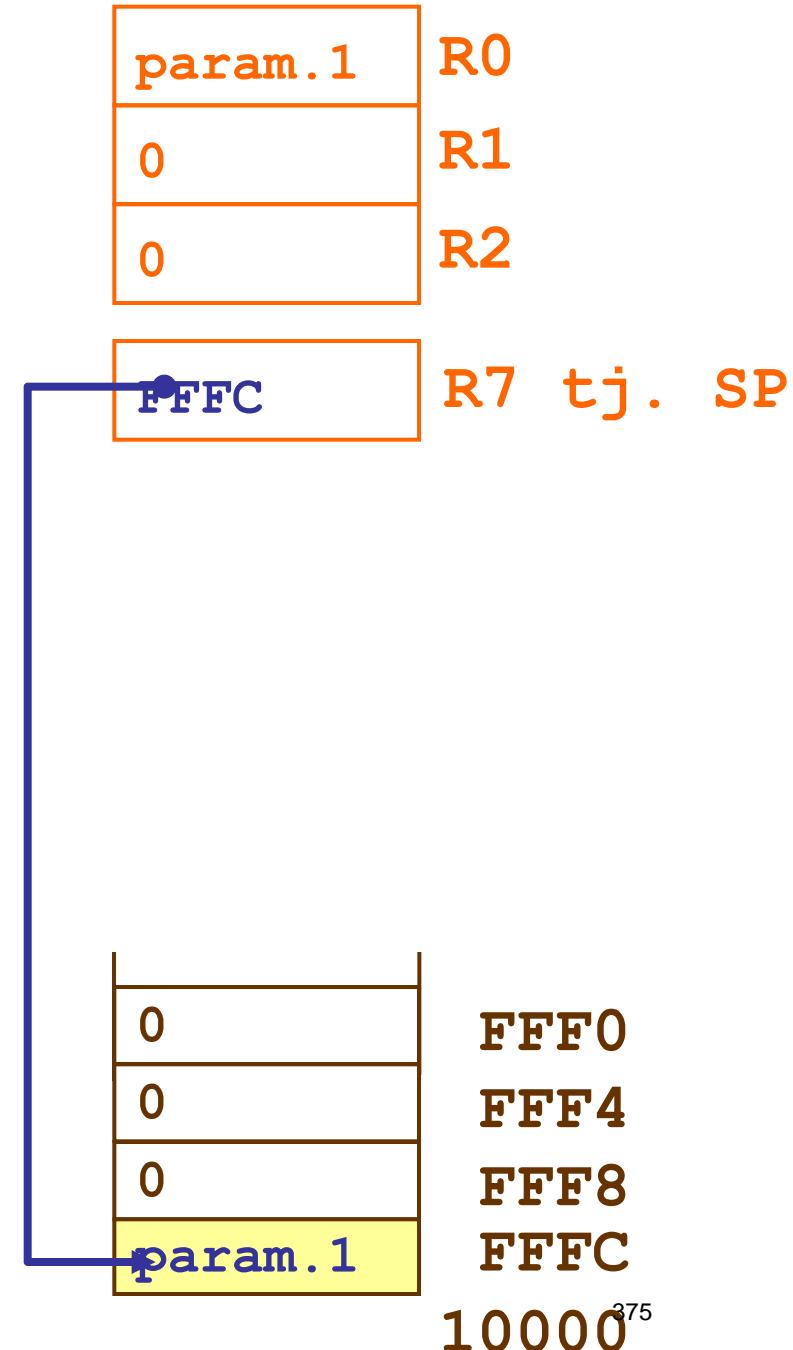


0	FFF0
0	FFF4
0	FFF8
0	FFFC

10000³⁷⁴

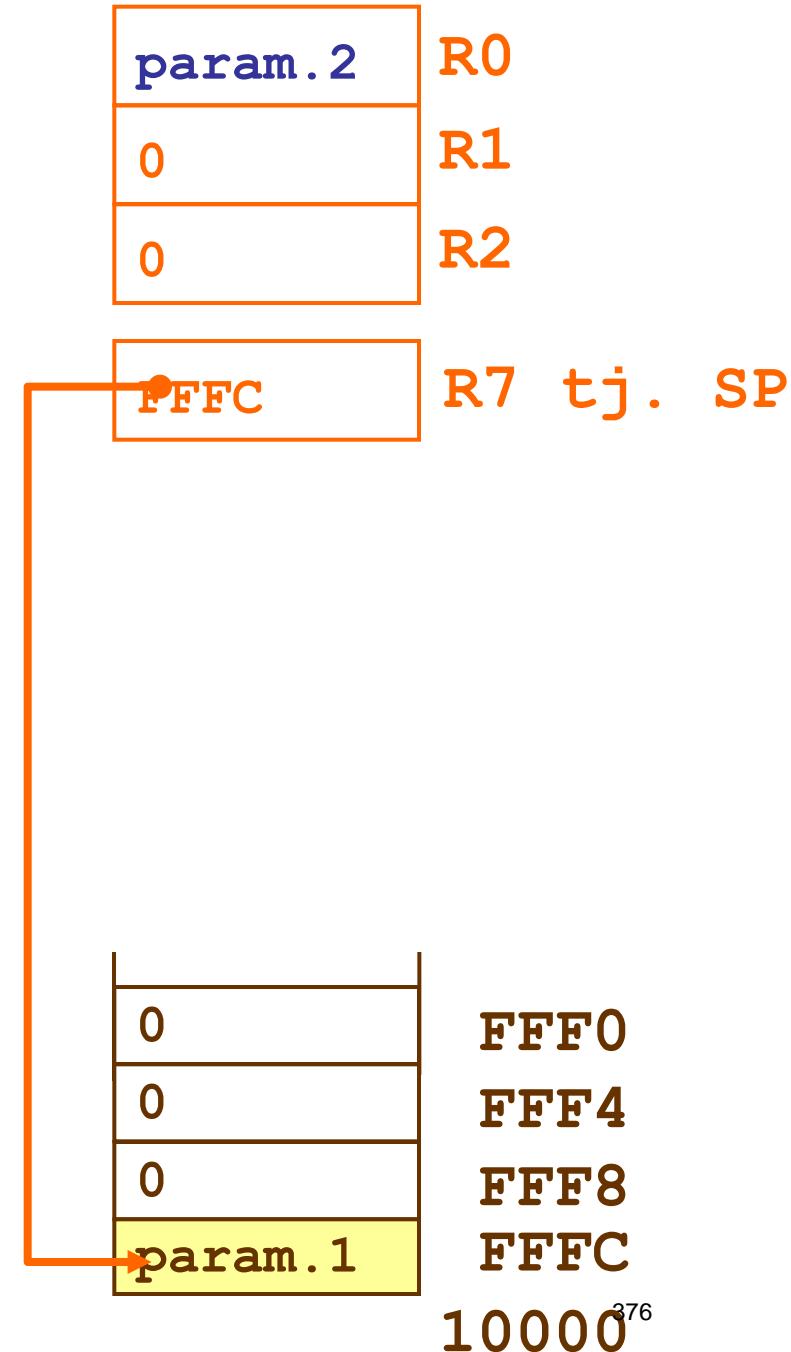
<<< Izvođenje programa:

```
GLAVNI MOVE 10000, SP  
          LOAD R0, (PRVI)  
          PUSH R0  
  
          LOAD R0, (DRUGI)  
          PUSH R0  
  
          CALL NILI  
  
          POP R0  
          STORE R0, (REZ)  
  
          HALT
```



<<< Izvođenje programa:

```
GLAVNI MOVE 10000, SP  
          LOAD R0, (PRVI)  
          PUSH R0  
  
          LOAD R0, (DRUGI) ←  
          PUSH R0  
  
          CALL NILI  
  
          POP R0  
          STORE R0, (REZ)  
  
          HALT
```



<<< Izvođenje programa:

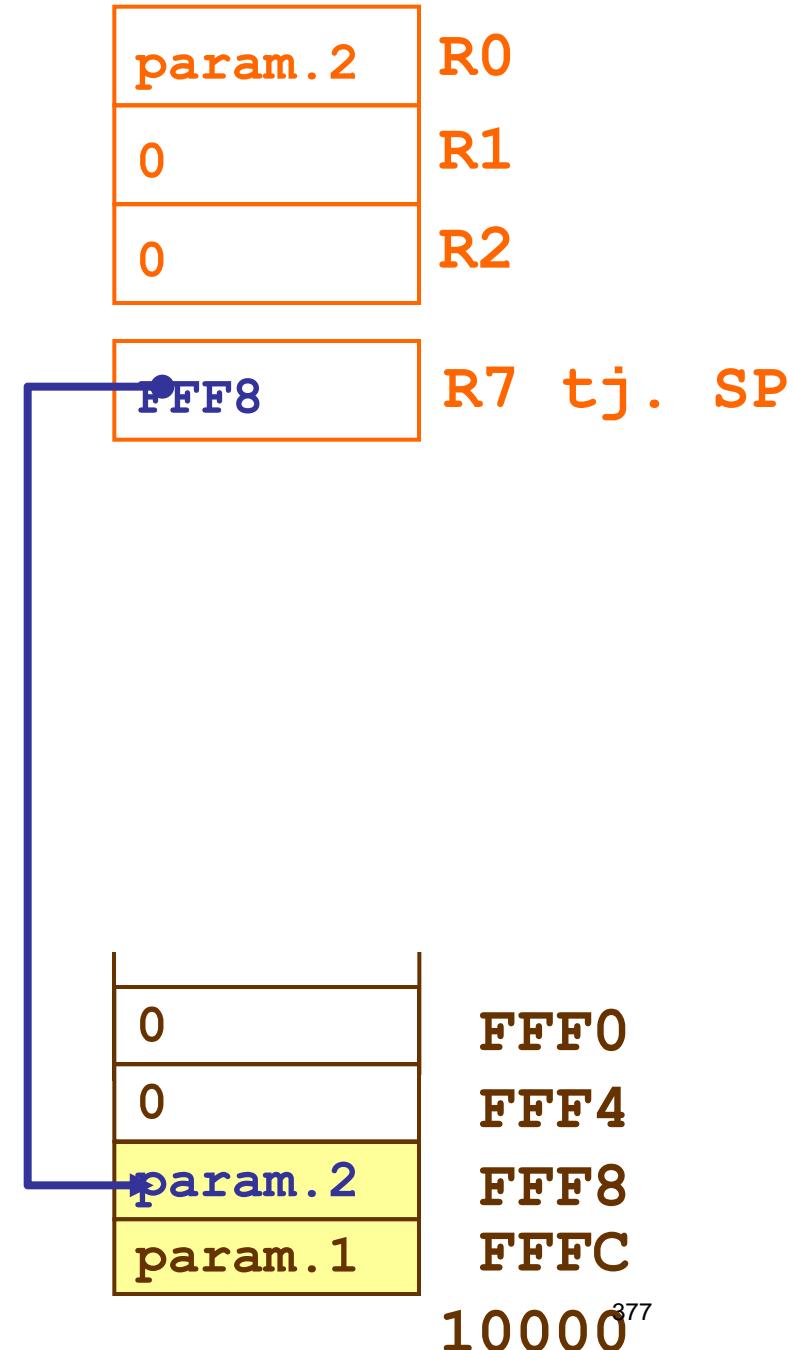
GLAVNI MOVE 10000, SP
LOAD R0, (PRVI
PUSH R0

LOAD R0, (DRUG
PUSH R0

CALL NILI

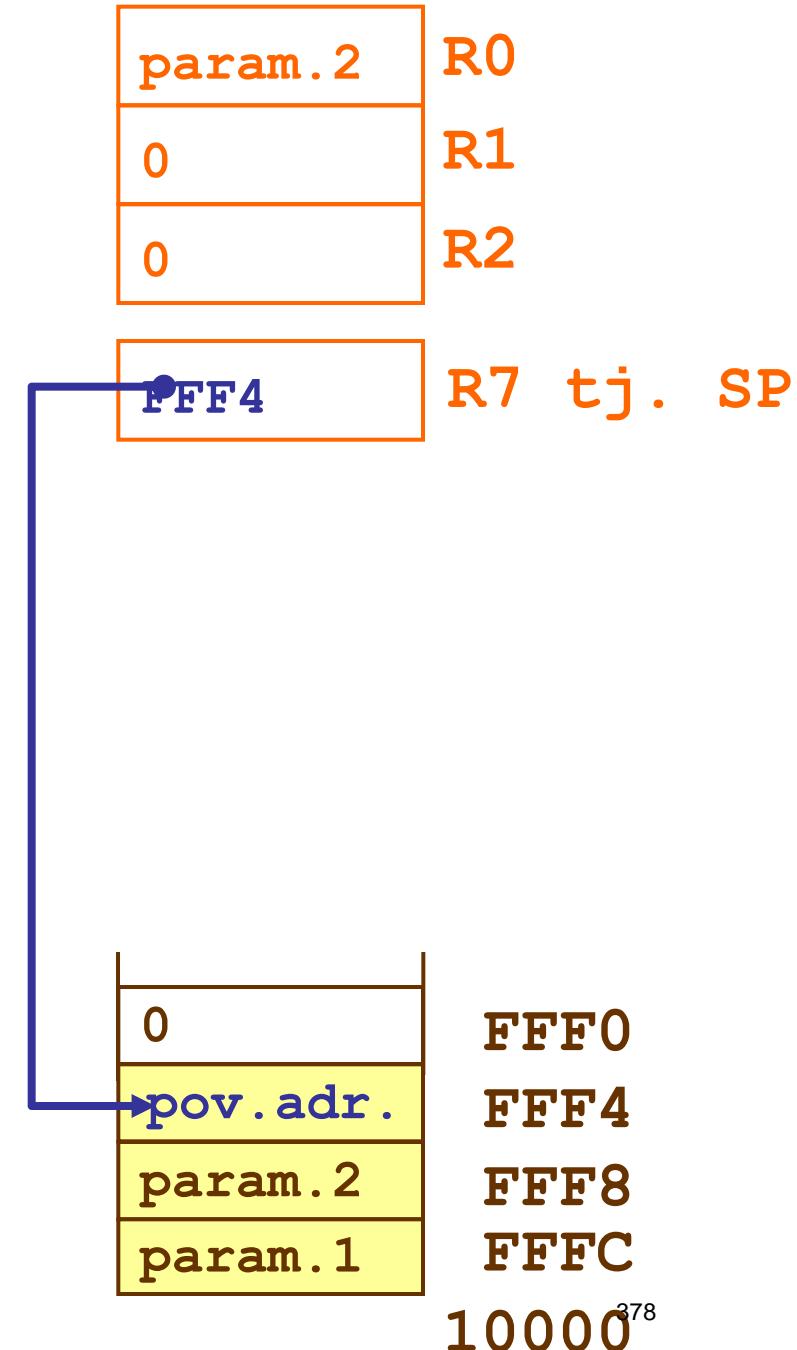
POP R0
STORE R0, (REZ)

HALT



<<< Izvođenje programa:

```
GLAVNI MOVE 10000, SP  
          LOAD R0, (PRVI)  
          PUSH R0  
  
          LOAD R0, (DRUGI)  
          PUSH R0  
  
          CALL NILI  
  
          POP R0  
          STORE R0, (REZ)  
  
          HALT
```



<<< Izvođenje programa:

NILI POP R2
 POP R0
 POP R1

OR R0 , R1 , R0
XOR R0 , -1 , R0

PUSH R0
PUSH R2
RET

param.2	R0
0	R1
0	R2

FFF4	R7 tj. SP
------	-----------

0	FFF0
pov.adr.	FFF4
param.2	FFF8
param.1	FFFC



<<< Izvođenje programa:

NILI POP R2
 POP R0
 POP R1

OR R0 , R1 , R0
XOR R0 , -1 , R0

PUSH R0
PUSH R2
RET



param.2	R0
0	R1
pov.adr.	R2



0	FFF0
pov.adr.	FFF4
param.2	FFF8
param.1	FFFC

<<< Izvođenje programa:

NILI POP R2
 POP R0
 POP R1



OR R0 , R1 , R0
XOR R0 , -1 , R0

PUSH R0
PUSH R2
RET

param . 2	R0
0	R1
pov . adr .	R2

FFFC	R7 tj. SP
------	-----------

0	FFF0
pov . adr .	FFF4
param . 2	FFF8
param . 1	FFFC

<<< Izvođenje programa:

NILI POP R2
 POP R0
 POP R1



OR R0 , R1 , R0
XOR R0 , -1 , R0

PUSH R0
PUSH R2
RET

param.2	R0
param.1	R1
pov.adr.	R2



0	FFF0
pov.adr.	FFF4
param.2	FFF8
param.1	FFFC

10000³⁸²

<<< Izvođenje programa:

NILI POP R2
POP R0
POP R1

OR R0 , R1 , R0 ←
XOR R0 , -1 , R0

PUSH R0
PUSH R2
RET

rezult	R0
param.1	R1
pov.adr.	R2

10000	R7 tj. SP
-------	-----------

0	FFF0
pov.adr.	FFF4
param.2	FFF8
param.1	FFFC

<<< Izvođenje programa:

NILI POP R2
POP R0
POP R1

OR R0 , R1 , R0
XOR R0 , -1 , R0

PUSH R0
PUSH R2
RET

rezult	R0
param.1	R1
pov.adr.	R2

FFFC

R7 tj. SP



0	FFF0
pov.adr.	FFF4
param.2	FFF8
rezult	FFFC

10000³⁸⁴

<<< Izvođenje programa:

NILI POP R2
POP R0
POP R1

OR R0 , R1 , R0
XOR R0 , -1 , R0

PUSH R0
PUSH R2
RET

rezult	R0
param.1	R1
pov.adr.	R2

FFF8	R7 tj. SP
------	-----------

0	FFF0
pov.adr.	FFF4
pov.adr.	FFF8
rezult	FFFC

<<< Izvođenje programa:

NILI POP R2
POP R0
POP R1

OR R0 , R1 , R0
XOR R0 , -1 , R0

PUSH R0
PUSH R2
RET

rezult	R0
param.1	R1
pov.adr.	R2

FFFC

R7 tj. SP

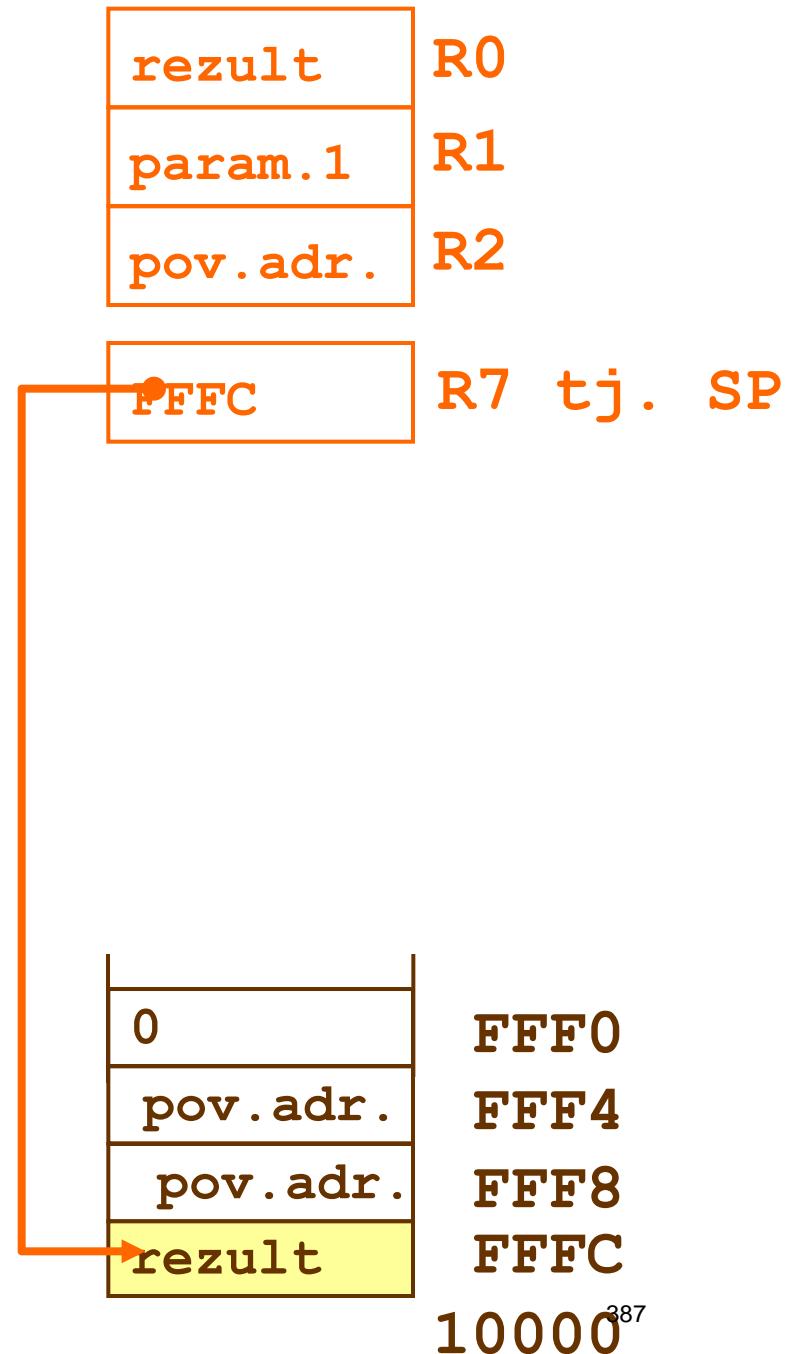


0	FFF0
pov.adr.	FFF4
pov.adr.	FFF8
rezult	FFFC

10000³⁸⁶

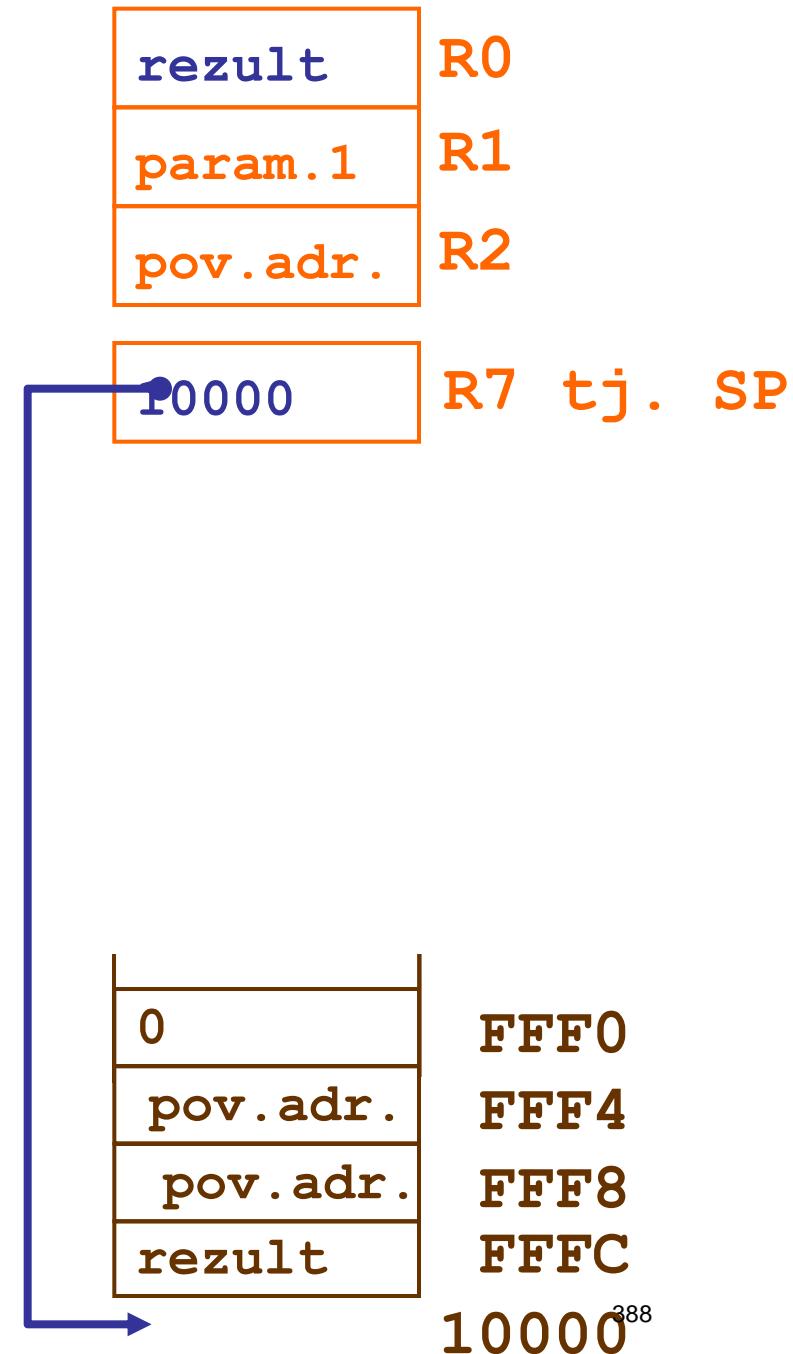
<<< Izvođenje programa:

```
GLAVNI MOVE 10000, SP  
LOAD R0, (PRVI)  
PUSH R0  
  
LOAD R0, (DRUGI)  
PUSH R0  
  
CALL NILI  
  
POP R0  
STORE R0, (REZ)  
  
HALT
```



<<< Izvođenje programa:

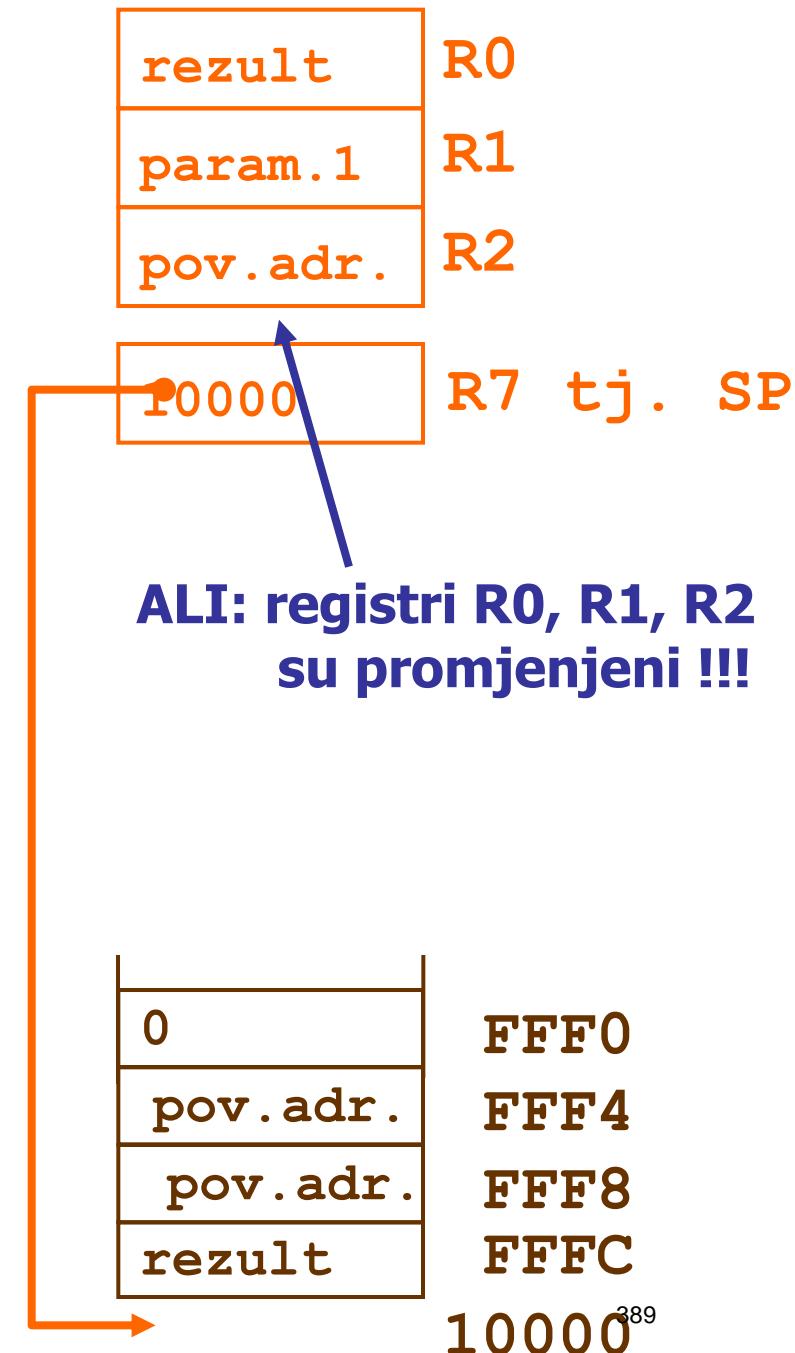
```
GLAVNI MOVE 10000, SP  
LOAD R0, (PRVI)  
PUSH R0  
  
LOAD R0, (DRUGI)  
PUSH R0  
  
CALL NILI  
  
POP R0  
STORE R0, (REZ)  
  
HALT
```



<<< Izvođenje programa:

- dohvaćeni rezultat se koristi
- stog je u početnom stanju

```
GLAVNI MOVE 10000, SP  
          LOAD R0, (PRVI)  
          PUSH R0  
  
          LOAD R0, (DRUGI)  
          PUSH R0  
  
          CALL NILI  
  
          POP R0  
          STORE R0, (REZ) ←  
  
          HALT
```



<<< (kompletan listing s komentarima)

; glavni program

GLAVNI MOVE 10000, SP ;važno: inicijaliziraj SP !!!

; stavi vrijednost prvog parametra na stog

LOAD R0, (PRVI)

PUSH R0

; stavi vrijednost drugog parametra na stog

LOAD R0, (DRUGI)

PUSH R0

CALL NILI ;poziv potprograma

; uzmi rezultat sa stoga i spremi ga

POP R0

STORE R0, (REZ) ;spremanje rezultata

HALT

>>>

<<<

```
; potprogram NILI
NILI    POP    R2      ; uzmi povratnu adresu sa stoga
        POP    R0      ; dohvati prvog parametra
        POP    R1      ; dohvati drugog parametra

        OR     R0, R1, R0   ; Izracunavanje
        XOR    R0, -1, R0   ; rezultata.

        PUSH   R0      ; stavi rezultat na stog
        PUSH   R2      ; vrati povratnu adresu na stog
        RET
```

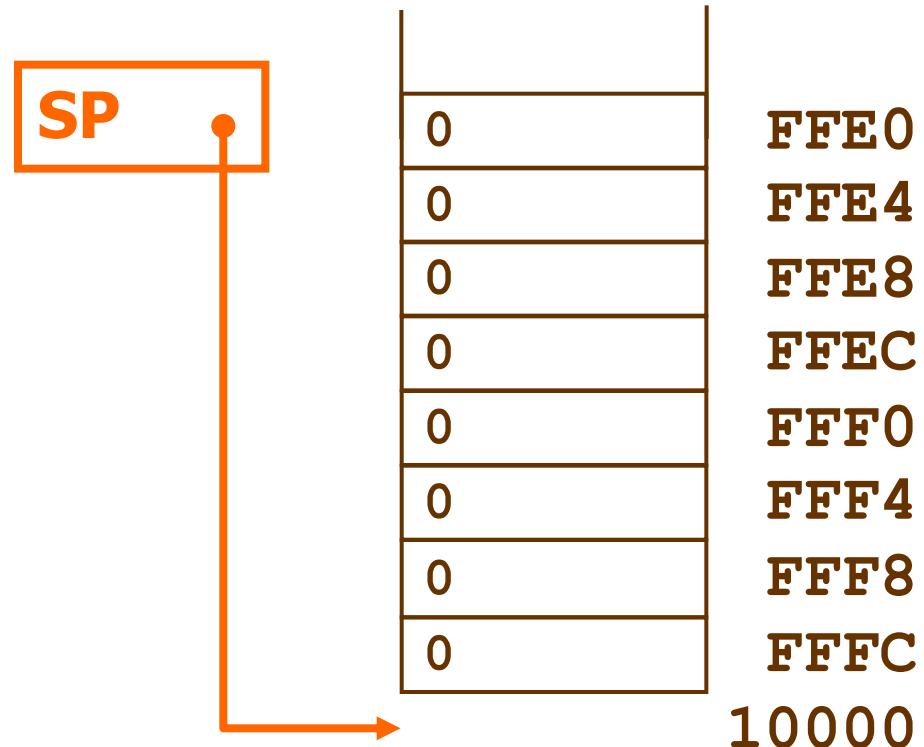
; podatci i mjesto za rezultat

```
PRVI   DW 1234ABCD
DRUGI  DW 22445599
REZ    DW 0
```

Potprogrami - Prijenos stogom

- Pokušajmo riješiti prethodni zadatak tako da se čuvaju stanja registara koje potprogram mijenja
- Već smo vidjeli da je najzgodnije **registre spremiti na stog**

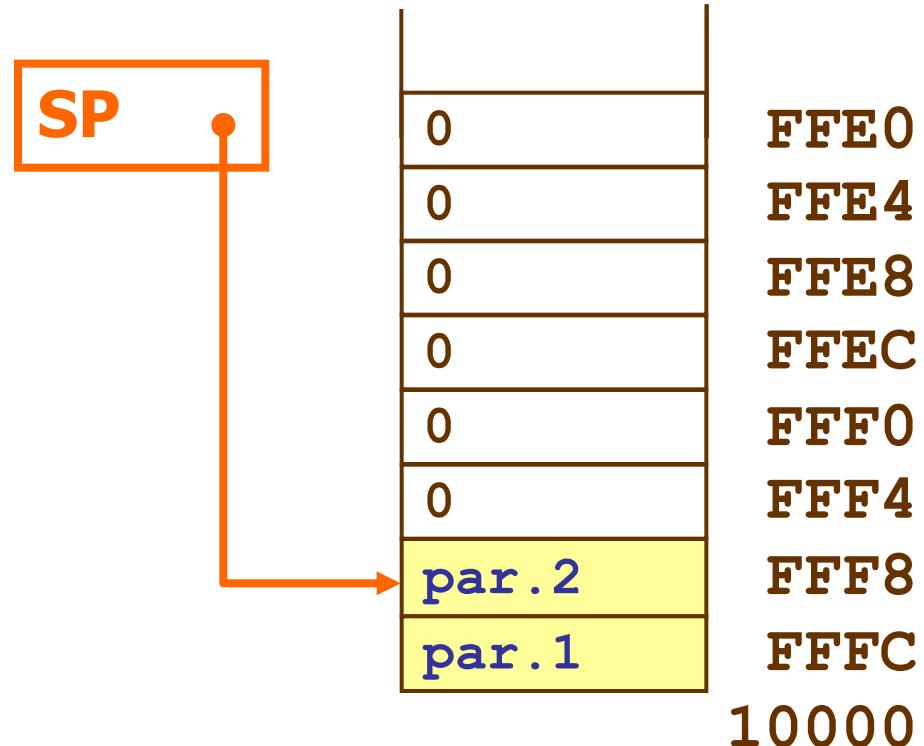
Potprogrami - Prijenos stogom



Potprogrami - Prijenos stogom

Izvodi se:

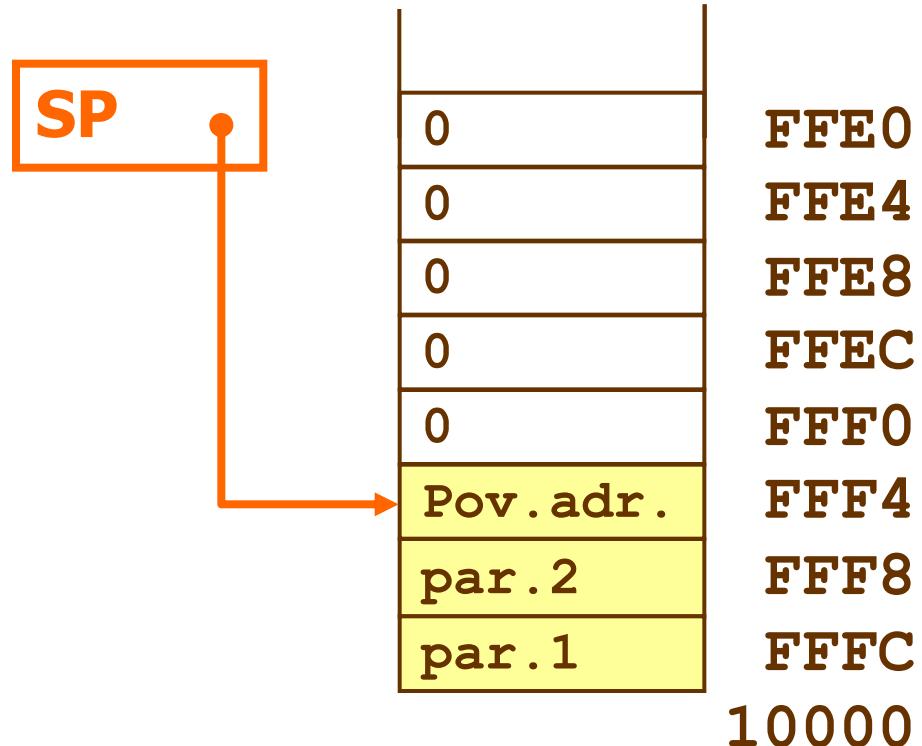
Glavni program stavlja dva parametra na stog



Potprogrami - Prijenos stogom

Izvodi se:

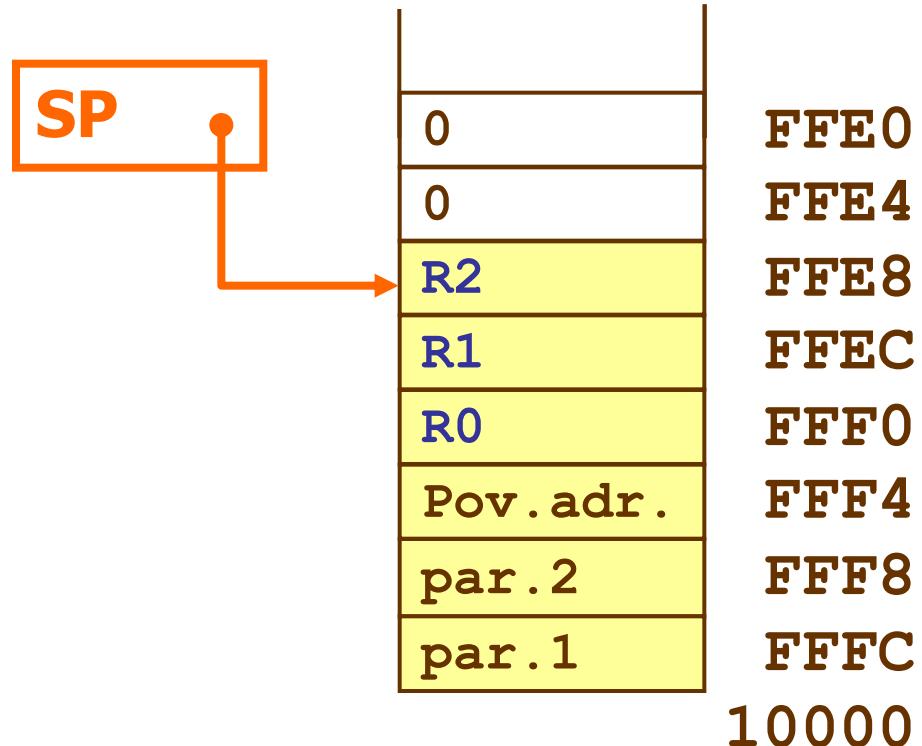
Glavni program poziva
potprogram



Potprogrami - Prijenos stogom

Izvodi se:

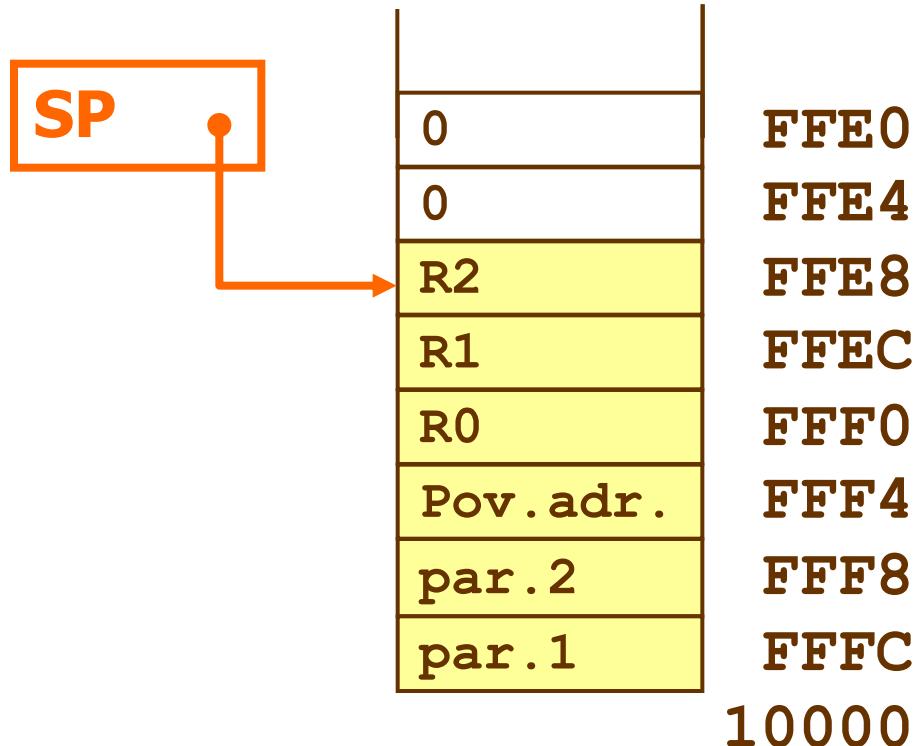
Potprogram sprema registre
(koje će mijenjati) na stog
(npr. tri registra R0, R1, R2)



Potprogrami - Prijenos stogom

?

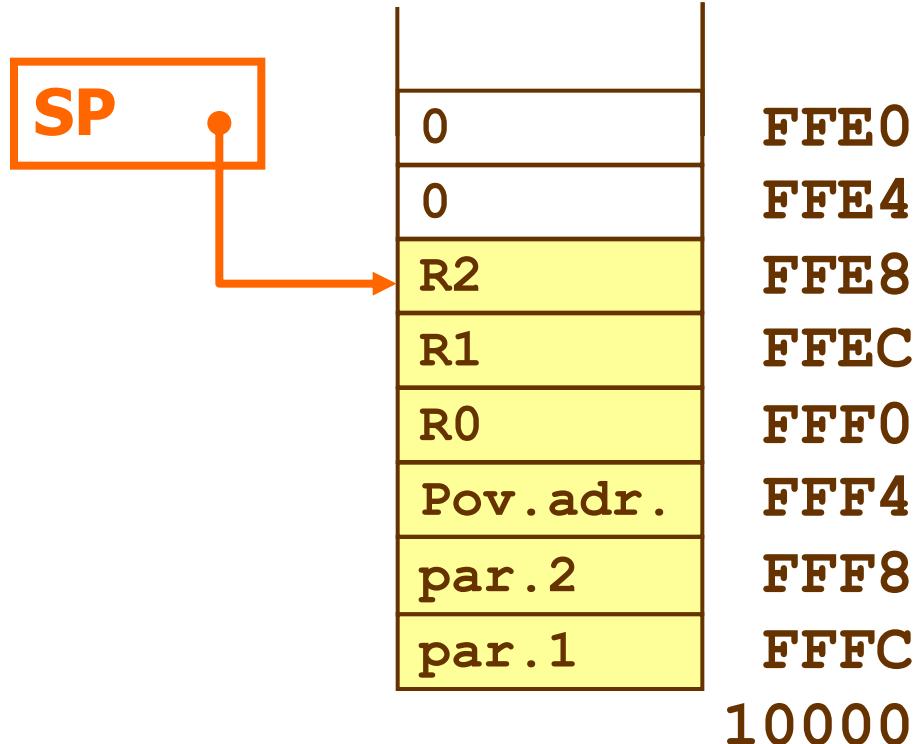
Kako dohvati parametre ?



Potprogrami - Prijenos stogom

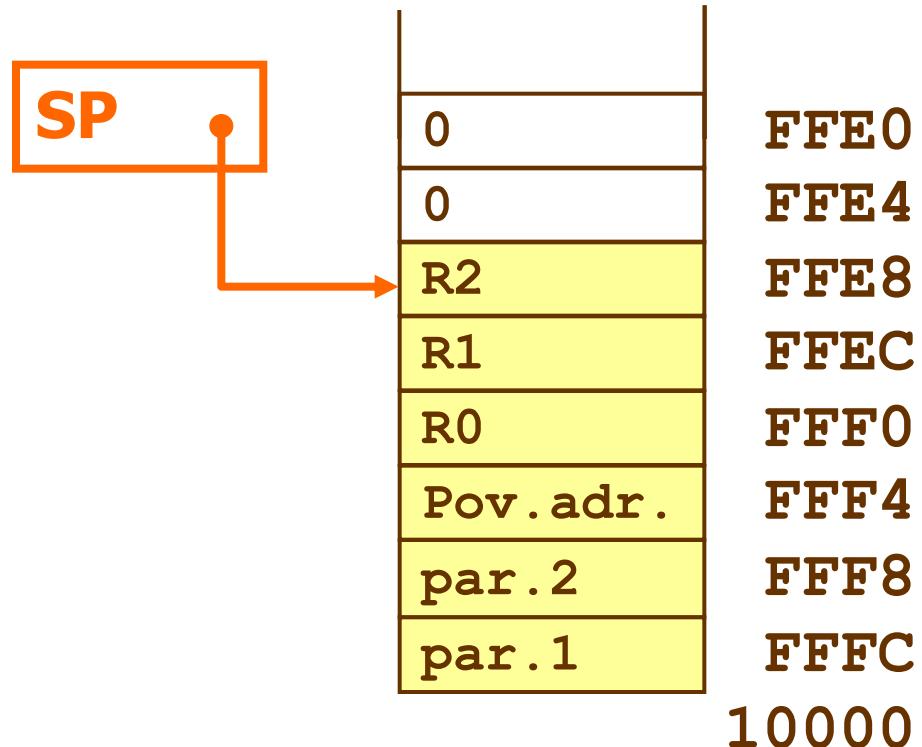
Da smo prije spremanja registara dohvatali parametre, onda bi uništili registre prije nego ih spremimo:

?



Potprogrami - Prijenos stogom

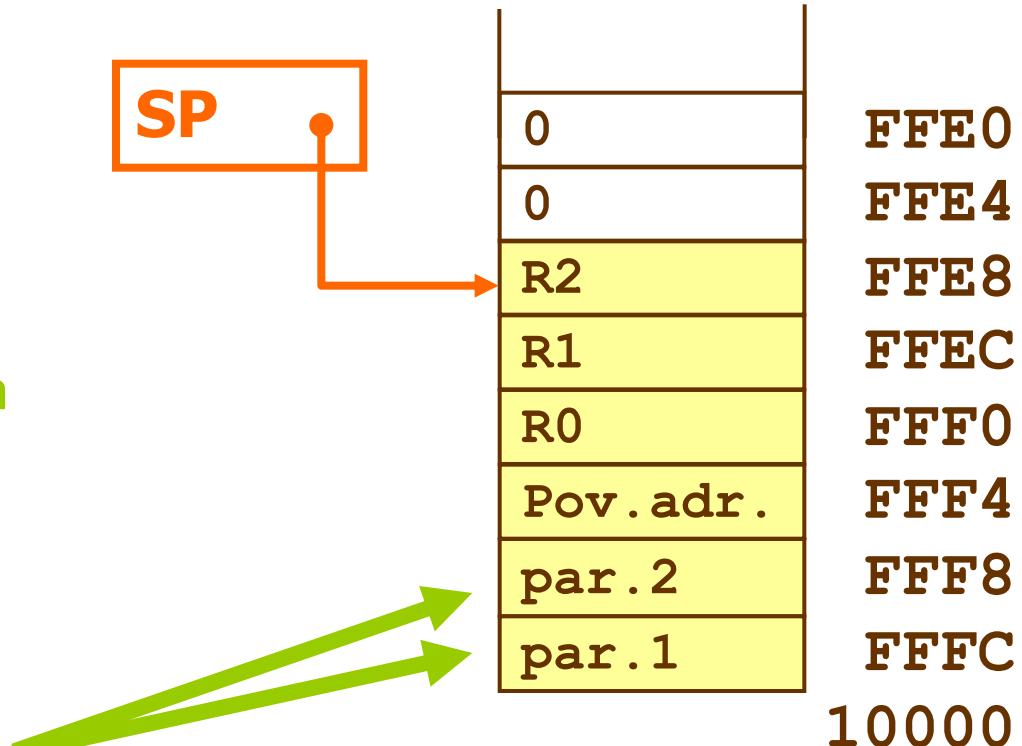
Jedna mogućnost (LOŠA!!!) je da presložimo podatke na stogu korištenjem fiksnih memorijskih lokacija (npr. tako da parametri dođu na vrh stoga)



Potprogrami - Prijenos stogom

Rješenje bi bilo kad ne bi morali pristupati podatcima na stogu samo pomoću PUSH i POP.

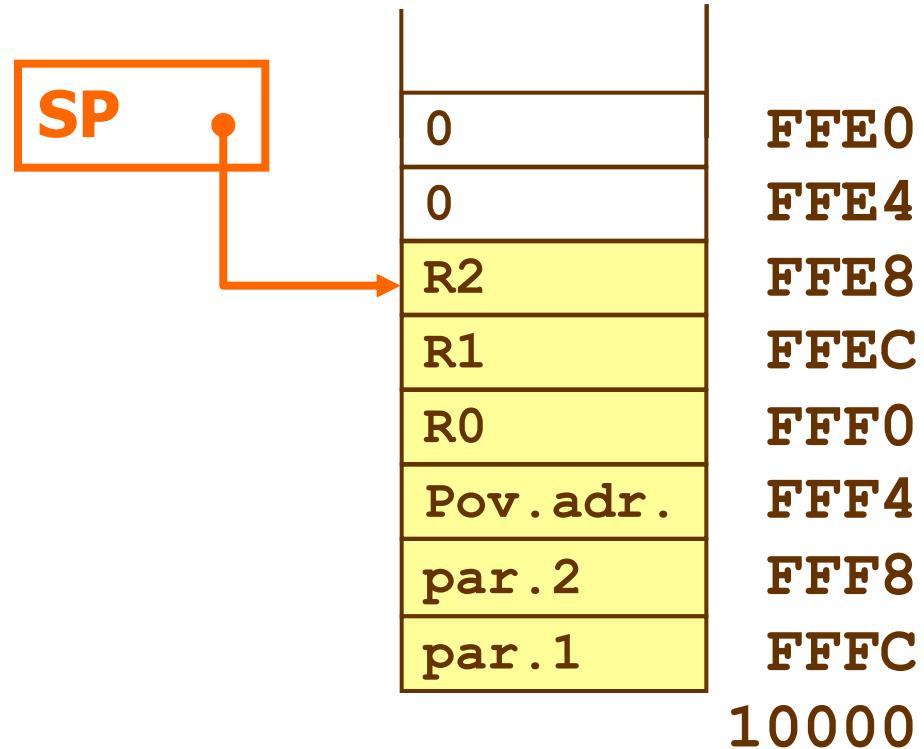
Kako možemo pristupiti podatcima "ispod" vrha stoga, bez pomicanja vrha stoga?



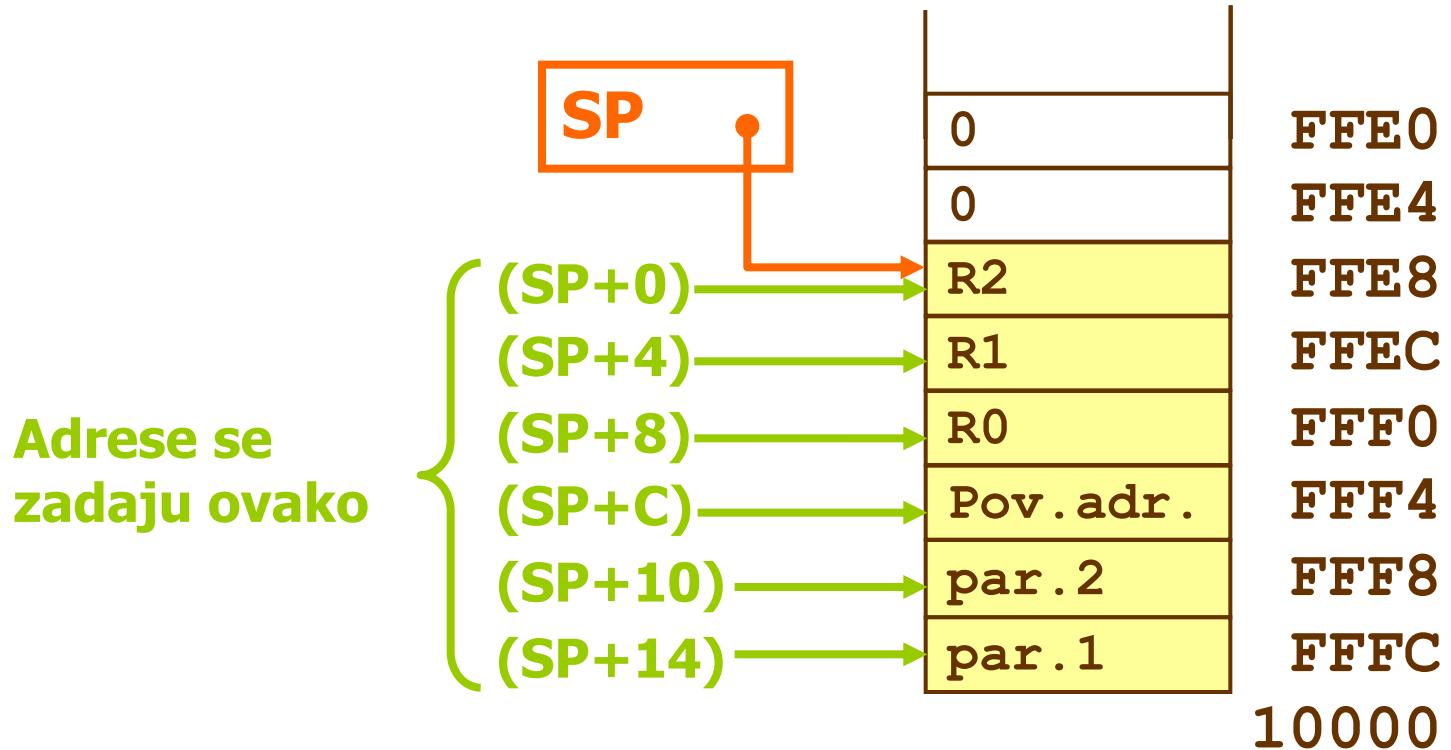
Potprogrami - Prijenos stogom

Rješenje je jednostavno:

Registarsko indirektno adresiranje s odmakom, pri čemu kao adresni register koristimo SP



Potprogrami - Prijenos stogom

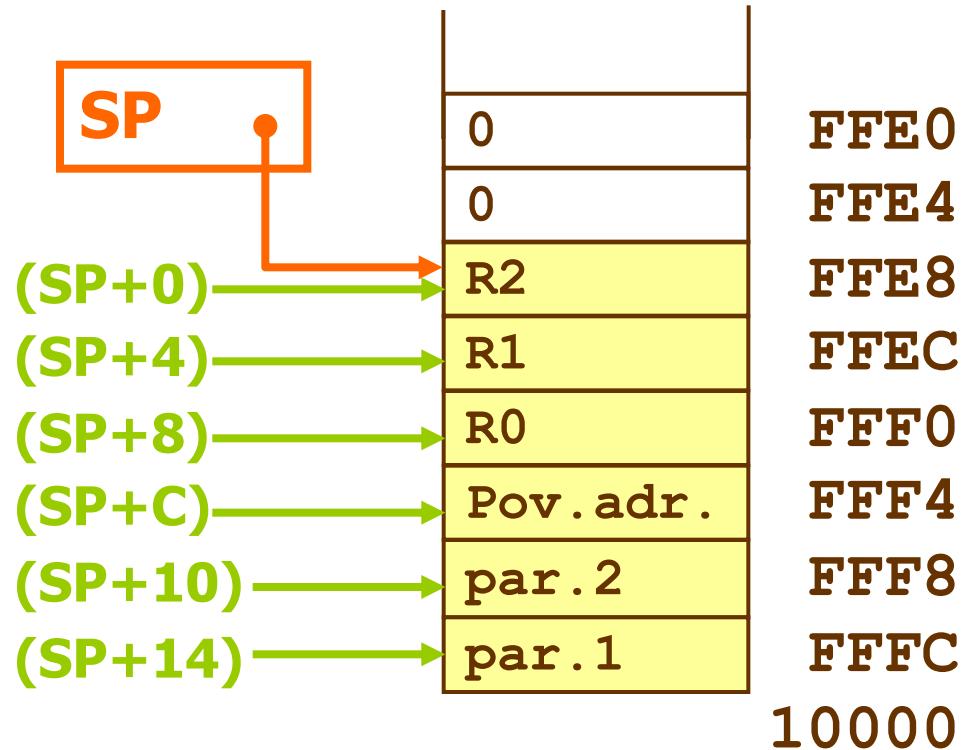


Adrese se
zadaju ovako

Naredbama LOAD i STORE
možemo pristupati
podatcima "unutar" stoga.

Potprogrami - Prijenos stogom

- Dok pišemo potprogram, točno znamo koliko on parametara ima i koliko registara mora spremiti na stog.
- Zato lako možemo izračunati odmake pojedinih parametara u odnosu na SP (vrh stoga).



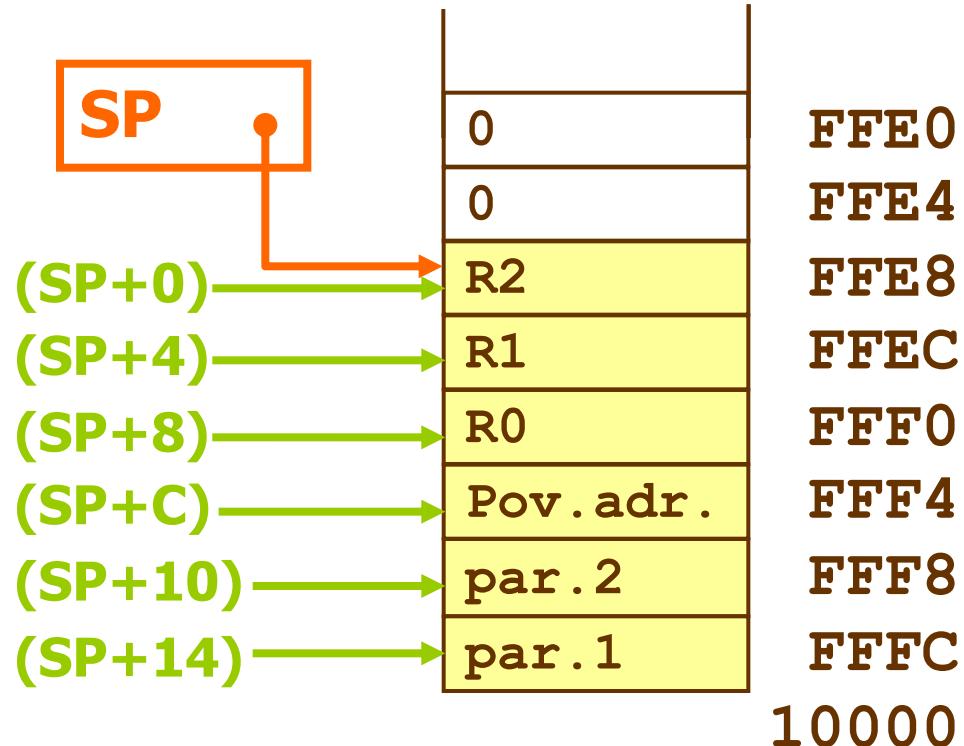
Potprogrami - Prijenos stogom

Promatramo stanje na stogu za potprogram koji bi spremao registre na stog.

Zadnji problem je povratna vrijednost.

Neka je upravo završeno izračunavanje rezultata potprograma i neka je rezultat npr. u registru R0.

Želimo se vratiti iz potprograma:



Potprogrami - Prijenos stogom

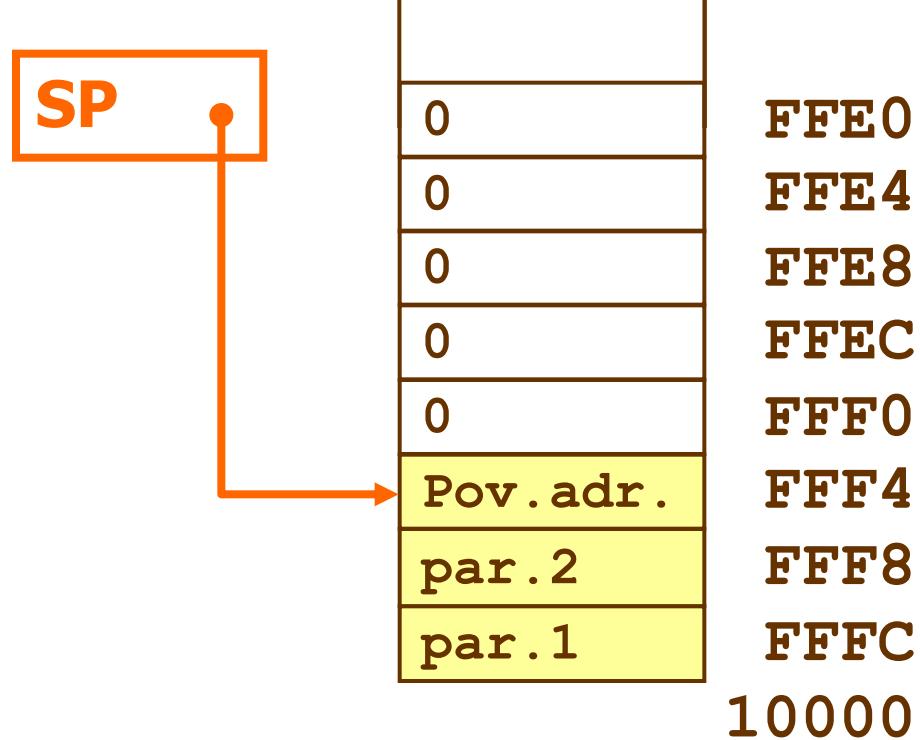
Promatramo stanje na stogu za potprogram koji bi spremao registre na stog.

Izvodi se:

Obnavljaju se registri
R0, R1 i R2.

Povratak se lako
ostavaruje sa RET

Ali, obnavljanjem R0
gubi se rezultat !!!



Potprogrami - Prijenos stogom

Rješenje koje se koristi u praksi:

- Za **povratak rezultata** iz potprograma koristi se **prijenos registrom**, a ne stogom.
- Ovo rješenje je efikasno i jednostavno. Ograničenje je da se može vratiti samo jedan rezultat, ali je u praksi to obično dovoljno.
- Obično se odabire jedan registar koji svi potprogrami koriste za povratak vrijednosti, a na mjestu pozivanja se zna da u tom registru ne smije biti koristan podatak.

Potprogrami - Prijenos stogom

Napisati potprogram koji izračunava logičku operaciju NILI.
Parametri se prenose stogom, a rezultat se vraća registrom R0.
Glavni program iz memorije učitava dva podatka za koje računa
NILI (pomoću potprograma) i rezultat spremi natrag u
memoriju.

Potprogram mora čuvati vrijednosti svih registara koje mijenja.

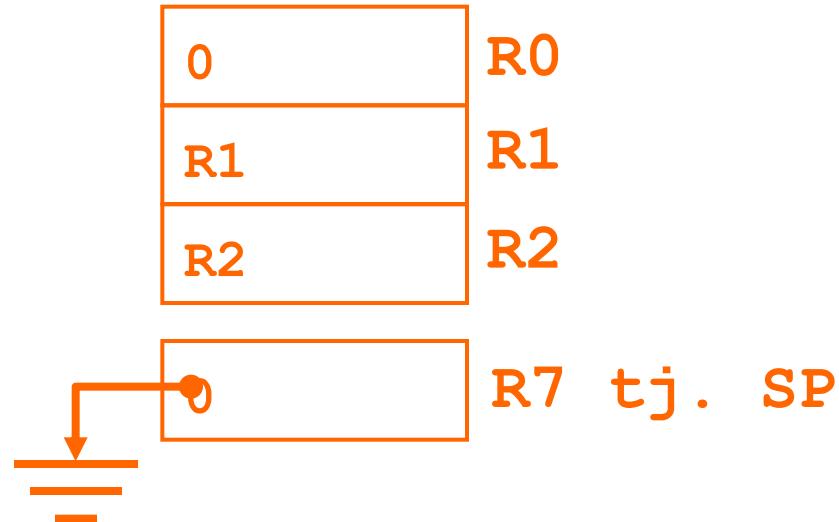
Rješenje:

- U ovom rješenju potprogram mijenja registre R1 i R2 i spremi ih na stog (R0 se također mijenja, ali preko njega se ionako vraća povratna vrijednost)
- Parametrima se pristupa registarskim indirektnim adresiranjem s odmakom (korištenjem SP-a kao adresnog registra)

>>>

<<< Izvođenje programa:

```
GLAVNI MOVE 10000, SP  
          LOAD R0, (PRVI)  
          PUSH R0  
  
          LOAD R0, (DRUGI)  
          PUSH R0  
  
          CALL NILI  
  
          STORE R0, (REZ)  
  
          ADD SP, 8, SP  
  
          HALT
```



0	FFE8
0	FFEC
0	FFF0
0	FFF4
0	FFF8
0	FFFC

<<< Izvođenje programa:

GLAVNI MOVE 10000, SP ←

LOAD R0, (PRVI)

PUSH R0

LOAD R0, (DRUGI)

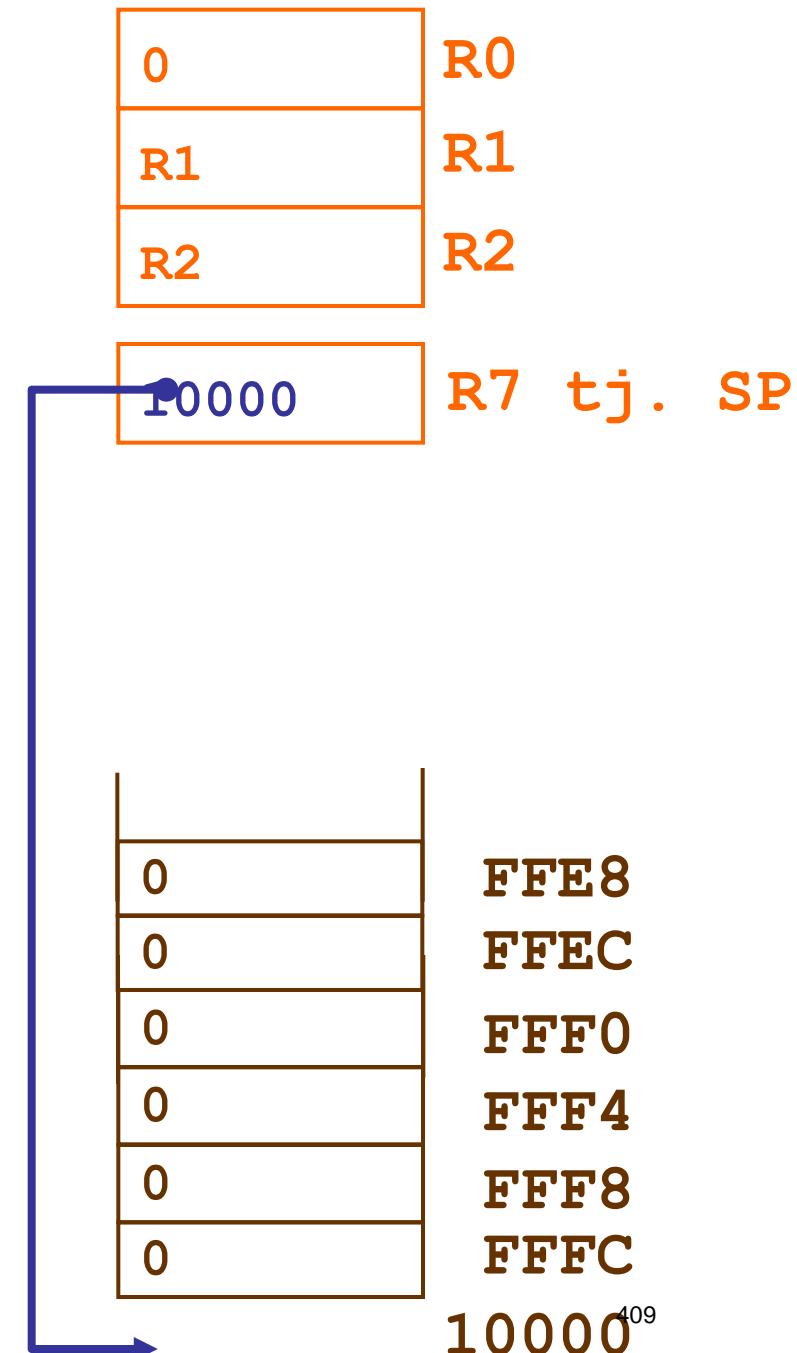
PUSH R0

CALL NILI

STORE R0, (REZ)

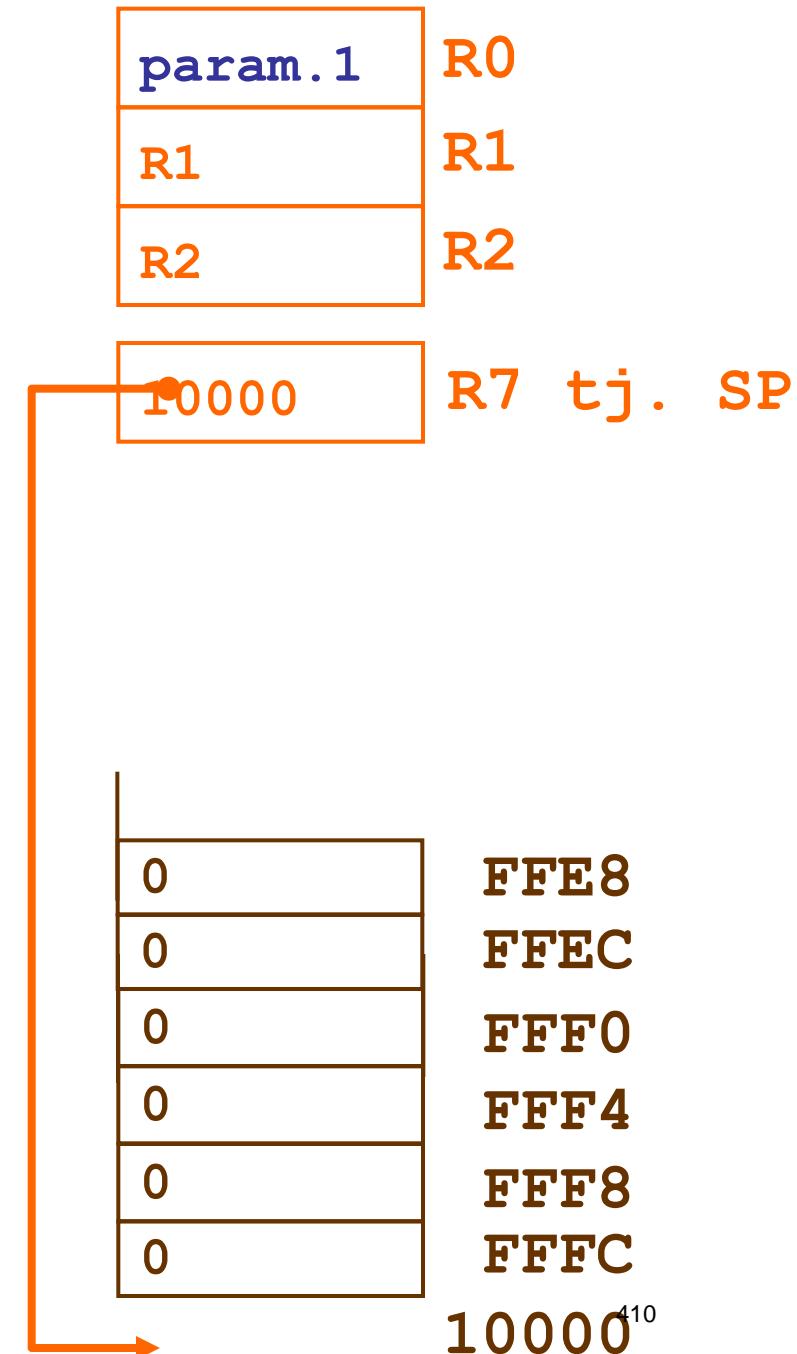
ADD SP, 8, SP

HALT



<<< Izvođenje programa:

```
GLAVNI MOVE 10000, SP  
LOAD R0, (PRVI) ←  
PUSH R0  
  
LOAD R0, (DRUGI)  
PUSH R0  
  
CALL NILI  
  
STORE R0, (REZ)  
  
ADD SP, 8, SP  
  
HALT
```



<<< Izvođenje programa:

GLAVNI MOVE 10000, SP

LOAD R0, (PRVI)

PUSH R0



LOAD R0, (DRUGI)

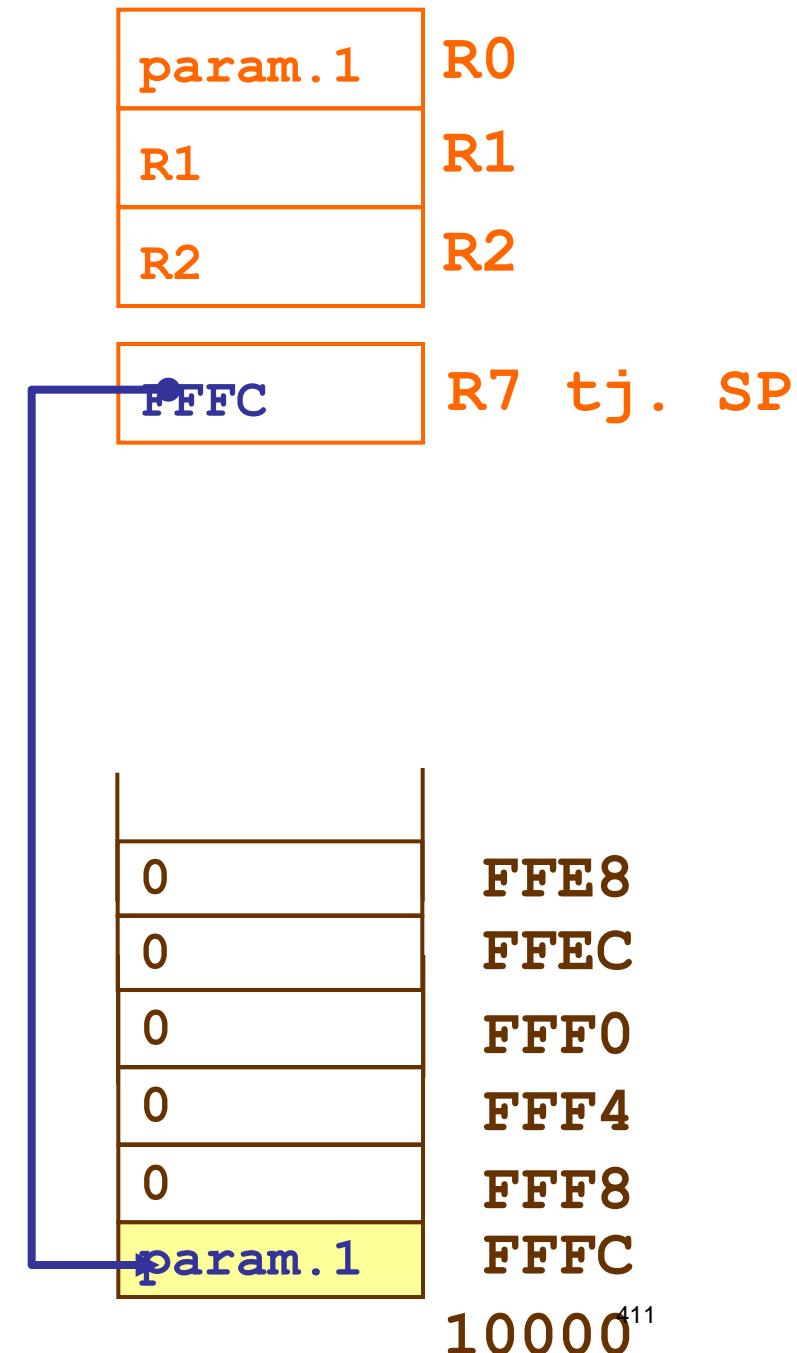
PUSH R0

CALL NILI

STORE R0, (REZ)

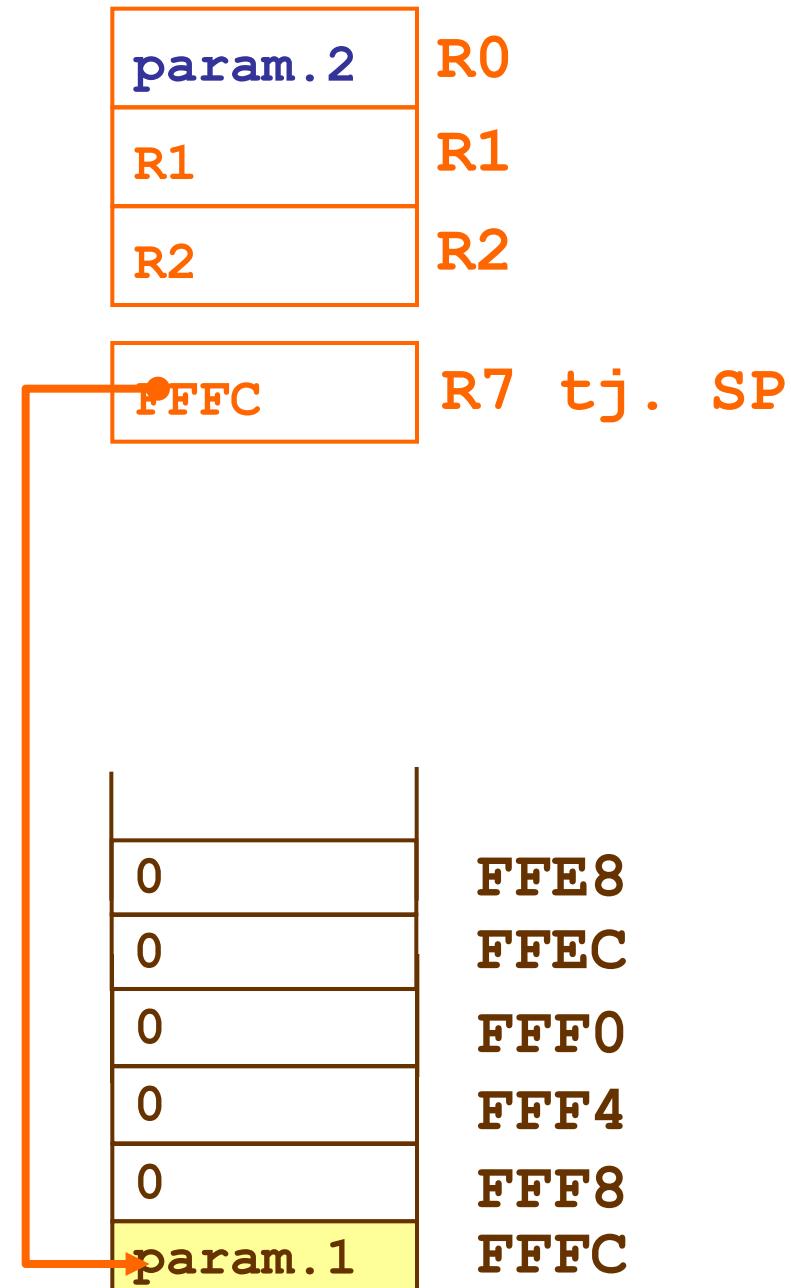
ADD SP, 8, SP

HALT



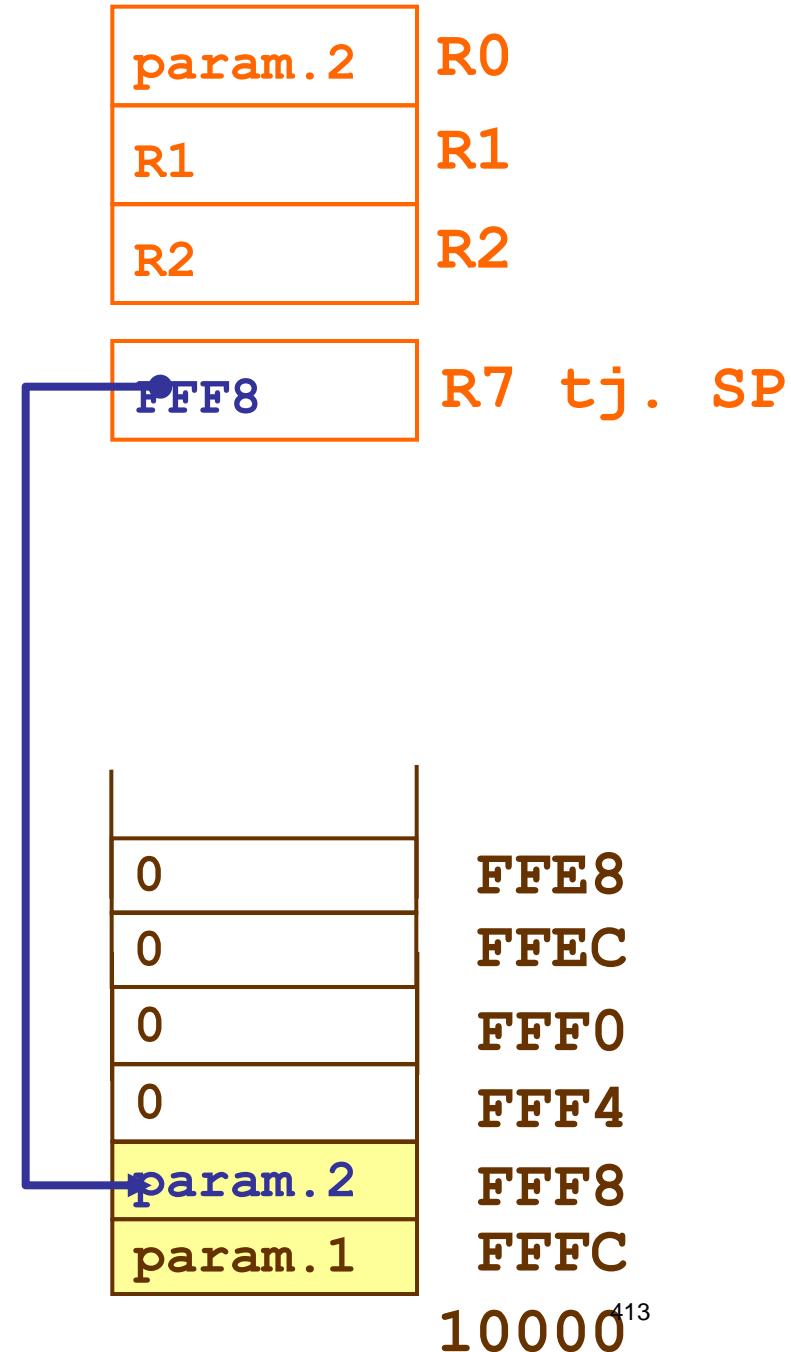
<<< Izvođenje programa:

```
GLAVNI MOVE 10000, SP  
          LOAD R0, (PRVI)  
          PUSH R0  
  
          LOAD R0, (DRUGI) ←  
          PUSH R0  
  
          CALL NILI  
  
          STORE R0, (REZ)  
  
          ADD SP, 8, SP  
  
          HALT
```



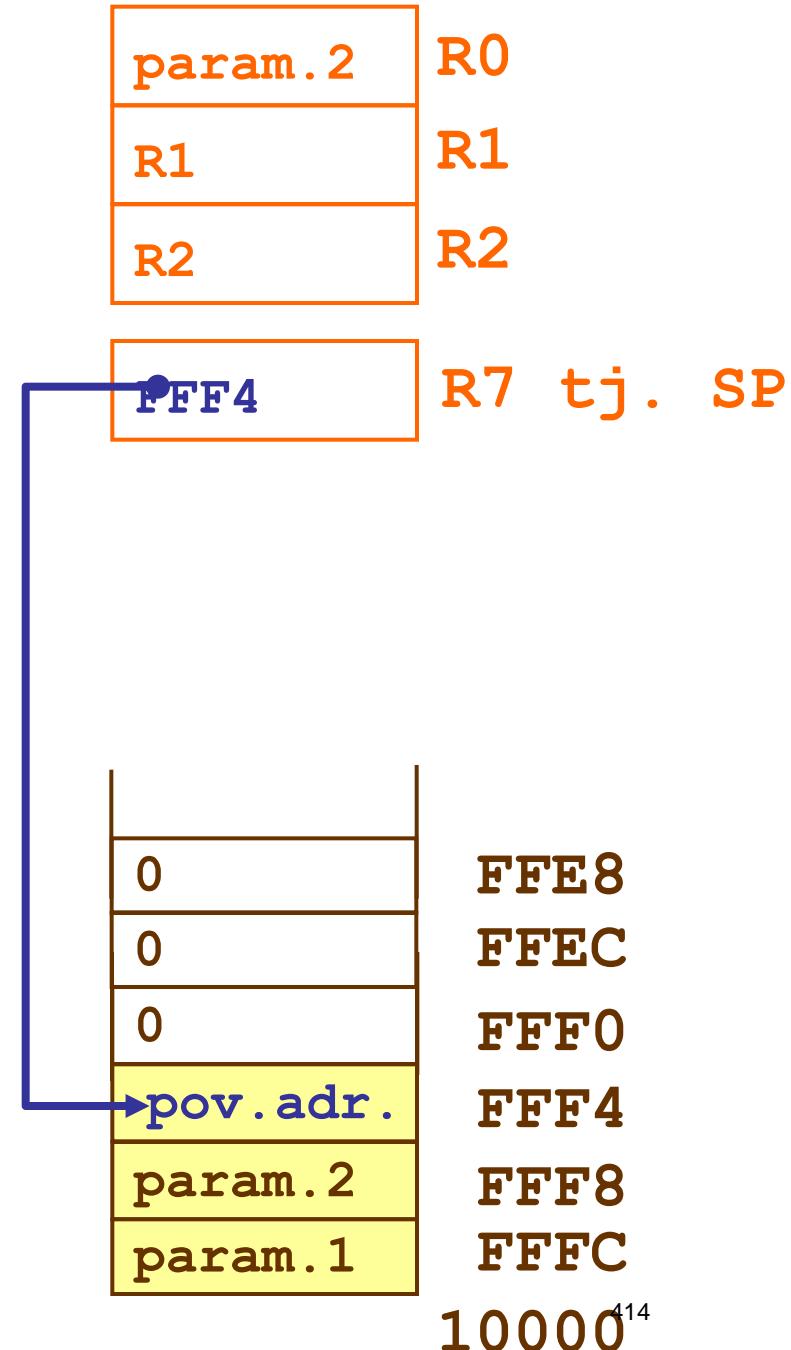
<<< Izvođenje programa:

```
GLAVNI MOVE 10000, SP  
          LOAD R0, (PRVI)  
          PUSH R0  
  
          LOAD R0, (DRUGI)  
          PUSH R0  
  
          CALL NILI  
  
          STORE R0, (REZ)  
  
          ADD SP, 8, SP  
  
          HALT
```



<<< Izvođenje programa:

```
GLAVNI MOVE 10000, SP  
          LOAD R0, (PRVI)  
          PUSH R0  
  
          LOAD R0, (DRUGI)  
          PUSH R0  
  
          CALL NILI  
  
          STORE R0, (REZ)  
  
          ADD SP, 8, SP  
  
          HALT
```



<<< Izvođenje programa:

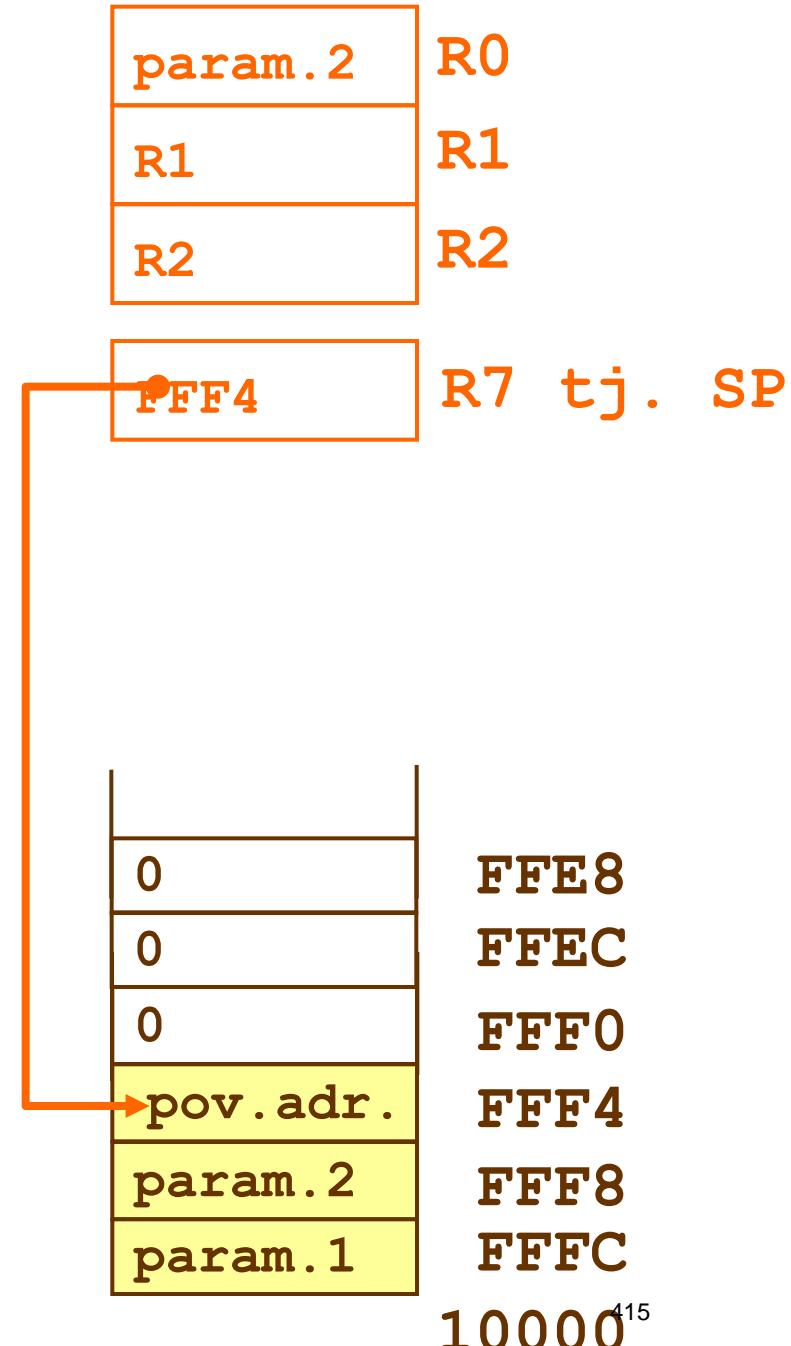
```
NILI    PUSH   R1
       PUSH   R2

       LOAD   R1 , (SP+0C)
       LOAD   R2 , (SP+10)

       OR     R1 , R2 , R0
       XOR   R0 , -1 , R0 ←

       POP   R2
       POP   R1

       RET
```



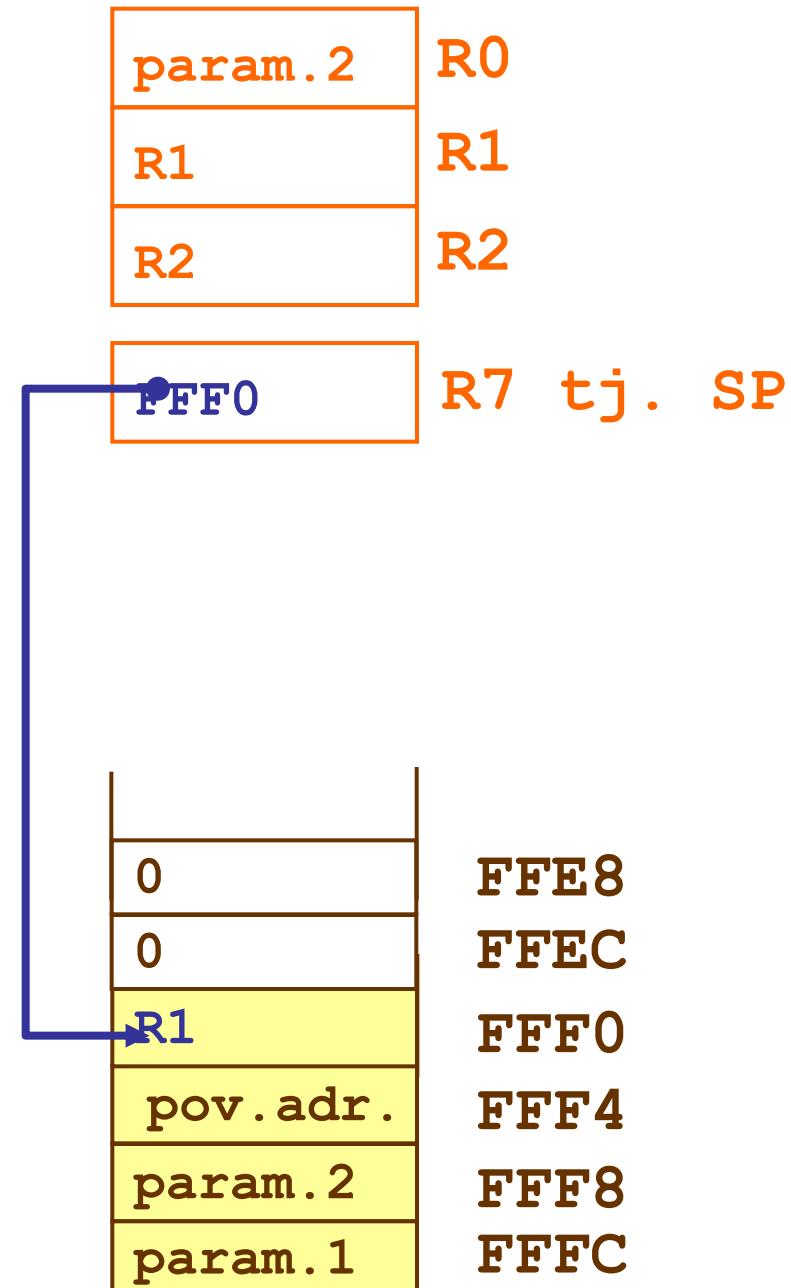
<<< Izvođenje programa:

```
NILI    PUSH   R1
       PUSH   R2
       LOAD   R1 , (SP+0C)
       LOAD   R2 , (SP+10)

       OR     R1 , R2 , R0
       XOR   R0 , -1 , R0

       POP   R2
       POP   R1

       RET
```



<<< Izvođenje programa:

NILI PUSH R1
 PUSH R2



LOAD R1 , (SP+0C)
LOAD R2 , (SP+10)

OR R1 , R2 , R0
XOR R0 , -1 , R0

POP R2
POP R1

RET

param.2	R0
R1	R1
R2	R2

PFEC	R7 tj. SP
------	-----------

0	FFE8
PFEC	FFEC
R1	FFF0
pov.adr.	FFF4
param.2	FFF8
param.1	FFFC

<<< Izvođenje programa:

NILI PUSH R1
 PUSH R2

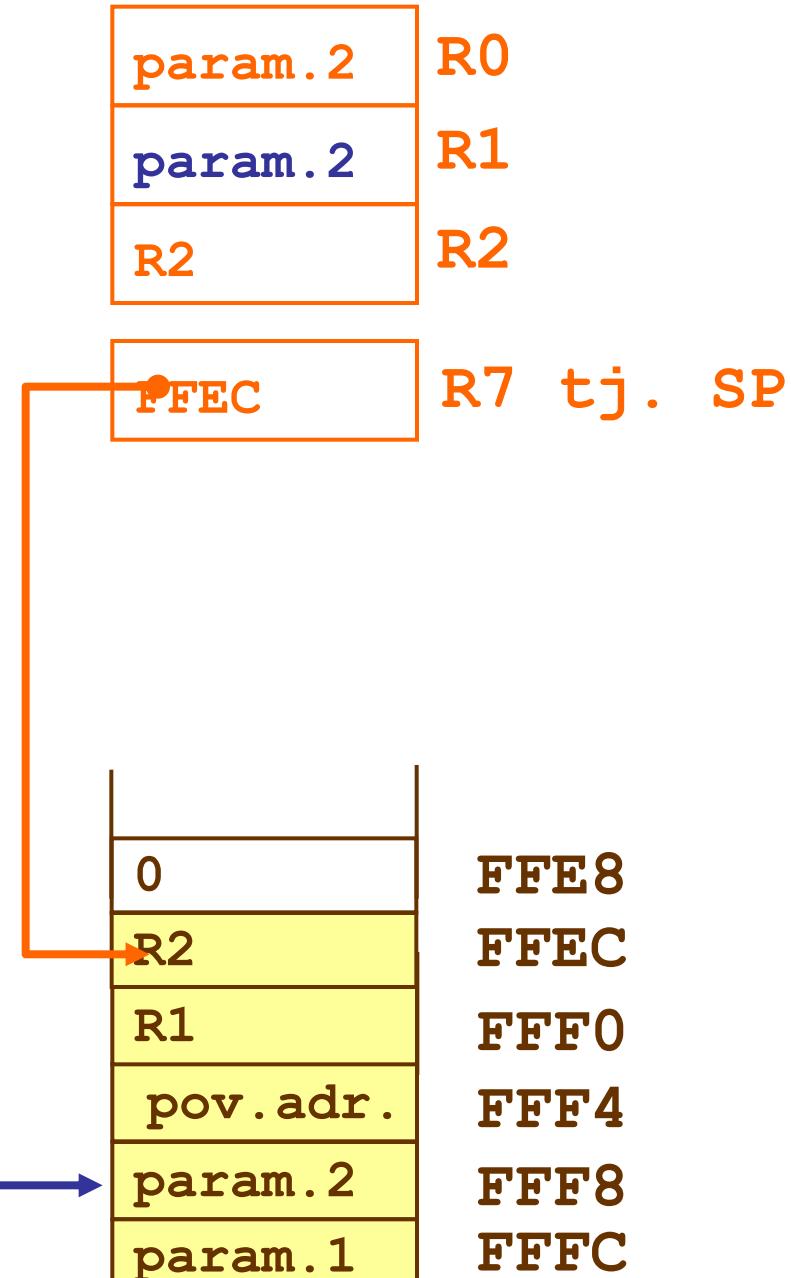
LOAD R1 , (SP+0C) ←
LOAD R2 , (SP+10)

OR R1 , R2 , R0
XOR R0 , -1 , R0

POP R2
POP R1

RET

(SP+0C) →



<<< Izvođenje programa:

NILI PUSH R1
 PUSH R2

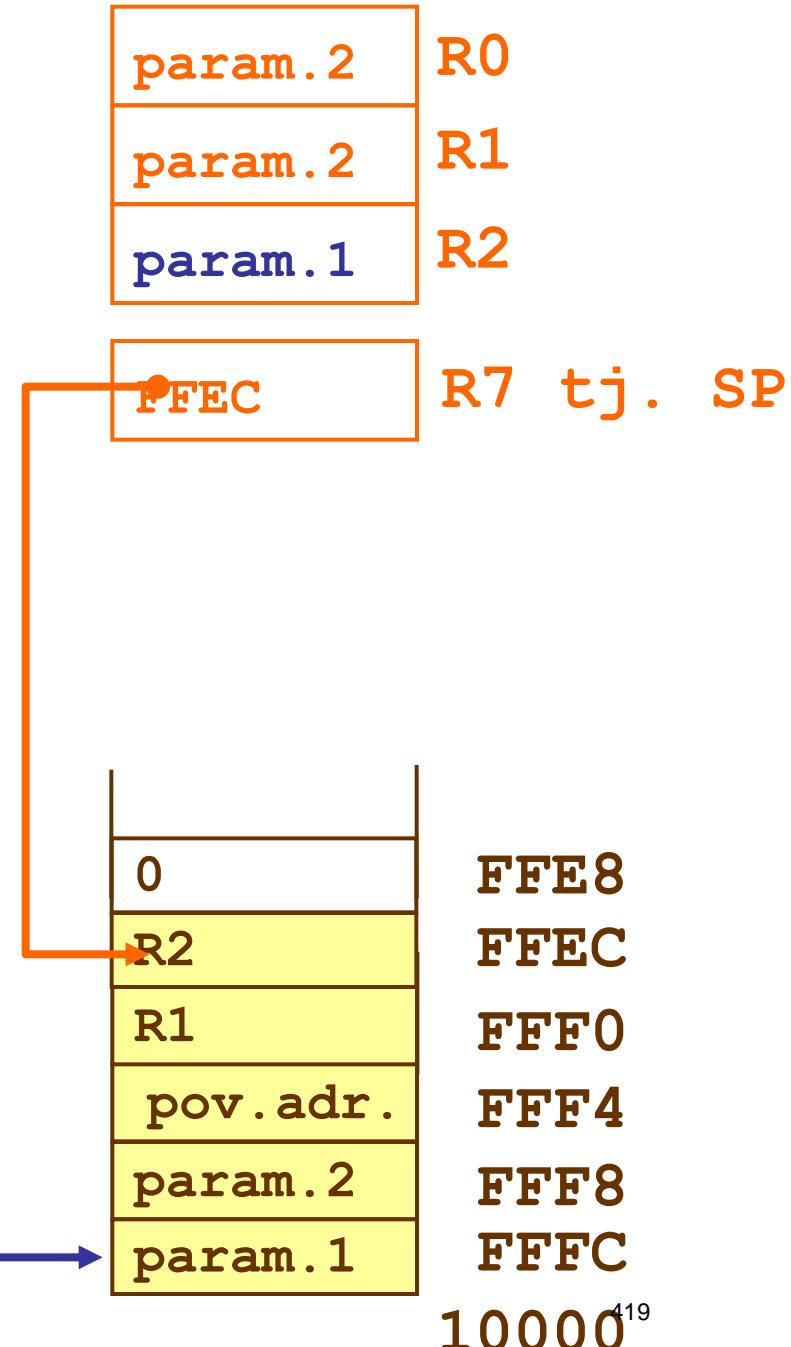
LOAD R1 , (SP+0C)
LOAD R2 , (SP+10) ←

OR R1 , R2 , R0
XOR R0 , -1 , R0

POP R2
POP R1

RET

(SP+10) →



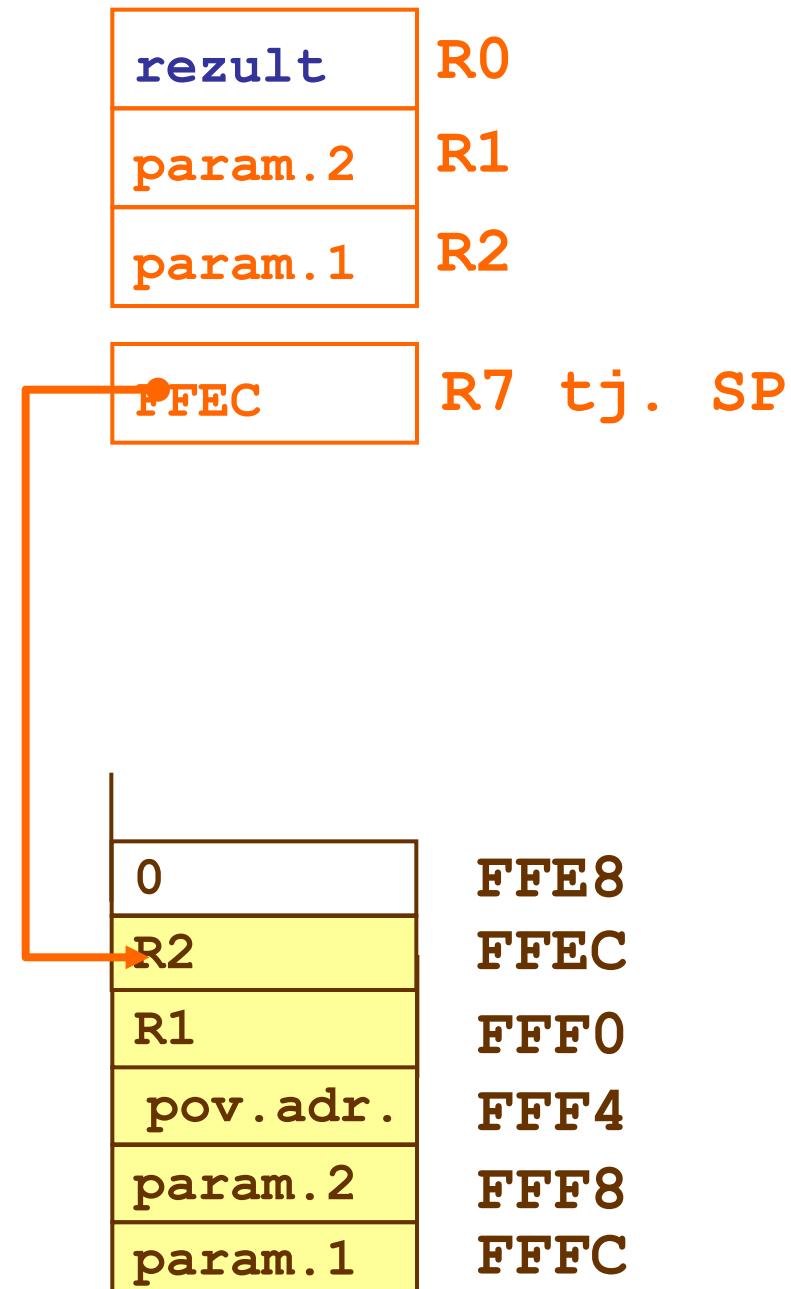
<<< Izvođenje programa:

```
NILI    PUSH   R1
       PUSH   R2
       LOAD   R1 , (SP+0C)
       LOAD   R2 , (SP+10)

       OR     R1 , R2 , R0
       XOR   R0 , -1 , R0 ←

       POP   R2
       POP   R1

       RET
```



<<< Izvođenje programa:

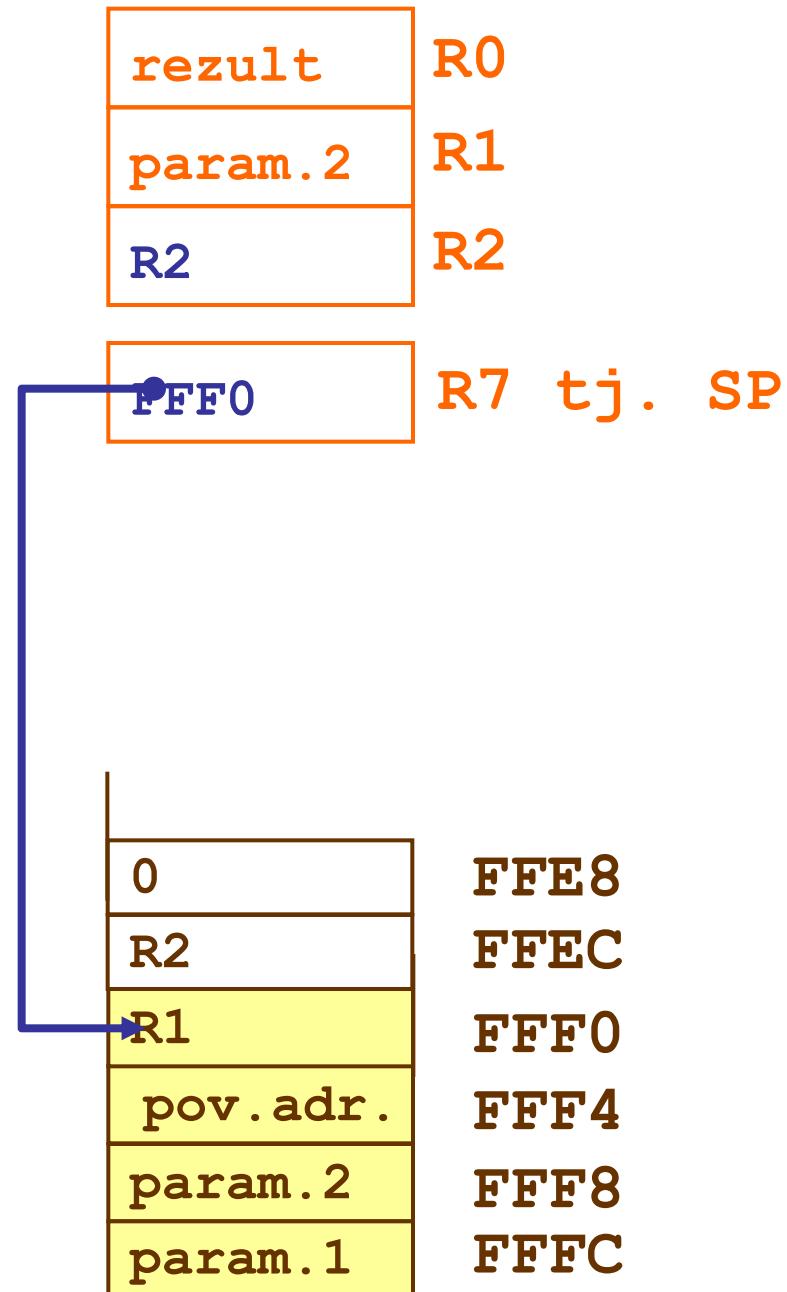
```
NILI    PUSH   R1
       PUSH   R2

       LOAD   R1 , (SP+0C)
       LOAD   R2 , (SP+10)

       OR     R1 , R2 , R0
       XOR   R0 , -1 , R0

       POP   R2
       POP   R1

       RET
```



<<< Izvođenje programa:

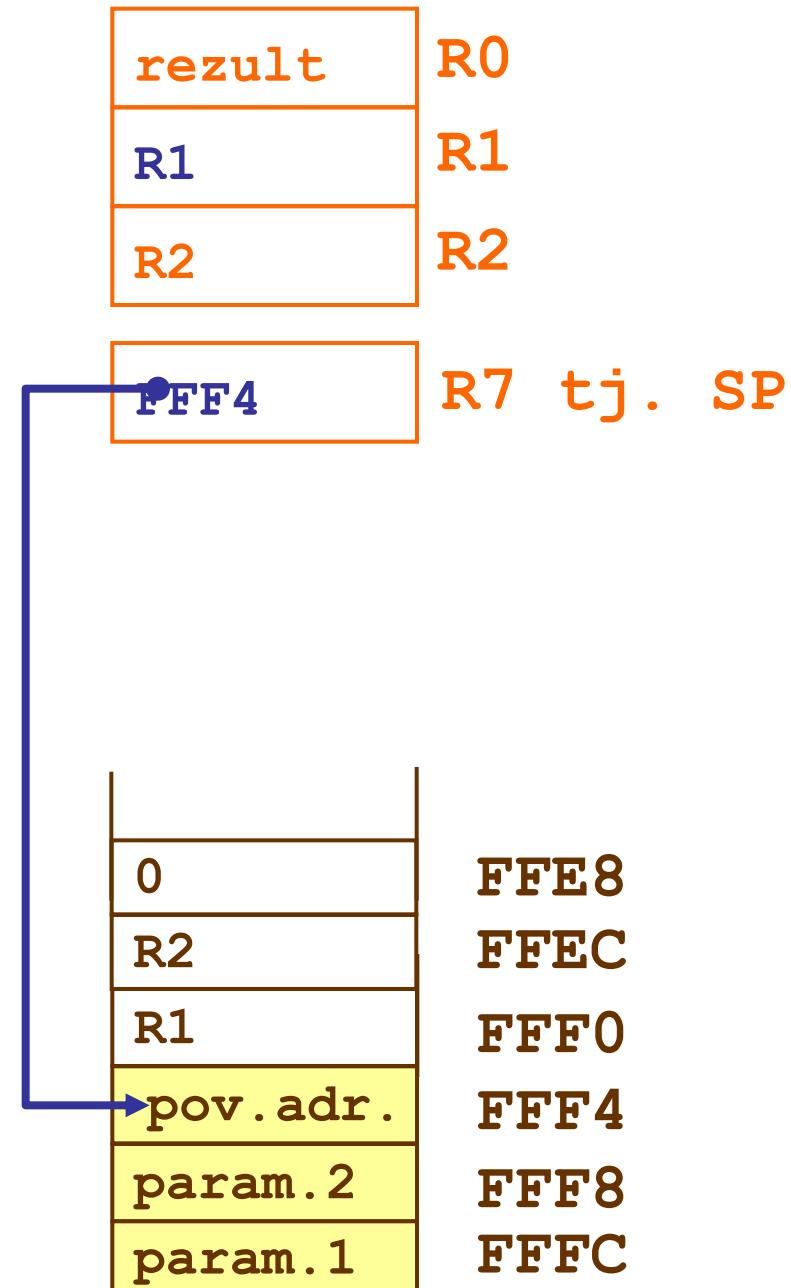
```
NILI    PUSH   R1
       PUSH   R2

       LOAD   R1 , (SP+0C)
       LOAD   R2 , (SP+10)

       OR     R1 , R2 , R0
       XOR   R0 , -1 , R0

       POP   R2
       POP   R1

       RET
```



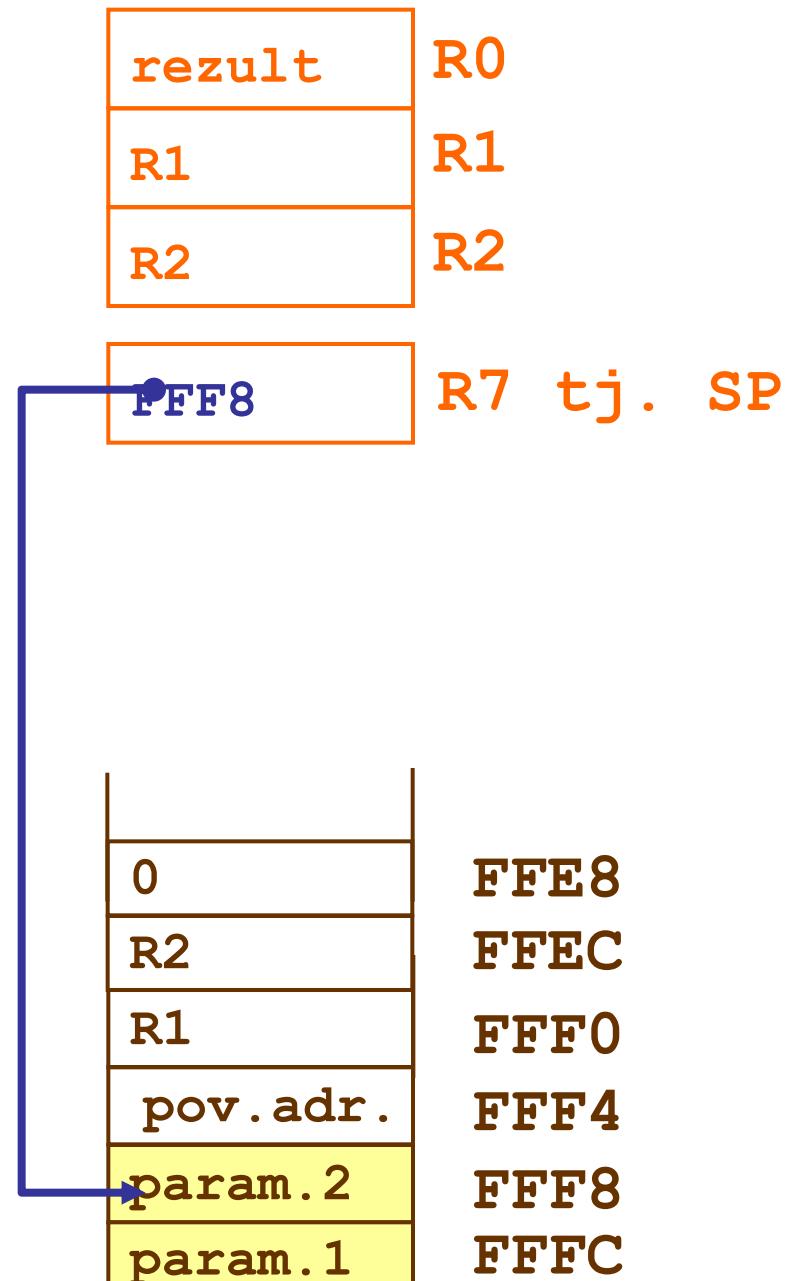
<<< Izvođenje programa:

```
NILI    PUSH   R1
       PUSH   R2
       LOAD   R1 , (SP+0C)
       LOAD   R2 , (SP+10)

       OR     R1 , R2 , R0
       XOR   R0 , -1 , R0

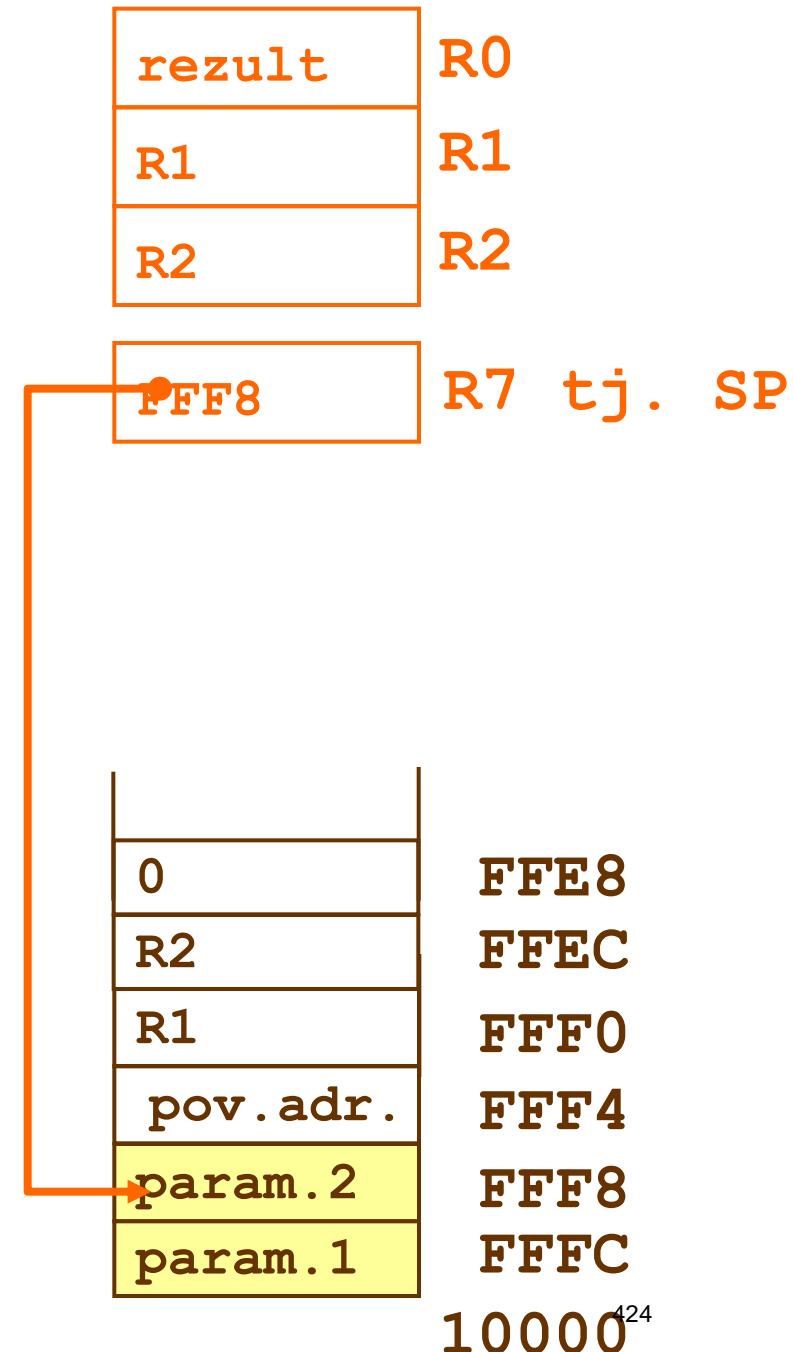
       POP   R2
       POP   R1

       RET
```



<<< Izvođenje programa:

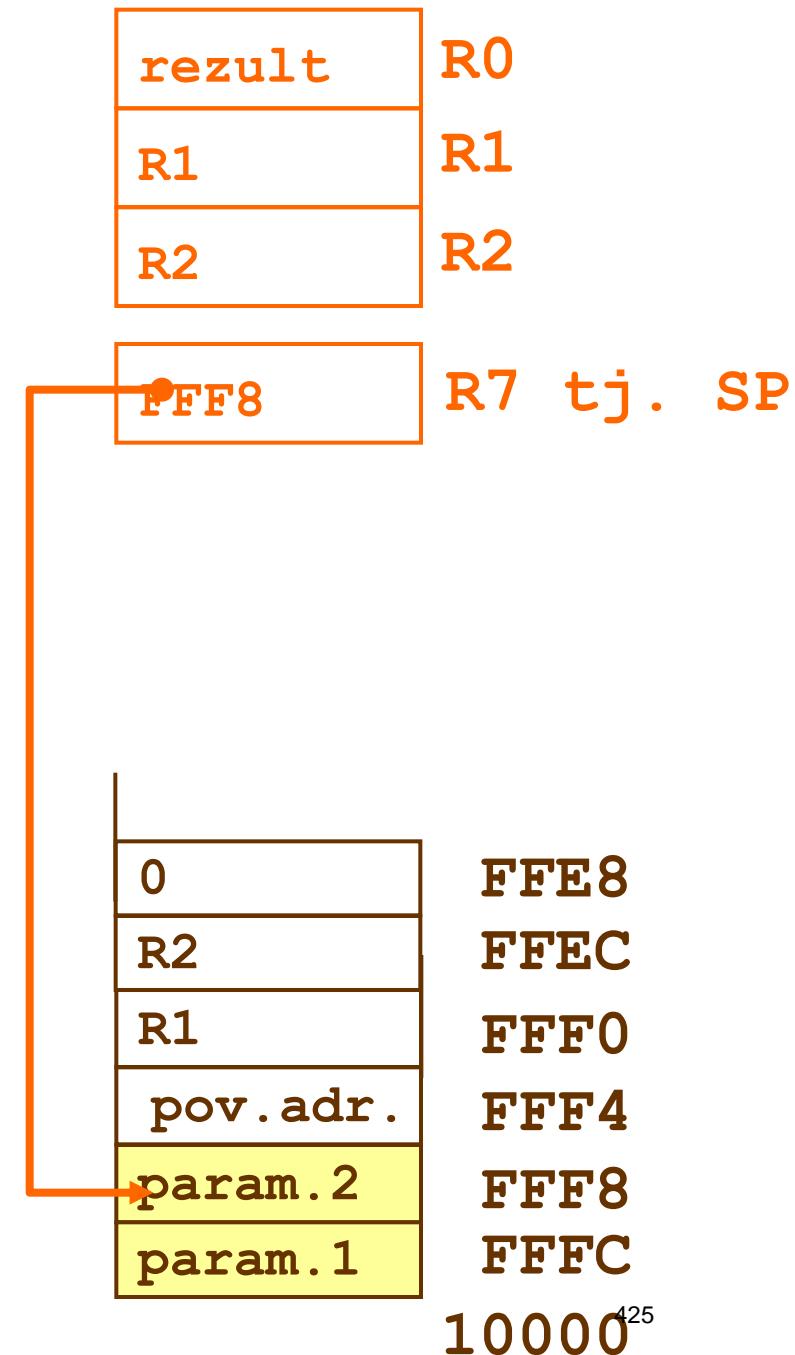
```
GLAVNI MOVE 10000, SP  
          LOAD R0, (PRVI)  
          PUSH R0  
  
          LOAD R0, (DRUGI)  
          PUSH R0  
  
          CALL NILI  
  
          STORE R0, (REZ)  
  
          ADD SP, 8, SP  
  
HALT
```



<<< Izvođenje programa:

```
GLAVNI MOVE 10000, SP  
          LOAD R0, (PRVI)  
          PUSH R0  
  
          LOAD R0, (DRUGI)  
          PUSH R0  
  
          CALL NILI  
  
          STORE R0, (REZ) ←  
  
          ADD SP, 8, SP  
  
          HALT
```

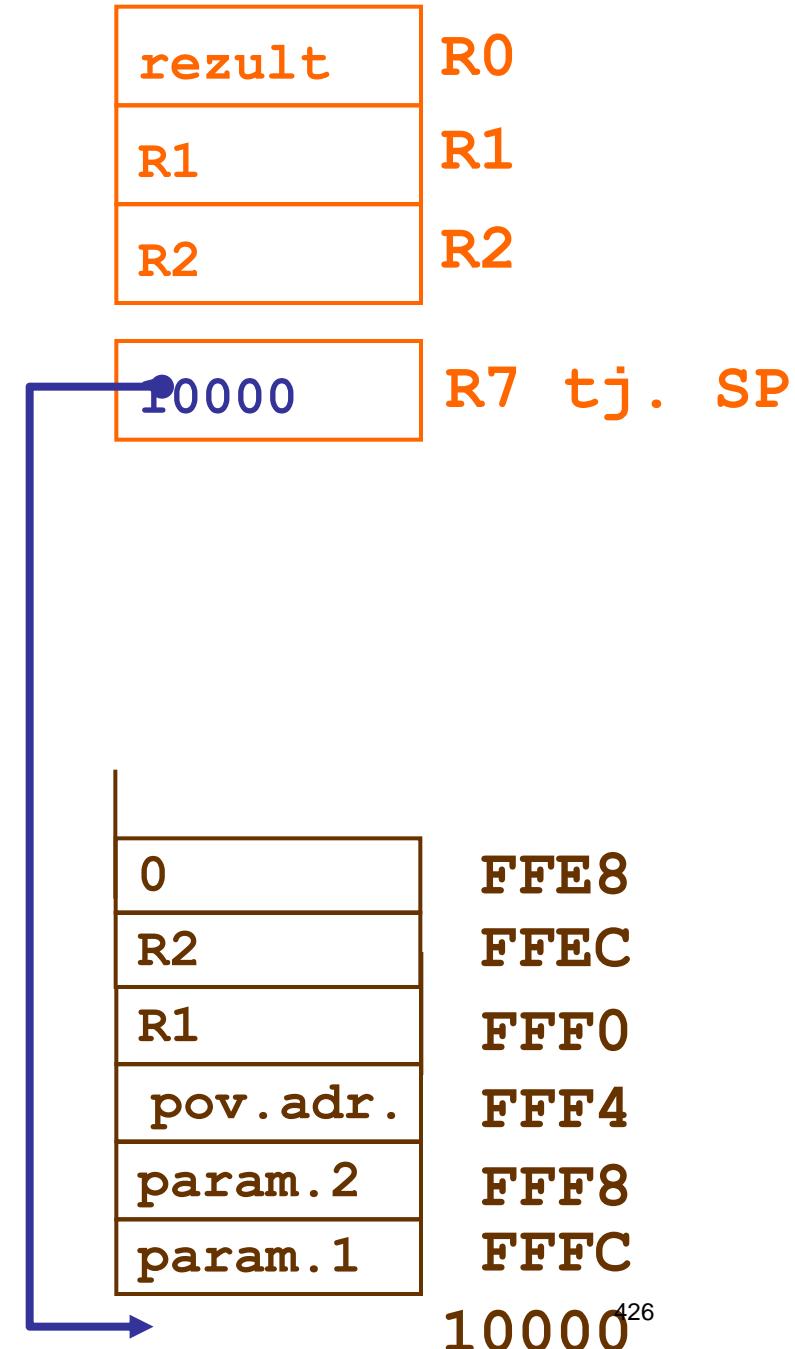
rezultat se sprema
izravno iz R0



<<< Izvođenje programa:

```
GLAVNI MOVE 10000, SP  
          LOAD R0, (PRVI)  
          PUSH R0  
  
          LOAD R0, (DRUGI)  
          PUSH R0  
  
          CALL NILI  
  
          STORE R0, (REZ)  
  
          ADD SP, 8, SP ←  
  
          HALT
```

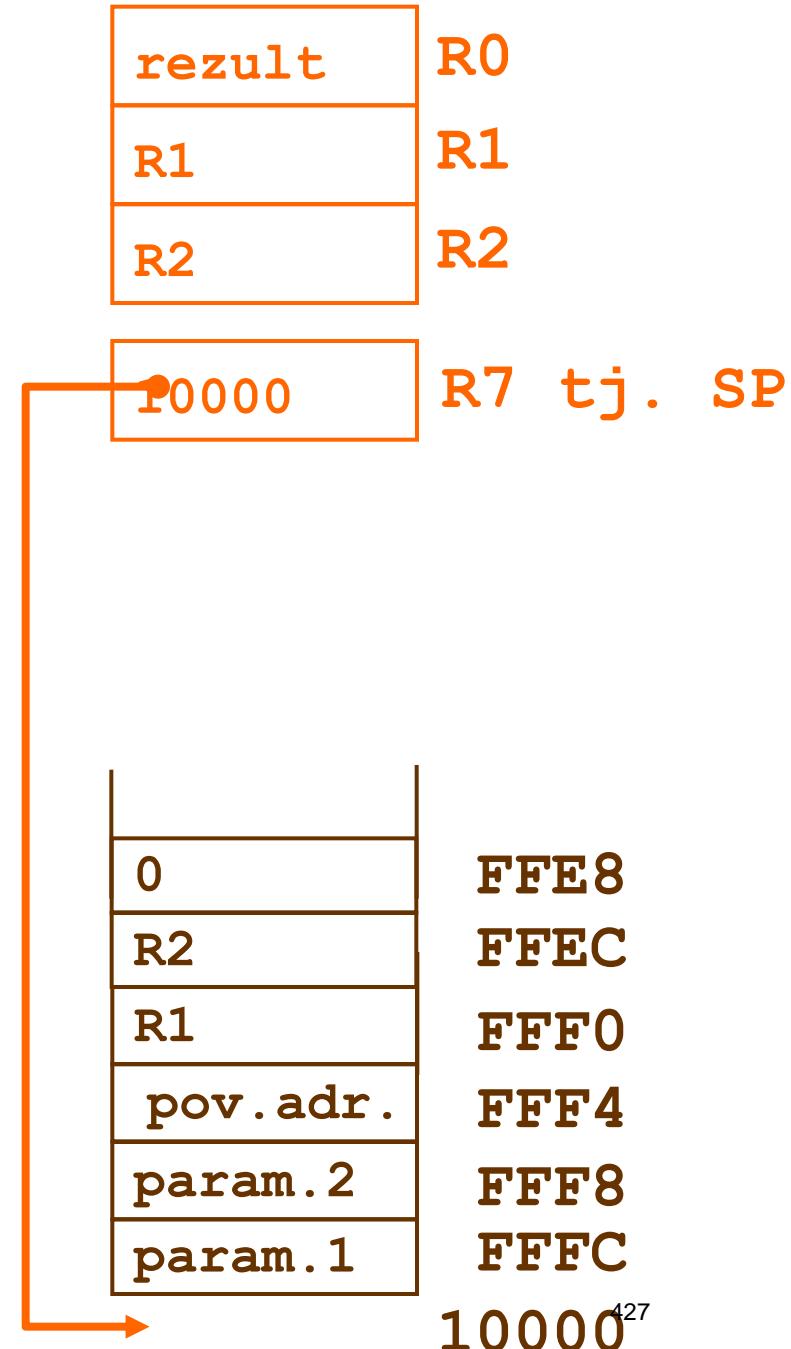
glavni program uklanja
parametre sa stoga



Komentari:

- Za razliku od prethodnih primjera, gdje je potprogram pomoću naredaba POP "potrošio" svoje parametre, ovdje parametre uklanja pozivatelj.
- Ovo je "čišće" rješenje, jer je logičnije da je parametre sa stoga dužan ukloniti onaj tko ih je i stavio na stog.
- Umjesto niza naredaba POP, parametri se uklanjuju naredbom:
ADD SP, 8, SP

što je ne samo brže, nego dodatno čuva vrijednosti svih registara.



<<< (kompletan listing s komentarima)

```
; glavni program
GLAVNI MOVE 10000, SP      ;važno: inicijaliziraj SP !!!
                           ; stavi vrijednost
LOAD   R0, (PRVI)        ; prvog parametra na stog
PUSH   R0
                           ; stavi vrijednost
LOAD   R0, (DRUGI)        ; drugog parametra na stog
PUSH   R0
                           ; poziv potprograma
CALL   NILI
                           ; spremi rezultat iz R0
STORE  R0, (REZ)
                           ; ukloni parametre sa stoga
ADD    SP, 8, SP
HALT
```

; podatci i mjesto za rezultat

```
PRVI  DW 1234ABCD
DRUGI DW 22445599
REZ   DW 0
```

>>>

<<<

```
; potprogram NILI
NILI  PUSH   R1      ; Spremanje
      PUSH   R2      ; registara.

      LOAD   R1,  (SP+0C)    ; Čitanje parametara
      LOAD   R2,  (SP+10)    ; u registre R1 i R2

      OR     R1,  R2,  R0      ; Izračunavanje
      XOR   R0,  -1,  R0      ; rezultata.

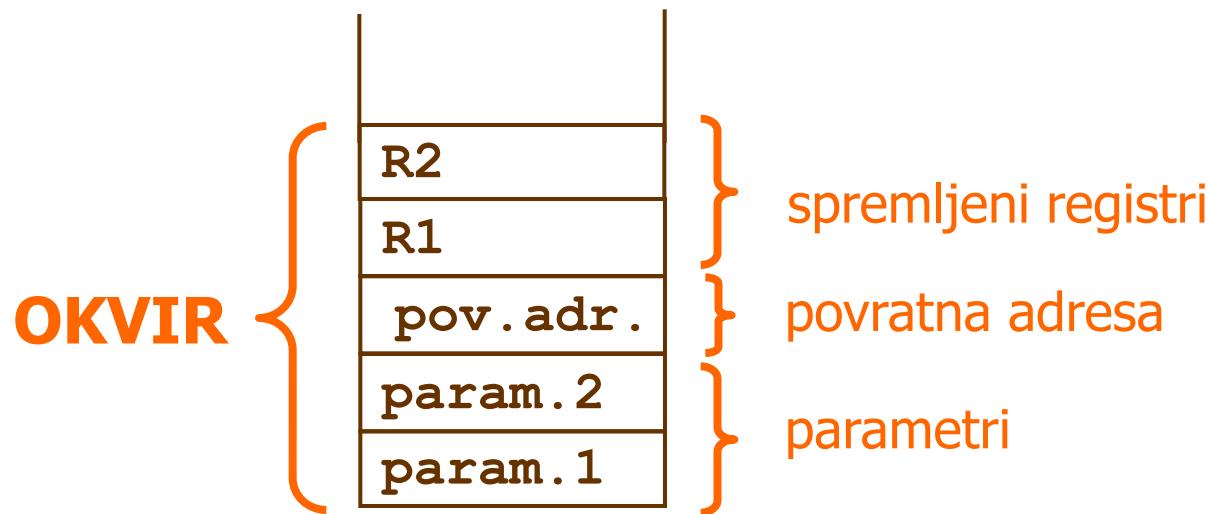
      POP   R2      ; Obnovi
      POP   R1      ; registre.

      RET
```

>>>

Komentari:

- Način rada s parametrima i način vraćanja rezultata pokazan u ovom primjeru vrlo je sličan stvarnim potprogramima dobivenim prevodenjem viših programskih jezika u asembler.
- Razlog za ovakvu organizaciju je njena praktičnost i efikasnost te općenitost koja omogućuje korištenje rekurzivnih i nerekurzivnih potprograma uz čuvanje stanja svih registara.
- Podatci na stogu su uniformno organizirani za svaki potprogram i takav **niz podataka za jedan potprogram** naziva se **okvir stoga** ili kraće okvir (frame, stack frame, activation record). Svaki potprogram ima svoj okvir.



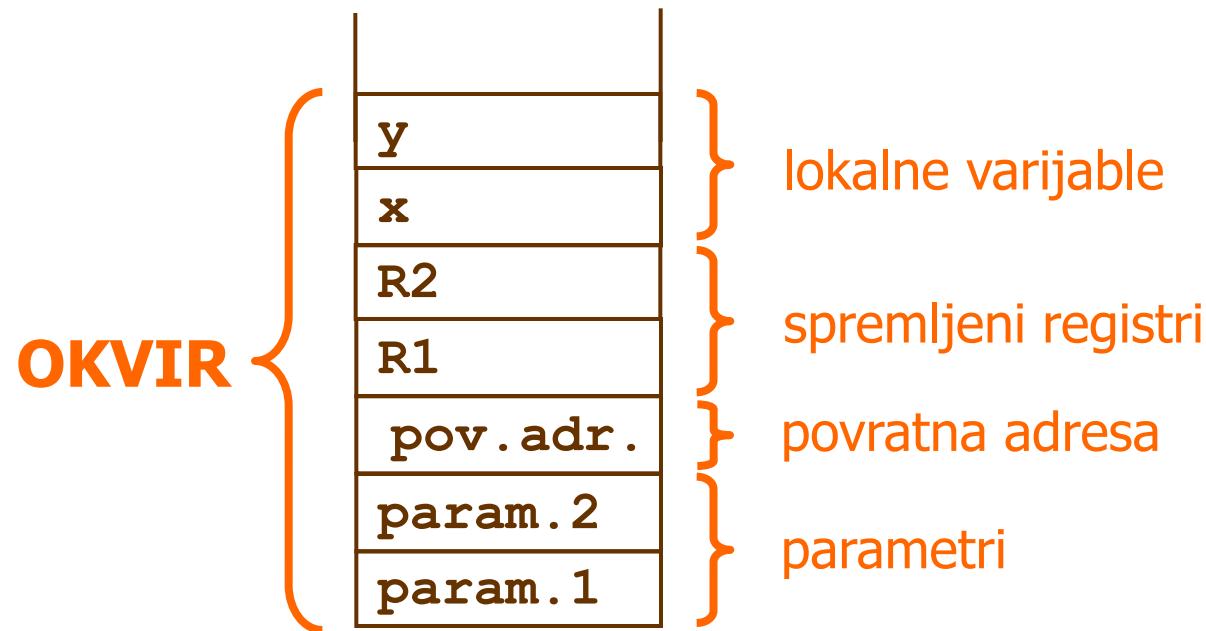
<<<

- Do sada nismo vidjeli kako u asembleru ostvariti lokalne varijable
- Lokalne varijable imaju sljedeća svojstva:
 - za svaki poziv potprograma postoje vlastite lokalne varijable
 - vidljive su samo unutar potprograma u kojem su definirane
 - stvaraju se prilikom poziva potprograma
 - nestaju prilikom povratka iz potprograma
- Vidimo da su lokalne varijable po svemu slične parametrima potprograma. Jedina je razlika u početnoj vrijednosti koja se za parametar definira od strane pozivatelja potprograma
- Dakle, prirodno je rješenje da se i lokalne varijable čuvaju na stogu, ili točnije u okviru stoga

>>>₄₃₁

<<<

- Potpunija verzija stogovnog okvira uključuje i lokalne varijable
- Lokalne varijable se stavljaju na stog nakon ulaska u potprogram, a mogu se staviti "ispod" ili "iznad" spremlijenih registara. Ovdje je odabранo da se stave "iznad" njih:

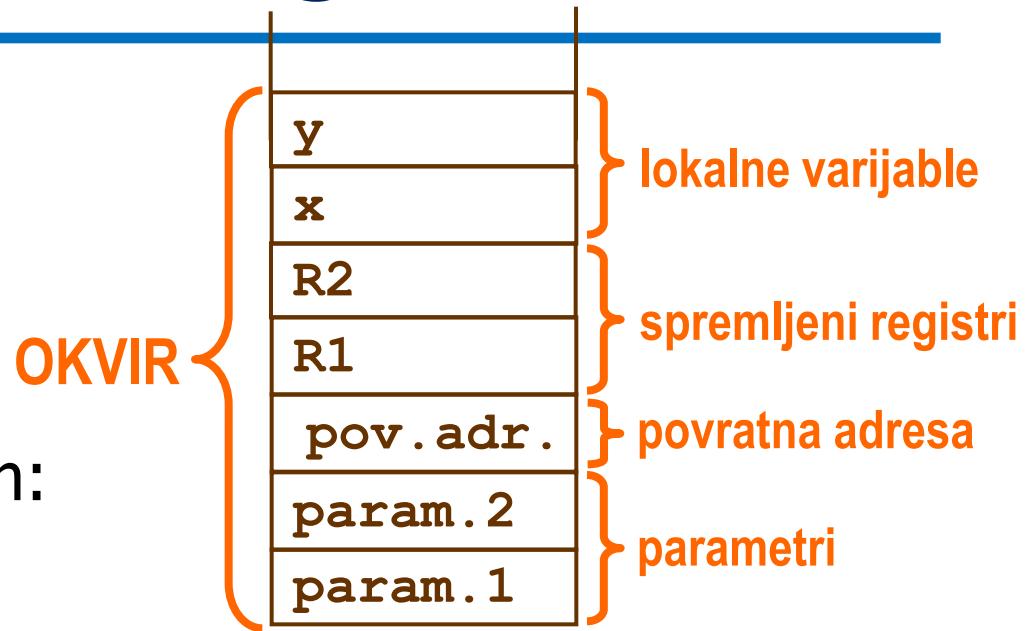


Važna napomena:

Prevoditelj u stvarnosti obično pokušava staviti lokalne varijable u registre, a tek kad se oni napune stavlja lokalne varijable na stog

Rekapitulacija okvira stoga

- Okvir stoga sadrži:
 - parametre
 - povratnu adresu
 - spremljene registre
 - lokalne varijable



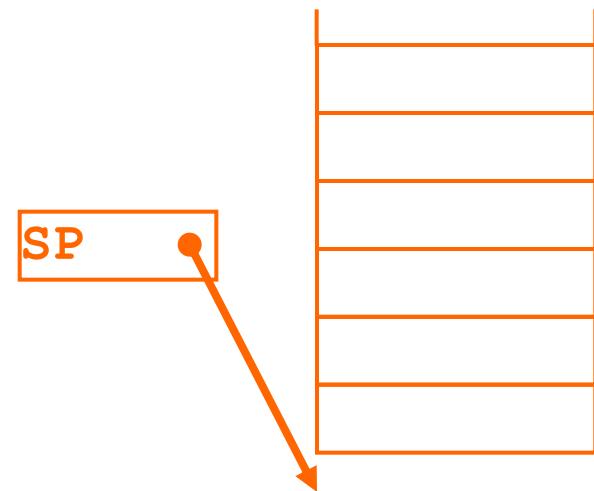
- Način rada s okvirom:
 - Glavni program:
 - stavља parametre
 - stavља povratnu adresu (CALL)
 - uklanja parametre
 - Potprogram:
 - sprema i obnavlja registre
 - stvara i uklanja lokalne varijable
 - uklanja povratnu adresu (RET)

Rekapitulacija rada s okvirom stoga:

Pozivatelj:

Potpogram:

Okvir:



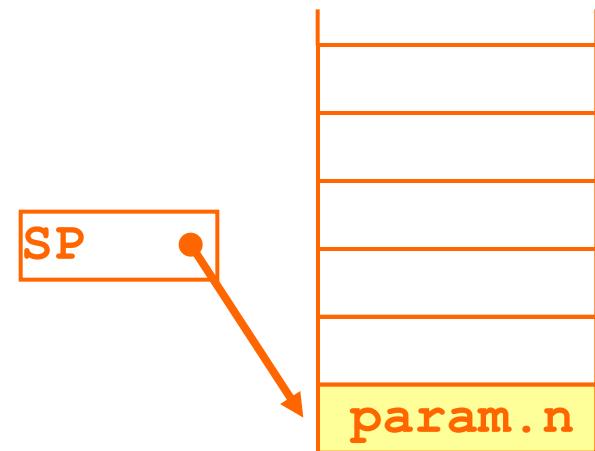
Rekapitulacija rada s okvirom stoga:

Pozivatelj:

Potprogram:

Okvir:

Stavi parametre na stog



Rekapitulacija rada s okvirom stoga:

Pozivatelj:

Potprogram:

Okvir:

Stavi parametre na stog



Pozovi potprogram



SP



Rekapitulacija rada s okvirom stoga:

Pozivatelj:

Potprogram:

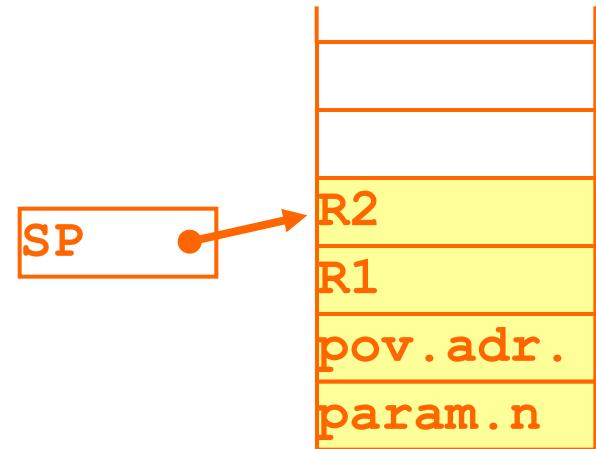
Okvir:

Stavi parametre na stog



Pozovi potprogram

Spremi registre



Rekapitulacija rada s okvirom stoga:

Pozivatelj:

Potprogram:

Okvir:

Stavi parametre na stog



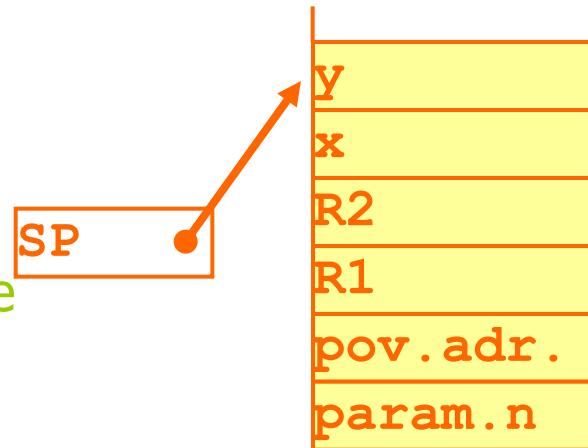
Pozovi potprogram



Spremi registre



Stvorи lokalne varijable



Rekapitulacija rada s okvirom stoga:

Pozivatelj:

Potprogram:

Okvir:

Stavi parametre na stog

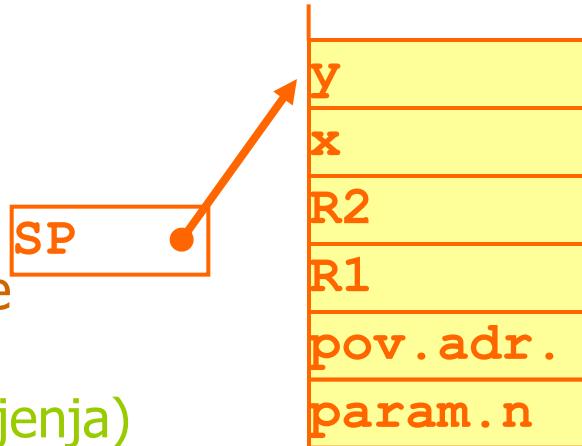


Pozovi potprogram

Spremi registre

Stvori lokalne varijable

Izvođenje (okvir se ne mijenja)
(ali se mogu mijenjati podatci u okviru,
npr. parametri i lokalne varijable)



Rekapitulacija rada s okvirom stoga:

Pozivatelj:

Potprogram:

Okvir:

Stavi parametre na stog



Pozovi potprogram



Spremi registre



Stvorи lokalne varijable



Izvođenje (okvir se ne mijenja)

Povratak: ukloni lokalne



Rekapitulacija rada s okvirom stoga:

Pozivatelj:

Potprogram:

Okvir:

Stavi parametre na stog



Pozovi potprogram

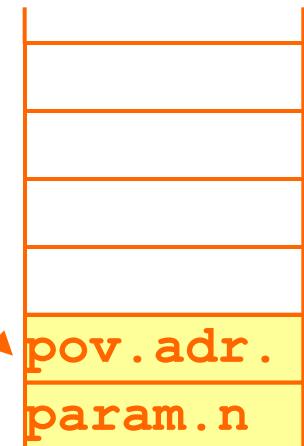
Spremi registre

Stvori lokalne varijable

Izvođenje (okvir se ne mijenja)

Povratak: ukloni lokalne
obnovi registre

SP



Rekapitulacija rada s okvirom stoga:

Pozivatelj:

Potprogram:

Okvir:

Stavi parametre na stog



Pozovi potprogram (CALL)

Spremi registre

Stvori lokalne varijable

SP



Izvođenje (okvir se ne mijenja)

Povratak: ukloni lokalne
obnovi registre
vrati se (RET)



Rekapitulacija rada s okvirom stoga:

Pozivatelj:

Potprogram:

Okvir:

Stavi parametre na stog



Pozovi potprogram (CALL)

Spremi registre

Stvori lokalne varijable

SP



Izvođenje (okvir se ne mijenja)

Povratak: ukloni lokalne
obnovi registre
vraći se (RET)



Ukloni parametre

Rekurzivni potprogrami

Potprogrami - Prijenos stogom



- Rekurzivni potprogrami su oni koji mogu pozvati sami sebe
 - (izravno, ili neizravno - preko drugih potprograma)
- Budući da imamo višestruki poziv istog potprograma, to je isto kao da je rekurzivni potprogram istovremeno "više puta aktiviran"
 - Zato se parametri ne mogu prenositi registrima i fiksnim lokacijama (npr. prvi poziv bi napunio vrijednosti u parametre, a već bi drugi poziv prepisao preko njih svoje vrijednosti, čime bi prve vrijednosti za prvi poziv bile izgubljene)
- Zato rekurzivni potprogrami koriste prijenos stogom i okvir stoga
- Uočite da povratna vrijednost nije problem i za njeno vraćanje se koristi prijenos registrom

Potprogrami - Rekurzija

Treba napisati rekurzivni potprogram za računanje faktorijela:

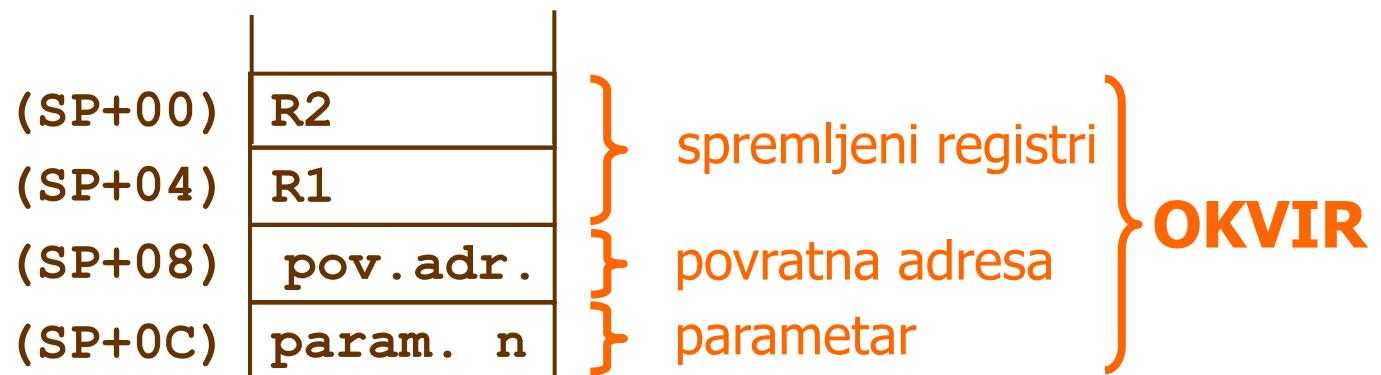
$$fakt(1) = 1$$

$$fakt(n) = fakt(n-1)*n \quad \text{za } n= 2, 3, 4, \dots$$

Parametar se prenosi stogom, a rezultat se vraća registrom R0.

Rješenje:

Pokažimo prvo kako će izgledati okvir stoga:



<<<

; potprogram fakt(n)

FAKT

PUSH R1
PUSH R2

} Spremi registre

LOAD R0, (SP+0C)
CMP R0, 1
JR_EQ VRATISE

} Ako je n jednak 1, onda vrati 1 u R0

SUB R0, 1, R0
PUSH R0
CALL FAKT
ADD SP, 4, SP

} Pozovi fakt(n-1) ovako:
- izračunaj n-1 i stavi ga na stog
- pozovi fakt(n-1), rezultat će biti u R0
- ukloni parametar (tj. n-1) sa stoga

LOAD R1, (SP+0C)
MOVE 0, R2

} Brojač R1=n; početni umnožak R2=0

ADD R0, R2, R2
SUB R1, 1, R1
JR_NZ MNOZI

} Množenje uzastopnim pribrajanjem:
R1 je brojač, umnošku R2 pribraja
se R0 u kojem je fakt(n-1)

MOVE R2, R0

} Umnožak iz R2 treba vratiti preko R0

MNOZI

POP R2
POP R1
RET

} Obnovi registre i vrati se

>>>

<<<

Pokažimo još samo kako bi mogao izgledati glavni program koji poziva fakt(3) i sprema rezultat na memoriju lokaciju REZULT.

```
GLAVNI    MOVE 10000, SP
           MOVE 3, R0
           PUSH R0
           CALL FAKT
           STORE R0, (REZULT)
           ADD   SP, 4, SP
           HALT
REZULT    DW    0
```



Potprogrami - Rekurzija

Treba napisati rekurzivni potprogram za računanje niza Fibonaccijevih brojeva:

$$Fib(1) = 1$$

$$Fib(2) = 1$$

$$Fib(n) = Fib(n-1) + Fib(n-2) \quad \text{za } n=3,4, \dots$$

Parametar se prenosi stogom, a rezultat se vraća registrom R0. Glavni program treba izračunati $Fib(8)$.

Rješenje:

- Radi lakšeg objašnjenja, pokažimo prvo kako bi rješenje moglo izgledati u programskom jeziku C

>>>

<<<

```
int fib ( int n ) {  
    int x, y;      // lokalne varijable za medurezultate  
    if ( n == 1 || n == 2 ) // Fib(1) i Fib(2) = 1  
        return ( 1 );  
    x = fib ( n-1 ); }  
    y = fib ( n-2 ); } // Fib(n) = Fib(n-1) + Fib(n-2)  
    return ( x+y ); }  
  
main () {  
    int rez;  
    rez = fib ( 8 );  
}
```

>>>

<<<

Rješenje u C-u se moglo napisati i nešto kraće:

```
int fib ( int n ) {  
    if ( n == 1 || n == 2 )  
        return ( 1 );  
  
    return ( fib ( n-1 ) + fib ( n-2 ) )  
}
```

- Ovdje nema lokalnih varijabli x i y. Međutim, treba voditi računa da rezultat od $\text{fib}(n-1)$ mora biti negdje pohranjen dok se izračunava $\text{fib}(n-2)$.
- Zato će C-prevodilac stvoriti asemblerski program koji je sličniji verziji s prethodnog slajda, iako je to za programera u C-u nevidljivo.

>>>

Mogući prijevod C-programa u asembler:

; glavni program

MAIN LOAD SP, 10000 ; inicijalizacija stoga

MOVE 8, R0 ; Stavi željenu vrijednost

PUSH R0 ; parametra (n=8) na stog.

CALL FIB ; poziv potprograma

ADD SP, 4, SP ; ukloni parametar sa stoga

STORE R0, (REZ) ; spremi rezultat iz R0

HALT

REZ DW 0 ; mjesto za rezultat

>>>

<<<

; potprogram FIB

FIB

```

    PUSH   R1      ; Spremanje
    PUSH   R2      ; registara na stog

    ; Stvaranje lokalnih varijabli x i y
    SUB    SP, 8, SP

    ; if( n==1 || n==2 ) return (1);

    MOVE   1, R0 ; Pripremi rez. za slučaj povratka

    LOAD   R1, (SP+14) ; dohvati parametar n
    CMP    R1, 1        ; je li n==1 ?
    JR_EQ VRATI_SE     ; Da: idi na dio za povratak

    CMP    R1, 2        ; Ne: ispitaj je li n==2 ?
    JR_EQ VRATI_SE     ; Da: idi na dio za povratak

```

>>>

<<< ; n nije ni 1 ni 2 - nastavak izvođenja

; x = fib(n-1)

LOAD R1, (SP+14) ; dohvati parametar n
SUB R1, 1, R1 ; Oduzmi n-1 i stavi na stog
PUSH R1 ; za prvi rekursivni poziv.

CALL FIB ; pozovi fib(n-1)
ADD SP, 4, SP ; ukloni n-1 sa stoga
STORE R0, (SP+4) ; spremi rezultat u x

; y = fib(n-2)

LOAD R1, (SP+14) ; dohvati parametar n
SUB R1, 2, R1 ; Oduzmi n-2 i stavi na stog
PUSH R1 ; za drugi rekursivni poziv.

CALL FIB ; pozovi fib(n-2)
ADD SP, 4, SP ; ukloni n-2 sa stoga
STORE R0, (SP+0) ; spremi rezultat u y

>>>

<<<

```
; return ( x + y )

LOAD  R1,  (SP+4)    ; dohvati x
LOAD  R2,  (SP+0)    ; dohvati y
ADD   R1,  R2,  R0    ; izračunaj povratnu vrijednost

VRATI_SE ADD    SP,  8,  SP    ; ukloni x i y sa stoga

POP   R2           ; Obnovi sadržaje
POP   R1           ; spremlijenih registara.

RET              ; povratak iz FIB
```

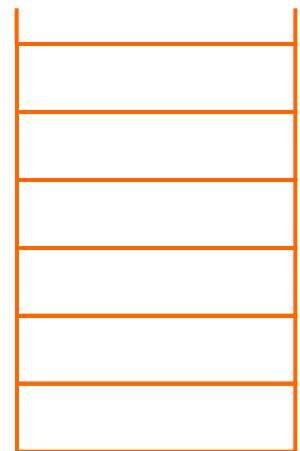
- Pokažimo ponašanje okvira na stogu ako je u glavnom programu pozvano Fib(4).
- Nećemo pratiti pojedine podatke u okvirima na stogu već promatramo pojedine **okvire kao cjeline**.



okviri na stogu

main

izvodi se main



okviri na stogu

main
main → fib(4)

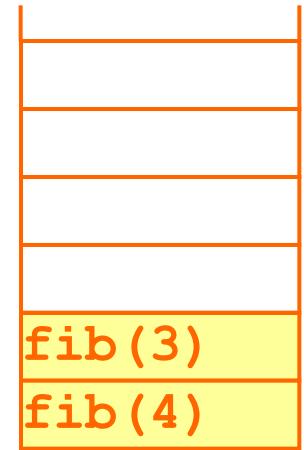
main poziva fib(4)



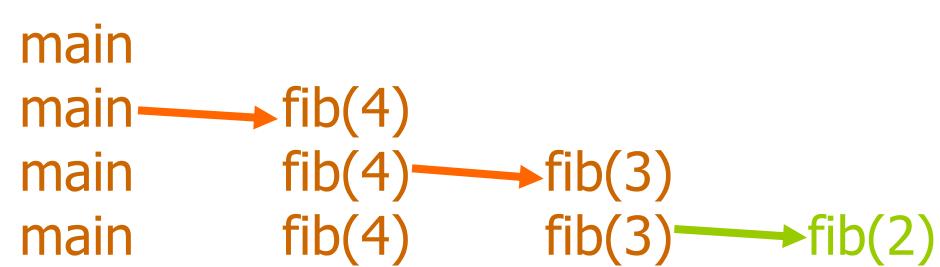
okviri na stogu

main
main → fib(4)
main fib(4) → fib(3)

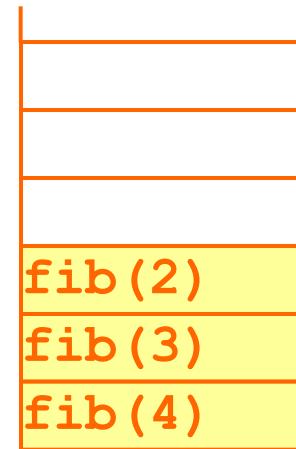
fib(4) poziva fib(3)



okviri na stogu



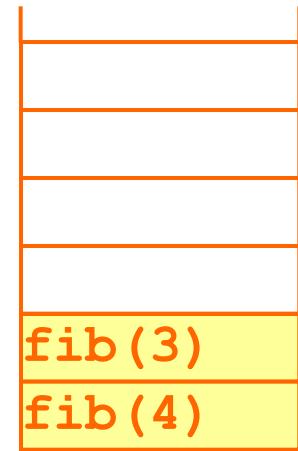
fib(3) poziva fib(2)



okviri na stogu

main
main → fib(4)
main fib(4) → fib(3)
main fib(4) fib(3) → fib(2)
main fib(4) fib(3) ← 1 fib(2)=1

fib(2) vraća 1

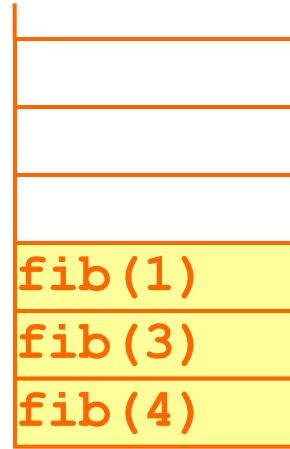


okviri na stogu

```
graph LR; main1[main] --> fib4_1[fib(4)]; fib4_1 --> fib3_1[fib(3)]; fib3_1 --> fib2_1[fib(2)]; fib2_1 --> fib2_val["fib(2)=1"]; fib3_1 --> fib3_2[fib(3)]; fib3_2 --> fib1_1[fib(1)];
```

The diagram illustrates the execution flow of a recursive Fibonacci function. It starts with a call from the `main` function to `fib(4)`. This leads to two parallel paths: one for `fib(4)` which further calls `fib(3)`, and another for `fib(3)` which further calls `fib(2)`. The `fib(2)` call is annotated with the value `1`, indicating its base case result. Finally, the `fib(3)` call leads to a call to `fib(1)`.

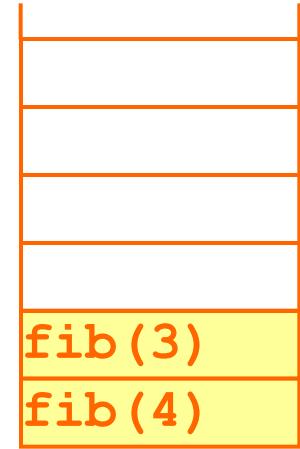
fib(3) poziva fib(1)



okviri na stogu

main
main → fib(4)
main fib(4) → fib(3)
main fib(4) fib(3) → fib(2)
main fib(4) fib(3) ← 1 fib(2)=1
main fib(4) fib(3) → fib(1)
main fib(4) fib(3) ← 1 fib(1)=1

fib(1) vraća 1



okviri na stogu

```

main
main → fib(4)
main   fib(4) → fib(3)
main   fib(4)   fib(3) → fib(2)
main   fib(4)   fib(3) ← 1 fib(2)=1
main   fib(4)   fib(3) → fib(1)
main   fib(4)   fib(3) ← 1 fib(1)=1
main   fib(4) ← 2 fib(3)=1+1

```

fib(3) vraća fib(2)+fib(1)



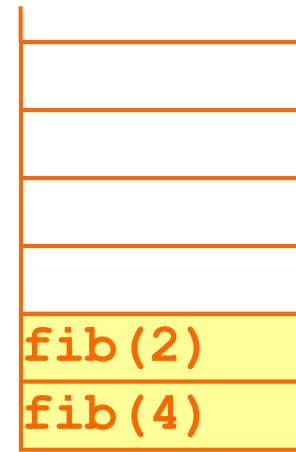
okviri na stogu

```

main
main → fib(4)
main   fib(4) → fib(3)
main   fib(4)   fib(3) → fib(2)
main   fib(4)   fib(3) ← 1 fib(2)=1
main   fib(4)   fib(3) → fib(1)
main   fib(4)   fib(3) ← 1 fib(1)=1
fib(4) ← 2 fib(3)=1+1
main   fib(4) → fib(2)

```

fib(4) poziva fib(2)



okviri na stogu

```

main
main → fib(4)
main   fib(4) → fib(3)
main   fib(4)   fib(3) → fib(2)
main   fib(4)   fib(3) ← 1 fib(2)=1
main   fib(4)   fib(3) → fib(1)
main   fib(4)   fib(3) ← 1 fib(1)=1
fib(4) ← 2 fib(3)=1+1
fib(4) → fib(2)
main   fib(4) ← 1 fib(2)=1

```

fib(2) vraća 1



okviri na stogu

```

main
main → fib(4)
main   fib(4) → fib(3)
main   fib(4)   fib(3) → fib(2)
main   fib(4)   fib(3) ← 1 fib(2)=1
main   fib(4)   fib(3) → fib(1)
main   fib(4)   fib(3) ← 1 fib(1)=1
main   fib(4) ← 2 fib(3)=1+1
main   fib(4) → fib(2)
main   fib(4) ← 1 fib(2)=1
main ← 3 fib(4)=2+1
main

```

fib(4) vraća fib(3)+fib(2)



okviri na stogu

```

main
main → fib(4)
main   fib(4) → fib(3)
main   fib(4)   fib(3) → fib(2)
main   fib(4)   fib(3) ← 1 fib(2)=1
main   fib(4)   fib(3) → fib(1)
main   fib(4)   fib(3) ← 1 fib(1)=1
main   fib(4) ← 2 fib(3)=1+1
main   fib(4) → fib(2)
main   fib(4) ← 1 fib(2)=1
main ← 3 fib(4)=2+1
main

```

nastavlja se izvoditi main



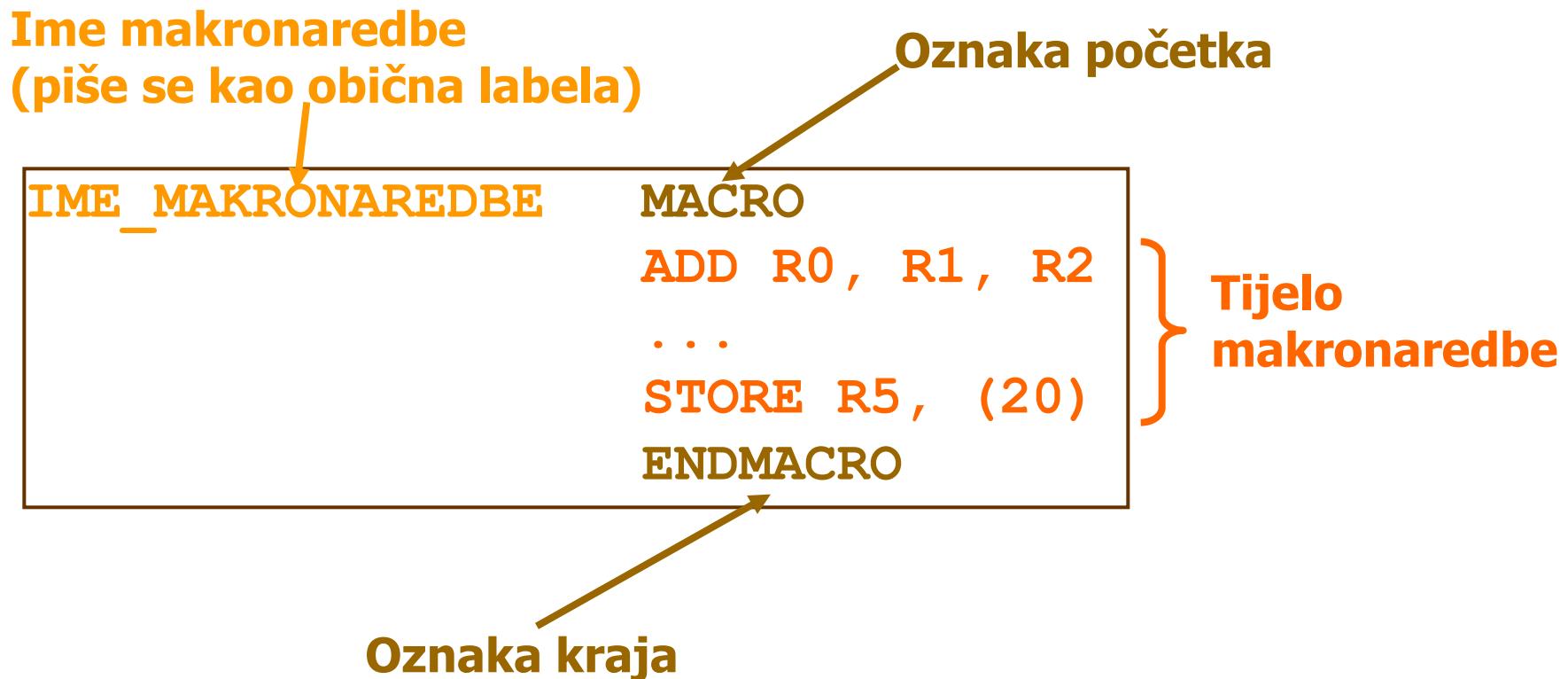
okviri na stogu

Makronaredbe

Makronaredbe

- Makronaredbe imaju istu namjenu kao i potprogrami
- Razlika je u načinu njihove izvedbe
- Potprogrami su podržani od strane procesora, pomoću naredaba CALL i RET kojima se obavljuju poziv i povratak iz potprograma
- Makronaredbe su podržane od strane asemblerskog prevoditelja koji ih prevodi u obične naredbe procesora
 - Napomena: u ATLAS-u nije moguće koristiti makronaredbe

Makronaredbe - definicija



Makronaredbe - Primjer

Napisati makronaredbu koja izračunava logičku operaciju NILI između sadržaja registara R0 i R1 i vraća rezultat u registru R2. Napisati i glavni program koji će pozvati makronaredbu za dva podatka iz memorije i rezultat također upisati u memoriju.

Rješenje:

; DEFINICIJA MAKRONAREDBE

```
NILI      MACRO
          OR     R0 , R1 , R2
          XOR    R2 , -1 , R2
ENDMACRO
```

>>>

```
; ; ; ; ; ; GLAVNI PROGRAM  
  
LOAD    R0 , (PRVI)   ; DOHVATI PODATKE  
LOAD    R1 , (DRUGI)   ; U R0 i R1  
  
NILI          ; POZOVI MAKRONAREDBU  
  
STORE   R2 , (REZ)    ; SPREMI REZULTAT  
HALT
```

```
; ; ; ; ; ; PODATCI I MJESTO ZA REZULTAT  
PRVI   DW  81282C34  
DRUGI   DW  29A82855  
REZ    DW  0
```

Napomena: Poziv se ostvaruje navođenjem imena makronaredbe. Poziv **nije naredba procesora**, već je sličniji pseudonaredbi asemblerskog prevoditelja.

Makronaredbe - Način prevodenja

- Asembleri koji podržavaju makronaredbe moraju biti troprolazni ili četveroprolazni i nazivaju se makroasemblerima
- U troprolaznom asembleru se svaka definicija makronaredbe mora nalaziti **ISPRED** njenog pozivanja
- U četveroprolaznom asembleru se definicija makronaredbe smije nalaziti **IZA** njenog pozivanja
- Objasnimo kako rade ove dvije vrste asemblera . . .

Makronaredbe - Troprolazni asembler

Prvi prolaz:

- Asembler čita redak po redak datoteke:
 - Ako nađe na definiciju makronaredbe, onda u tablici makronaredbi zapamti njeni ime i tijelo (slično kao što dvoprolazni asembler nailaskom na labelu u tablici labela pamti njeno ime i vrijednost)
 - Ako nađe na poziv makronaredbe, zamjenjuje ga njenim tijelom koje je zapamćeno u tablici (ovo je uvijek moguće napraviti zbog obaveznog redoslijeda u kojem definicija uvijek prethodi pozivu makronaredbe)
 - Prevoditelj ne mijenja ostale retke mnemoničkog programa
- Na kraju prvog prolaska, mnemonička datoteka više nema ni definicija makronaredaba ni poziva makronaredaba, već sadrži samo obične naredbe procesora i pseudonaredbe

Makronaredbe - Troprolazni asembler

<<<

Drugi i treći prolaz:

- Budući da je rezultat prvog prolaska obični mnemonički program, drugi i treći prolaz u potpunosti odgovaraju radu običnog dvoprolaznog asemblera:
 - Drugi prolaz: ekvivalentan prvom prolasku dvoprolaznog asemblera
 - Treći prolaz: ekvivalentan drugom prolasku dvoprolaznog asemblera

>>>

Makronaredbe - Tro prolazni asembler

<<< Kako izgleda prevodenje prethodnog primjera?



```
NILI MACRO  
    OR R0, R1, R2  
    XOR R2, -1, R2  
ENDMACRO
```

```
GLAVNI LOAD R0, (PRVI)  
    LOAD R1, (DRUGI)  
NILI  
STORE R2, (REZ)  
HALT
```

originalni program s
makronaredbama i
njihovim pozivima

```
GLAVNI LOAD R0, (PRVI)  
    LOAD R1, (DRUGI)
```

```
OR R0, R1, R2  
XOR R2, -1, R2
```

```
STORE R2, (REZ)  
HALT
```

rezultat prvog prolaza:
obični mnemonički
program

Makronaredbe - Četveroprolazni asembler

Četveroprolazni asembler dozvoljava da poziv makronaredbe prethodi njenoj definiciji. Zato je potreban dodatni prolaz (kao što je u simboličkom dvoprolaznom asembleru potreban dodatni prolaz zbog labela koje se koriste prije nego što su definirane)

Prvi prolaz:

- Asembler čita redak po redak datoteke:
 - Ako nađe na definciju makronaredbe, onda u tablici makronaredbi zapamti njeni ime i tijelo (slično kao što dvoprolazni asembler nailaskom na labelu u tablici labela pamti njeni ime i vrijednost)
 - Prevoditelj ne mijenja ostale retke mnemoničkog programa
- Na kraju prvog prolaska, mnemonička datoteka više nema definicija makronaredaba, ali sadrži pozive makronaredaba.

Makronaredbe - Četveroprolazni asembler

<<<

Drugi prolaz:

- Drugi prolaz zamjenjuje sve pozive makronaredba njihovim tijelima što daje obični mnemonički program

Treći i četvrti prolaz:

- Treći i četvrti prolaz u potpunosti odgovaraju prvom i drugom prolazu običnog dvoprolaznog asemblera (odnosno odgovaraju radu drugog i trećeg prolaza troprolaznog asemblera)

>>>

Makronaredbe - Parametri

- Jedna od mogućnosti koju makroasembleri obično nude je i korištenje **parametara makronaredaba**
- Parametri se obično navode kao lista imena iza *MACRO*
- Unutar tijela se parametri koriste na bilo kojim mjestima u naredbama
- Kod poziva makronaredbe programer zadaje argumente, tj. navodi npr. registre, adrese i sl. Ovi argumenti zamjenjuju parametre prilikom proširivanja makronaredbe.
- Ovisno o mjestima korištenja parametara, programeru je pri pozivanju ograničena njihova upotreba

Makronaredbe - Parametri

Primjer:

Napišite makronaredbu koja će izračunati logičku operaciju NILI. Ulazni podatci i rezultat mogu biti smješteni na bilo kojim memorijskim lokacijama koje se zadaju kao parametri makronaredbe.

Napišite glavni program koji će izračunati NILI između podataka s lokacija PRVI i DRUGI, a rezultat će spremiti na REZULT.

Rješenje:

na sljedećem slajdu

>>>

<<<

; DEFINICIJA MAKRONAREDBE

NILI MACRO P1, P2, REZ
LOAD R0, (P1)
LOAD R1, (P2)
OR R0, R1, R2
XOR R2, -1, R2
STORE R2, (REZ)
ENDMACRO

; GLAVNI PROGRAM

NILI PRVI, DRUGI, REZULTAT ;;;; POZIV
HALT

; PODATCI I MJESTO ZA REZULTAT

PRVI DW 81282C34
DRUGI DW 29A82855
REZULT DW 0

>>>
482

<<<

Nakon prvog prolaza prevođenja, glavni program izgleda ovako:

```
LOAD  R0 ,  (PRVI)
LOAD  R1 ,  (DRUGI)
OR    R0 ,  R1 ,  R2
XOR   R2 ,  -1 ,  R2
STORE R2 ,  (REZULT)
```



NILI PRVI , DRUGI , REZULT

HALT

```
PRVI     DW    81282C34
DRUGI    DW    29A82855
REZULT   DW    0
```

>>>
483

Usporedba makronaredaba i potprograma

- Makronaredbe općenito troše više memorije jer njihovo tijelo u memoriji postoji u više primjeraka (onoliko koliko ima poziva). Potprogrami se u memoriji nalaze samo u jednom primjerku.
- Potprogrami su sporiji jer se troši vrijeme na njihovo pozivanje i povratak, a također dio vremena troši i prijenos parametara i povratne vrijednosti. Makronaredbe se ne pozivaju nego se jednostavno izvode na mjestu na kojem je to potrebno.
- Što odabrat? Ovisno o tome što je kritično u pojedinom dijelu programa - brzina ili zauzeće memorije.



Napomena: Mjesto upotrebe parametara u tijelu makronaredbe ograničava argumente koje možemo slati prilikom poziva. U ovom primjeru kao argumente možemo navoditi samo adrese zadane absolutnim ili simboličkim adresiranjem.

Dodatno se smije (iako to nije bila namjera pri pisanju makronaredbe) kao argument poslati i registar opće namjene, jer se u naredbama STORE i LOAD u zagradama smije koristiti i indirektno registarsko adresiranje s odmakom (odmak će u ovom slučaju biti 0)

```
NILI    MACRO P1 , P2 , REZ
        LOAD  R0 , (P1)
        LOAD  R1 , (P2)
        OR    R0 , R1 , R2
        XOR  R2 , -1 , R2
        STORE R2 , (REZ)
ENDMACRO
```

osim adrese
smije se napisati
i registar



<<<

Napomena: Uočite da ovako definirana makronaredba mijenja sadržaje registara R0, R1 i R2 što nije poželjno (kao što nije bilo ni kod potprograma).

Registri se slično potprogramima mogu spremati na stog (ali i na fiksne lokacije jer se makronaredbe ne mogu pozivati rekurzivno*)

```
NILI    MACRO P1 , P2 , REZ
        LOAD  R0 , (P1)   ← mijenja R0
        LOAD  R1 , (P2)   ← mijenja R1
        OR    R0 , R1 , R2 ← mijenja R2
        XOR  R2 , -1 , R2 ← mijenja R2
        STORE R2 , (REZ)
ENDMACRO                                >>>
```

* Neki makroasembleri dozvoljavaju rekurzivno i uvjetno pozivanje makronaredaba, ali to prelazi opseg gradiva na ovom predmetu



<<<

Budući da se ovoj makronaredbi smiju poslati registri kao argumenti, što bi se dogodilo da se pozove ovako:

NILI PRVI , DRUGI , R0

Zašto makronaredba ne bi radila ispravno?

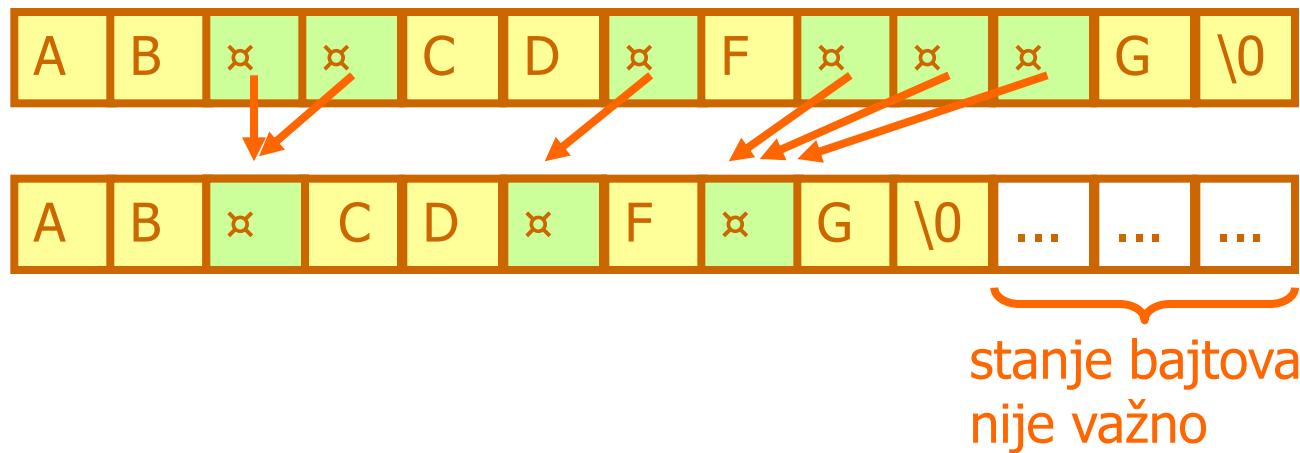
```
NILI    MACRO P1 , P2 , REZ
        LOAD  R0 , (P1)
        LOAD  R1 , (P2)
        OR    R0 , R1 , R2
        XOR  R2 , -1 , R2
        STORE R2 , (REZ)
ENDMACRO
```

Ostali primjeri

Ostali primjeri

Primjer:

Napisati potprogram SAZMI kojemu se preko R1 predaje adresa znakovnog niza u memoriji. Znakovi su zapisani ASCII kodom u bajtovima. Potprogram treba u znakovnom nizu sva uzastopna pojavljivanja razmaka sažeti u jednostrukе razmake. Niz je zaključen ASCII-znakom NUL.



- "Slova" (žuti) se kopiraju
- Razmaci označeni sa ✕ (zeleni) se sažimaju

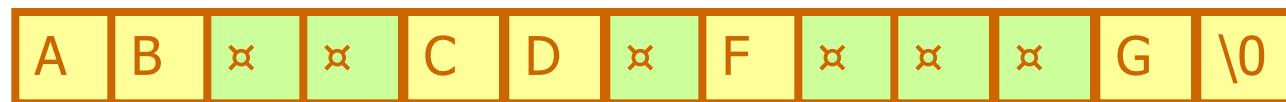
>>>

Ostali primjeri

Rješenje:

Upotrijebiti dva pokazivača:

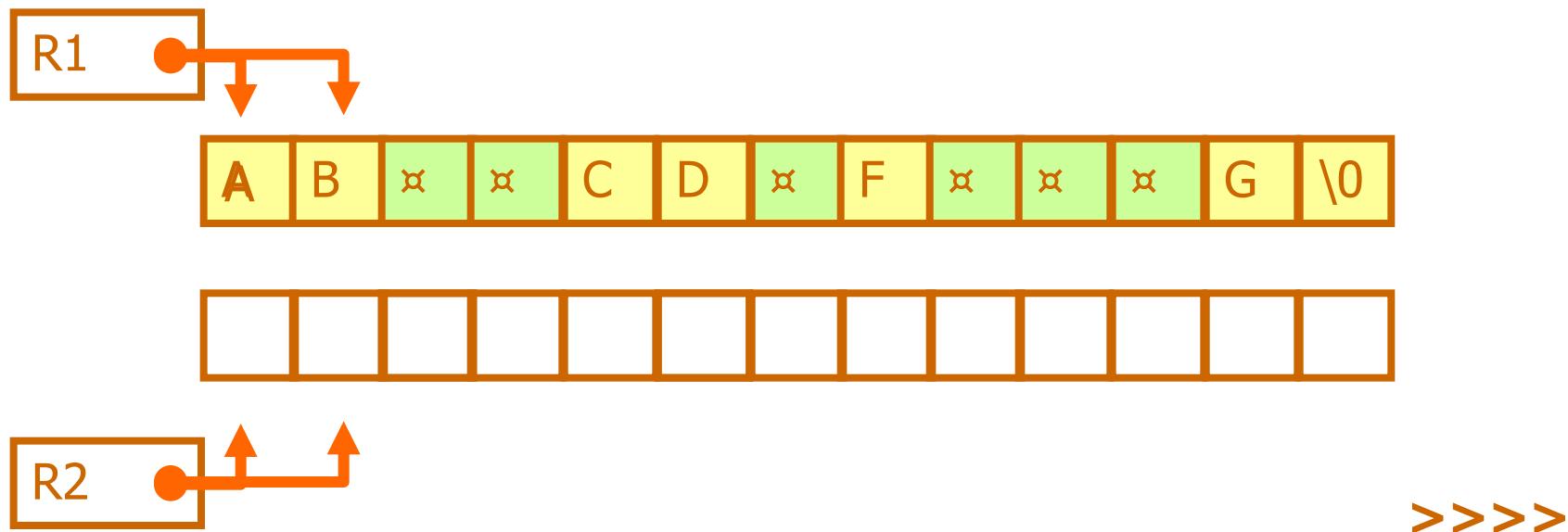
- jedan čita znakove izvornog niza (R1)
- drugi pokazuje mjesto na koji se kopira znak izvornog niza (R2)



>>>

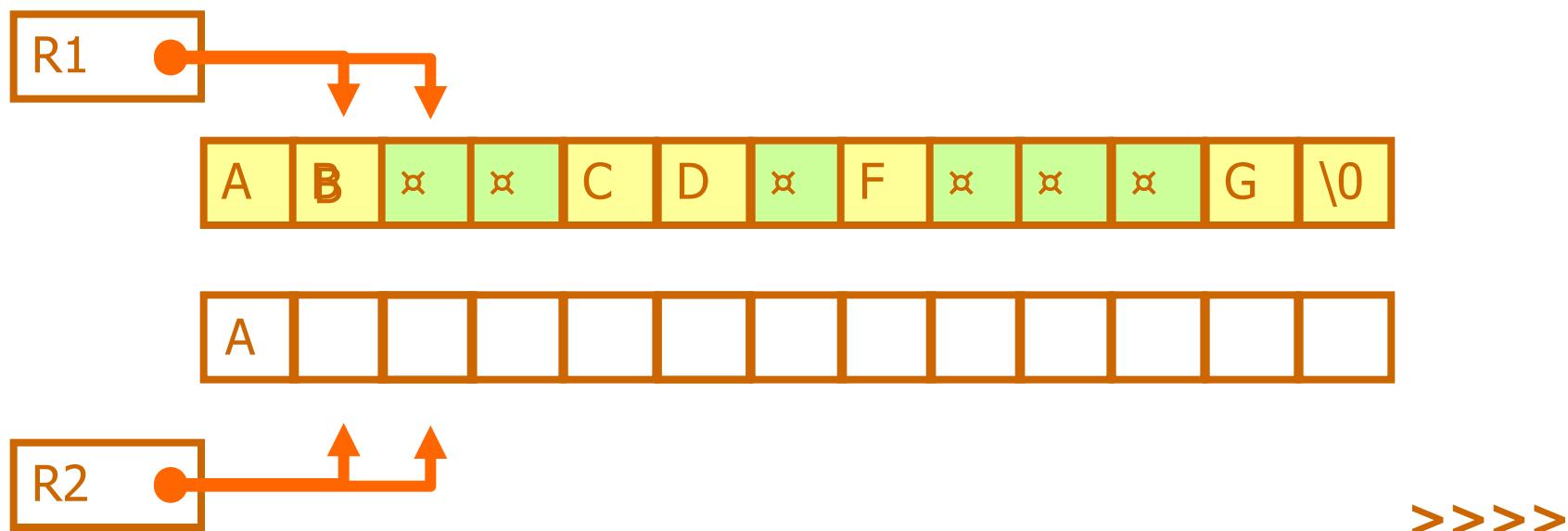
Ostali primjeri

- Ako je "ispod" R1 slovo, onda
 - slovo se kopira tamo gdje pokazuje R2
 - R1 i R2 se pomiču za jedno mjesto



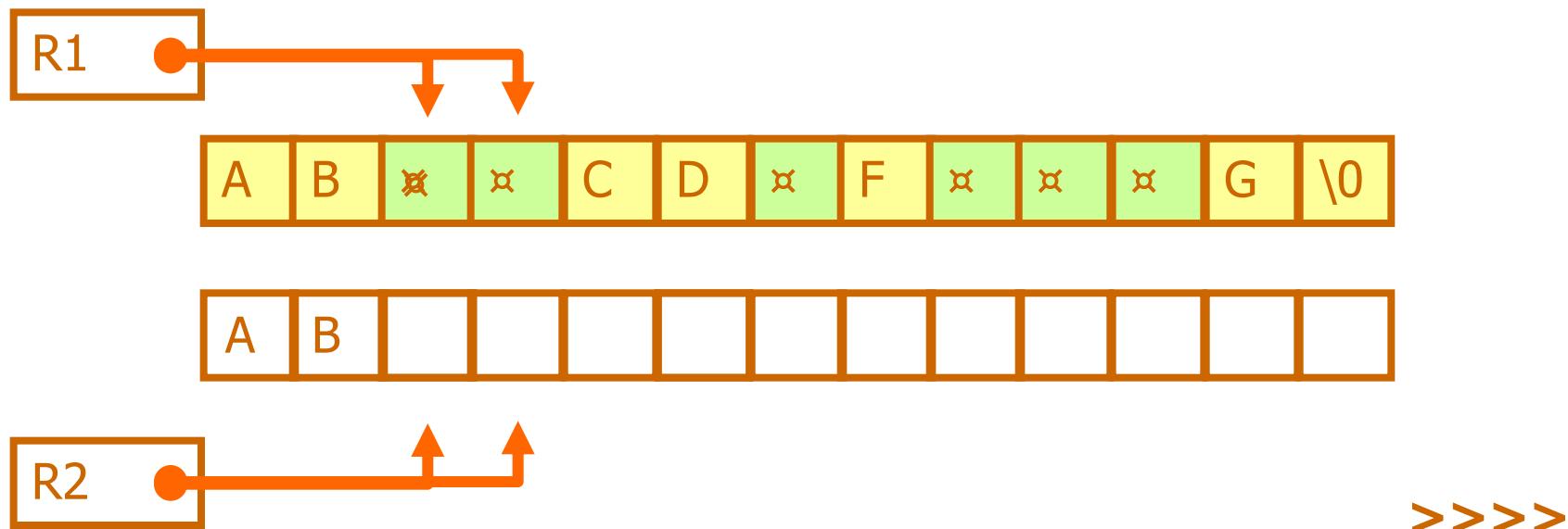
Ostali primjeri

- Ako je "ispod" R1 slovo, onda
 - slovo se kopira tamo gdje pokazuje R2
 - R1 i R2 se pomiču za jedno mjesto



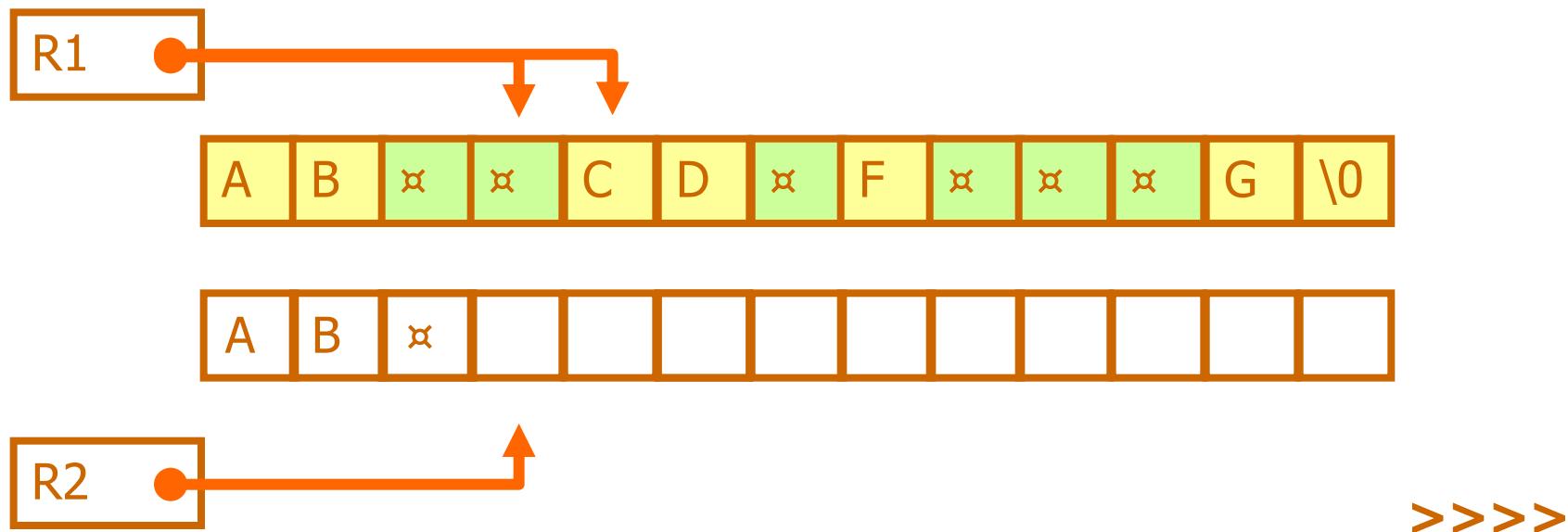
Ostali primjeri

- Ako je "ispod" R1 razmak, onda
 - ako je prethodilo slovo, onda
 - trenutačni razmak se kopira
 - R1 i R2 se pomiču za jedno mjesto



Ostali primjeri

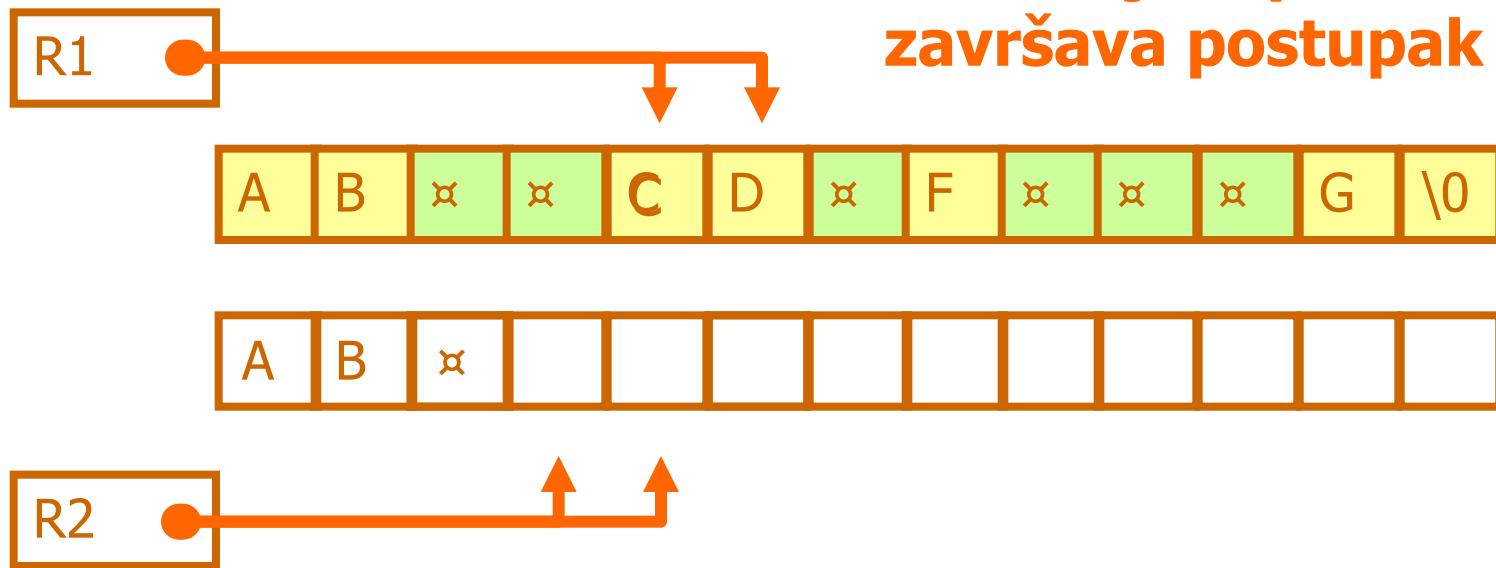
- Ako je "ispod" R1 razmak, onda
 - ako je prethodio razmak, onda
 - trenutačni razmak se zanemaruje
 - R1 se pomiče za jedno mjesto



Ostali primjeri

- Ako je "ispod" R1 slovo, onda
 - trenutačno slovo se kopira
 - R1 i R2 se pomiču za jedno mjesto

itd... do znaka NUL koji se zadnji kopira čime završava postupak



>>>>>

Ostali primjeri

- U registru R0 pamtit ćeмо prethodno stanje, tj. koji je bio prethodni znak:
 - SLO će značiti slovo
 - RAZ će značiti razmak
- Prilikom prelaska iz slova u razmak, mijenjat ćemo stanje u RAZ
- Prilikom prelaska iz razmaka u slovo, mijenjat ćemo stanje u SLO
- Početno stanje bit će SLO
- Registrar R3 služit će samo za učitavanje trenutačnog znaka i njegovo ispitivanje

>>>

<<<

; parametar R1 = adresa niza

SAZMI	PUSH	R0	; Spremi registre
	PUSH	R1	
	PUSH	R2	
	PUSH	R3	
	MOVE	SLO, R0	; Početno: stanje = slovo
	MOVE	R1, R2	; Inicijalizacija R2
PETLJA	LOADB	R3, (R1)	; Učitaj znak ispod R1
	CMP	R3, 0	; Ako je NUL, onda
	JR_Z	KRAJ	; idi na kraj.
	; JE LI TRENUTAČNI ZNAK RAZMAK ILI NIJE		
	CMP	R3, 20	; 20 = ASCII razmak
	JR_EQ	RAZMAK	

>>>
497

<<<

; TRENUTAČNI ZNAK JE SLOVO

; JE LI PRETHODNI BIO RAZMAK ILI NIJE

SLOVO CMP R0 , RAZ

 JR_EQ RAZMAK_PA_SLOVO

SLOVO_PA_SLOVO

STOREB R3 , (R2) ; Kopiraj znak

ADD R1 , 1 , R1 ; Pomakni R1 i R2

ADD R2 , 1 , R2

JR PETLJA

RAZMAK_PA_SLOVO

MOVE SLO , R0 ; stanje = slovo

STOREB R3 , (R2) ; Kopiraj znak

ADD R1 , 1 , R1 ; Pomakni R1 i R2

ADD R2 , 1 , R2

JR PETLJA

>>>
498

<<<

; TRENUTAČNI ZNAK JE RAZMAK

; JE LI PRETHODNI BIO RAZMAK ILI NIJE

RAZMAK CMP R0 , RAZ

 JR_EQ RAZMAK_PA_RAZMAK

SLOVO_PA_RAZMAK

MOVE RAZ , R0 ; stanje = razmak

STOREB R3 , (R2) ; Kopiraj znak

ADD R1 , 1 , R1 ; Pomakni R1 i R2

ADD R2 , 1 , R2

JR PETLJA

RAZMAK_PA_RAZMAK

ADD R1 , 1 , R1 ; Pomakni R1

JR PETLJA

>>>

<<<

KRAJ STOREB R3, (R2) ; Kopiraj NUL-znak
 POP R3 ; Obnovi registre
 POP R2
 POP R1
 POP R0

 RET

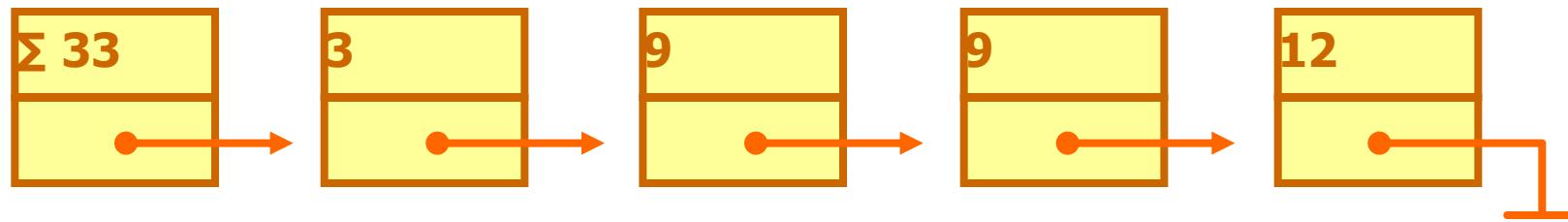
; definiranje "konstanti"

SLO EQU 0
RAZ EQU 1

Ostali primjeri

Primjer:

Jednostruko povezana lista ima čvorove koji u memoriji zauzimaju dvije 32-bitne riječi: prva riječ sadrži NBC-broj (koji predstavlja vrijednost čvora), a druga riječ je pokazivač na sljedeći čvor liste (tj. sadrži adresu sljedećeg čvora). Čvorovi su sortirani prema svojoj vrijednosti.



Prvi čvor liste ima specijalno značenje i u njemu se pamti zbroj svih vrijednosti preostalih čvorova liste. Prvi čvor je uvijek prisutan, bez obzira postoje li ostali čvorovi. Pretpostavka je da u zbroju nikada neće doći do prekoračenja opsega.

Zadnji čvor u listi prepoznaje se po NULL-pokazivaču (tj. lokacija s pokazivačem sljedećeg čvora sadrži nulu).

Ostali primjeri

<<<

Treba napisati potprogram UBACI koji ubacuje novi čvor u postojeću sortiranu listu tako da ona ostane sortirana.

Parametri potprograma su adresa prvog elementa liste i adresa novog čvora. Parametri se šalju preko stoga.

Povratna vrijednost je novi zbroj iz prvoga čvora liste. Vrijednost se vraća pomoću R0.

Rješenje:

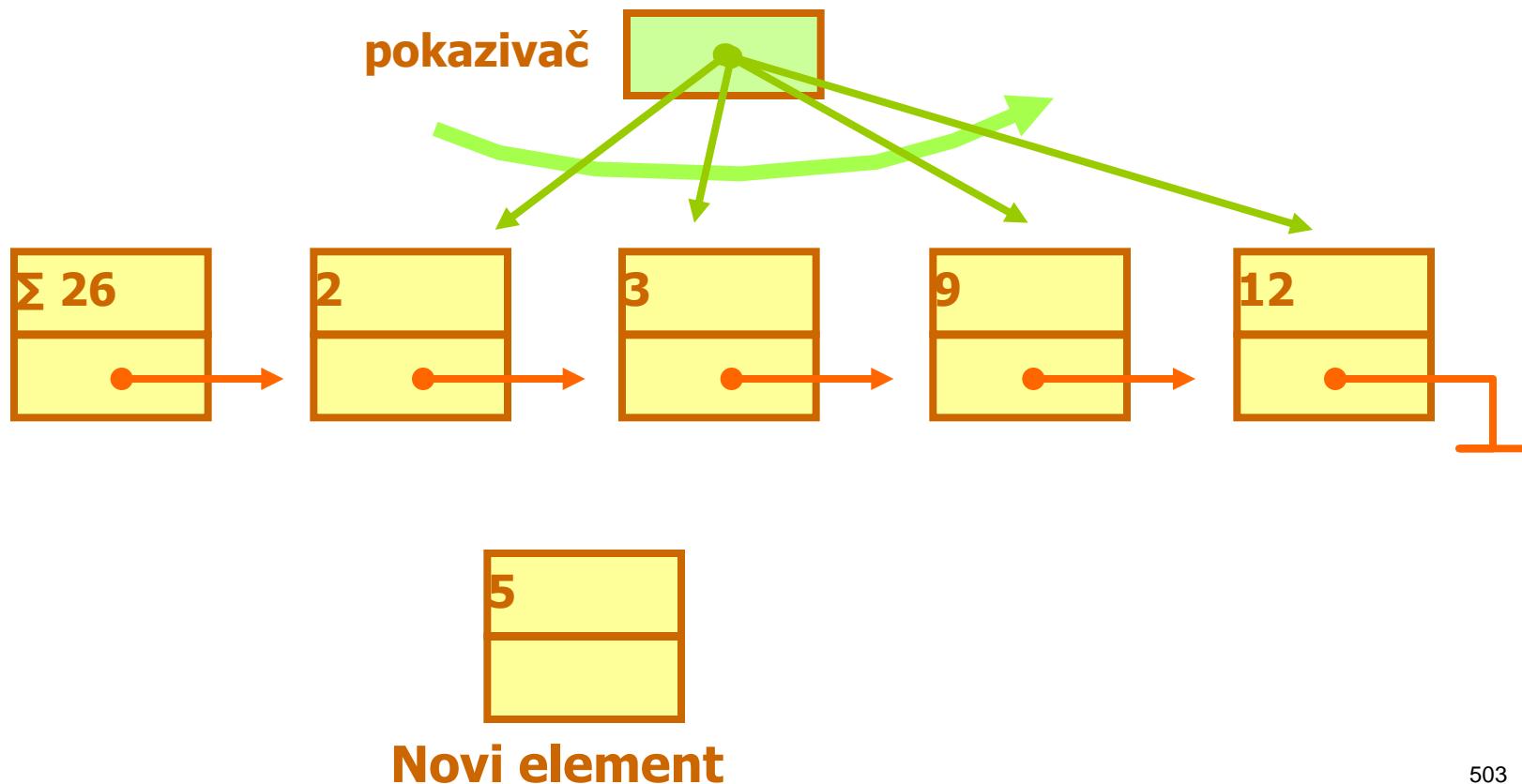
Potprogram će čuvati vrijednosti registara, a parametre će sa stoga uklanjati glavni program.

>>>

Ostali primjeri

<<<< Idejno rješenje:

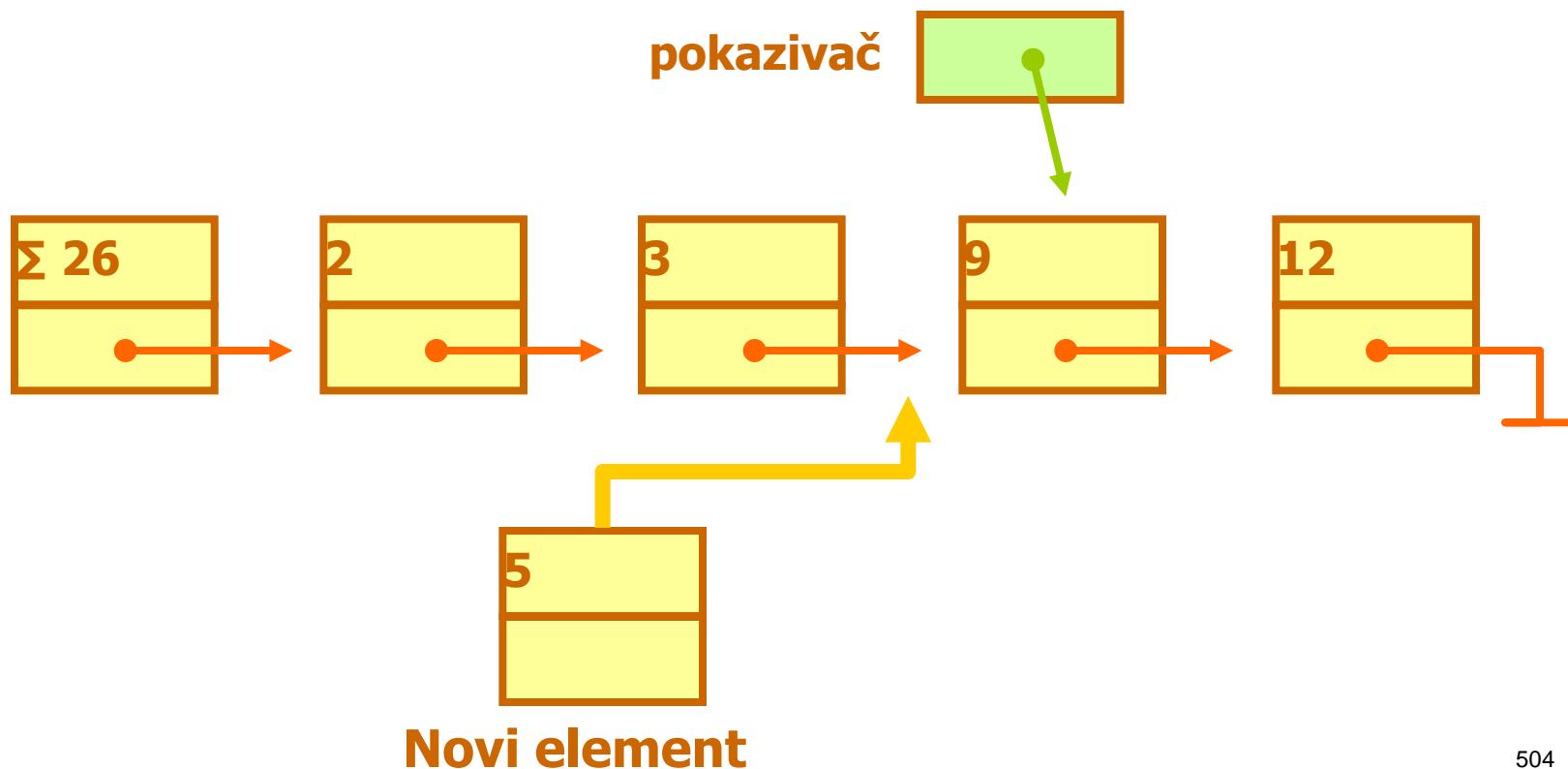
- Lista se pretražuje dok se ne nađe mjesto za ubacivanje.
- Jednim pokazivačem krećemo se po čvorovima koje ispitujemo.



Ostali primjeri

<<< Idejno rješenje:

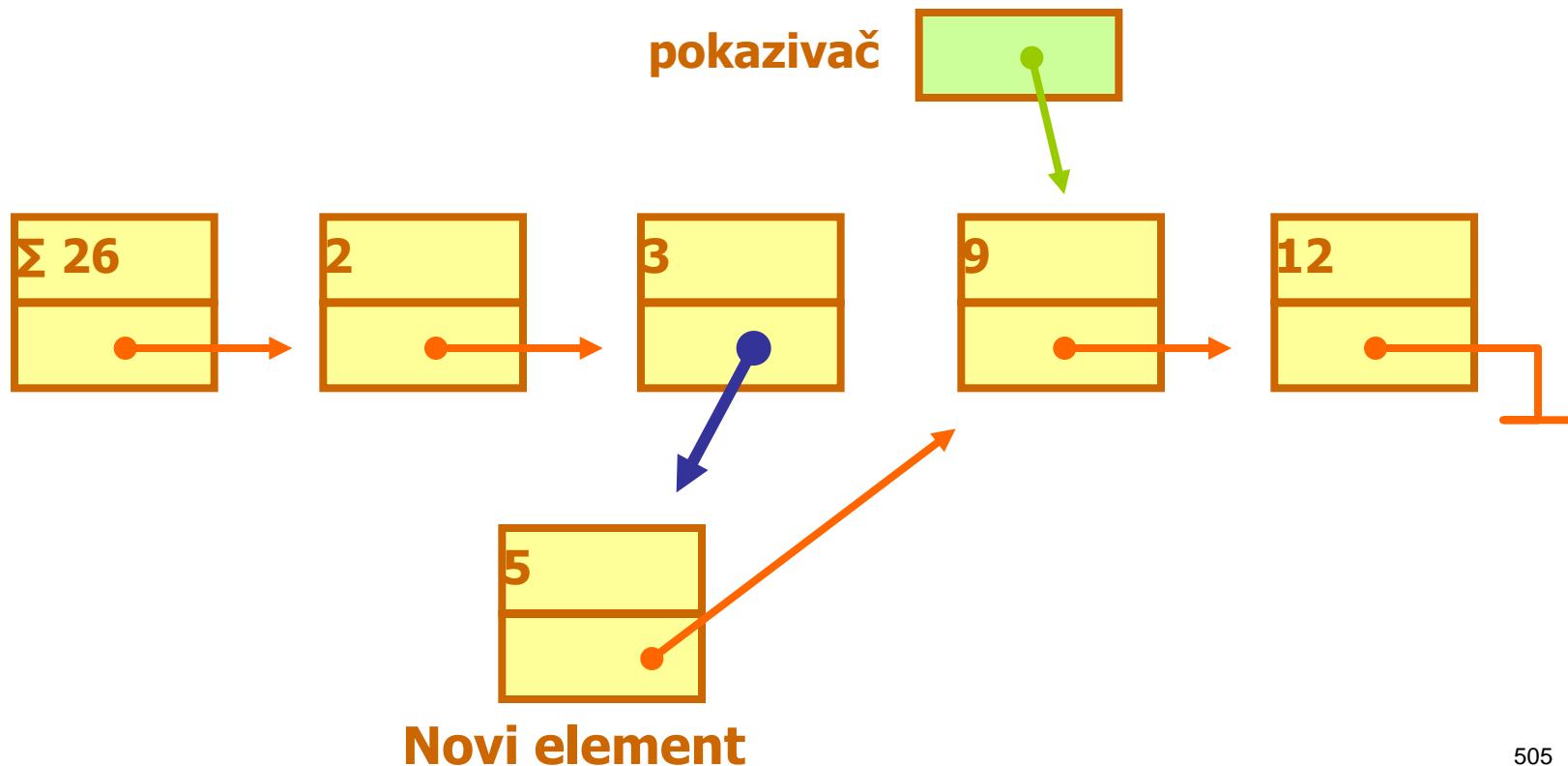
- Mjesto za ubacivanje je ISPRED prvog čvora koji ima veću ili jednaku vrijednost novom čvoru.



Ostali primjeri

<<< Idejno rješenje:

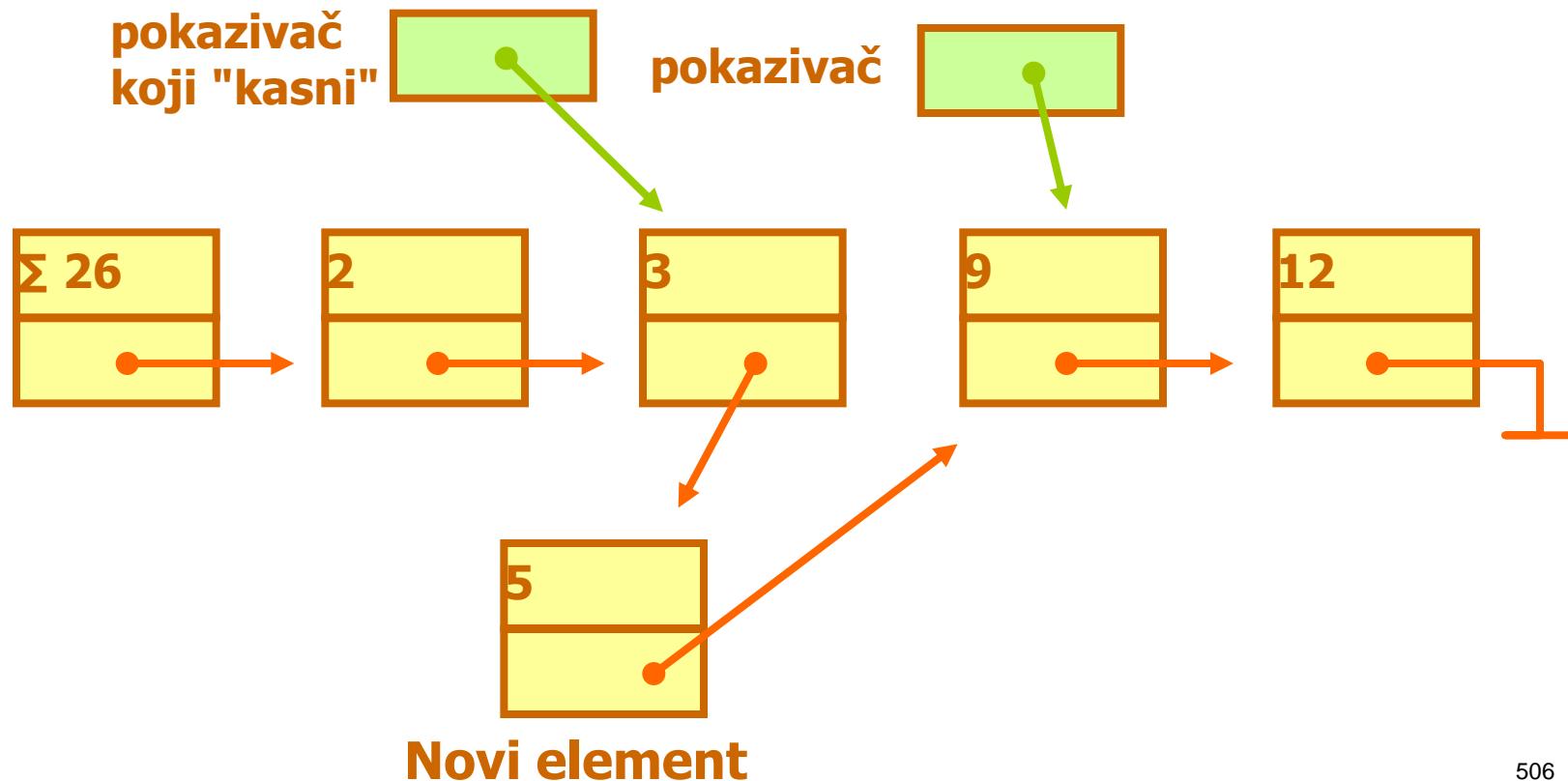
- Budući da se čvor mora ubaciti ISPRED čvora na kojem pokazuje pokazivač, onda ne možemo dohvatiti prethodni čvor u kojem treba promijeniti pokazivač na sljedeći (plava strelica)



Ostali primjeri

<<< Idejno rješenje:

- Zato trebamo još jedan pokazivač koji će uvijek "kasniti" za jedan čvor u odnosu na pokazivač ispitivanog čvora, tj. pokazivač će pokazivati jedan čvor ispred onog čvora kojeg ispitujemo

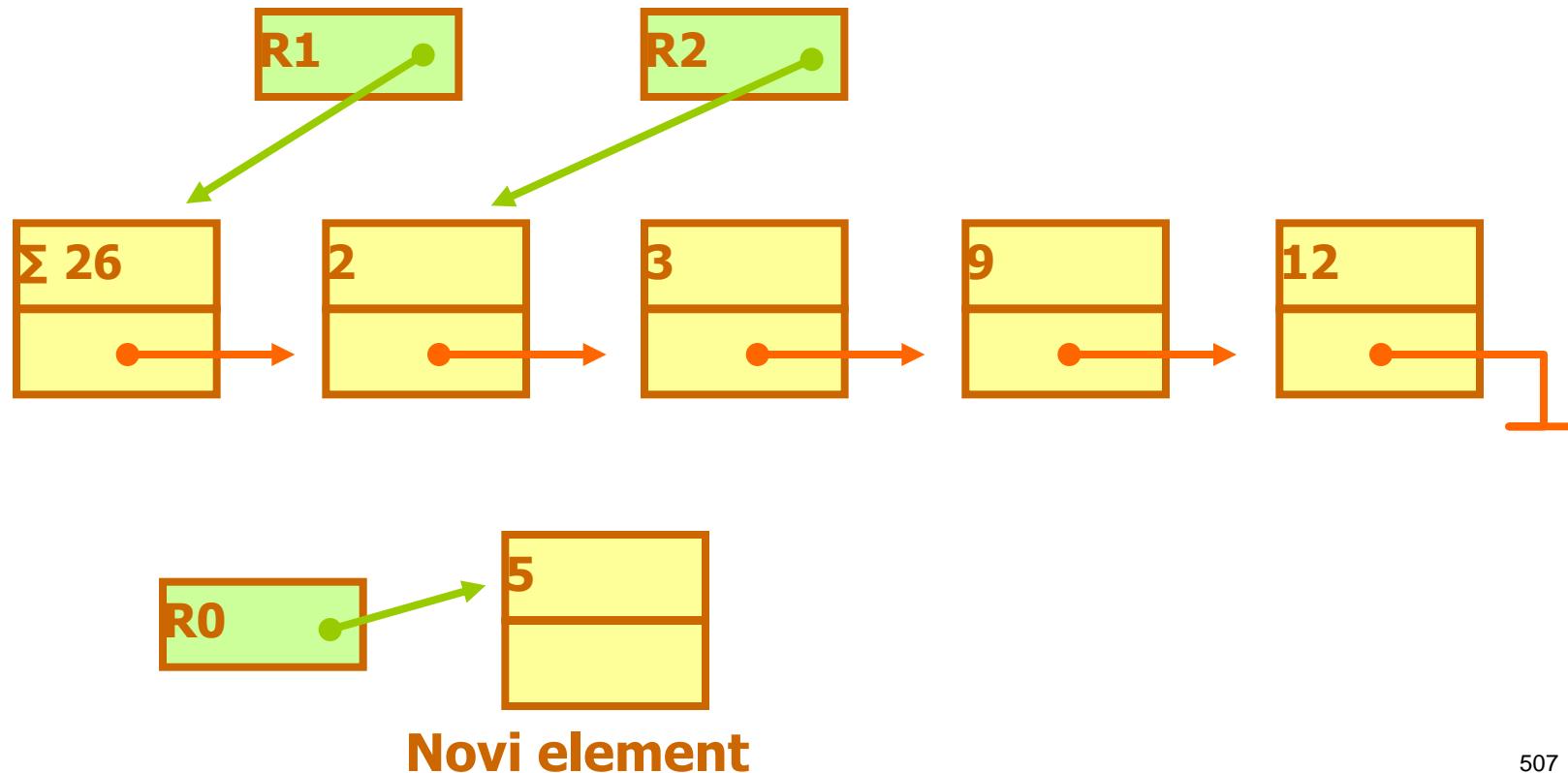


Ostali primjeri

Način rada potprograma:

Pokazuje čvor iza kojeg se ubacuje novi čvor (početno na prvi čvor)

Pokazuje čvor čiju vrijednost uspoređujemo s novim čvorom (jedan čvor dalje od R1)

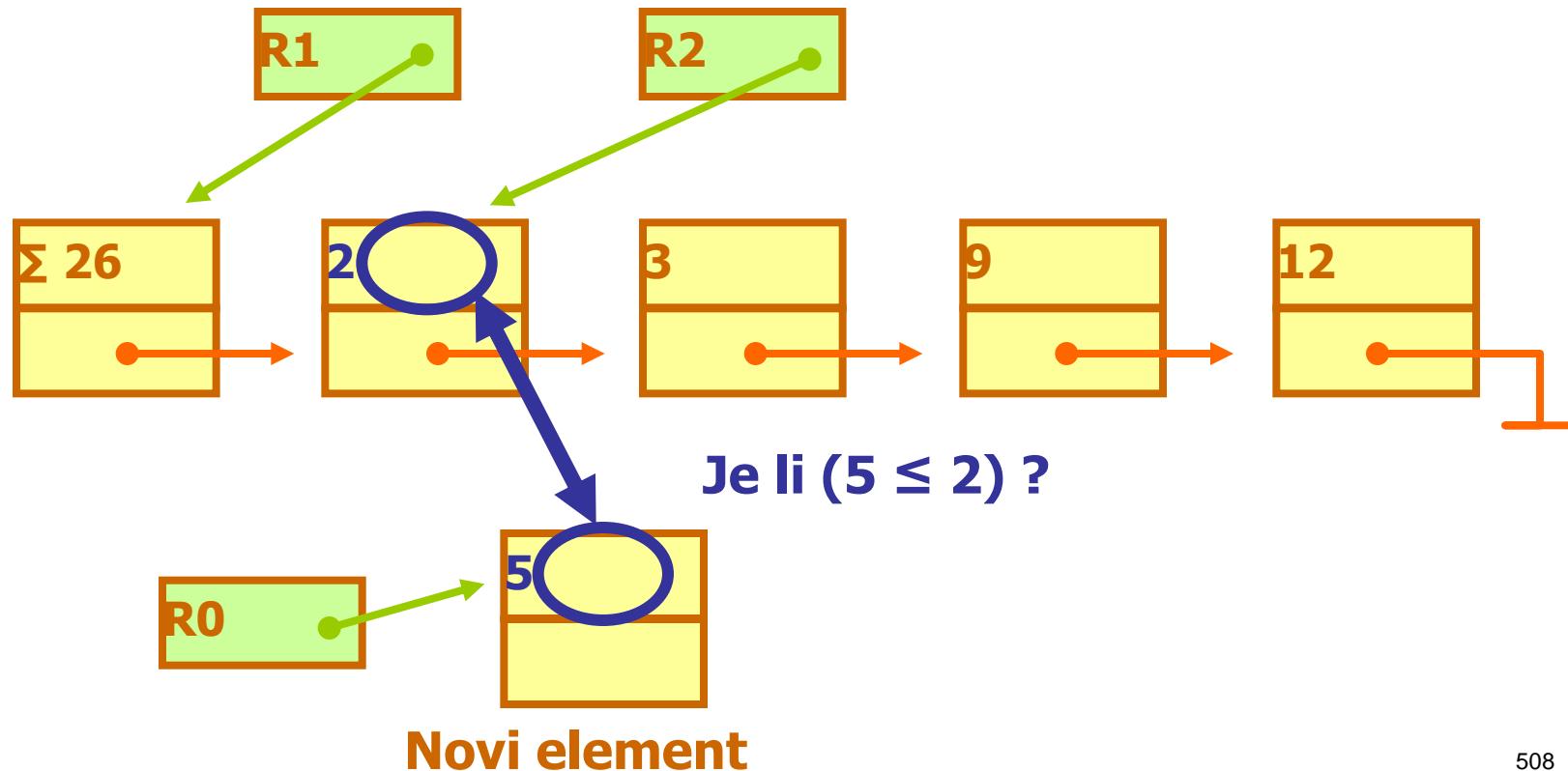


Ostali primjeri

Način rada potprograma:

Pokazuje čvor iza kojeg se ubacuje novi čvor (početno na prvi čvor)

Pokazuje čvor čiju vrijednost uspoređujemo s novim čvorom (jedan čvor dalje od R1)

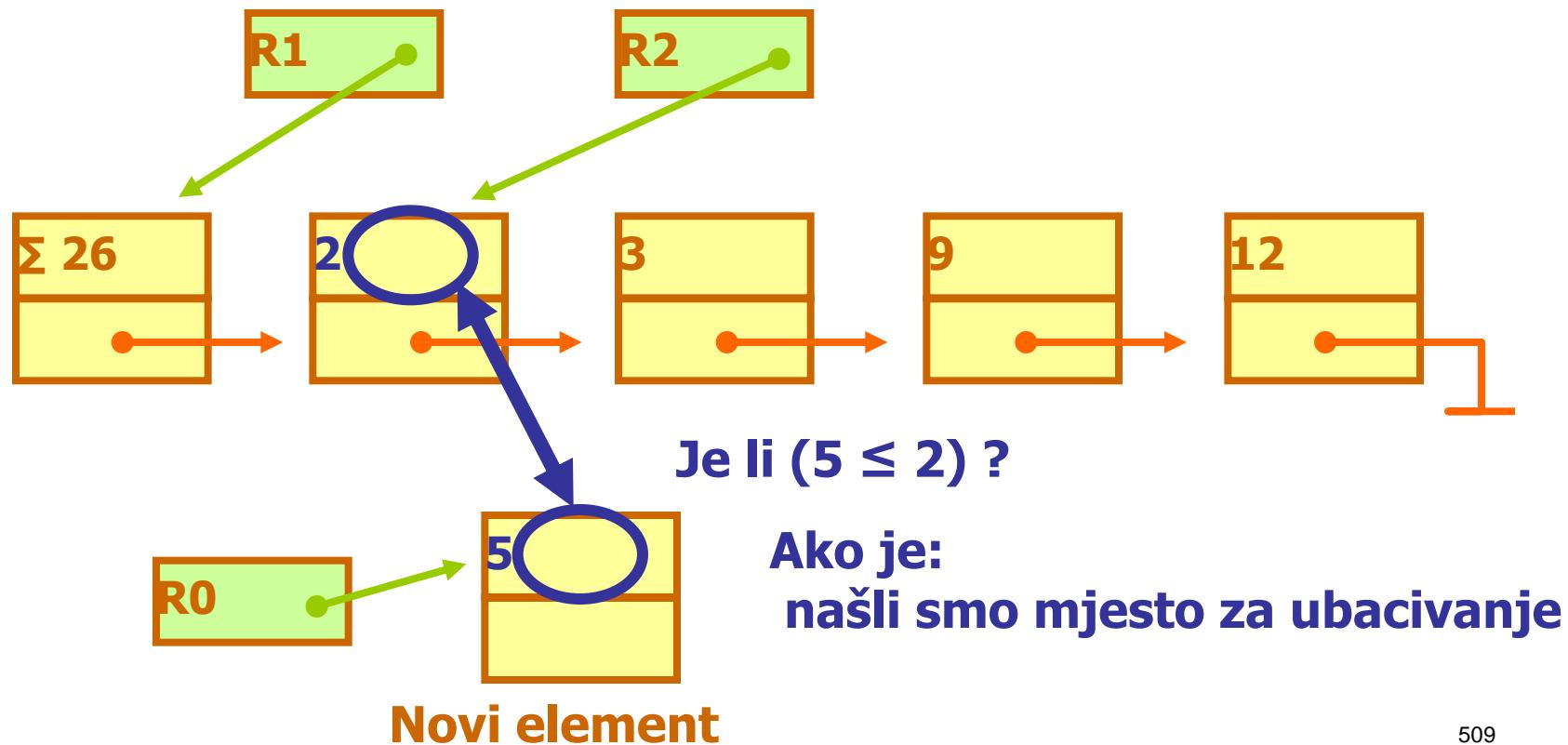


Ostali primjeri

Način rada potprograma:

Pokazuje čvor iza kojeg se ubacuje novi čvor (početno na prvi čvor)

Pokazuje čvor čiju vrijednost uspoređujemo s novim čvorom (jedan čvor dalje od R1)

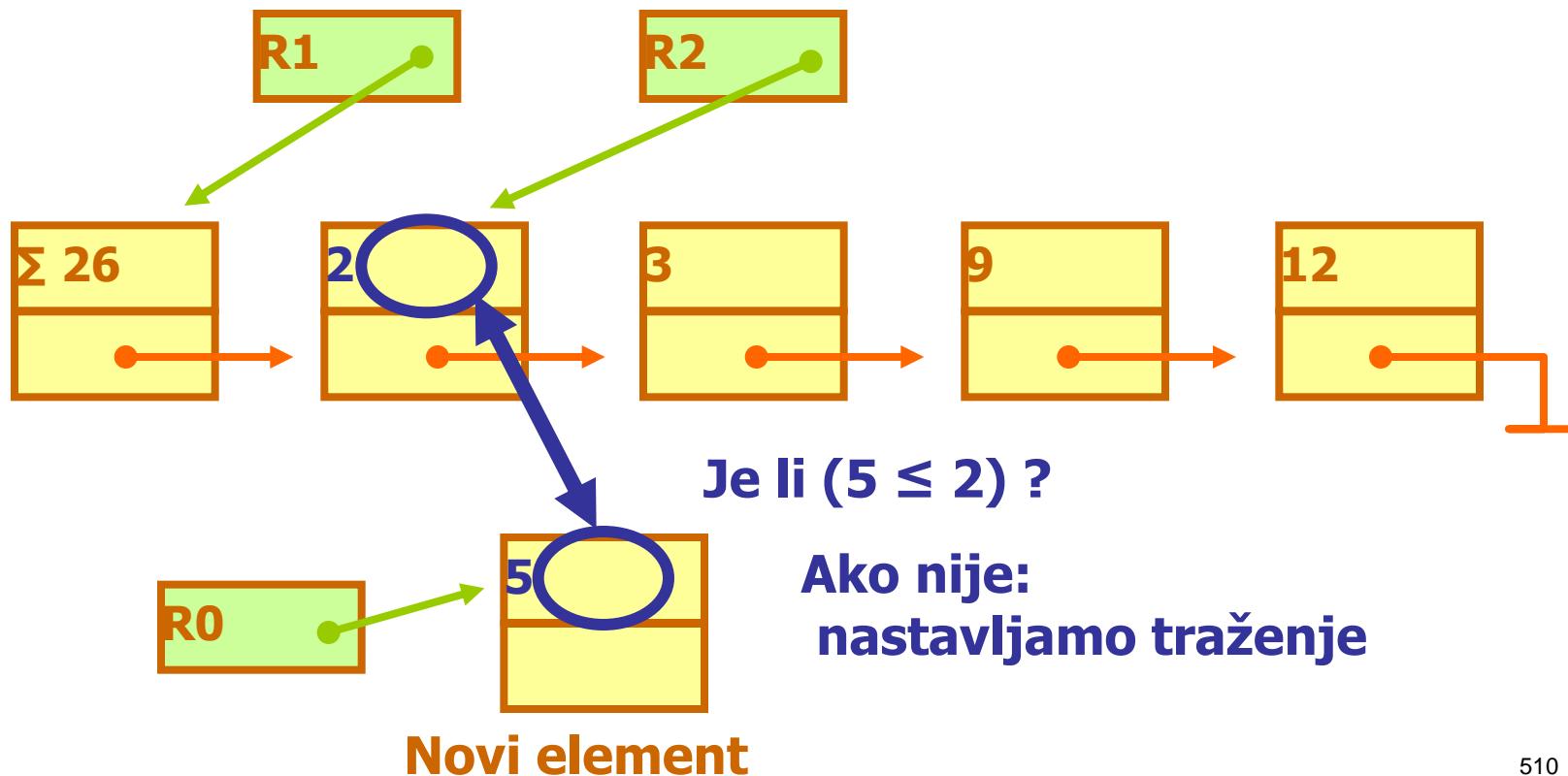


Ostali primjeri

Način rada potprograma:

Pokazuje čvor iza kojeg se ubacuje novi čvor (početno na prvi čvor)

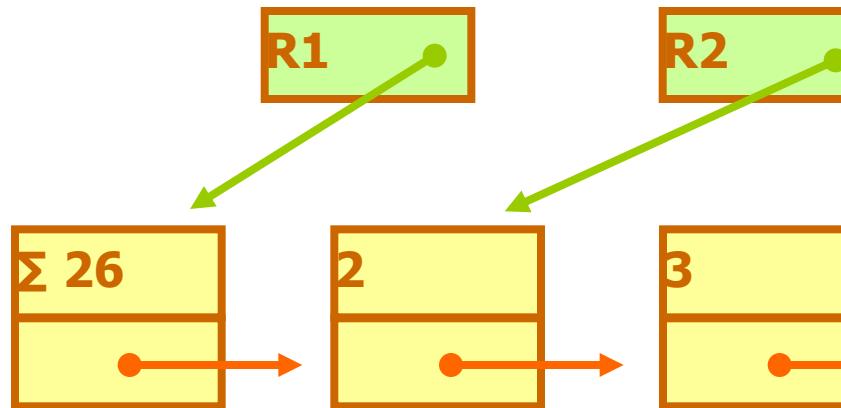
Pokazuje čvor čiju vrijednost uspoređujemo s novim čvorom (jedan čvor dalje od R1)



Ostali primjeri

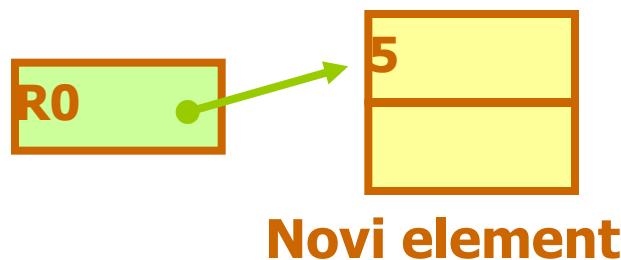
Način rada potprograma:

Pokazuje čvor iza kojeg se ubacuje novi čvor (početno na prvi čvor)



Pokazuje čvor čiju vrijednost uspoređujemo s novim čvorom (jedan čvor dalje od R1)

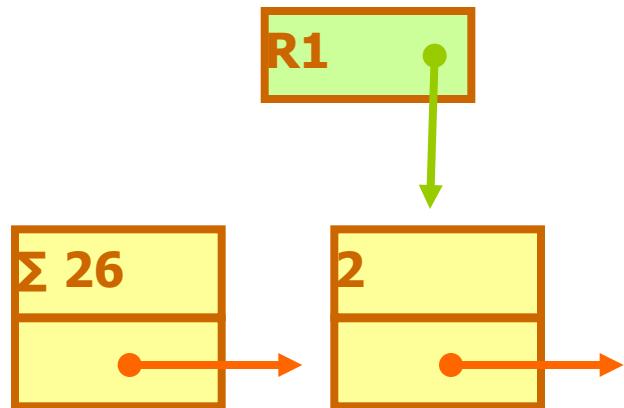
Pomičemo R1 i R2 za jedan čvor dalje



Ostali primjeri

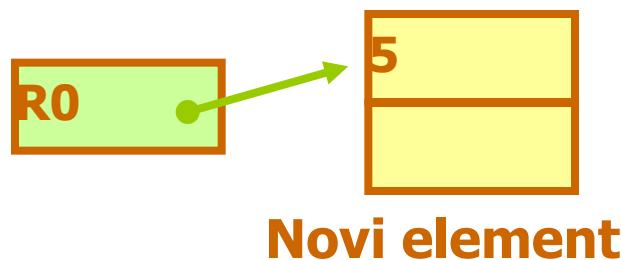
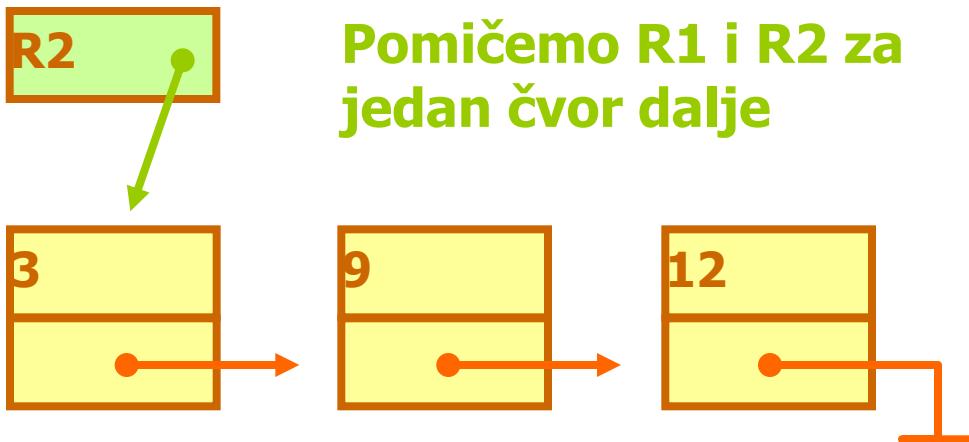
Način rada potprograma:

Pokazuje čvor iza kojeg se ubacuje novi čvor (početno na prvi čvor)



Pokazuje čvor čiju vrijednost uspoređujemo s novim čvorom (jedan čvor dalje od R1)

Pomičemo R1 i R2 za jedan čvor dalje

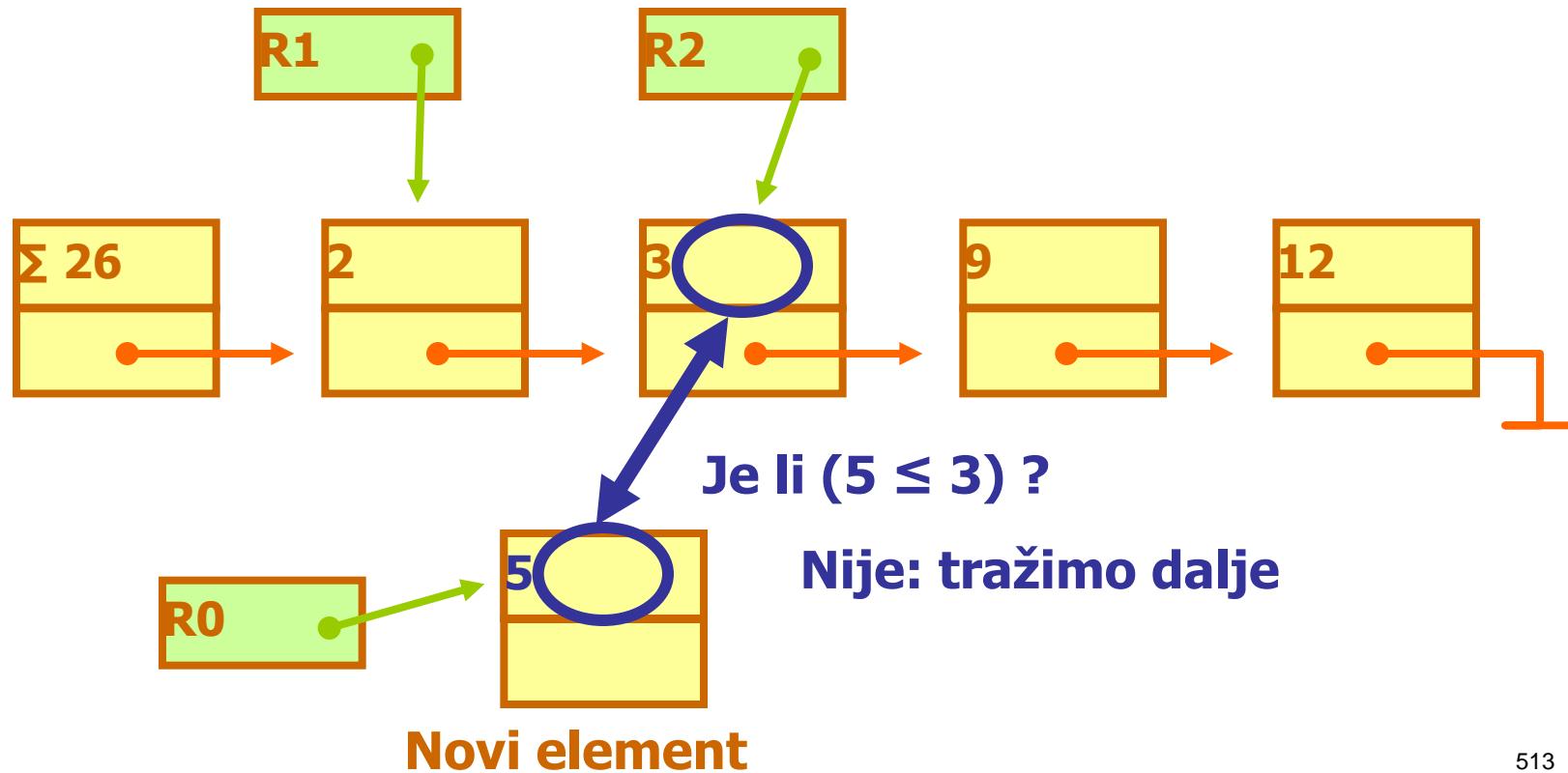


Ostali primjeri

Način rada potprograma:

Pokazuje čvor iza kojeg se ubacuje novi čvor (početno na prvi čvor)

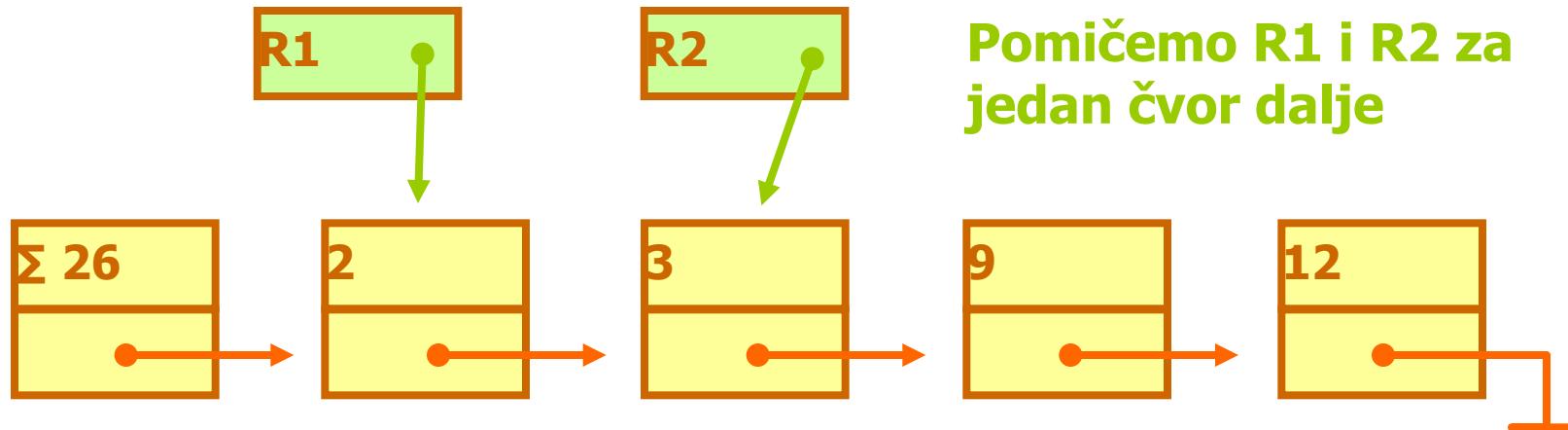
Pokazuje čvor čiju vrijednost uspoređujemo s novim čvorom (jedan čvor dalje od R1)



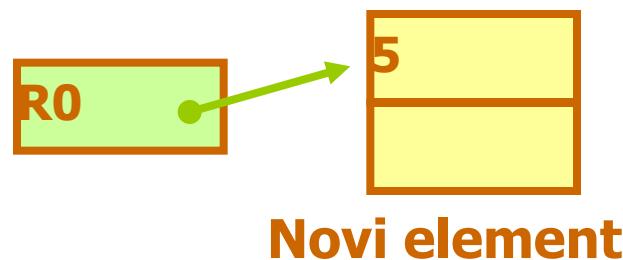
Ostali primjeri

Način rada potprograma:

Pokazuje čvor iza kojeg se ubacuje novi čvor (početno na prvi čvor)



Pokazuje čvor čiju vrijednost uspoređujemo s novim čvorom (jedan čvor dalje od R1)



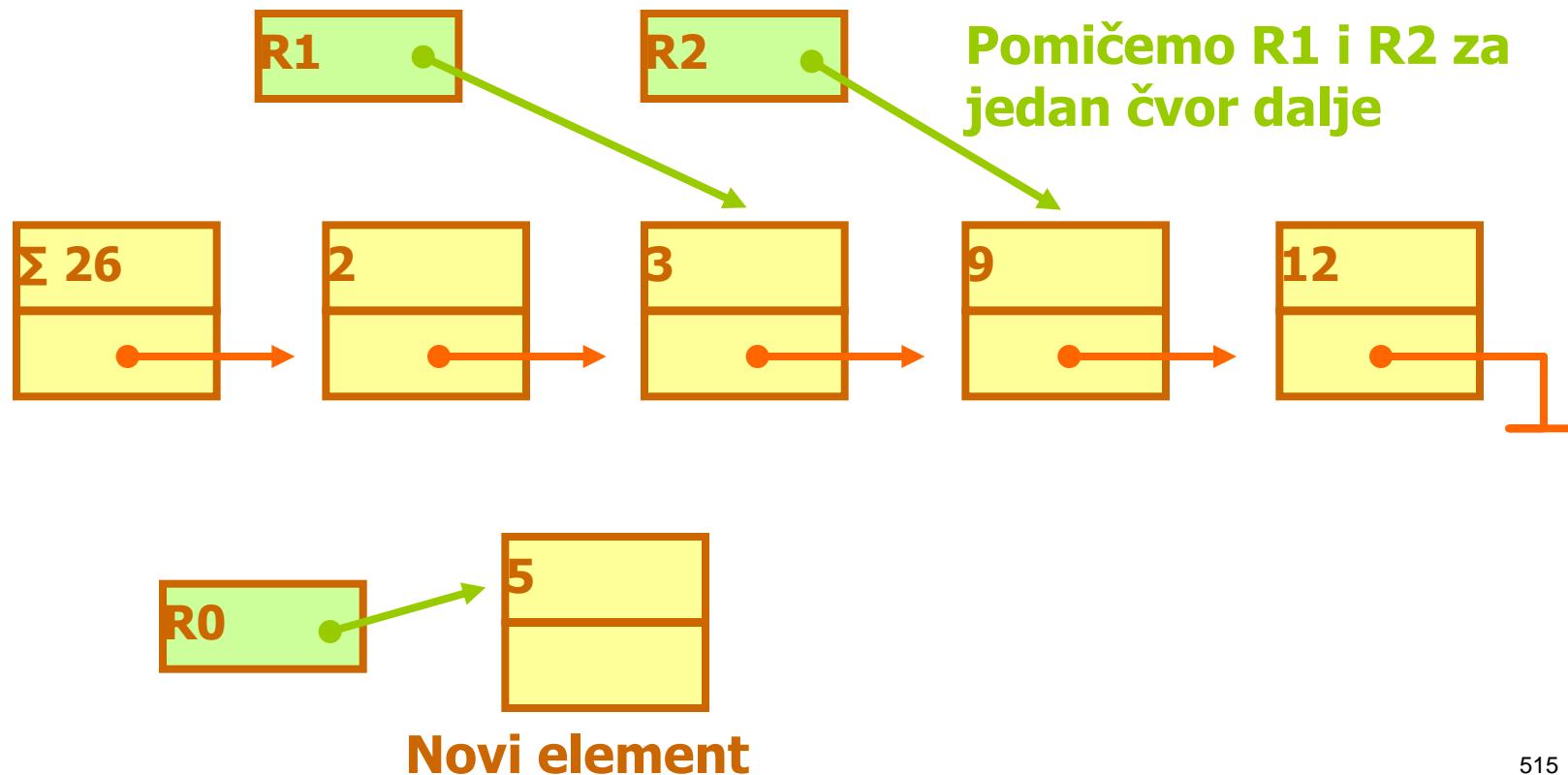
Pomičemo R1 i R2 za jedan čvor dalje

Ostali primjeri

Način rada potprograma:

Pokazuje čvor iza kojeg se ubacuje novi čvor (početno na prvi čvor)

Pokazuje čvor čiju vrijednost uspoređujemo s novim čvorom (jedan čvor dalje od R1)

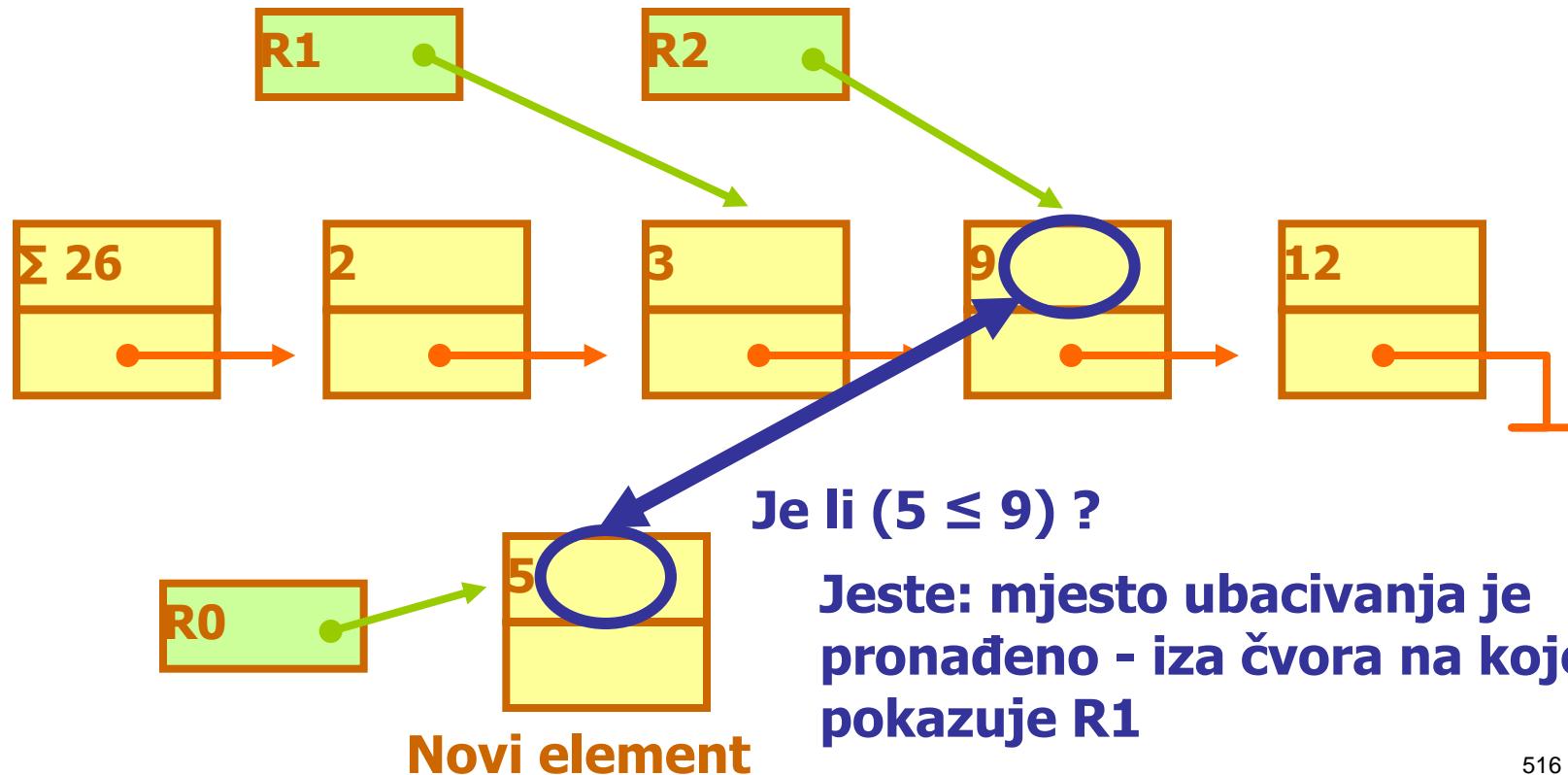


Ostali primjeri

Način rada potprograma:

Pokazuje čvor iza kojeg se ubacuje novi čvor (početno na prvi čvor)

Pokazuje čvor čiju vrijednost uspoređujemo s novim čvorom (jedan čvor dalje od R1)

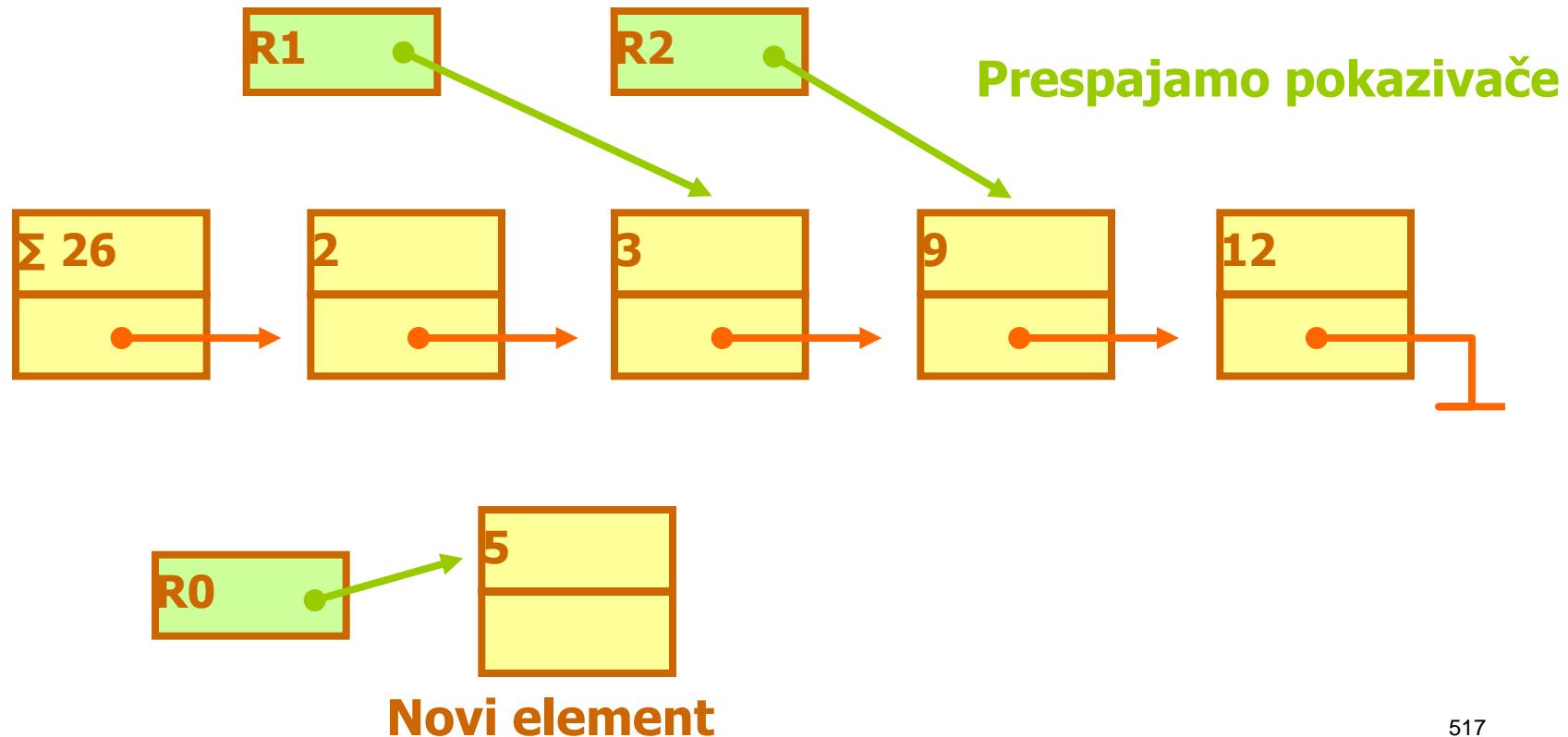


Ostali primjeri

Način rada potprograma:

Pokazuje čvor iza kojeg se ubacuje novi čvor (početno na prvi čvor)

Pokazuje čvor čiju vrijednost uspoređujemo s novim čvorom (jedan čvor dalje od R1)

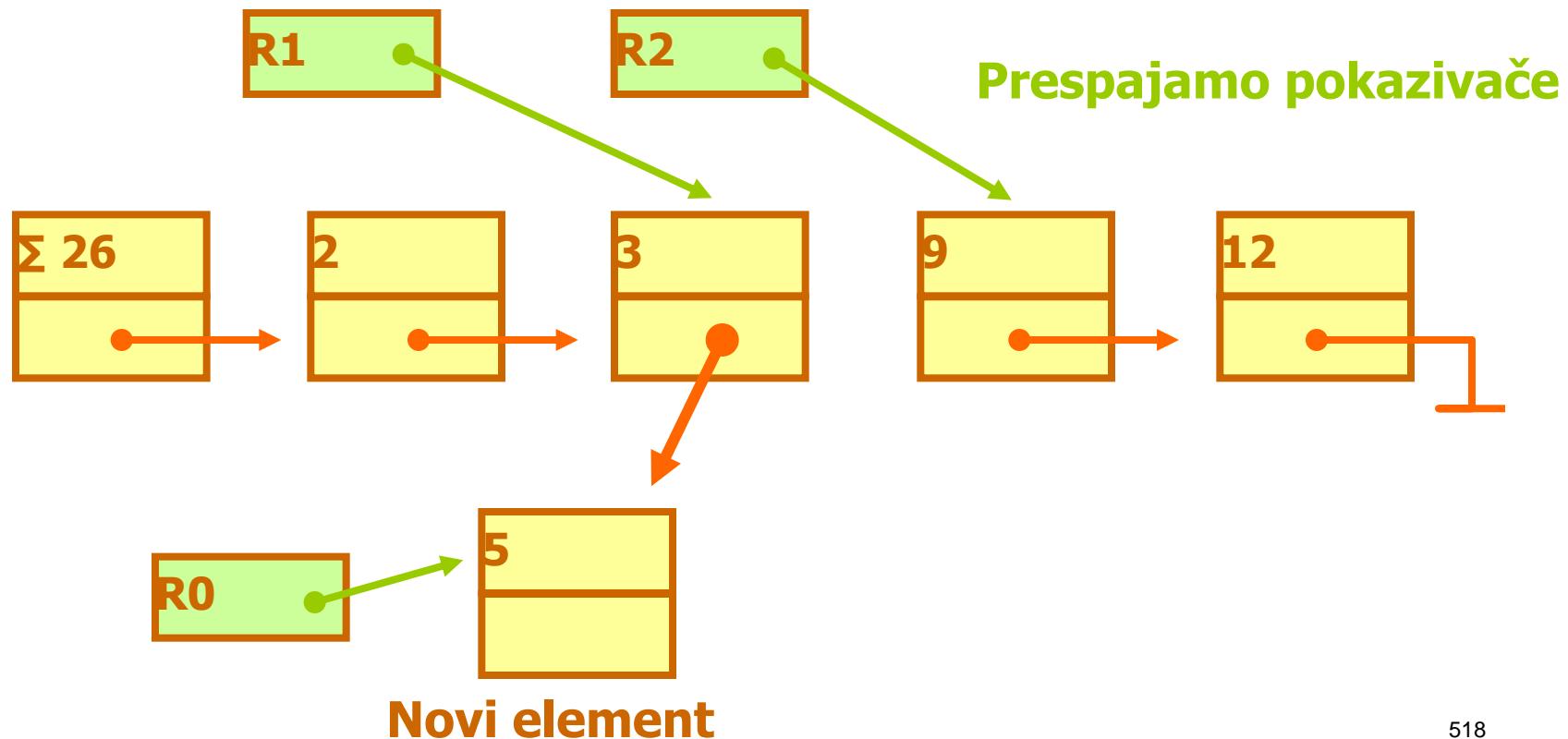


Ostali primjeri

Način rada potprograma:

Pokazuje čvor iza kojeg se ubacuje novi čvor (početno na prvi čvor)

Pokazuje čvor čiju vrijednost uspoređujemo s novim čvorom (jedan čvor dalje od R1)

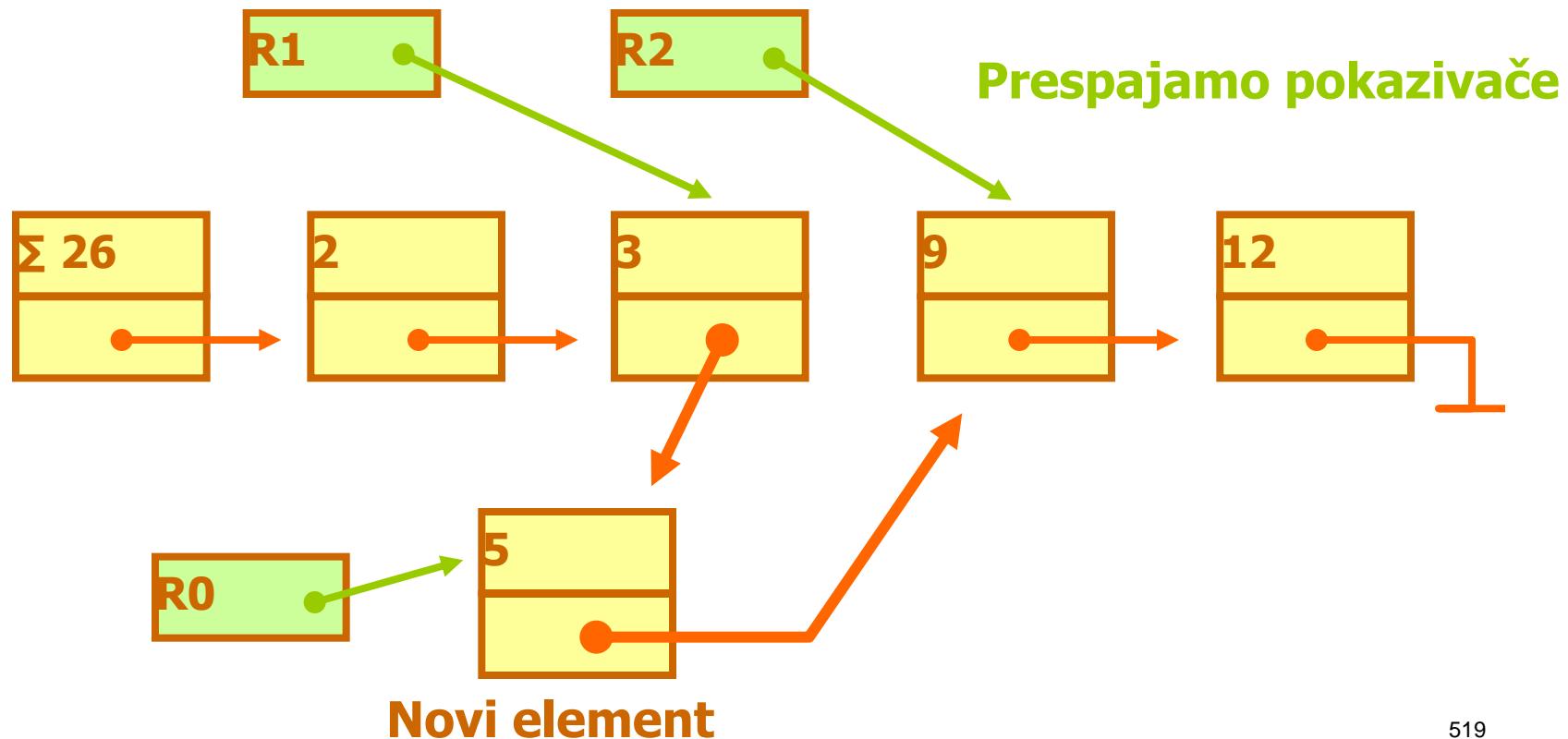


Ostali primjeri

Način rada potprograma:

Pokazuje čvor iza kojeg se ubacuje novi čvor (početno na prvi čvor)

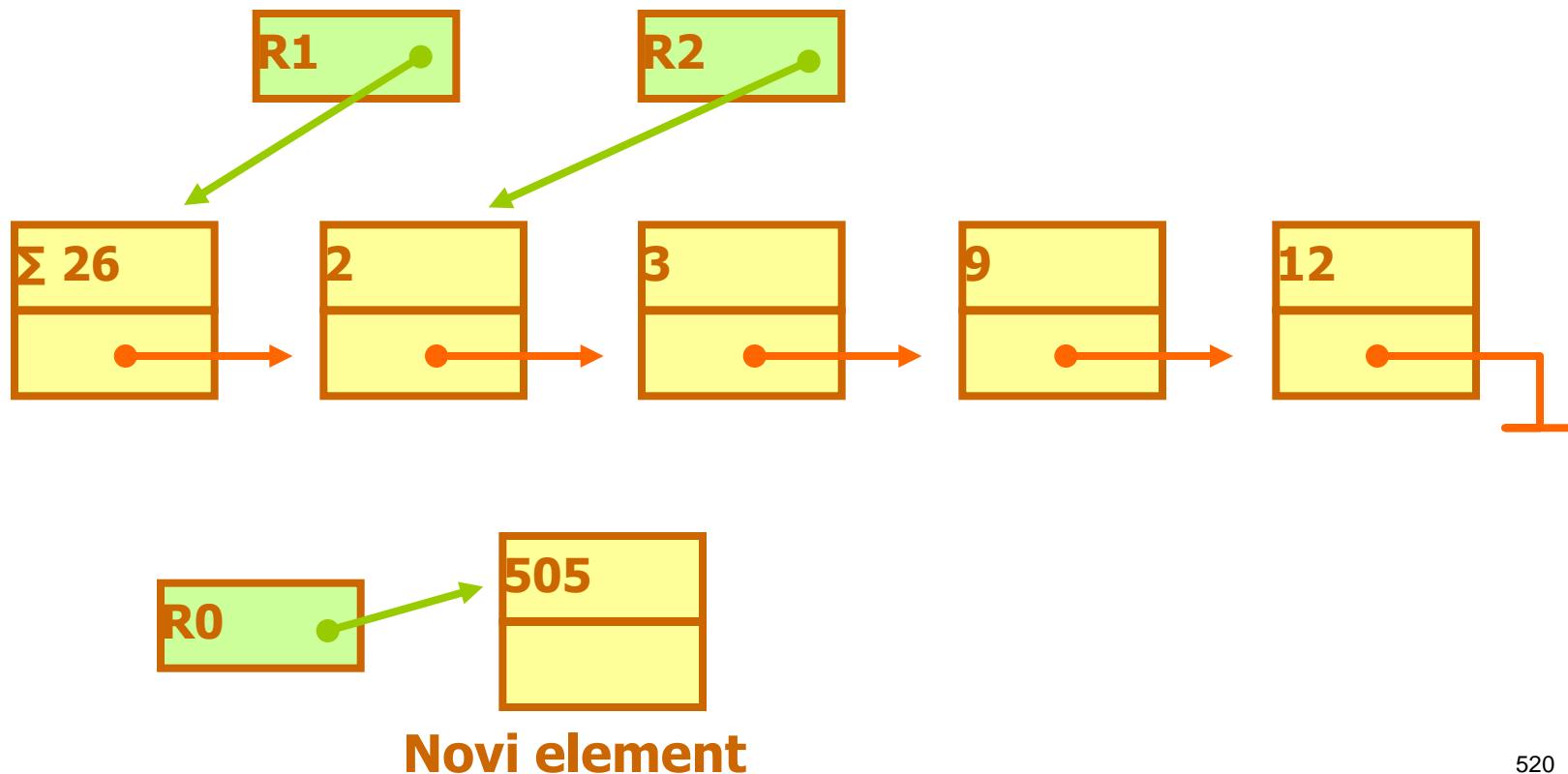
Pokazuje čvor čiju vrijednost uspoređujemo s novim čvorom (jedan čvor dalje od R1)



Ostali primjeri

Način rada potprograma:

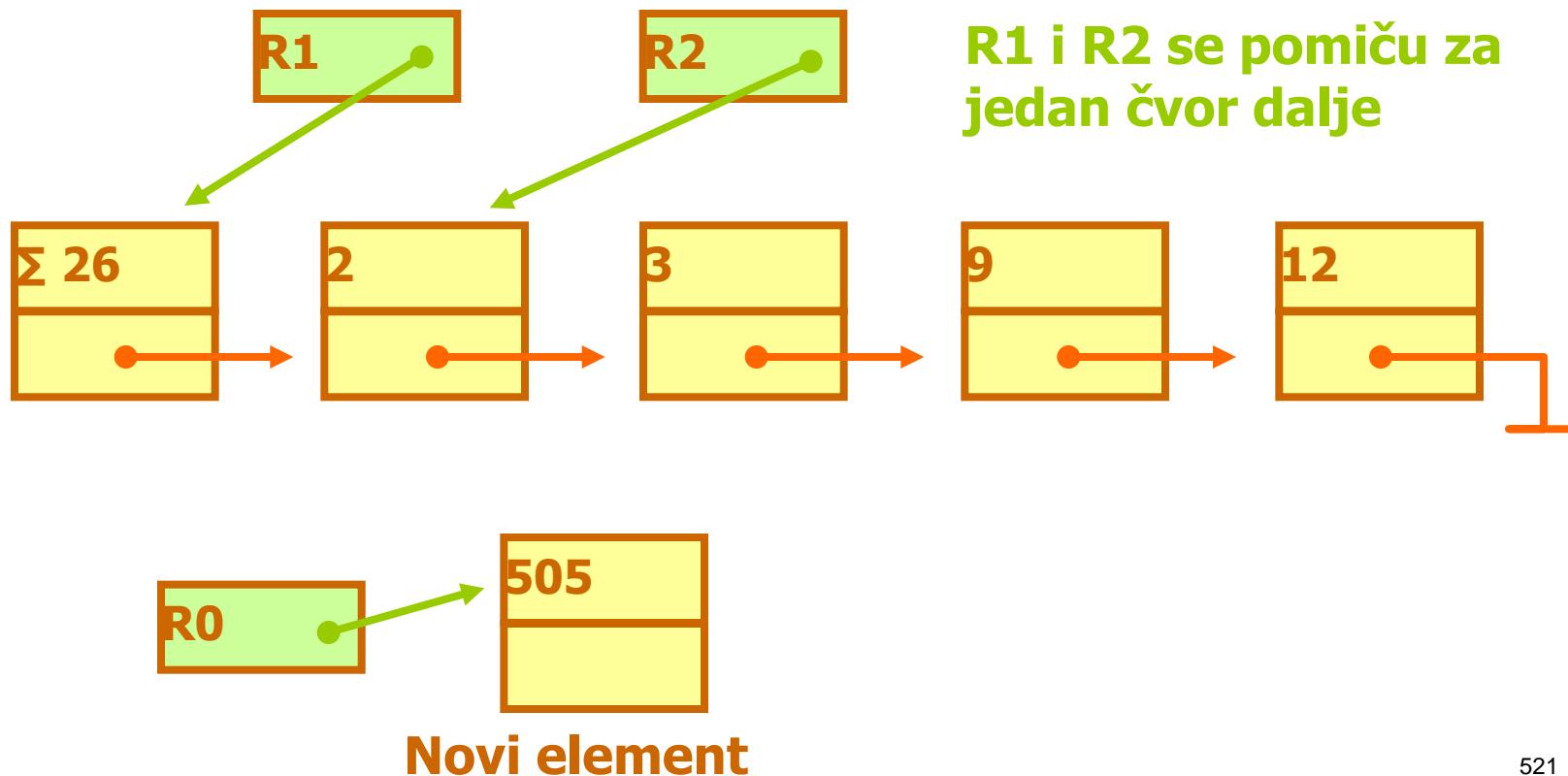
Pogledajmo još slučaj kad su u listi svi elementi manji od novog člana (ili je lista prazna)



Ostali primjeri

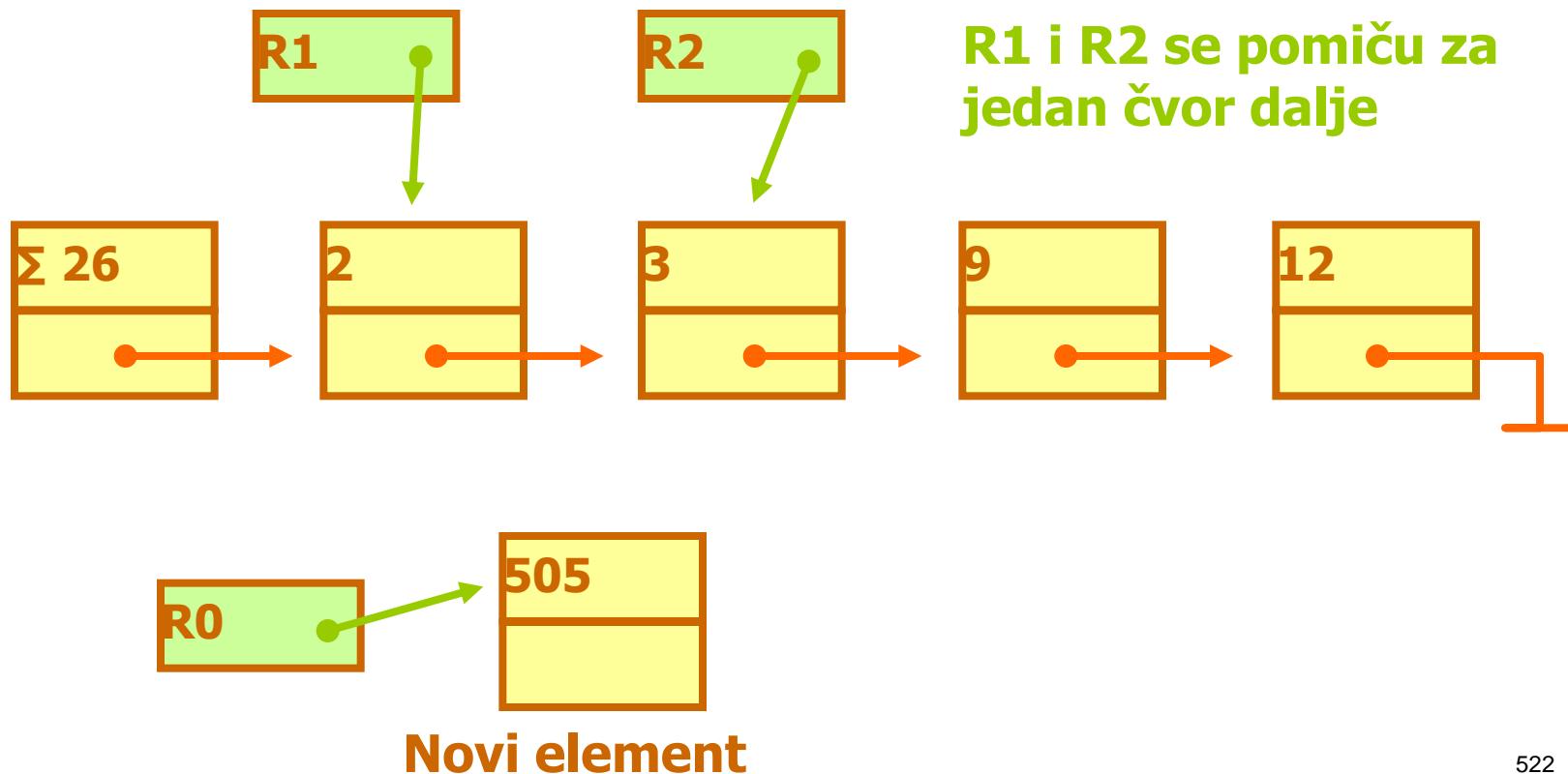
Način rada potprograma:

Pogledajmo još slučaj kad su u listi svi elementi manji od novog člana (ili je lista prazna)



Ostali primjeri

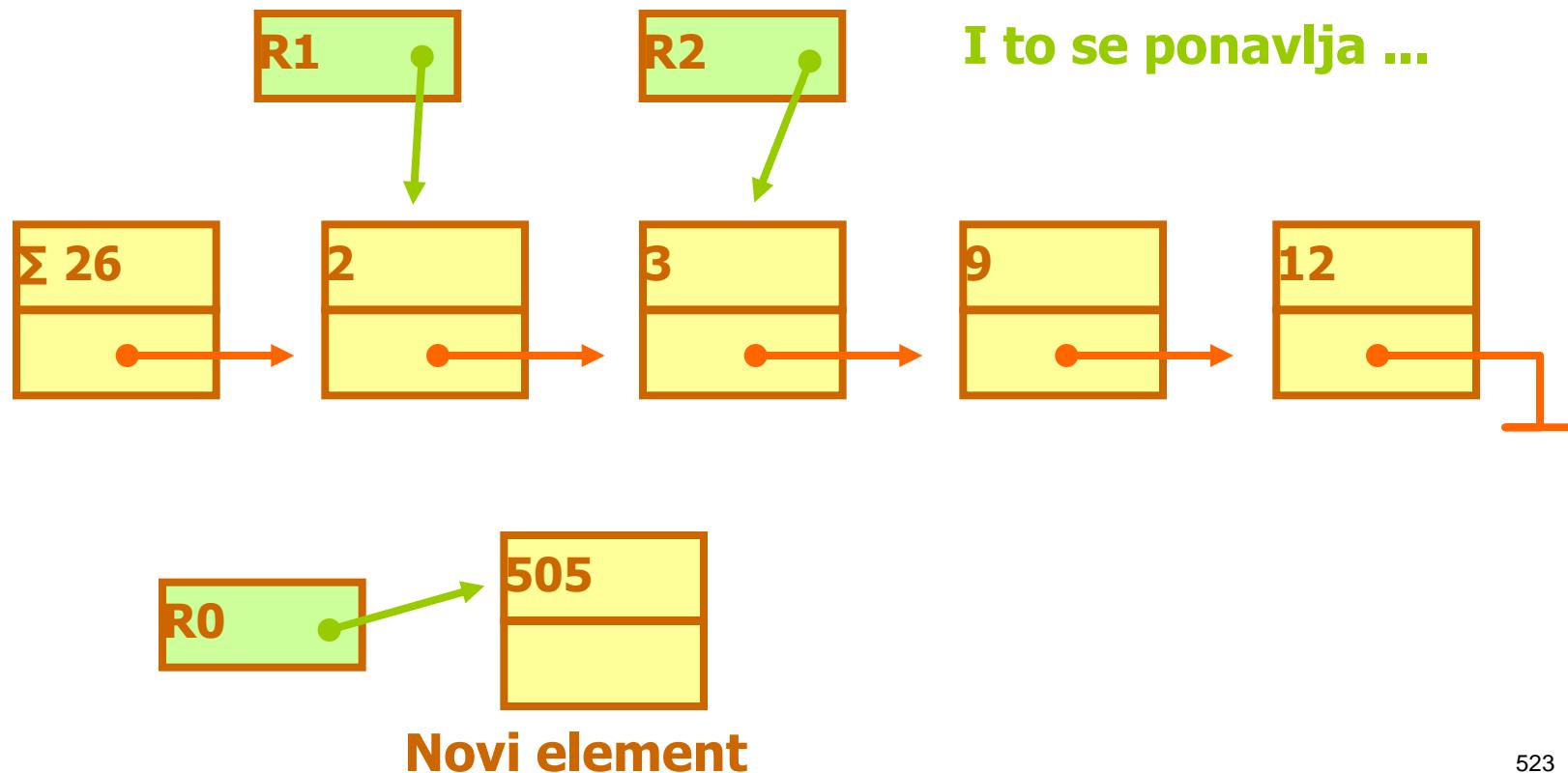
Pogledajmo još slučaj kad su u listi svi elementi manji od novog člana (ili je lista prazna)



Ostali primjeri

Način rada potprograma:

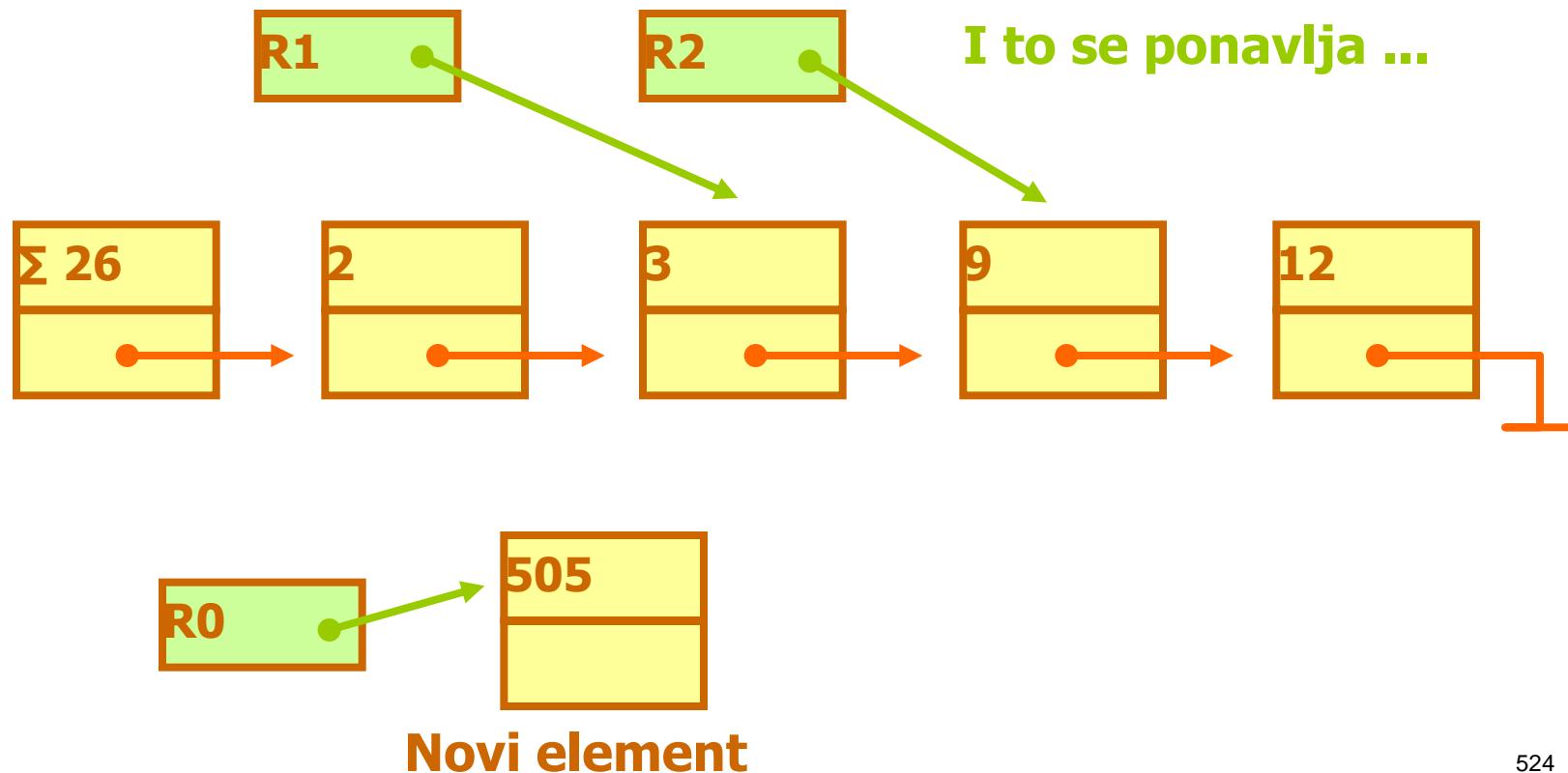
Pogledajmo još slučaj kad su u listi svi elementi manji od novog člana (ili je lista prazna)



Ostali primjeri

Način rada potprograma:

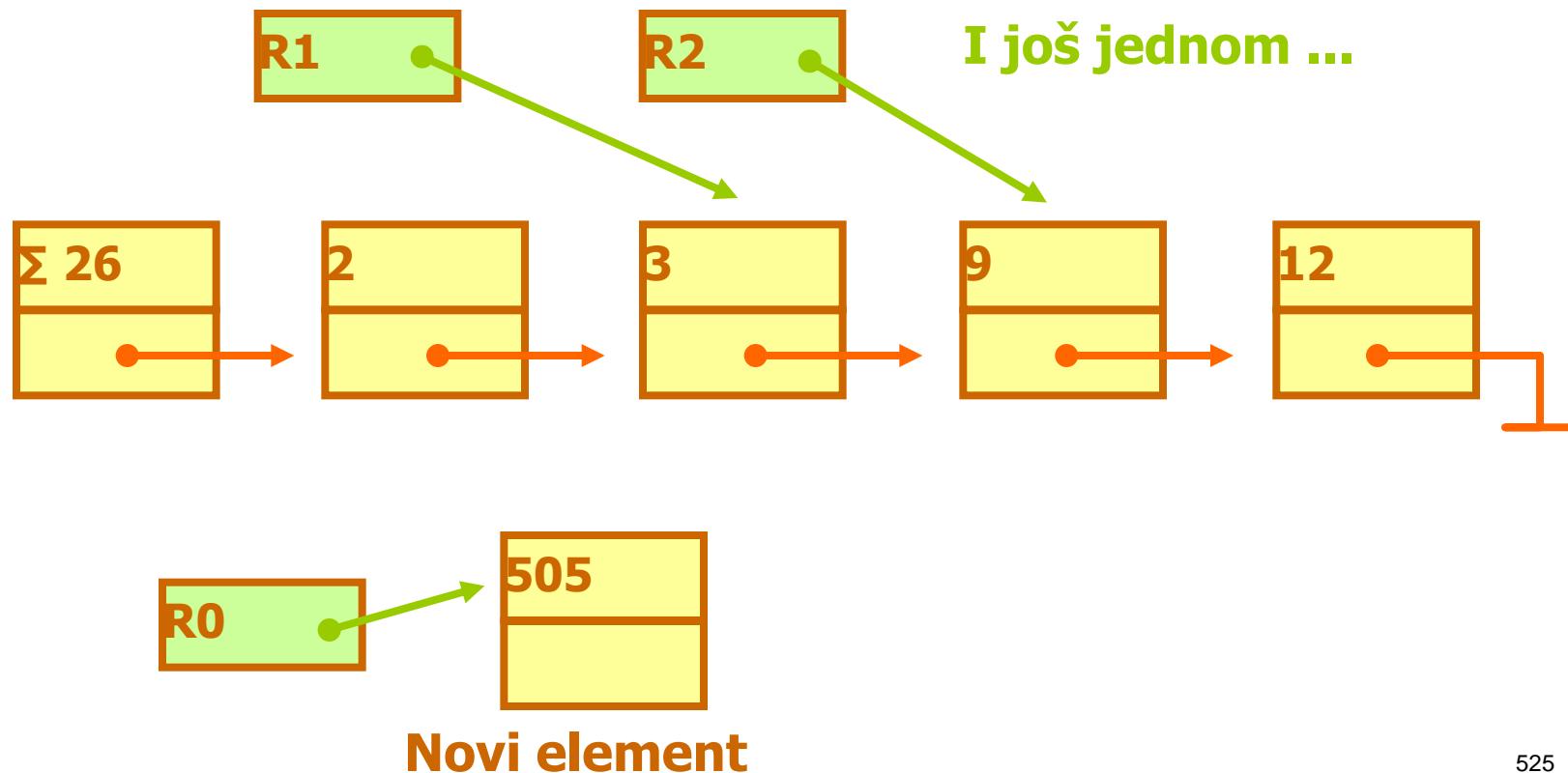
Pogledajmo još slučaj kad su u listi svi elementi manji od novog člana (ili je lista prazna)



Ostali primjeri

Način rada potprograma:

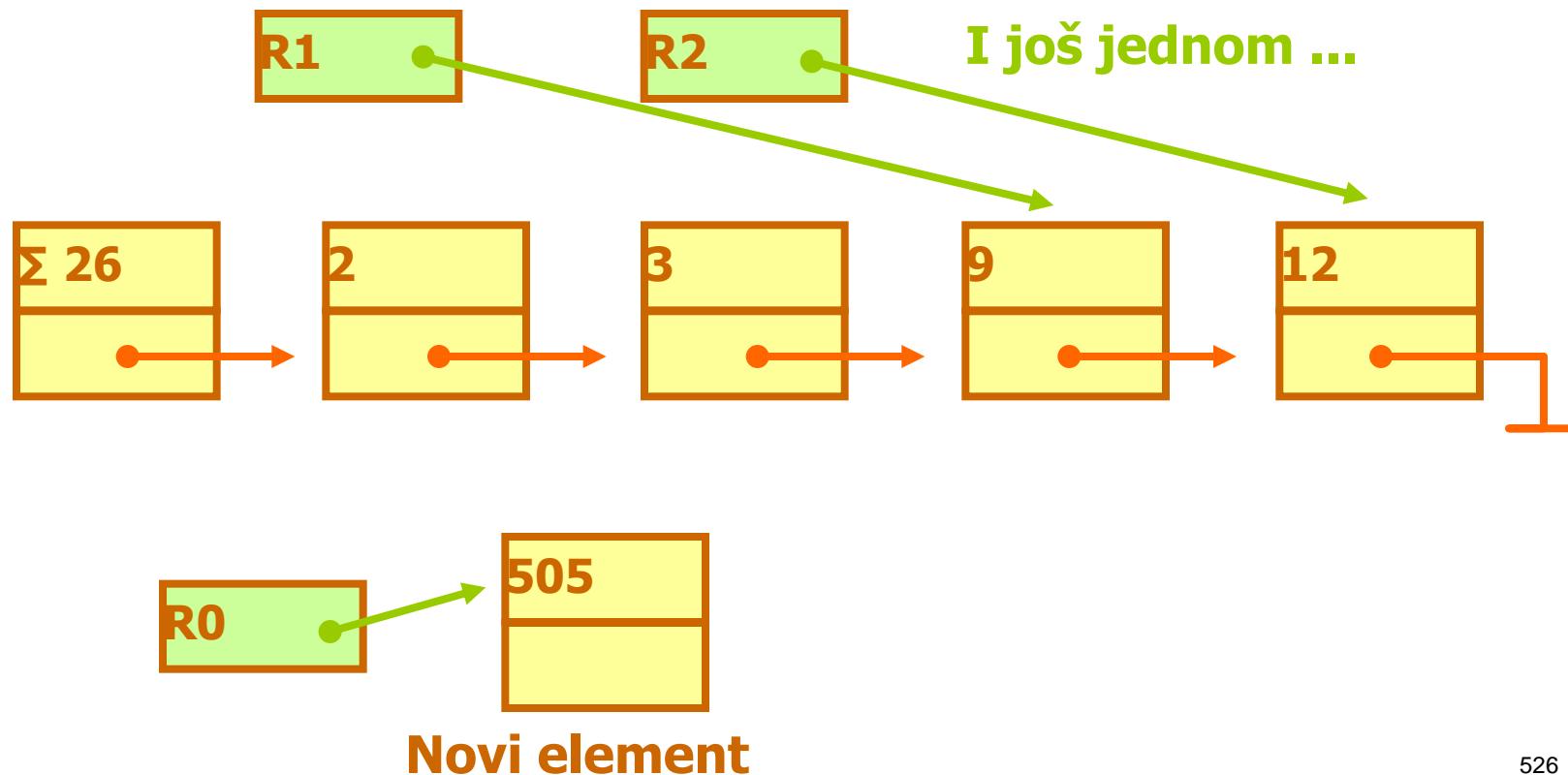
Pogledajmo još slučaj kad su u listi svi elementi manji od novog člana (ili je lista prazna)



Ostali primjeri

Način rada potprograma:

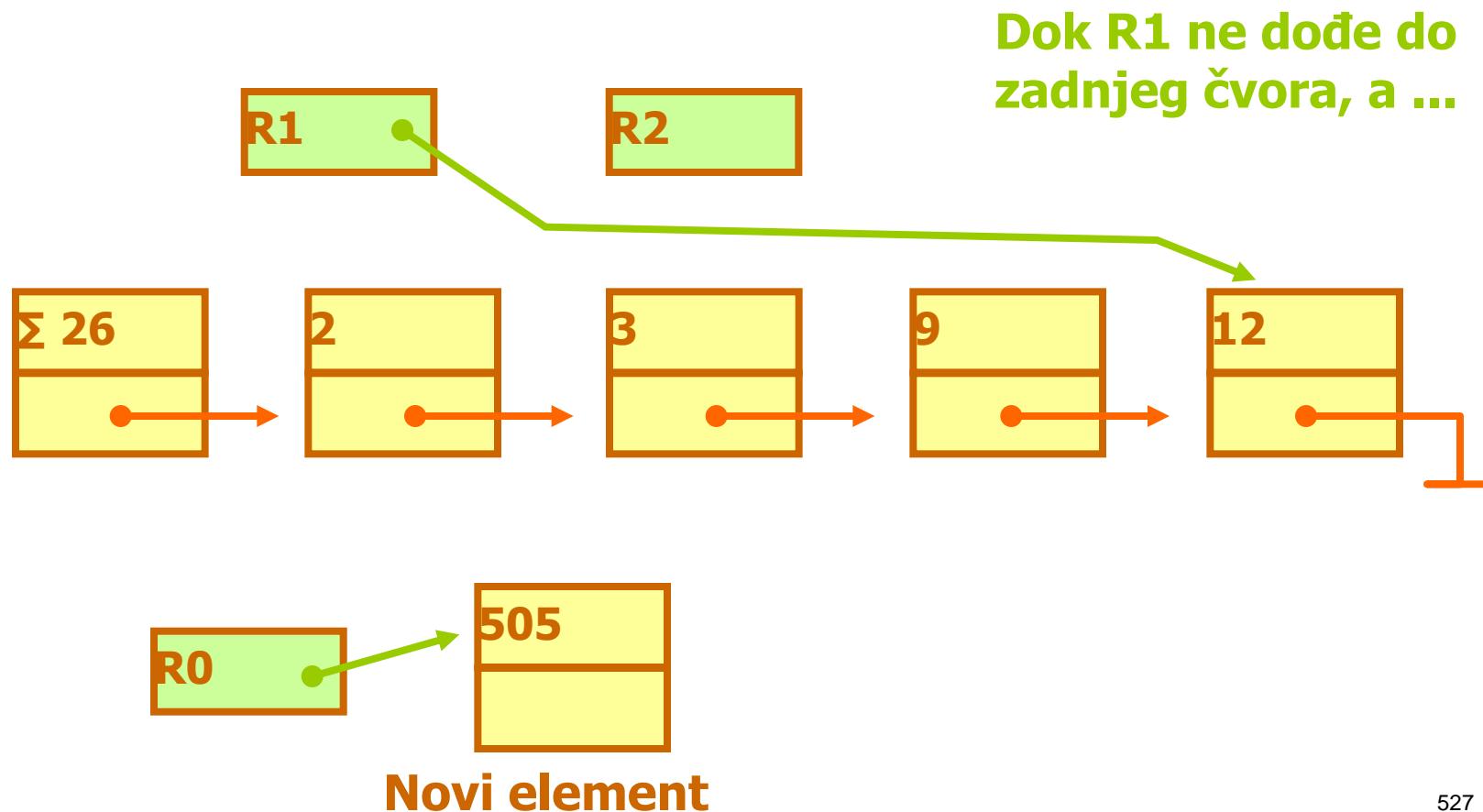
Pogledajmo još slučaj kad su u listi svi elementi manji od novog člana (ili je lista prazna)



Ostali primjeri

Način rada potprograma:

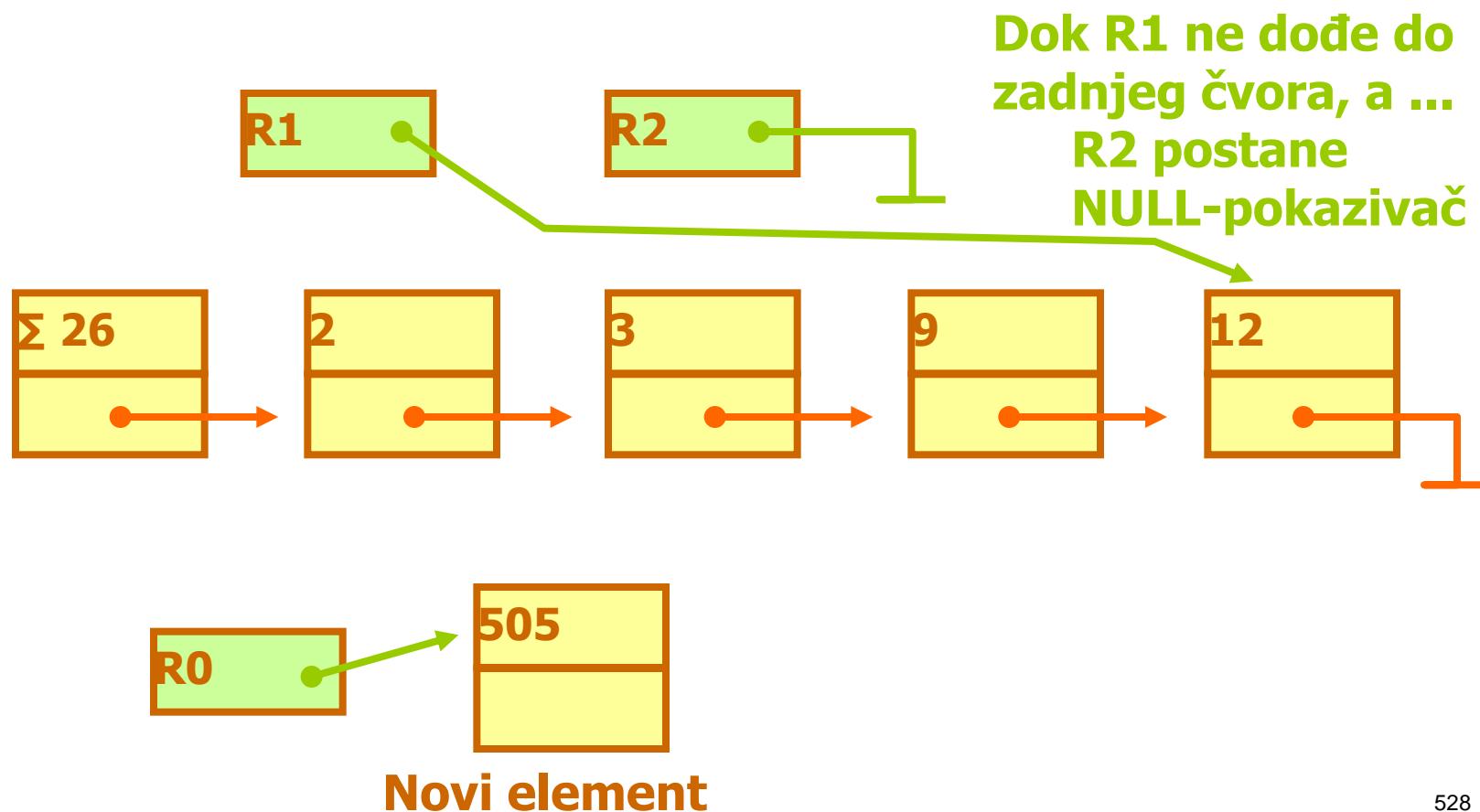
Pogledajmo još slučaj kad su u listi svi elementi manji od novog člana (ili je lista prazna)



Ostali primjeri

Način rada potprograma:

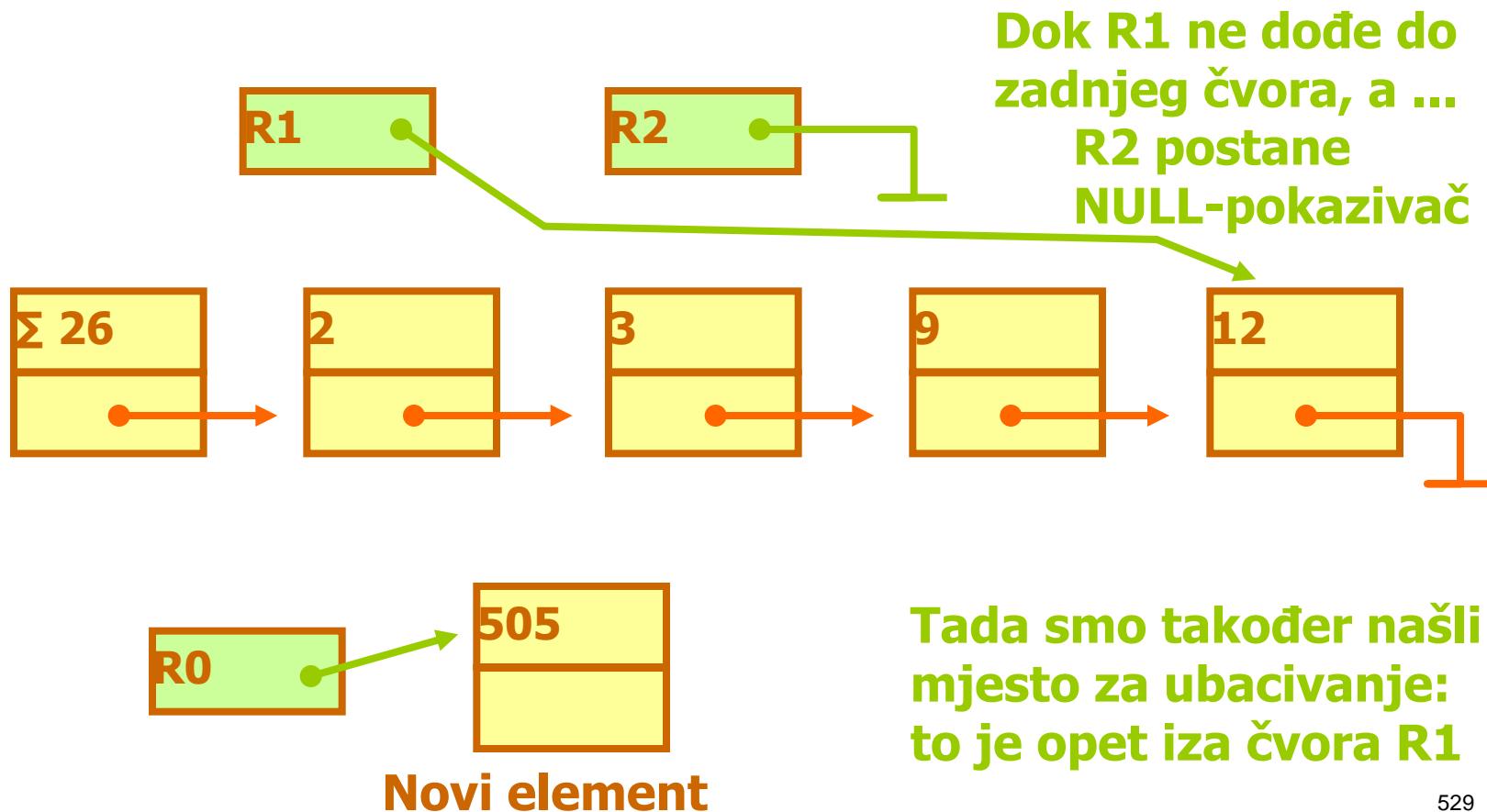
Pogledajmo još slučaj kad su u listi svi elementi manji od novog člana (ili je lista prazna)



Ostali primjeri

Način rada potprograma:

Pogledajmo još slučaj kad su u listi svi elementi manji od novog člana (ili je lista prazna)



UBACI ; ; ; ; Potprogram UBACI

; Parametri na stogu:

; 1. parametar:

; adresa prvog čvora (glava)

; 2. parametar:

; adresa novog čvora

PUSH R1 ; Spremi sve

PUSH R2 ; registre koje

PUSH R5 ; potprogram

PUSH R6 ; mijenja (osim R0).



; dohvati vrijednosti novog čvora za pretraživanje

LOAD R0, (SP+14) ; adresa novog čvora u R0

LOAD R6, (R0) ; vrijednost novog čvora u R6

; priprema pokazivača R1 i R2 za pretraživanje

LOAD R1, (SP+18) ; adresa prvog čvora u R1

LOAD R2, (R1+4) ; adresa drugog čvora u R2

<<<

; ; ; ; PETLJA ZA TRAŽENJE MJESTA ZA UBACIVANJE

```

TRAZI CMP      R2, 0          ; Ako je kraj liste:
    JR_Z    NASAO          ; => onda ubacujemo na kraj

; dohvati u R5 vrijednost trenutačnog čvora i
; usporedi je s vrijednošću R6 novog čvora
LOAD   R5, (R2)
CMP    R6, R5

JR_ULE NASAO      ; ako je novi ≤ trenutačni =>
                   ; pronađeno je mjesto za ubacivanje

; inače treba nastaviti s petljom za traženje

MOVE   R2, R1          ; pomakni se na sljedeći čvor
LOAD   R2, (R1+4)

JR     TRAZI          ; nastavi s traženjem

```

>>>

<<<

; ; ; ; DIO ZA UMETANJE NOVOG ČVORA U LISTU

NASAO STORE R0, (R1+4) ; stavi novi čvor iza prethodnog
STORE R2, (R0+4) ; stavi trenutačni čvor iza novog

; izračunavanje novog zbroja u prvom čvoru:

LOAD R1, (SP+18) ; dohvati adresu prvog
LOAD R0, (R1) ; dohvati dosadašnji zbroj

ADD R0, R6, R0 ; novi zbroj stavi u R0 (rezultat)
STORE R0, (R1) ; spremi ga u prvi čvor

POP R6 ; obnovi
POP R5 ; vrijednosti
POP R2 ; spremeljenih
POP R1 ; registara

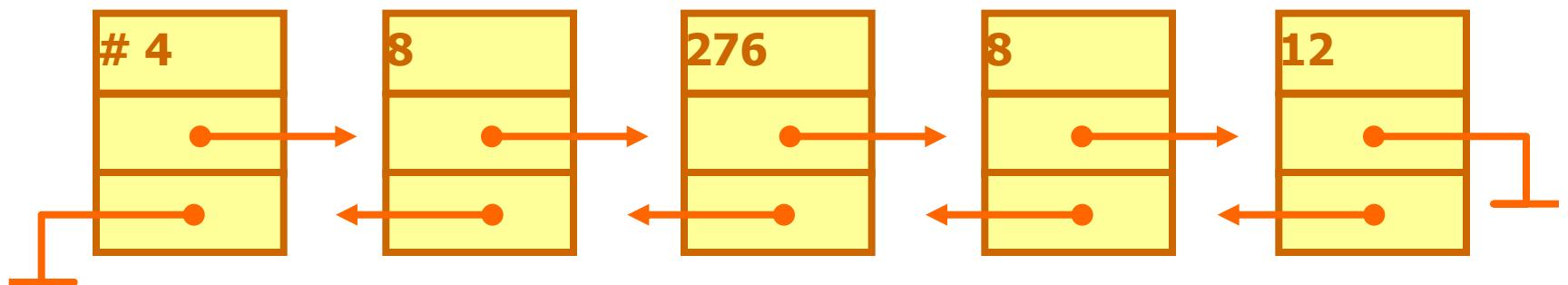
RET



Ostali primjeri

Primjer:

U memoriji se nalazi dvostruko povezana nesortirana lista. Svaki čvor zauzima tri memorijske lokacije: na prvoj je NBC broj, na drugoj je pokazivač na sljedeći čvor liste, a na trećoj je pokazivač na prethodni čvor liste. Prvi čvor liste ima specijalno značenje i uvijek je prisutan u listi. U njemu se pamti ukupan broj preostalih čvorova u listi.





Ostali primjeri

<<<

Treba napisati potprogram IZBACI koji prima preko stoga dva parametra: pokazivač na prvi čvor liste i NBC broj (X). Potprogram traži prvo pojavljivanje čvora u kojem je upisan broj X. Ako ga nađe, izbacuje taj čvor iz liste. Povratna vrijednost je adresa izbačenog čvora ili nula ako čvor nije pronađen. Vrijednost se vraća pomoću R1. Potprogram ne smije mijenjati sadržaje registara u glavnom programu.

Rješenje:

Parametre će sa stoga uklanjati glavni program.

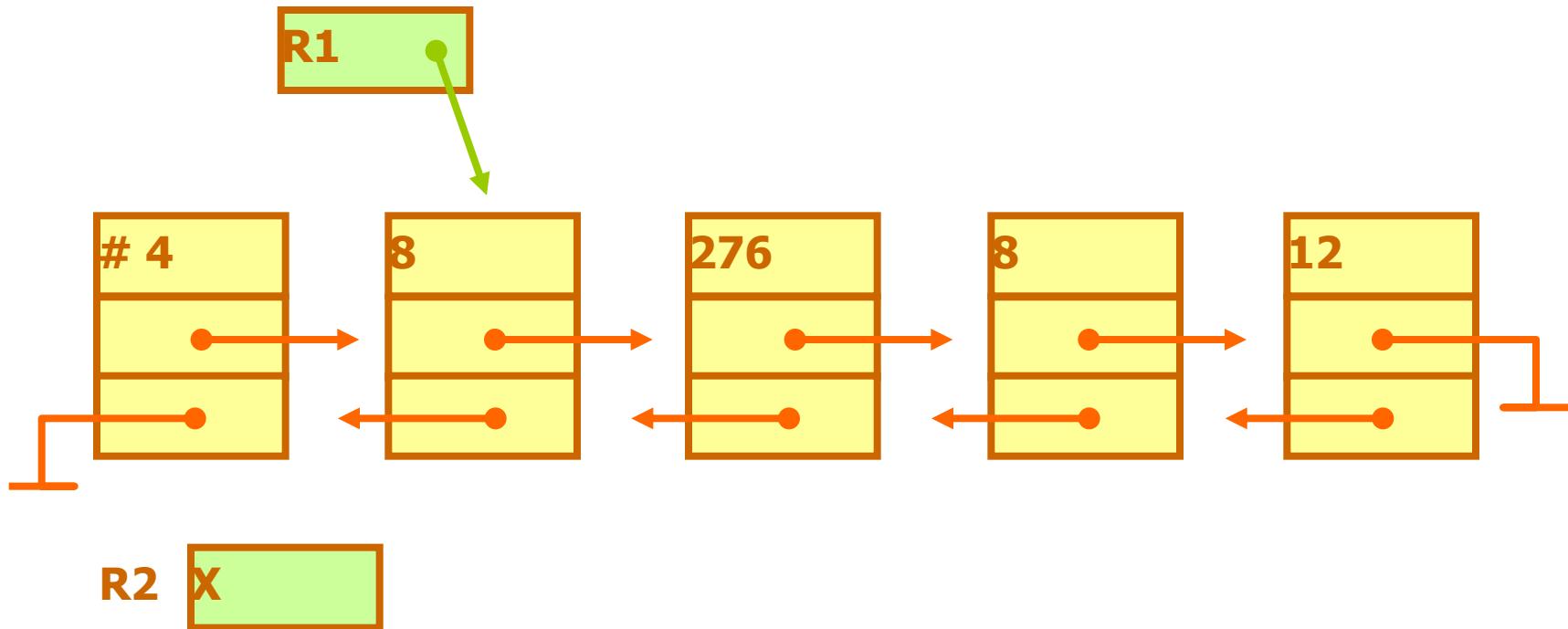
>>>



Ostali primjeri

<<< Način rada potprograma:

Pokazivač na čvor koji se uspoređuje s X (početno pokazuje na drugi čvor)



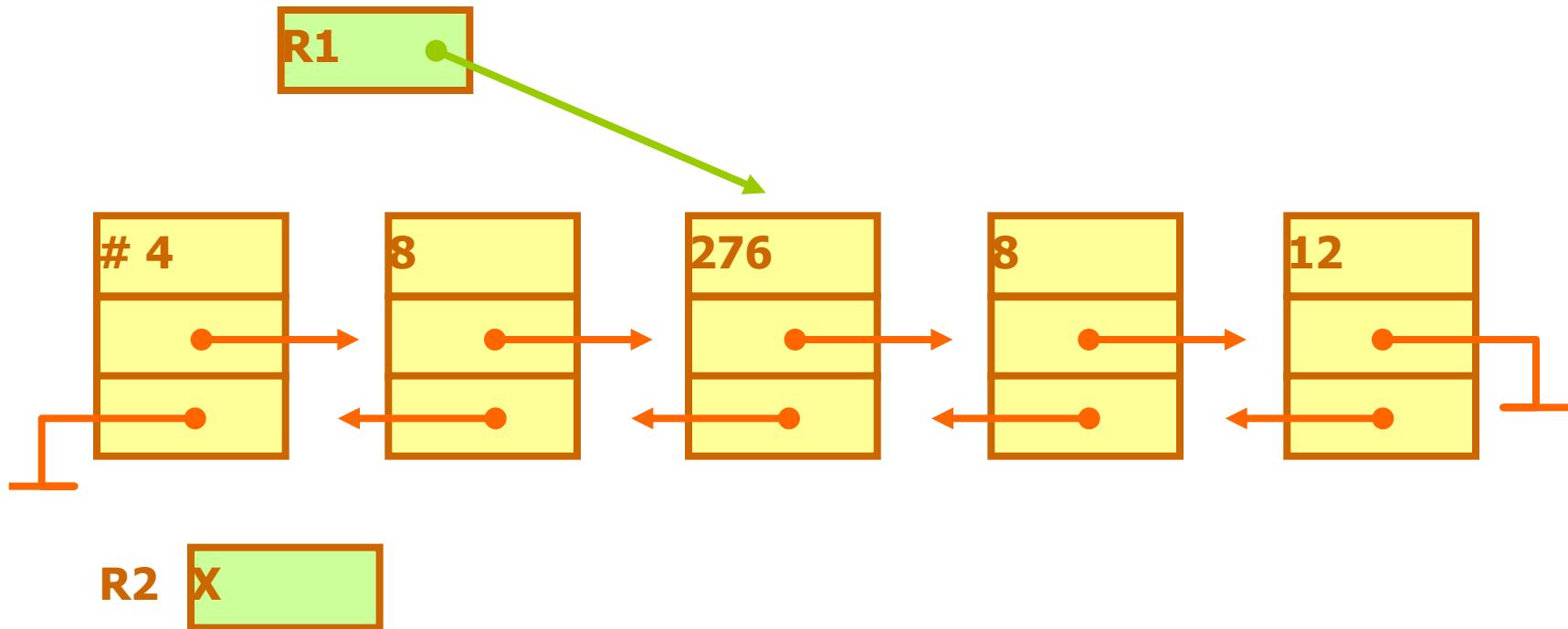
Broj X bit će stalno spremljen u R2



Ostali primjeri

<<< Način rada potprograma:

Pomičemo R1 za jedan
čvor dalje ...

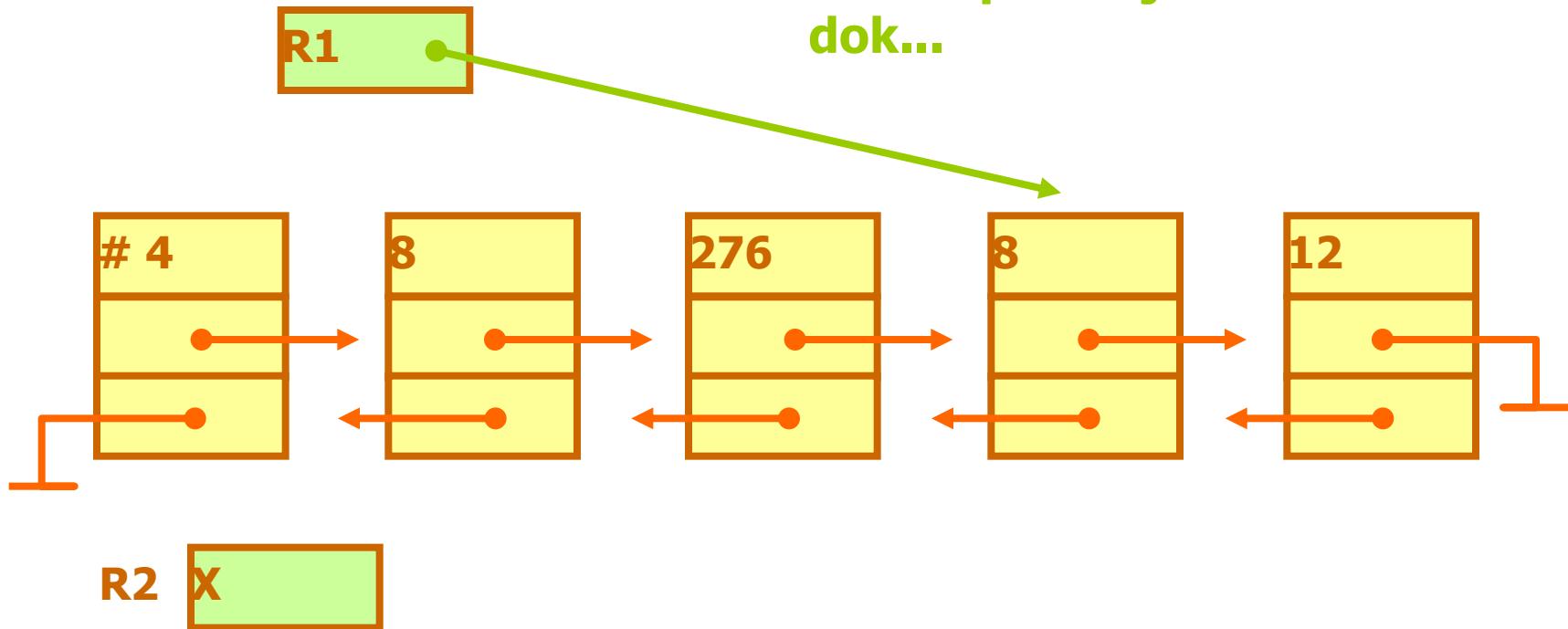




Ostali primjeri

<<< Način rada potprograma:

Pomičemo R1 za jedan
čvor dalje ...
I to se ponavlja sve
dok...

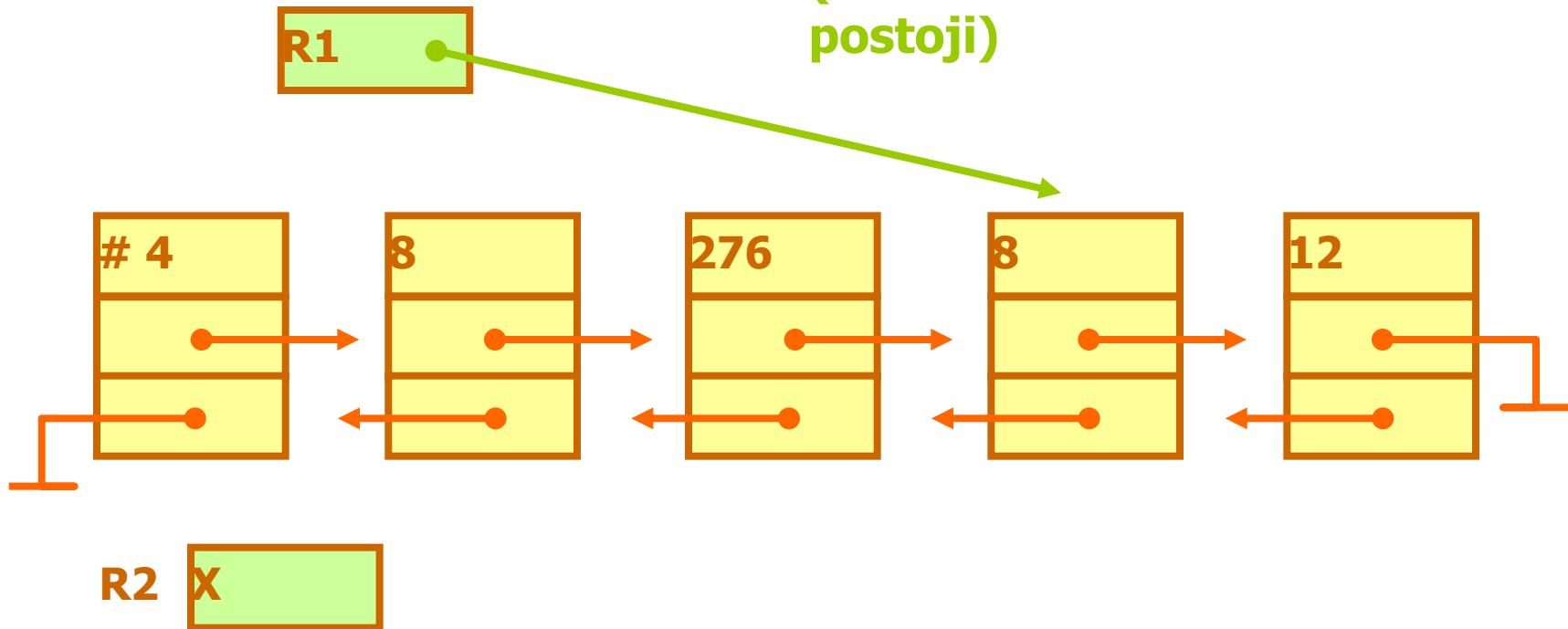




Ostali primjeri

<<< Način rada potprograma:

Ne pronađemo čvor ili
dođemo do kraja liste
(tada traženi čvor ne
postoji)

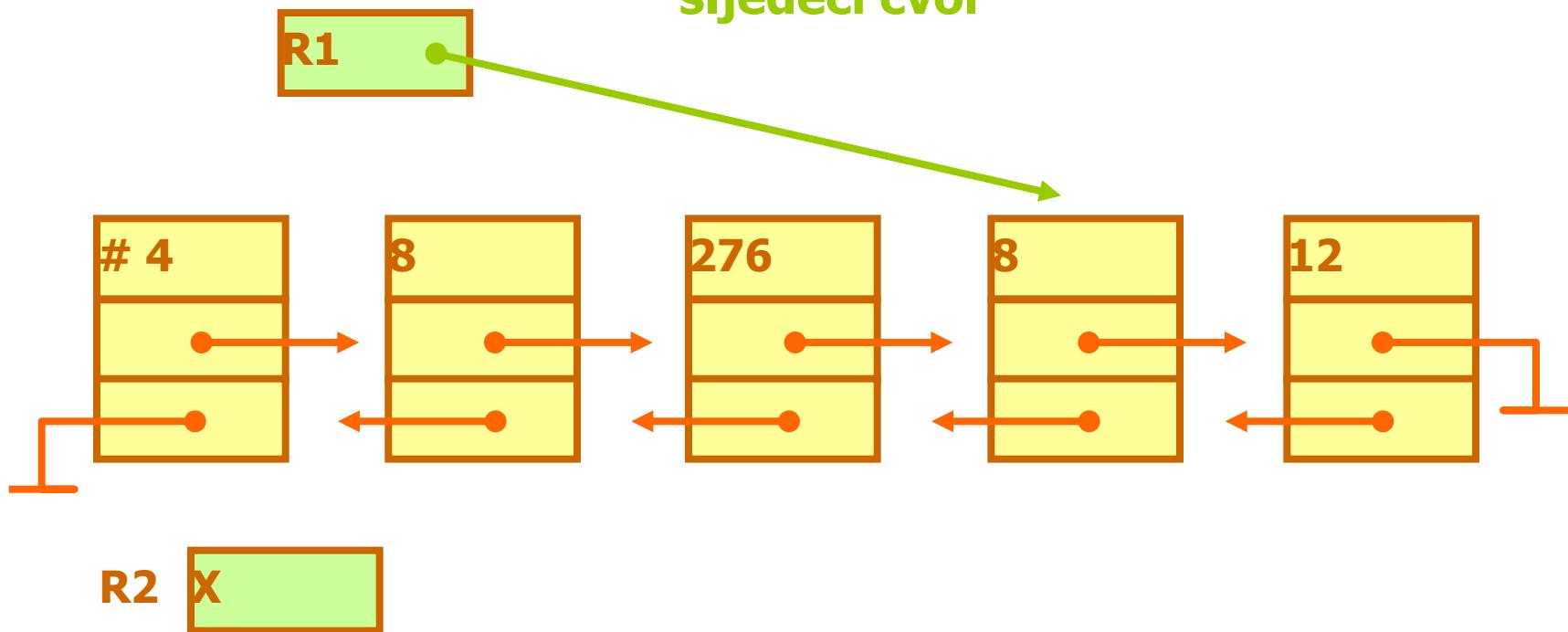




Ostali primjeri

<<< Način rada potprograma:

Ako pronađemo čvor, lako ga izbacujemo, jer u njemu postoje pokazivači i na prethodni i na sljedeći čvor

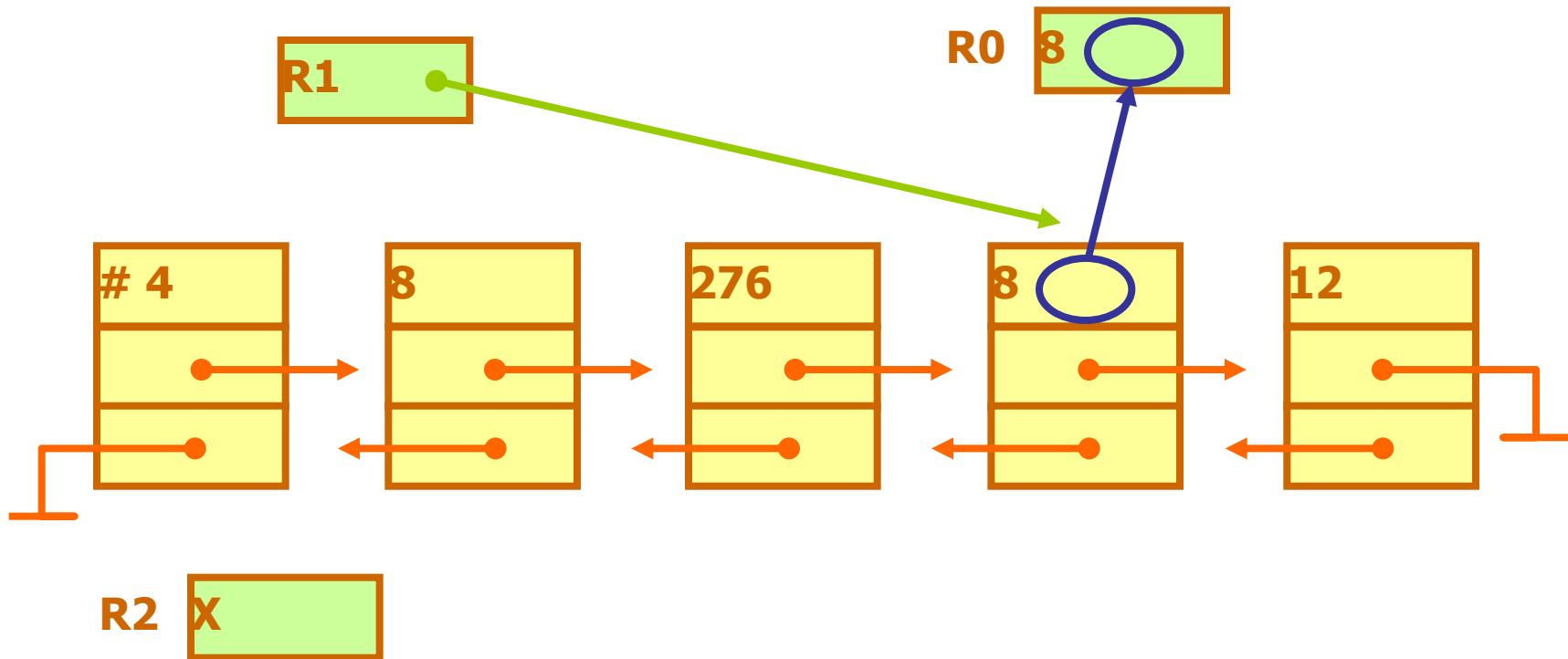




Ostali primjeri

<<< Način rada potprograma:

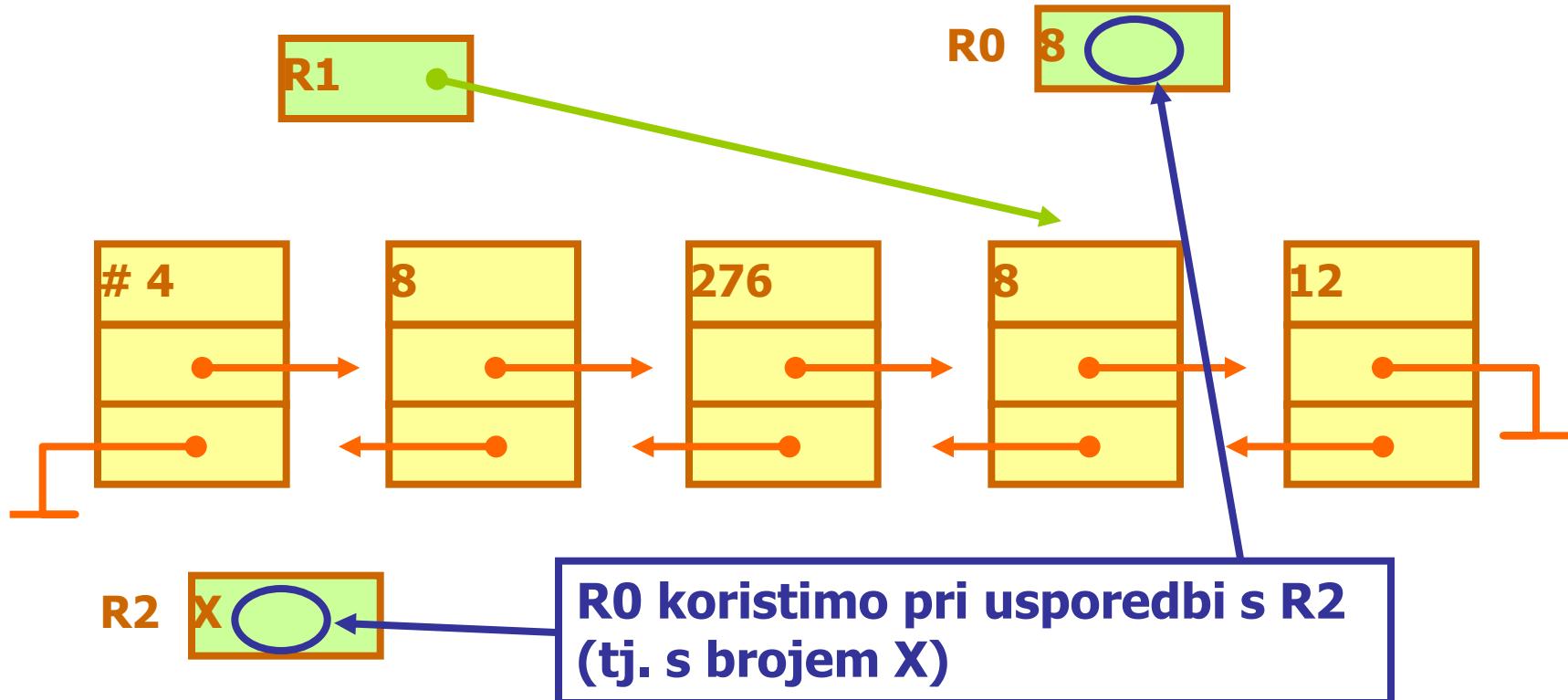
Prilikom traženja, u R0 ćemo učitati vrijednost čvora na kojeg pokazuje R1





Ostali primjeri

<<< Način rada potprograma:

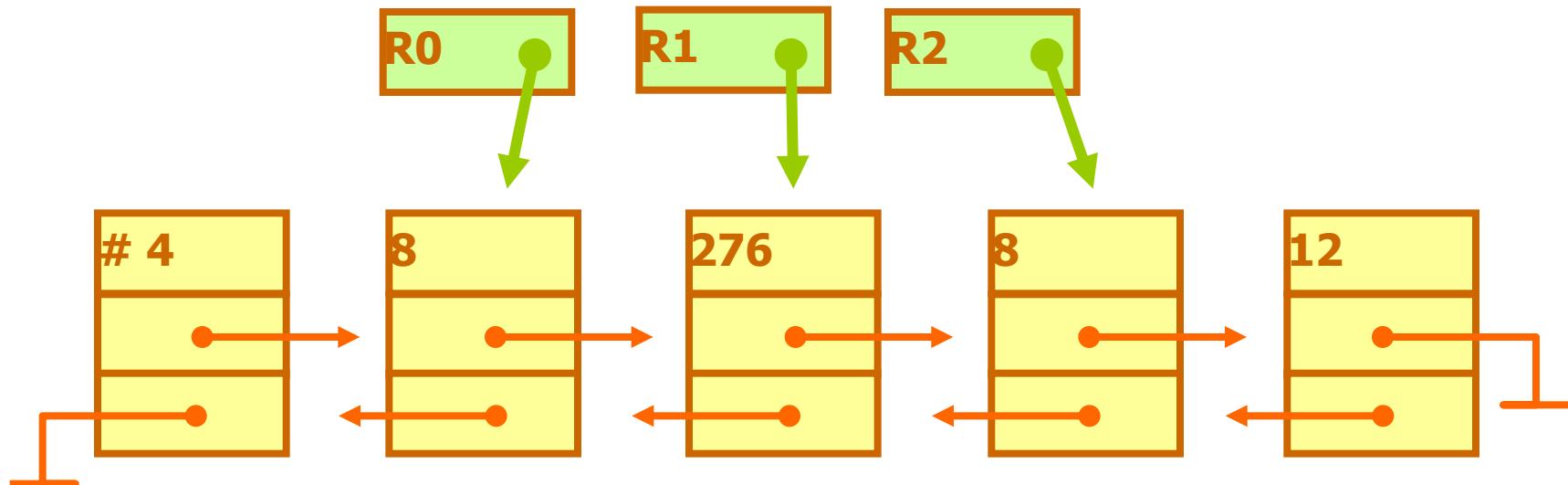




Ostali primjeri

<<< Način rada potprograma:

Kod izbacivanja čvora,
registri se postavljaju ovako
(na ovoj slici izbacuje se čvor 276):

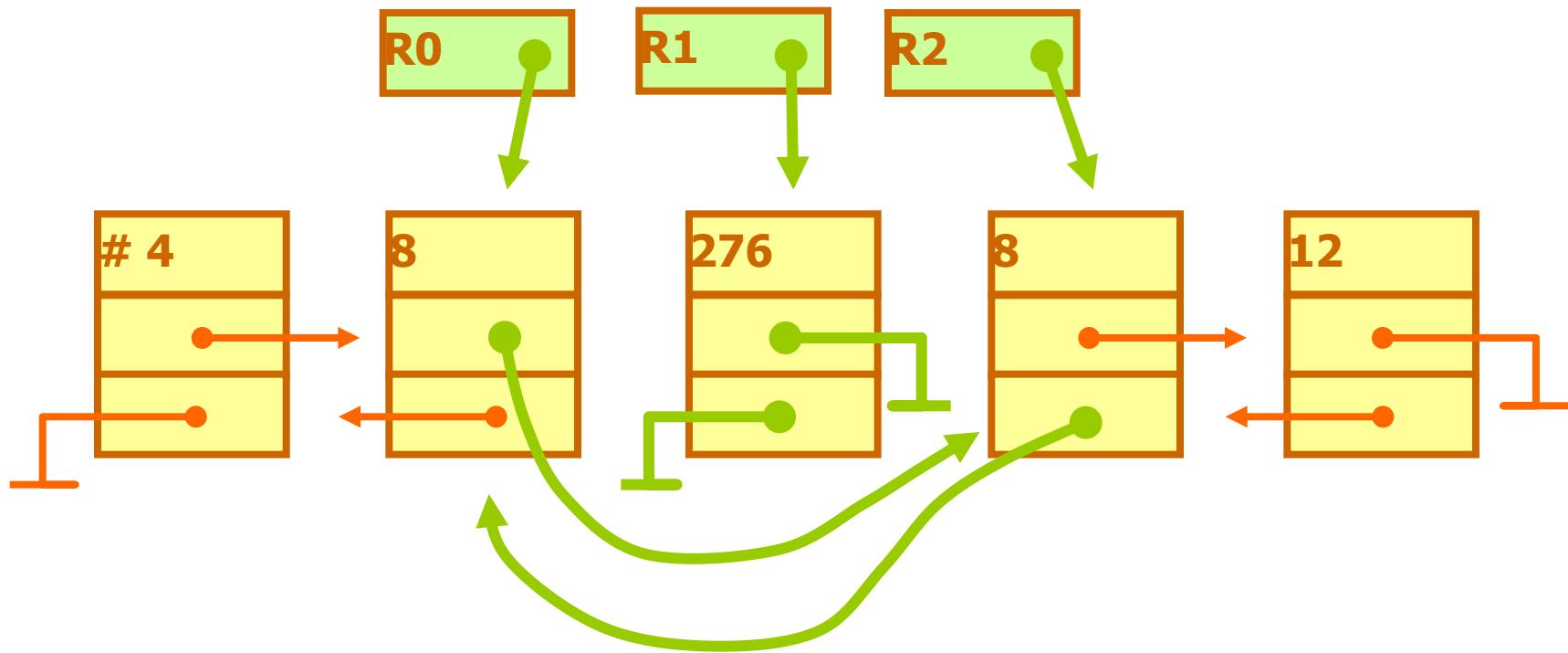




Ostali primjeri

<<< Način rada potprograma:

Pokazivači u čvorovima se
prespajaju ovako:





IZBACI ;;;; Potprogram IZBACI
; Parametri na stogu:
; 1. parametar:
; adresa prvog čvora
; 2. parametar:
; broj X koji se traži

PUSH R0 ; spremi registre
PUSH R2 ; iz glavnog programa

LOAD R2, (SP+0C) ; dohvati broj X
LOAD R1, (SP+10) ; dohvati adresu prvog čvora
LOAD R1, (R1+4) ; pripremi pokazivač za traženje

TRAZI CMP R1, 0
JR_Z IZLAZ ; ako je kraj => čvor nije nađen
LOAD R0, (R1+0) ; dohvati vrijednost čvora
CMP R0, R2 ; usporedi je sa X
JR_EQ NASAO ; ako su isti => čvor je nađen
LOAD R1, (R1+4) ; pomakni se na sljedeći čvor
JR TRAZI ; nastavi s traženjem

okvir

R2	SP+0
R0	SP+4
pov.adr.	SP+8
BROJ X	SP+C
GLAVA	SP+10



<<<

NASAO ;;; odspoji nađeni čvor iz liste

LOAD R0, (R1+8) ; stavi adresu prethodnog u R0
LOAD R2, (R1+4) ; stavi adresu sljedećeg u R2
STORE R2, (R0+4) ; stavi sljedeći iza prethodnog
OR R2, R2, R2 ; provjeri izbacuje li se zadnji
JR_Z DALJE ; čvor iz liste (tj. R2 je NULL)
STORE R0, (R2+8) ; stavi prethodni ispred sljedeć.

DALJE MOVE 0, R0 ; odspoji pokazivače
STORE R0, (R1+4) ; u čvoru kojeg
STORE R0, (R1+8) ; izbacuješ iz liste

;;; smanji brojač čvorova u 1. čvoru liste
LOAD R0, (SP+10) ; dohvati adresu 1. čvora
LOAD R2, (R0+0) ; dohvati broj čvorova liste
SUB R2, 1, R2 ; smanji broj čvorova
STORE R2, (R0+0) ; upiši ga natrag u 1. čvor

IZLAZ POP R2 ; obnovi registre
POP R0 ; iz glavnog programa
RET

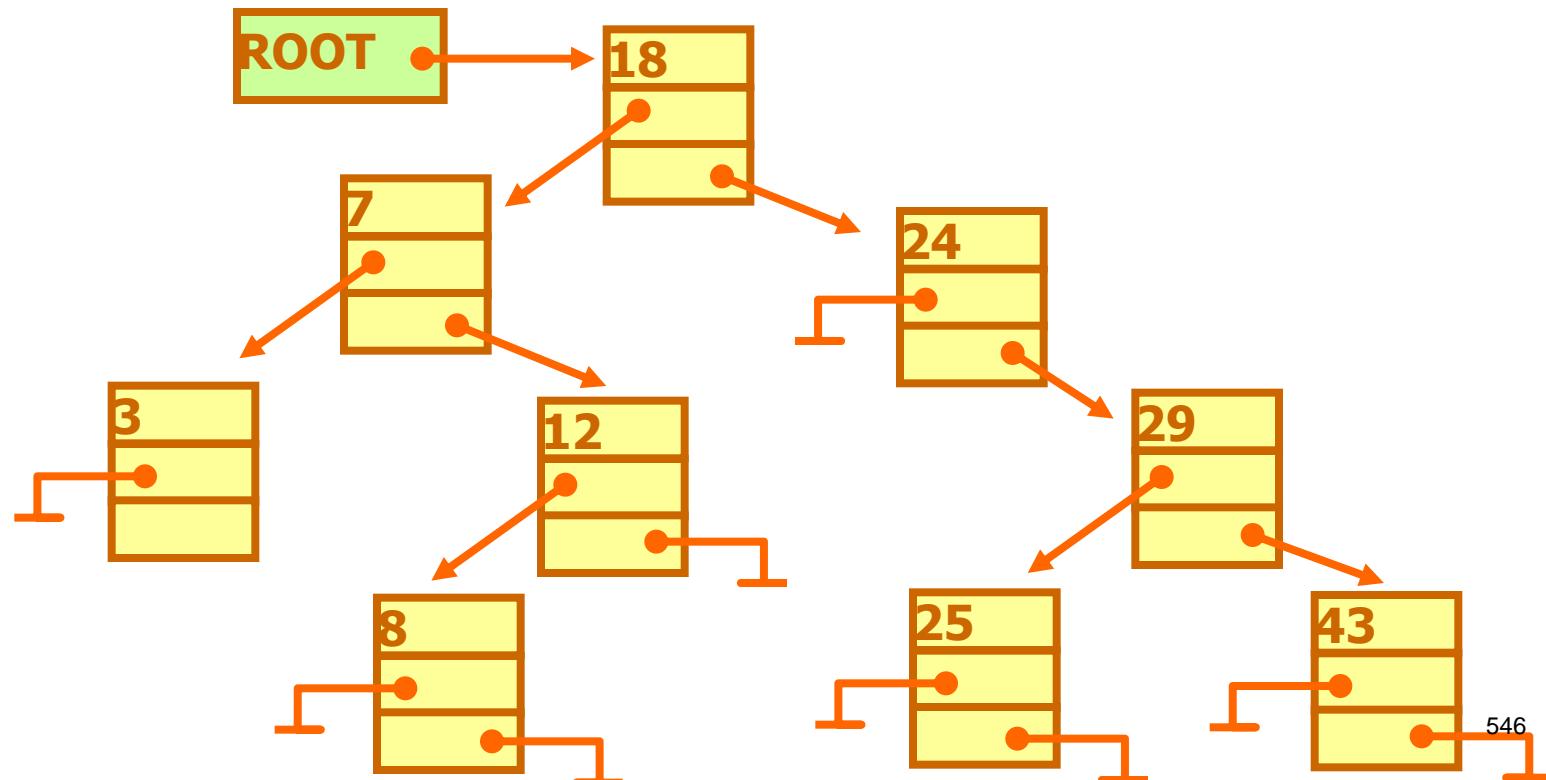


Ostali primjeri

Primjer:

U memoriji se nalazi sortirano binarno stablo. Svaki čvor zauzima tri memorijske lokacije: na prvoj je NBC broj, na drugoj je pokazivač na lijevo podstablo, a na trećoj je pokazivač na desno podstablo.

Poseban pokazivač (ROOT) u glavnom programu pokazuje na korijen stabla.





Ostali primjeri

<<<

Treba napisati potprogram PISI koji će ispisati sve brojeve iz čvorova stabla, ali u rastućem redoslijedu. Parametar potprograma PISI je adresa ishodišnog čvora, a prenosi se stogom. Povratna vrijednost potprograma je broj ispisanih čvorova, a vraća se preko R0.

Ispisivanje se obavlja tako da se pozove potprogram PRINT i kao parametar mu se pošalje broj koji se želi ispisati. Parametar za PRINT šalje se preko R0, a PRINT nema povratne vrijednosti. Prepostavite da potprogram PRINT već postoji negdje u memoriji.

Treba napisati i glavni program, koji će ispisati stablo čija adresa je u pokazivaču ROOT, a broj ispisanih čvorova treba spremiti U R5.

>>>



Ostali primjeri

<<<

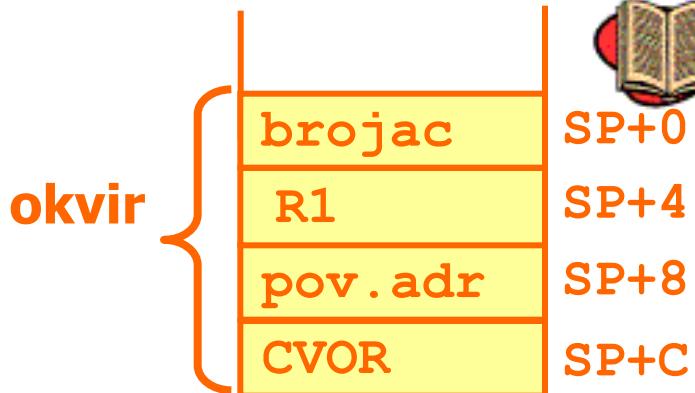
Idejno rješenje (u C-u):

```
int PISI ( struct cvor_stabla * cvor ) {  
    int brojac;  
    if( cvor == NULL )  
        return 0;  
  
    // "in-order" obilazak  
    brojac = PISI( cvor -> lijevi );  
    PRINT( cvor -> broj );  
    brojac += PISI( cvor -> desni );  
  
    return (brojac+1);  
}
```

>>>



```
GLAVNI LOAD R0, (ROOT)
    PUSH R0
    CALL PISI
    ADD SP, 4, SP
    MOVE R0, R5
    HALT
ROOT DW ... ; adresa stabla
```



```
PISI ;;;;; Potprogram PISI
; Parametar na stogu:
; 1. parametar:
;     adresa čvora
```

```
PUSH R1      ; spremi registre

SUB SP, 4, SP ; lokalna varijabla brojac
LOAD R1, (SP+C) ; dohvati adresu čvora
MOVE 0, R0      ; if( cvor == NULL)
CMP R1, 0       ; idi na return
JR_Z VAN
```



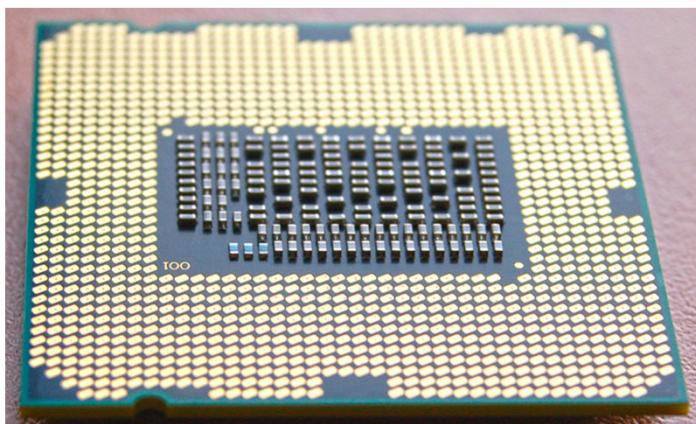
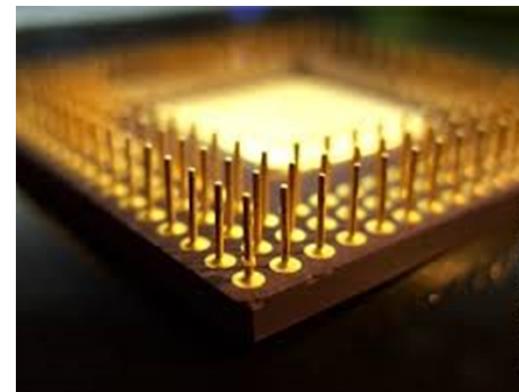
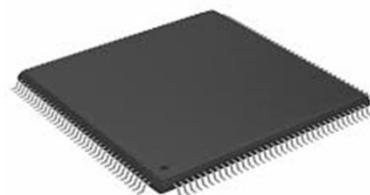
<<<

LOAD	R0 , (R1+4) ; cvor->lijevi	okvir	brojac	SP+0
PUSH	R0		R1	SP+4
CALL	PISI ; PISI()		pov.adr	SP+8
ADD	SP , 4 , SP		CVOR	SP+C
STORE	R0 , (SP+0) ; brojac = ; povratna vrijednost			
LOAD	R0 , (R1) ; cvor->broj			
CALL	PRINT ; PRINT()			
LOAD	R0 , (R1+8) ; cvor->desni			
PUSH	R0			
CALL	PISI ; PISI()			
ADD	SP , 4 , SP			
LOAD	R1 , (SP+0) ; stavi brojac u R1			
ADD	R1 , R0 , R0 ; brojac += povratna vrijednost			
ADD	R0 , 1 , R0 ; stavi (brojac+1) u R0 za return			
	; return			
VAN	ADD SP , 4 , SP ; ukloni lokalnu var.			
POP	R1 ; obnovi registre			
RET				

Priklučci - općenito

Priključci

- Svaki procesor, memorija ili bilo koja druga komponenta ostvarena kao čip ima određen broj priključaka ili izvoda (nazivaju se još i pinovi prema engleskom izvorniku)



Priključci - podjela po namjeni

- Sabirnice se prema namjeni obično dijele na:
 - adresnu sabirnicu (address bus)
 - podatkovnu sabirnicu (data bus)
 - upravljačku sabirnicu (control bus)
- Adresni priključci postavljaju adresu na adresnu sabirnicu, od procesora prema memoriji i vanjskim jedinicama. Procesor, kao aktivna i vodeća komponenta, adresira druge komponente zadajući im adresu s koje želi čitati ili pisati podatak
- Podatkovni priključci spojeni su na podatkovnu sabirnicu i služe za prijenos podataka između procesora i memorije ili između procesora i vanjske jedinice prilikom operacija čitanja i pisanja
- Upravljački priključci imaju razne funkcije vezane uz rad procesora i općenito služe za sinkronizaciju rada pojedinih dijelova računala

Priklučci - podjela po namjeni

- Priklučci koji imaju upravljačku namjenu (možemo promatrati kao da prenose logičko stanje true ili false):
 - mogu biti **aktivni u niskoj razini** i tada se označavaju imenom s potezom. Na primjer IREQ može označavati zahtjev za prekid. Ako je priključak nisko, onda znači da postoji zahtjev za prekid, a u suprotnom ne postoji.
 - mogu biti **aktivni u visokoj razini** i tada se nazivaju imenom bez dodatnih oznaka. Na primjer READ može označavati ciklus čitanja. Ako je priključak visoko, onda se trenutačno obavlja ciklus čitanja, a u suprotnom se ne obavlja ciklus čitanja

>>>

Priklučci - podjela po namjeni

<<<

- mogu označavati jedno od dva moguća stanja i tada u imenu sadrže nazive oba stanja. Na primjer READ/WRITE može označavati da li se trenutačno izvodi ciklus čitanja ili pisanja. Ako je priključak nisko, onda se izvodi čitanje (READ ima potez). Ako je priključak visoko, onda se izvodi ciklus pisanja (WRITE nema potez)
- U pravilu, komponente koje osluškuju upravljačke priključke aktiviraju se na brid signala (tipično u trenutku kad signal prelazi iz neaktivnog u aktivno stanje)
- Kad se ne promatra brid, nego stanje signala, onda se stanje "očitava" u točno definiranim trenutcima

Priklučci - "širina" priključaka

- Po širini priključci mogu biti:
 - Jednostruki priključci su oni koji imaju svoju samostalnu namjenu. To su obično upravljački priključci
 - Priključci grupirani u skupinu po nekoj zajedničkoj funkciji. Na primjer, mogu biti 32 adresna priključka, a pojedinačni se priključci označavaju nazivima ADR0, ADR1, ... ADR31

Priklučci - smjer priključaka

- Priklučci se mogu podijeliti po smjeru signala (podataka) koji njima putuje na:
 - ulazne
 - izlazne
 - dvosmjerne
- Smjer priključka uvijek se definira u odnosu na komponentu koju promatramo

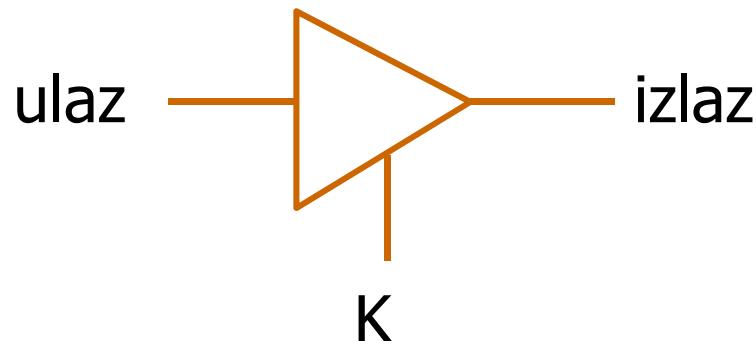
Priklučci - smjer priključaka

- Komponenta upravlja svojim izlaznim priključcima, a druge komponente (spojene preko sabirnice na njih) "osluškuju" njihovo stanje
- Za ulazne priključke komponenta samo "osluškuje" stanje na njima. Sabirnicama koje su povezane na ove priključke upravljaju druge komponente
- Dvosmjerni priključci spojeni su na sabirnicu kojom u različitim trenutcima upravljaju različite komponente. Pri tome **uvijek samo jedna komponenta upravlja sabirnicom u nekom trenutku**, a priključci svih ostalih komponenata su ili **ulazni** ili u **stanju visoke impedancije**



Priklučci - smjer priključaka

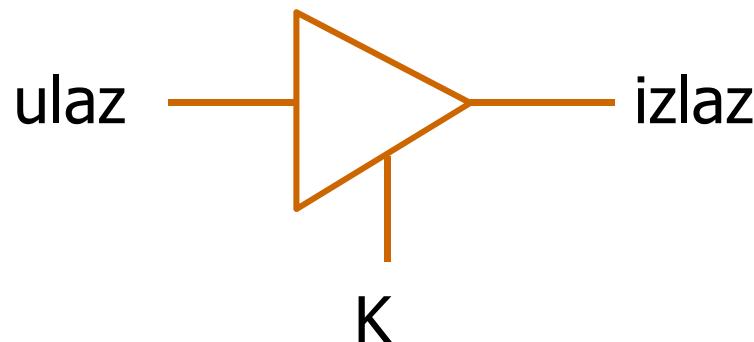
- Stanje visoke impedancije omogućuje jednostavno spajanje više komponenata na istu sabirnicu (**ponoviti iz "Digitalne"**)
- Sklopove s tri stanja simbolički prikazujemo ovako:





Priključci - smjer priključaka

- Ako je upravljački ulaz K neaktivan (0), onda je izlaz u stanju visoke impedancije (high Z)
 - Ako je upravljački ulaz K aktivan (1), onda je stanje ulaza prenosi na izlaz



K	ulaz	izlaz
0	0	X
0	1	X
1	0	0
1	1	1



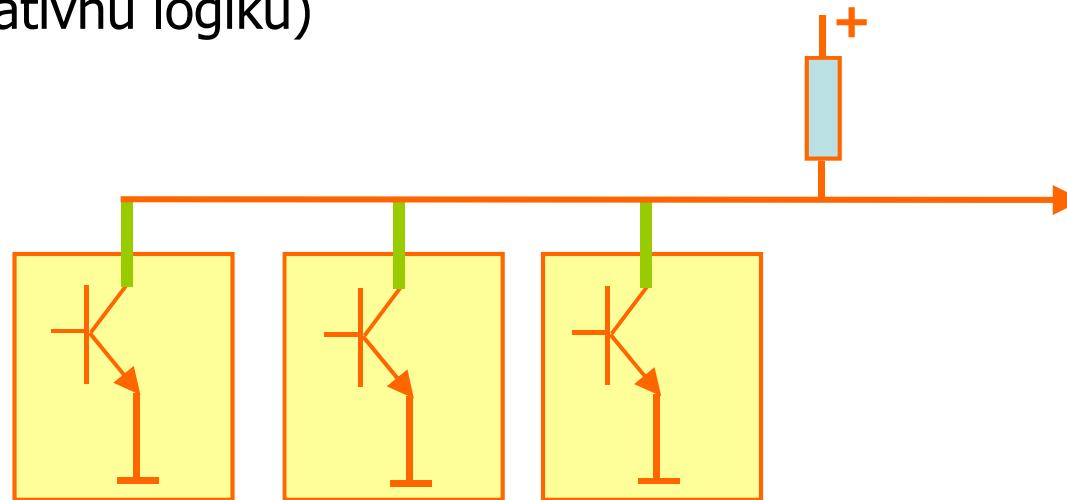
Priklučci - smjer priključaka

- Sklop s tri stanja omogućuje spajanje više priključaka na istu sabirnicu
- Sabirnicom smije upravljati najviše jedan priključak i on određuje stanje sabirnice (0 ili 1)
- Svi ostali priključci moraju biti neaktivni i oni ne utječu na stanje na sabirnici



Priklučci - smjer priključaka

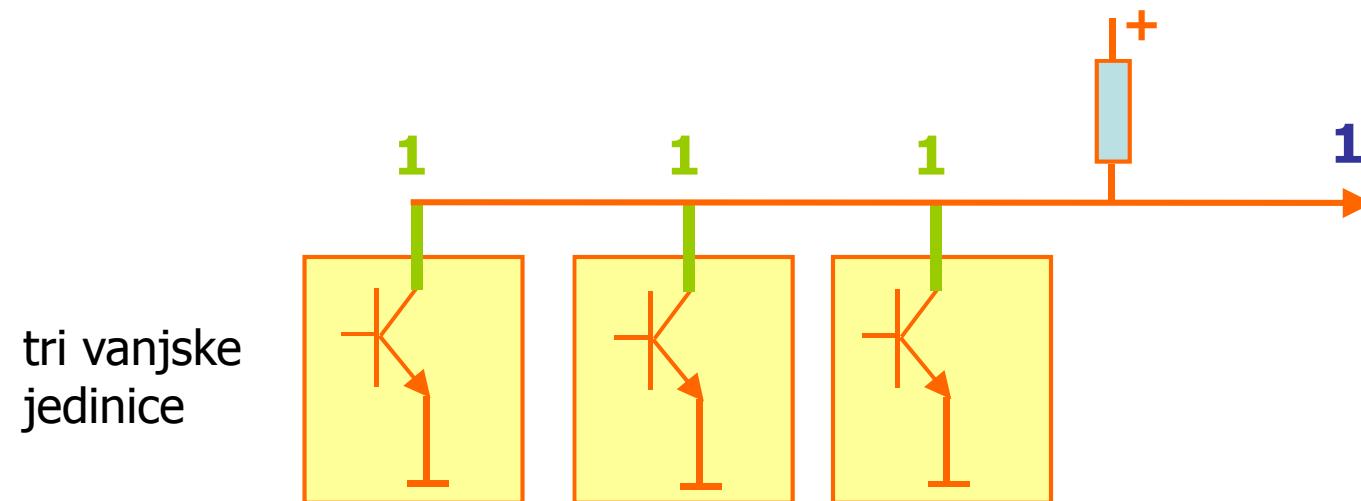
- Postoje i posebne vrste priključaka. To su tzv. *open collector* priključci (**ponoviti iz "Digitalne"**)
 - oni omogućuju da se **više izlaznih priključaka** spoji zajedno na jednu sabirnicu i da **svi zajedno njome upravljaju** u istom trenutku
 - stanje sabrnice određeno je logičkom funkcijom spojeni-I (wired-AND) između svih priključaka (naziva se još i wired-OR - za negativnu logiku)





Priklučci - smjer priključaka

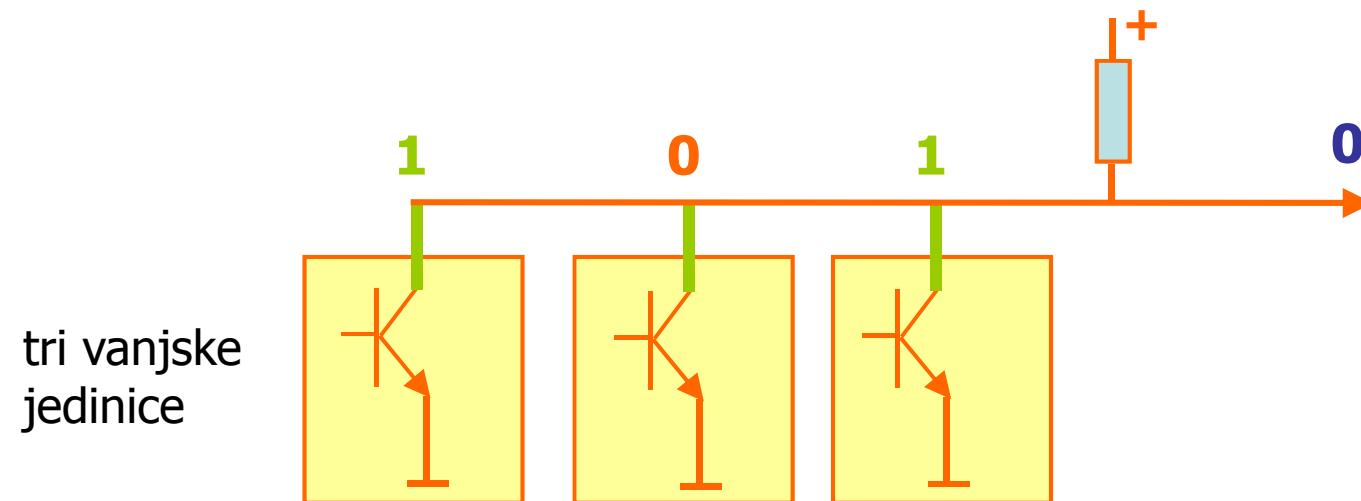
- Ako su svi prekidni priključci **neaktivni** (u visokoj su razini), onda je cijela sabirnica **neaktivna** (u visokoj je razini)





Priklučci - smjer priključaka

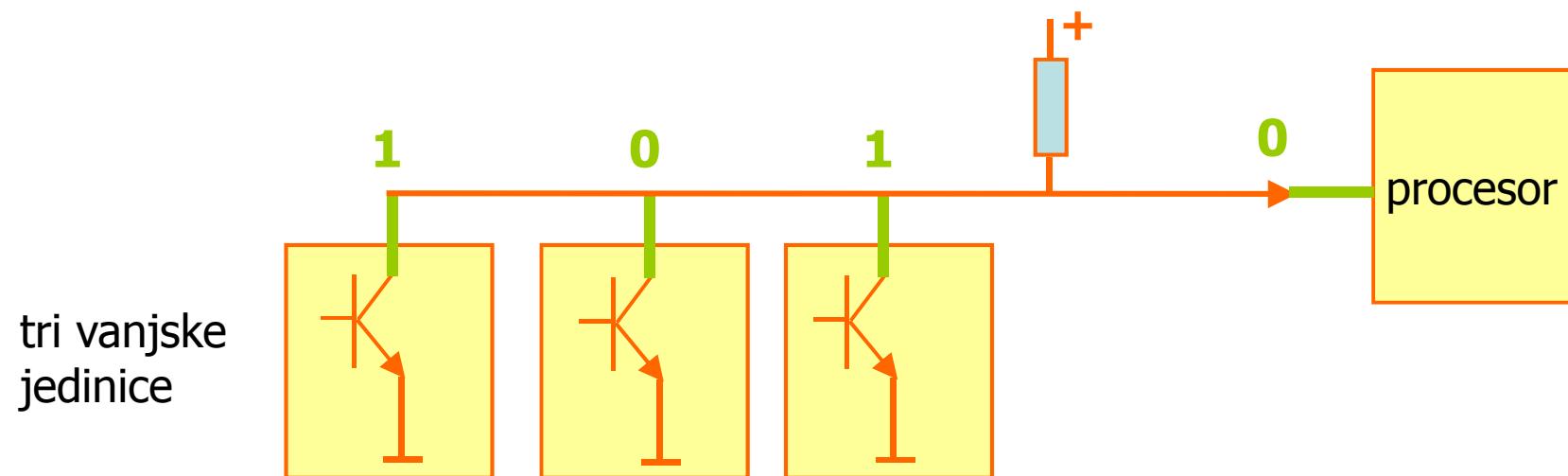
- Ako se bilo koji prekidni priključak **aktivira** (aktivna razina je u niskom), onda se **aktivira** cijela sabirnica, tj. prelazi u nisku razinu





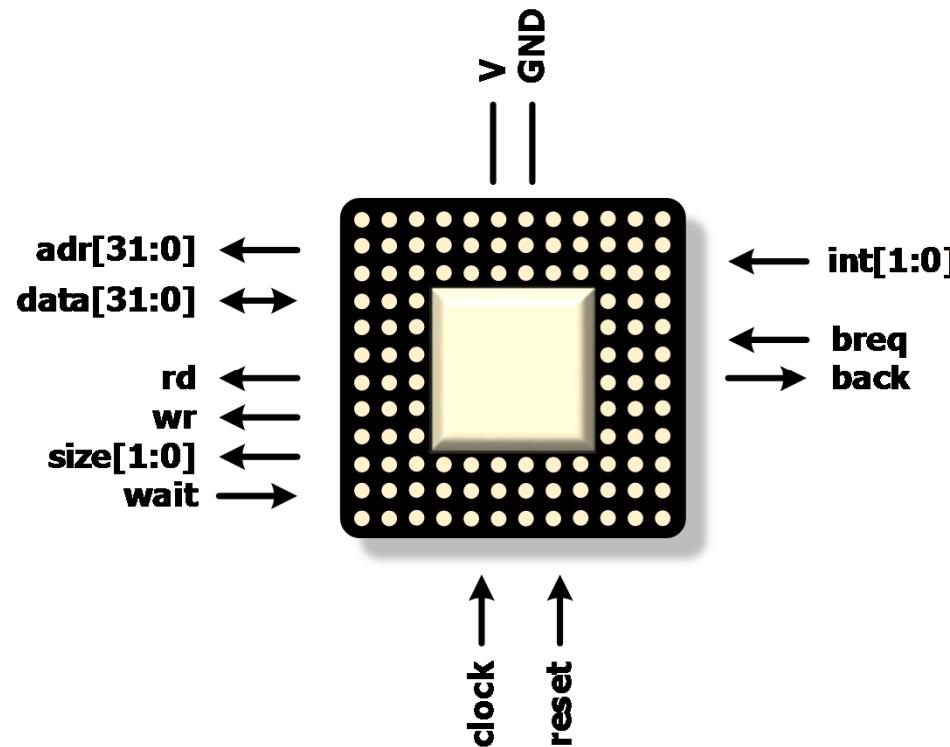
Priklučci - smjer priključaka

- *Open collector* priključci se koriste, npr. za spajanje prekidnih priključaka vanjskih jedinica na jednu sabirnicu
- Na ovaj način procesor koji osluškuje prekidnu liniju može detektirati da je netko postavio zahtjev za prekid



Priklučci procesora FRISC

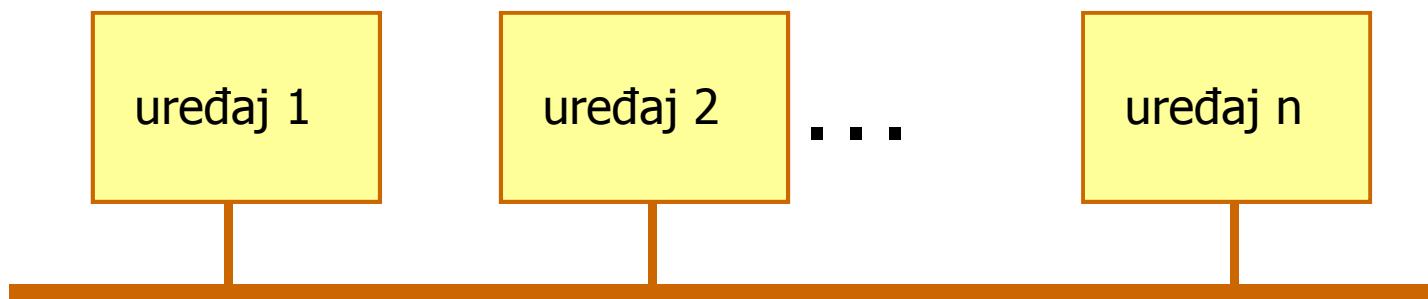
Priključci procesora FRISC



Sabirnice - Osnovno

Sabirnice

- Sabirnica (engl. bus) je spojni put koji povezuje više uređaja (tj. dijelova računalnog sustava), a sastoji se od skupa vodiča

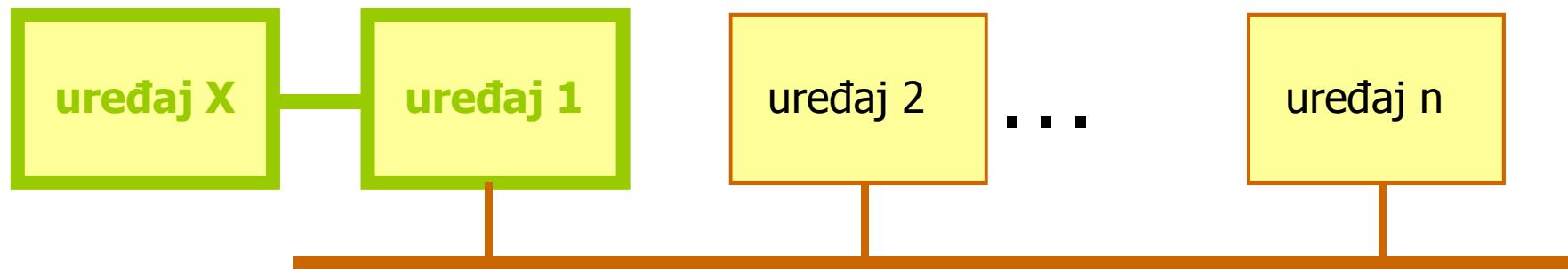


Sabirnice

- Prednosti sabirnice:
 - mala cijena (isti spojni put dijeli više uređaja)
 - prilagodljivost prilikom projektiranja i nadogradnje računala (jednostavno dodavanje uređaja)
 - standardiziranost
- Nedostatci sabirnice:
 - mala propusnost
 - ograničena duljina sabirnice
 - ograničen broj uređaja koji se mogu spojiti na jednu sabirnicu
 - problemi zbog uređaja različite brzine

Sabirnice

- Alternativa sabirnici je **spajanje dva uređaja** (point-to-point) pomoću vlastitog spojnog puta prilagođenog upravo tim uređajima
 - Prednost je veća brzina komunikacije
 - Nedostatak je veća cijena (više spojnih putova, više priključaka na čipu)



Sabirnice - memorjske i UI

- Sabirnice se mogu dijeliti i na:
 - memorjsku sabirnicu
 - ulazno-izlaznu (U/I) sabirnicu
 - sabirnice specijalne namjene (npr. grafička)
- **Memorijska sabirница:**
 - povezuje procesor i memoriju
 - male duljine
 - velike brzine rada
 - prilagođena brzini memorije

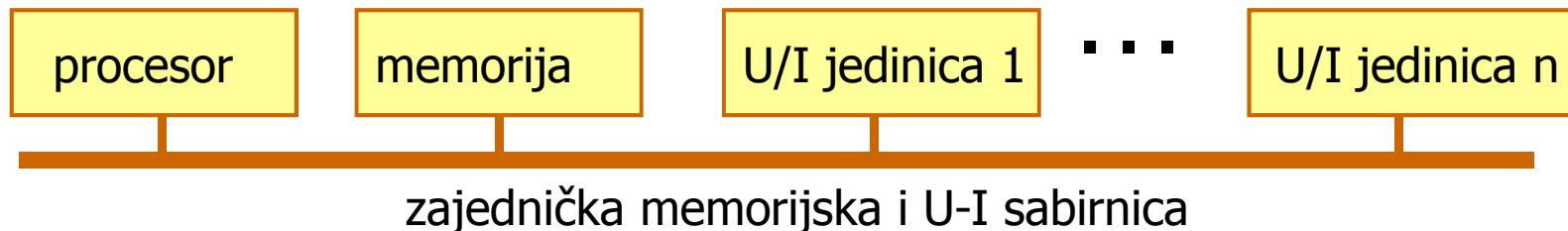
Sabirnice - memorijske i UI

- **U-I sabirnica:**
 - povezuje U-I jedinice s procesorom
 - velika duljina (npr. USB 2.0 - do 5 metara)
 - manja brzina rada nego memorijska sabirnica
 - prilagodljivost različitim brzinama rada pojedinih U-I jedinica
 - mogućnost spajanja velikog broja U-I jedinica (npr. 128)
 - U-I sabirnica se spaja na procesor i memoriju na dva načina (vidi sljedeći slajd)

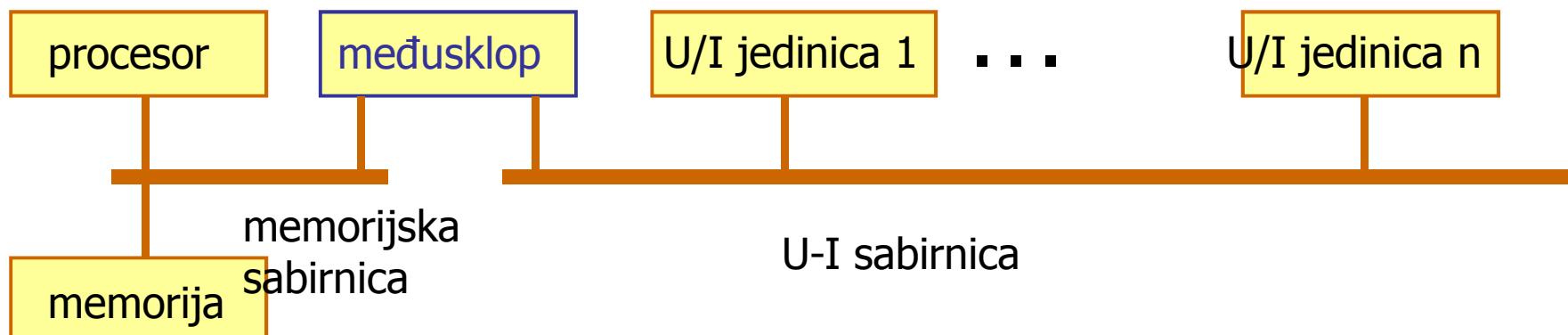
>>>

Sabirnice - memorijske i UI

- Zajednička memorijska i U-I sabirnica (backplane bus)



- Spajanje memorijske i UI sabirnice pomoću posebnog međusklopa (tj. neizravno)

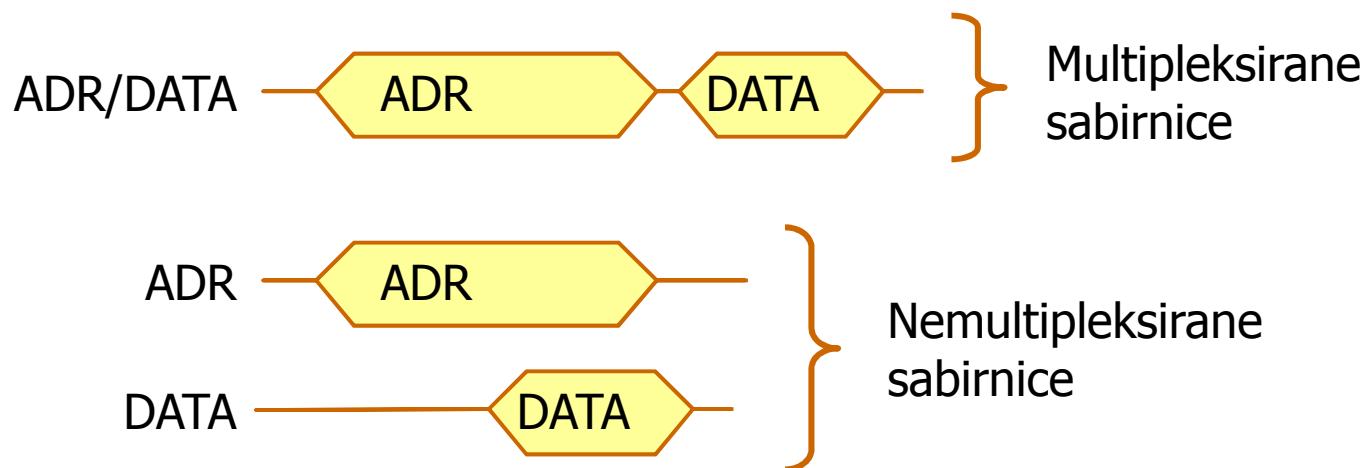


Multipleksirane sabirnice

- Kod procesora vrlo niske cijene zbog uštede
- Da bi mogli imati sabirnice većih širina (npr, podatkovna i adresna), koristile su se tzv. **multipleksirane sabirnice** što znači da **isti spojni putovi imaju više namjena, ali ne u isto vrijeme**

Multipleksirane sabirnice

- Na primjer, adresna i podatkovna sabirnica mogu dijeliti iste spojne putove i to tako da:
 - se u jednom trenutku prenosi adresa
 - a u drugom trenutku se prenosi podatak
- **Vremenski** ove funkcije moraju biti jasno **odijeljene** jer se ne mogu događati istodobno

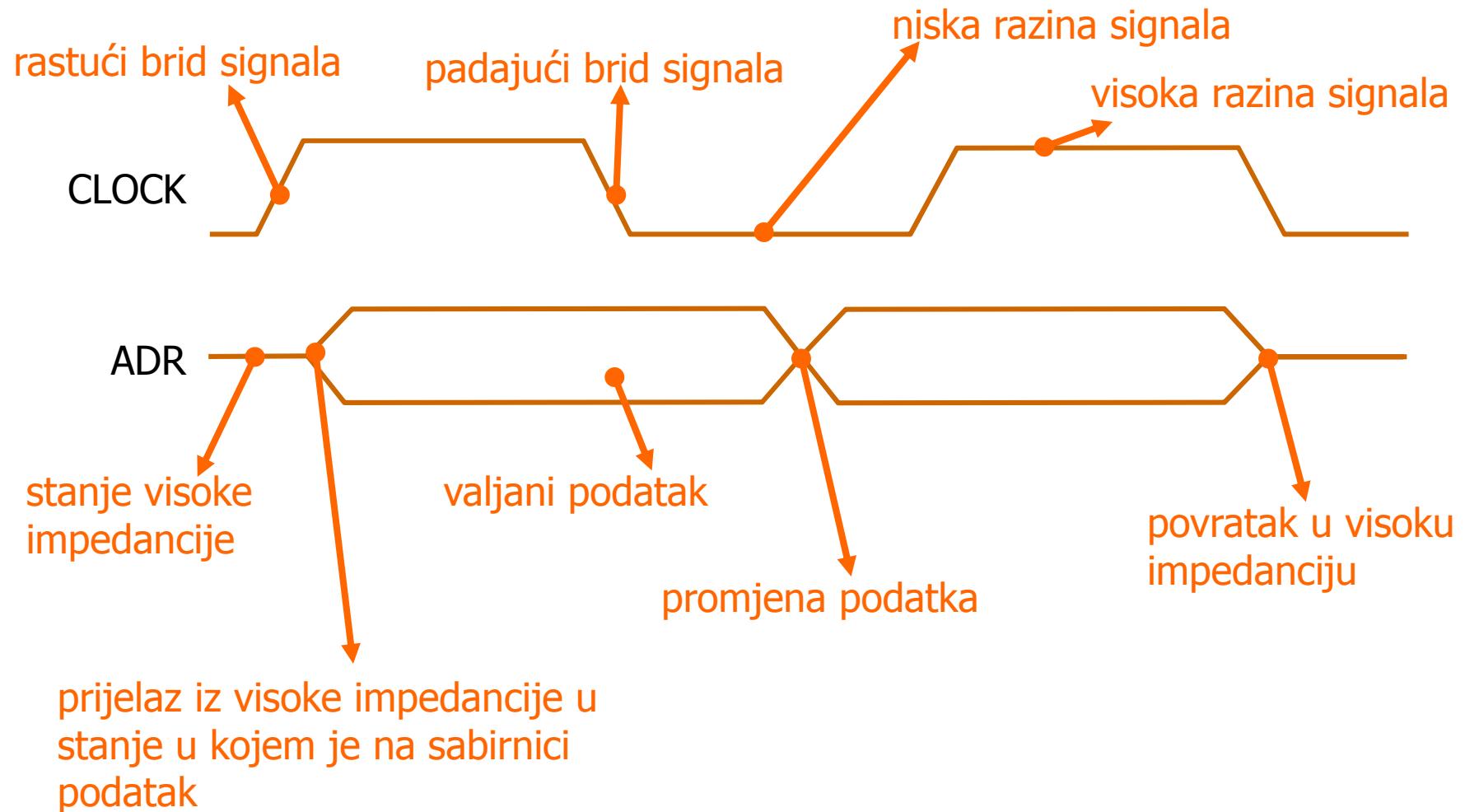


Sabirnice - sabirnički protokoli

Sabirnice - sabirnički protokoli

- Točan redoslijed svih koraka u komunikaciji naziva se **sabirnički protokol** (bus protocol)
- Sabirnička transakcija (bus transaction) je slijed koraka potrebnih da bi se na sabirnici izvela određena operacija, kao npr. operacija čitanja ili pisanja
- Sabirnička transakcija obično sadrži:
 - zahtjev (request)
 - odgovor (response)
- Unutar zahtjeva i/ili odgovora može se nalaziti podatak, adresa, naredba i sl.

Tumačenje oznaka na vrem. dijagramu:

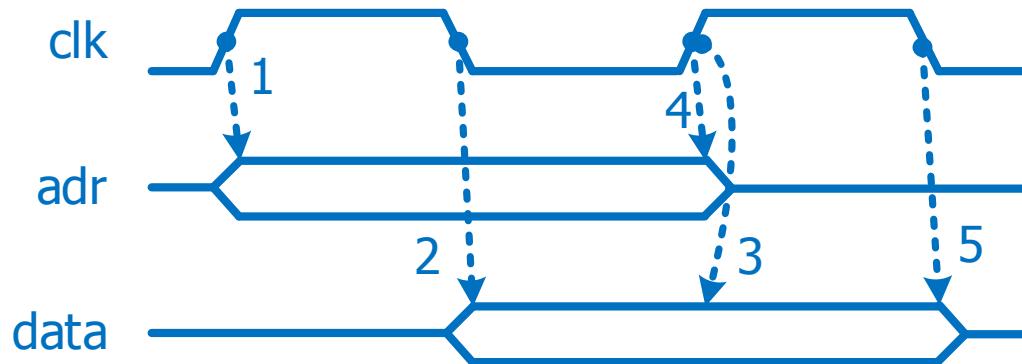


Sabirnice - sinkrone i asinkrone

- Sabirnice se prema načinu komunikacije dijele na*:
 - sinkrone
 - asinkrone
- **Sinkrone sabirnice:**
 - **sve operacije su sinkronizirane s taktom sustava (tj. clockom)**
 - jednostavne su za implementaciju
 - imaju veliku brzinu rada pa zato i malu duljinu
 - bolje su prilagođene za slučaj kad svi uređaji imaju jednaku brzinu
 - imaju mogućnost prilagodbe brzine rada, ali se komunikacija većinom odvija predviđenom brzinom
 - češće se koriste za memorijske sabirnice

Sabirnice - sinkrone

- Pojednostavljeni prikaz sinkrone komunikacije:



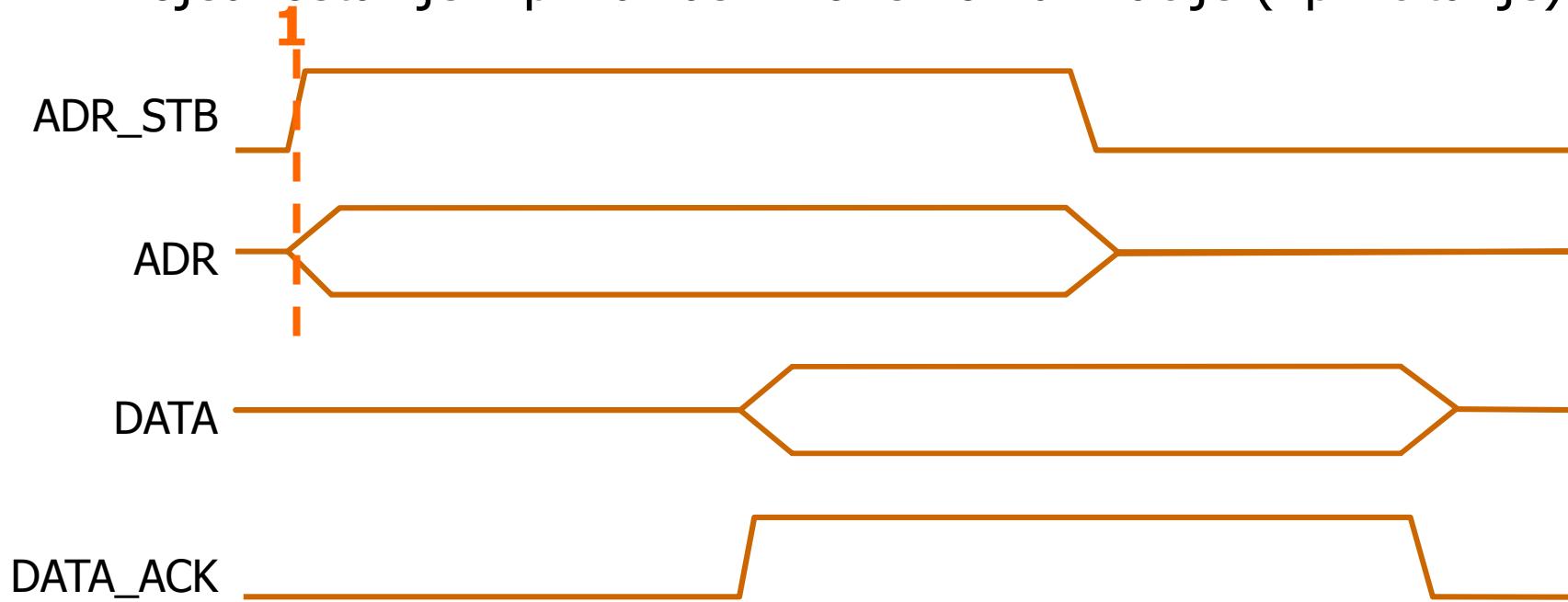
Sve operacije postavljanja adresa, čitanja podataka i
uklanjanja adresa i podataka odvijaju se u vremenskim
trenutcima **točno definiranim u odnosu na CLOCK**

Sabirnice - asinkrone

- **Asinkrone sabirnice:**
 - ne koriste CLOCK za sinkronizaciju
 - **uređaji se sinkroniziraju tzv. rukovanjem** (engl. handshaking protocol)
 - rukovanje je postupak u kojem strane koje komuniciraju prelaze na sljedeći korak komunikacije tek kad obje strane potvrde da je prethodni korak dovršen
 - složenije su za implementaciju
 - imaju manju brzinu rada
 - mogu imati veliku duljinu
 - bolje su prilagođene za slučaj kad uređaji imaju različite brzine
 - češće se koriste za U-I sabirnice

Sabirnice - asinkrone

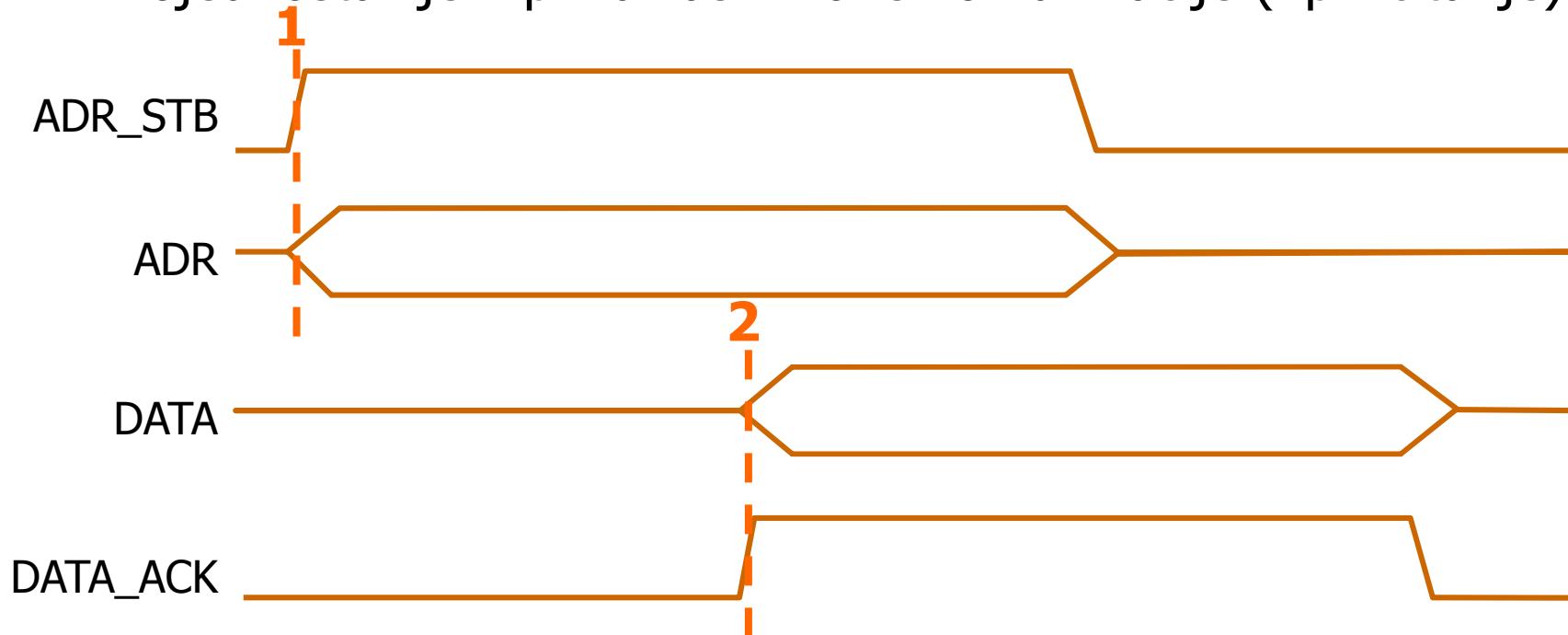
- Pojednostavljeni prikaz asinkrone komunikacije (npr. čitanje):



- 1) Procesor postavlja adresu na ADR i dojavljuje to memoriji aktiviranjem signala **ADR_STB** (address strobe)

Sabirnice - asinkrone

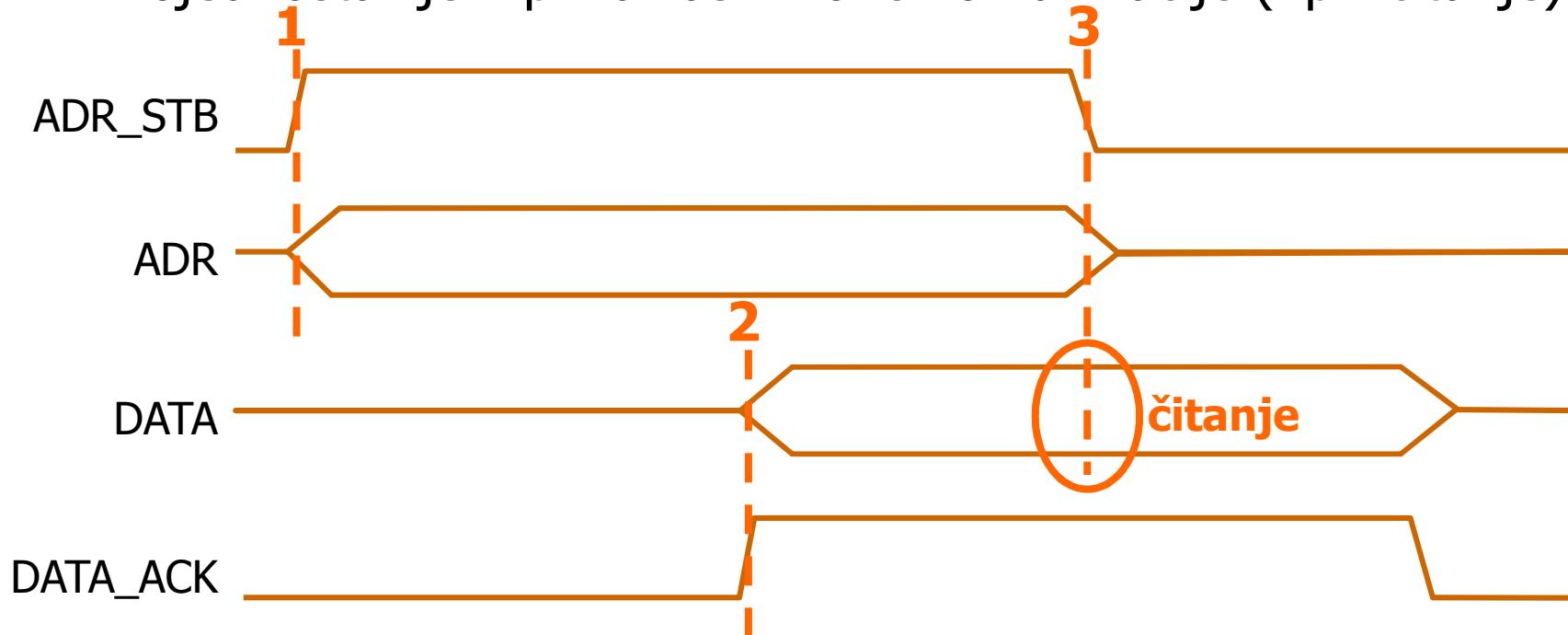
- Pojednostavljeni prikaz asinkrone komunikacije (npr. čitanje):



- Memorija je detektirala ADR_STB i započela operaciju čitanja; nakon nekog vremena postavlja podatak na DATA i dojavljuje to procesoru aktiviranjem DATA_ACK (acknowledge)

Sabirnice - asinkrone

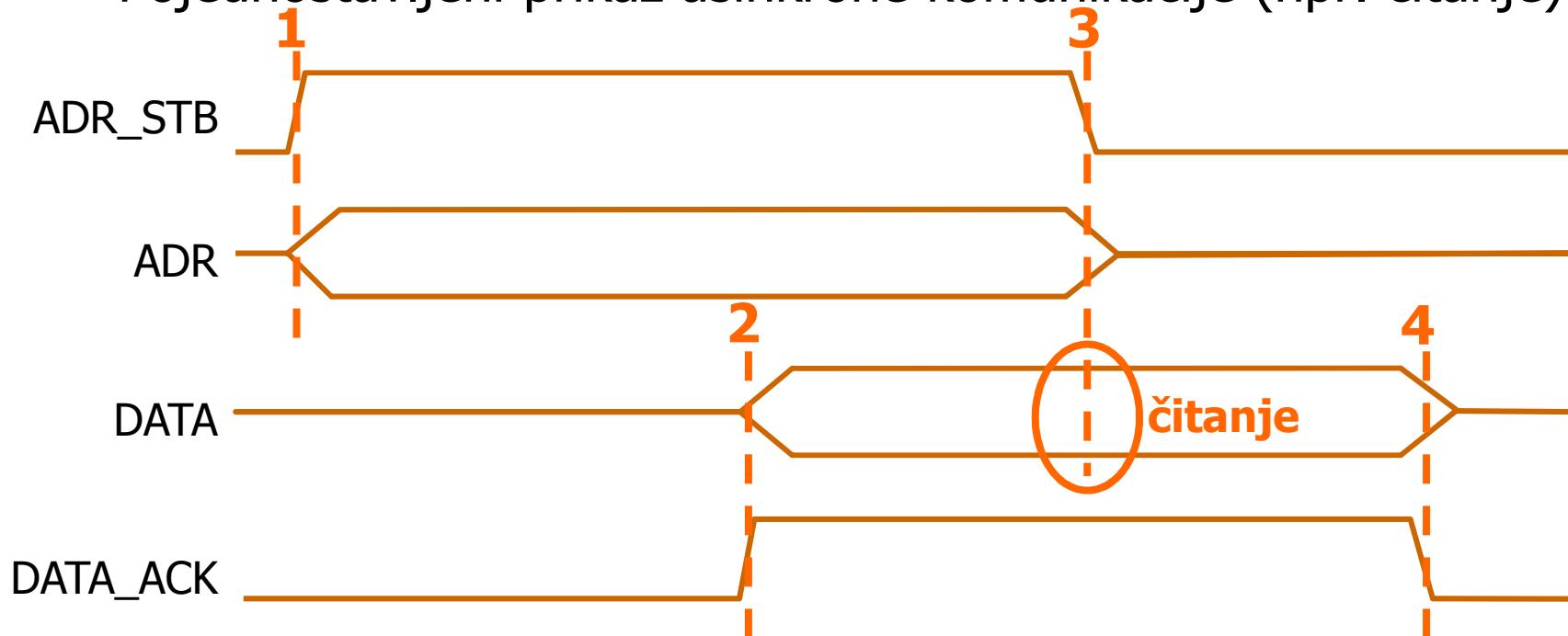
- Pojednostavljeni prikaz asinkrone komunikacije (npr. čitanje):



- 3) Procesor je detektirao DATA_ACK i zna da se na DATA nalazi podatak; procesor čita podatak sa DATA te uklanja adresu sa ADR i dojavljuje to memoriji deaktiviranjem ADR_STB

Sabirnice - asinkrone

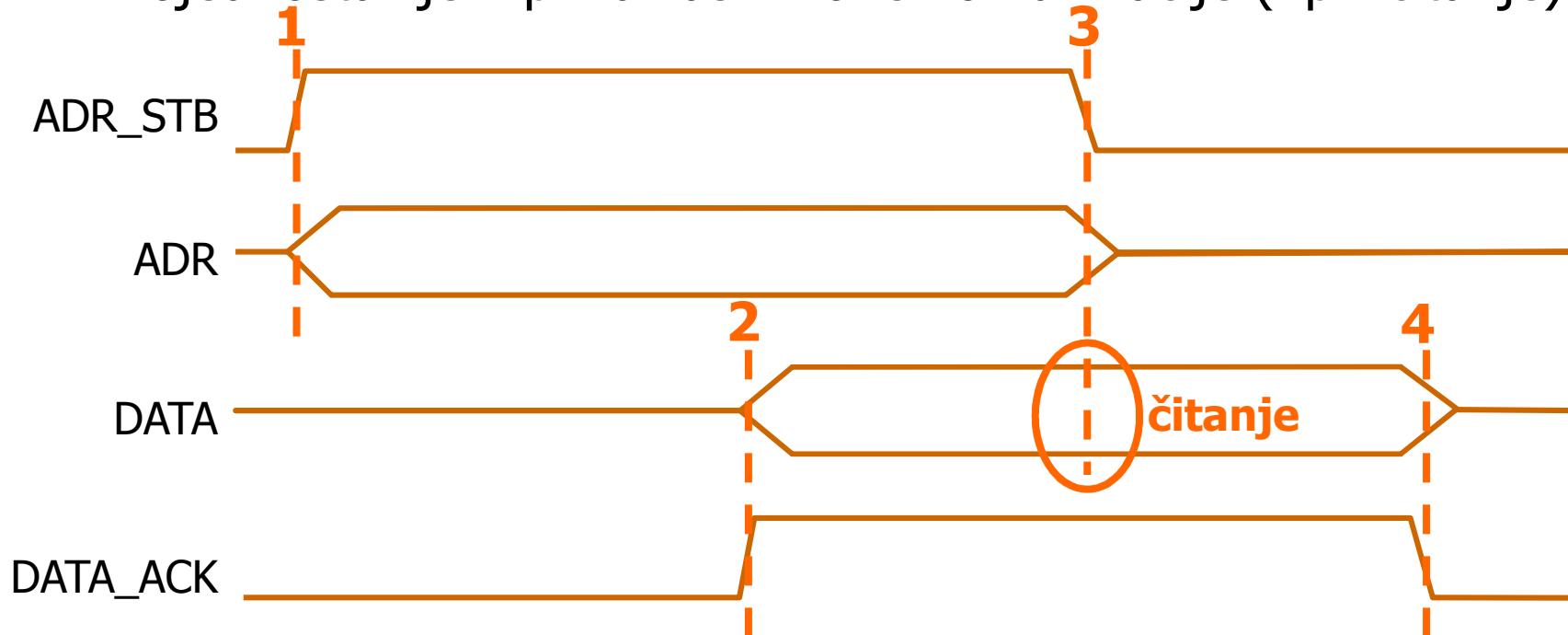
- Pojednostavljeni prikaz asinkrone komunikacije (npr. čitanje):



- 4) Memorija je detektirala deaktiviranje ADR_STB i zna da je procesor pročitao podatak; memorija uklanja podatak sa DATA i javlja to procesoru deaktiviranjem DATA_ACK

Sabirnice - asinkrone

- Pojednostavljeni prikaz asinkrone komunikacije (npr. čitanje):



Nakon koraka 4) procesor detektira deaktiviranje DATA_ACK i može započeti novo čitanje ili pisanje podatka, tj. može ponovno započeti s korakom 1)

Sabirnice - sinkrone i asinkrone

- Komentari:
 - I sinkrone i asinkrone sabirnice imaju mogućnost prilagodbe različitim brzinama uređaja spojenih na sabirnicu. Osnovna razlika je:
 - u tome što su asinkrone sabirnice upravo predviđene za spajanje uređaja različite brzine
 - spajanje uređaja različite brzine je više iznimka nego pravilo kod sinkronih sabirnica, ili takvi uređaji sudjeluju u manjem postotku sabirničkih transakcija
 - Asinkroni protokoli trebaju potvrdu da bi mogli nastaviti sa sljedećim koracima bez obzira koliko je vremena proteklo
 - Sinkroni protokoli nastavljaju s radom automatski kad se dosegne određeno stanje CLOCK-a, a usporenje rada se mora izričito zahtijevati

Sabirnice - sinkrone i asinkrone

- Komentari:
 - Neke memorijске sabirnice koriste asinkrone protokole
 - Međutim, to ne znači da te sabirnice nemaju CLOCK ili da ga uopće ne koriste
 - CLOCK se koristi u procesoru za njegov interni rad
 - Budući da interni rad procesora ovisi o njegovoj komunikaciji preko sabirnica, onda su i na asinkronoj sabirnici barem neki koraci (početni) sinkronizirani sa signalom CLOCK

Sabirnice procesora FRISC

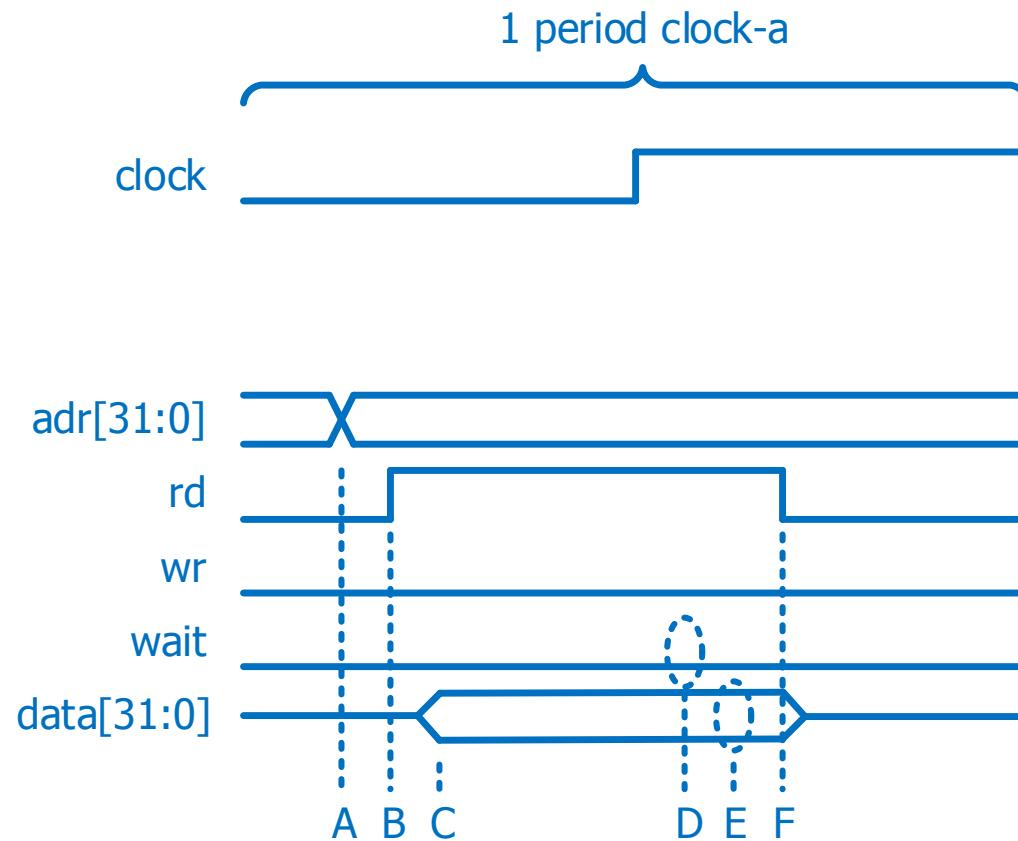
Sabirnice - FRISC

- Za FRISC odabiremo:
 - neće se koristiti posebni spojni putovi između raznih dijelova sustava
 - zajednička memorijska i U-I sabirnica (tzv. backplane)
 - sinkrona sabirnica s mogućnošću prilagodbe brzine
 - brzina se prilagođava umetanjem tzv. ciklusa čekanja

Sabirnice - FRISC

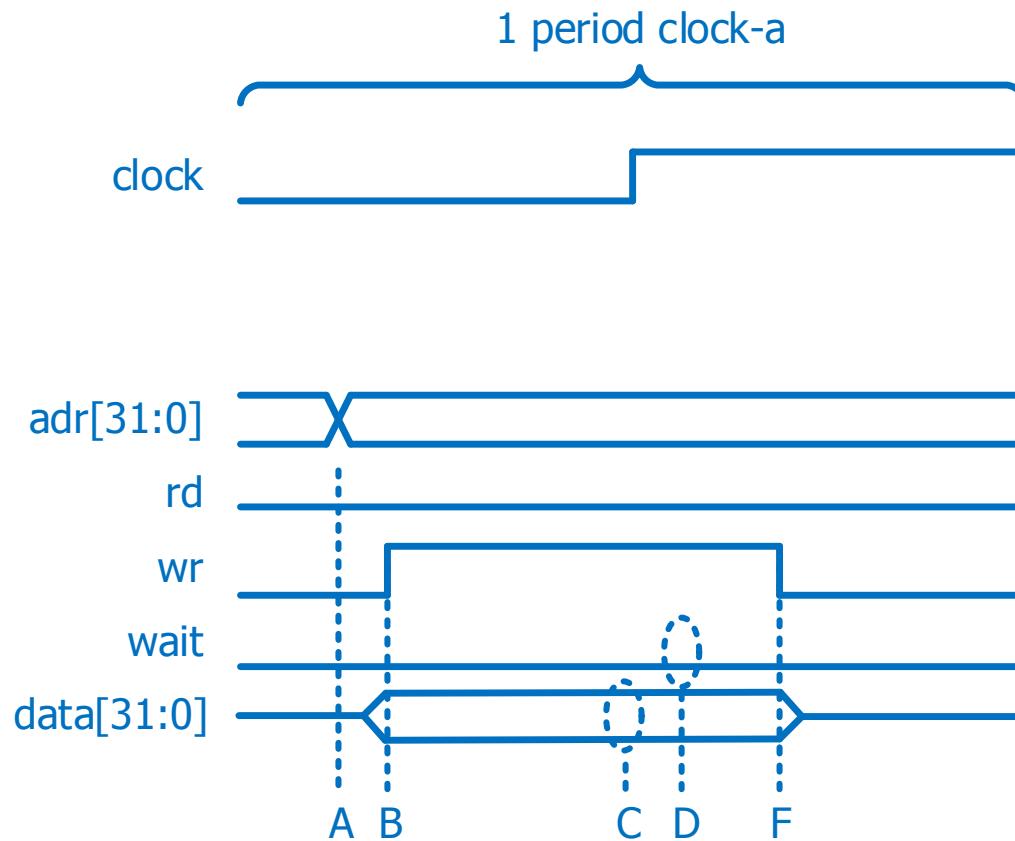
- Definirajmo sabirničke protokole za FRISC
- Par napomena:
 - Bit će jednaki za pristup memoriji ili U-I jedinicama (u objašnjenjima se spominje samo memorija, ali isto vrijedi i za U-I jedinice)
 - Pokazat ćemo protokole za čitanje i pisanje podataka
 - U slučaju normalne brzine, traju jedan takt CLOCK-a
 - U slučaju sporih memorija ili U-I jedinica, umeću se dodatni ciklusi čekanja (svaki traje po jedan takt CLOCK-a)
 - Promatramo samo one sabirničke vodove koji su relevantni za operacije čitanja i pisanja

Čitanje bez stanja čekanja



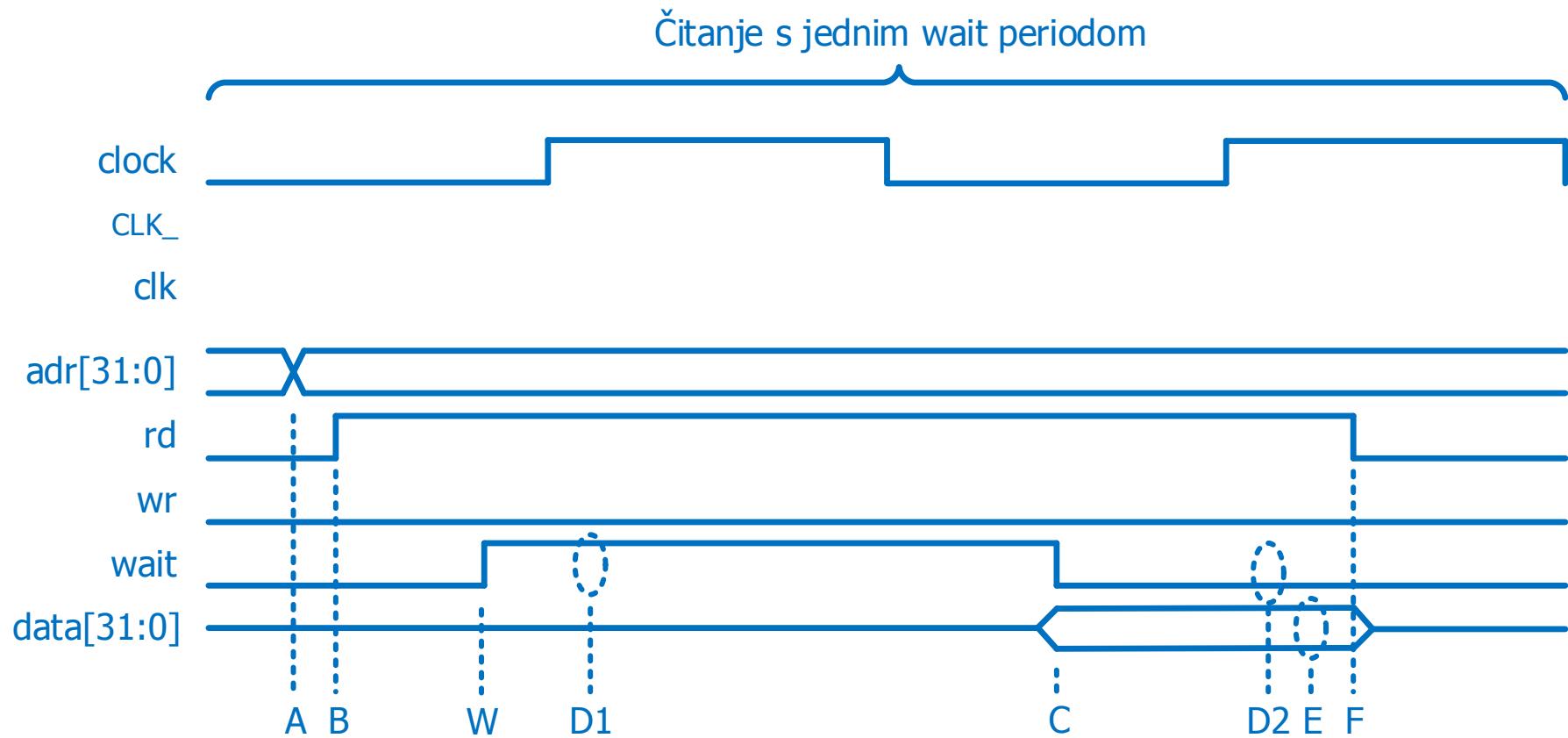
>>>

Pisanje bez stanja čekanja



>>>

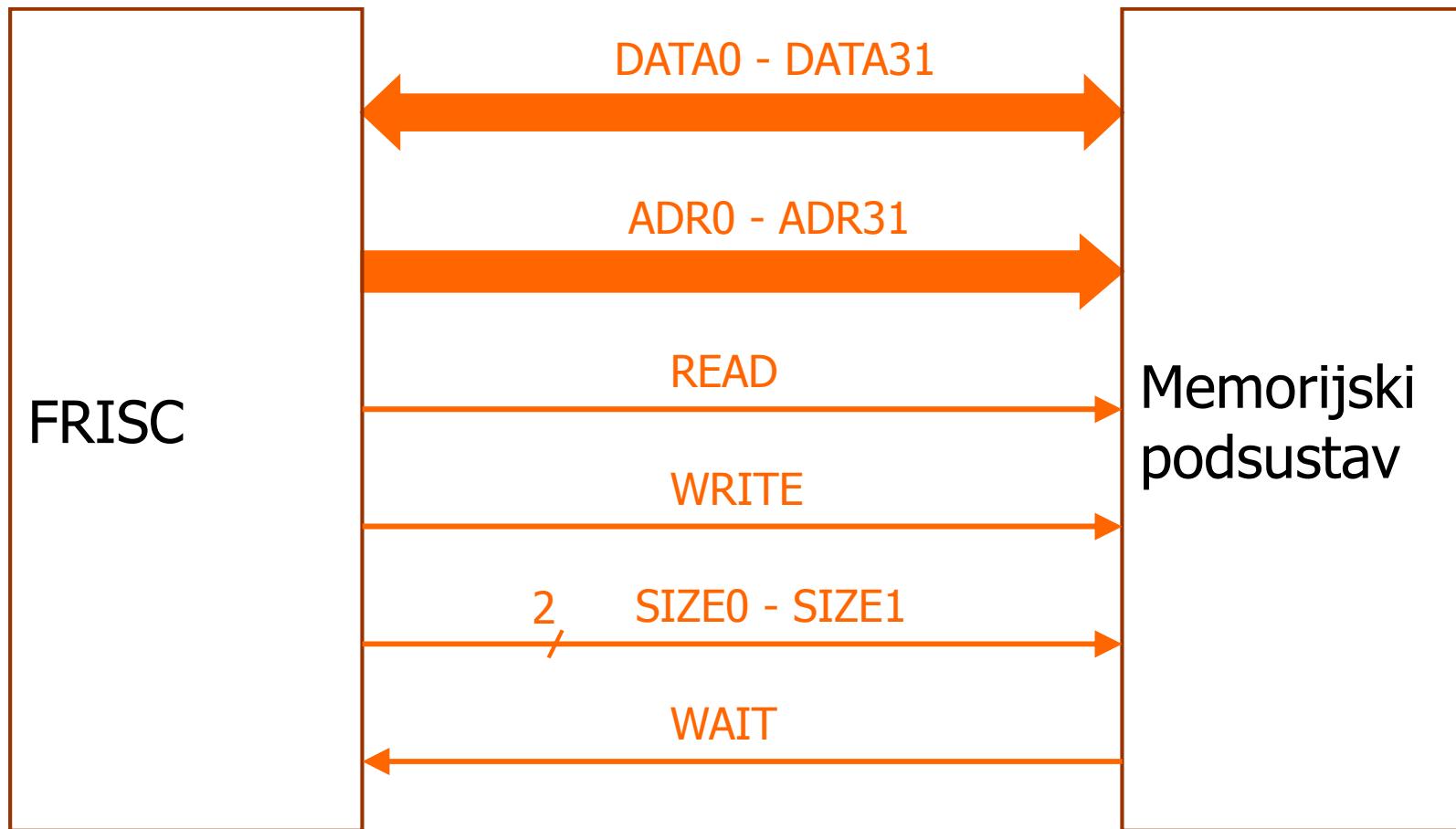
Čitanje sporih mem. s jednim stanjem čekanja



Spajanje FRISC-a i memorije

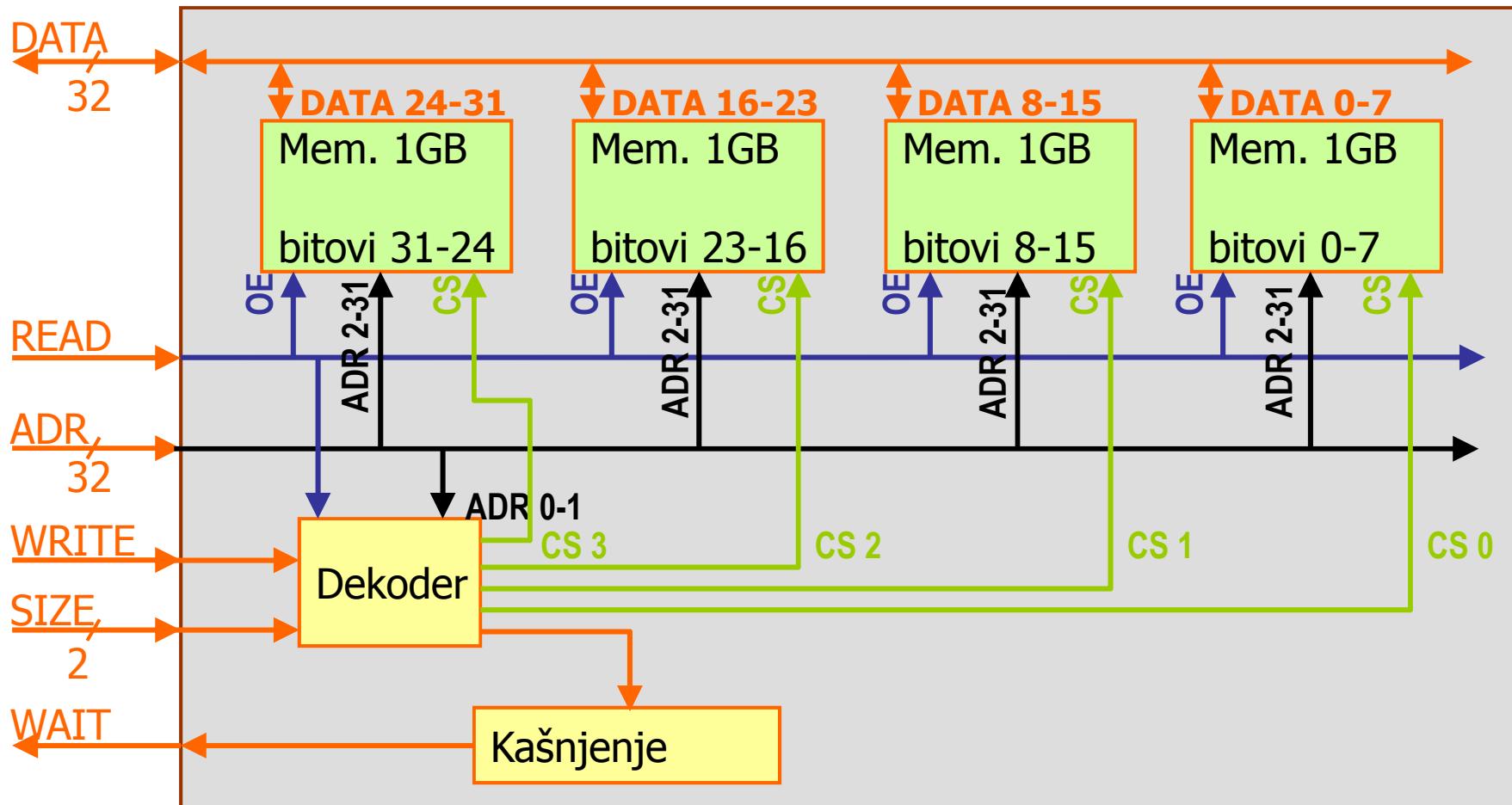
Spajanje FRISC-a i memorije

- Načelna shema spajanja:



Memorijski podsustav

- Načelna shema memorijskog podsustava ($4GB = 4 \times (1G \times 1B)$) :



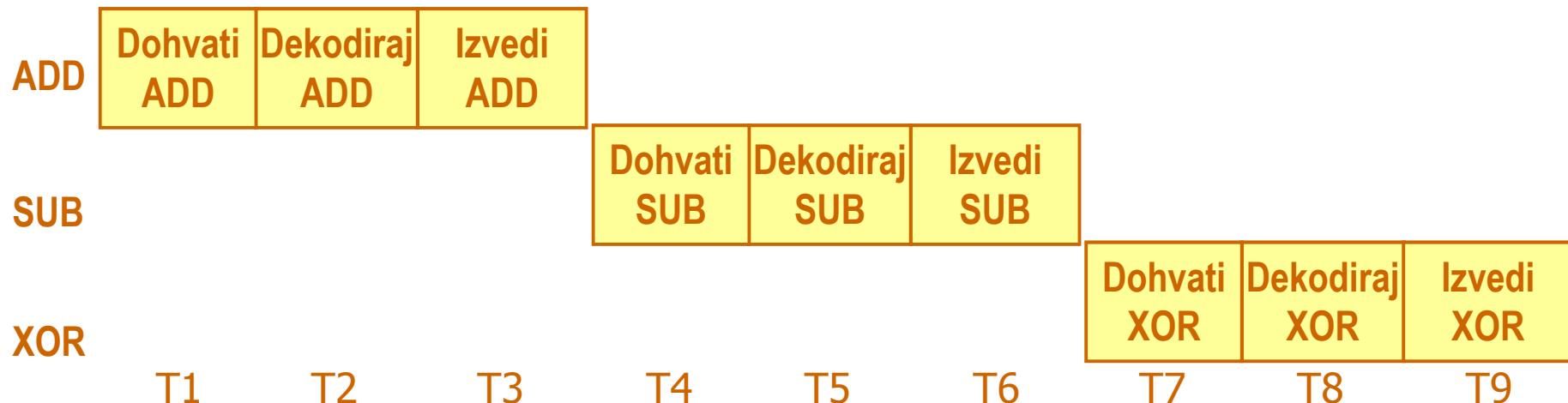
Izvođenje naredaba i protočna struktura

Izvođenje naredaba u procesoru

- **Podsjetnik** - tipične faze u izvođenju naredbe su:
 - dohvati naredbu iz memorije (fetch)
 - dekodiranje naredbe (decode)
 - izvođenje naredbe (execute)
- Kao što ćemo vidjeti kasnije kompleksniji procesori imaju puno više koraka...

Izvođenje naredaba u procesoru

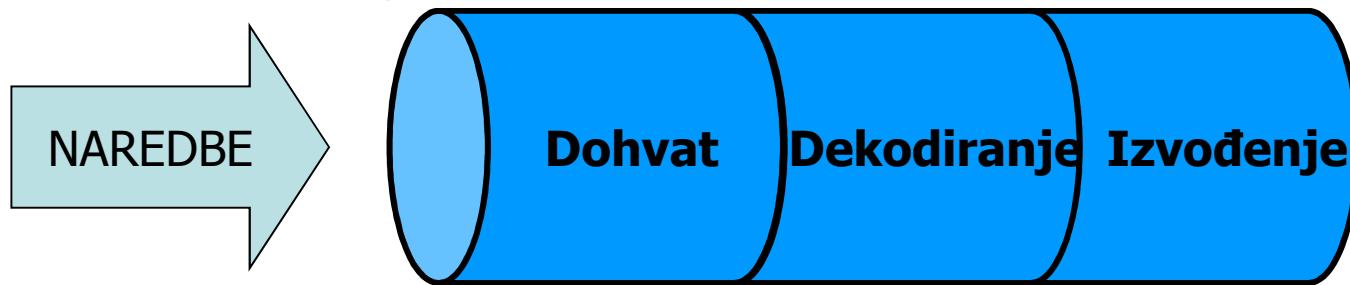
- Npr. neka imamo redom naredbe ADD, SUB i XOR koje se trebaju izvesti:



- Ako bi se naredbe izvodile slijedno po fazama, tada bi za svaku naredbu bila potrebna 3 vremenska perioda (uz pretpostavku da svaka faza traje jedan period).
 - Ovo predstavlja STARI način izvođenja ---- NEEFIKASNO
 - Još uvijek se koristi kod nekih jednostavnih procesora.

Izvođenje naredaba i protočna struktura

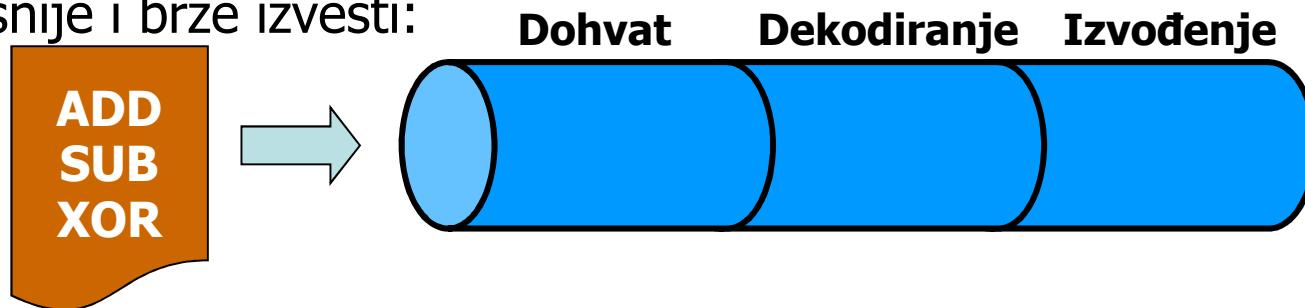
- Podjelimo li faze tako da su međusobno neovisne, možemo primjeniti načela pokretne trake, pa se sve tri faze mogu izvoditi istodobno:



- Ovakva organizacija izvođenja se u literaturi naziva cjevovod (engl. pipeline), a mi ga nazivamo **protočna struktura**.
- Svaka **razina** protočne strukture (engl. pipeline stage) izvodi jednu fazu.
- Glavna namjena je **ubrzanje izvođenja**

Izvođenje naredaba i protočna struktura

- Sada se prije spomenute tri naredbe ADD,SUB i XOR mogu puno efikasnije i brže izvesti:



ADD	Dohvati ADD	Dekodiraj ADD	Izvedi ADD	
		Dohvati SUB	Dekodiraj SUB	Izvedi SUB
SUB				
XOR		Dohvati XOR	Dekodiraj XOR	Izvedi XOR
	T1	T2	T3	T4
				T5

Izvođenje naredaba i protočna struktura

- Podijelimo li naredbu na **N faza**, onda možemo izračunati ubrzanje:
 - u slijednom izvođenju:
 - za svaku naredbu treba N vremenskih perioda
 - za M naredaba slijedno izvođenje traje: $M * N$
 - u protočnom izvođenju:
 - prva naredba traje N perioda, a svaka sljedeća traje 1 period
 - za M naredaba protočno izvođenje traje: $N + (M-1)$
- za veliki M vrijedi: $(M*N) / (N+M-1) \approx (M*N) / (M) = N$
- **Protočno izvođenje u N razina je u prosjeku N puta brže od izvođenja korak-po-korak** (uz pretpostavku linearog programa bez hazarda)

Izvođenje naredaba i protočna struktura

- Radi većeg ubrzanja, dobro bi bilo naredbu podijeliti na što više faza (tj. protočnu strukturu na što više razina)
- Brzina cijele protočne strukture ovisi o brzini najsporije razine, tj. najsporija razina predstavlja ograničenje (usko grlo)
- Zato se pokušavaju odabrati faze tako da svaka traje čim kraće, ali i tako da sve imaju podjednako trajanje

Izvođenje naredaba i protočna struktura

- RISC arhitekture upravo koriste navedena načela da se izvodi puno brzih i jednostavnih naredaba čime se vrijeme izvođenja skraćuje
- Zato se protočna struktura i počela koristiti s pojavom procesora RISC arhitekture
- Danas se, zbog napretka tehnologije, protočnost koristi u većini procesora bez obzira jesu li RISC ili CISC
- U nastavku ćemo definirati protočnu strukturu procesora FRISC

Protočna struktura FRISC-a

- Kako bi zadržali jednostavnost našeg procesora, za efikasnije izvođenje naredaba uvesti ćemo najjednostavniju moguću protočnu strukturu
- **Odabiremo protočnu strukturu FRISC-a sa dvije razine, koje ćemo nazvati:**
 - **razina za dohvata**
 - **razina za izvođenje**



Protočna struktura FRISC-a

- Podjela operacija između razina je ovakva:
 - **Razina za dohvat**
 - Dohvat naredbe
 - Dekodiranje naredbe
 - Dohvat operanada
 - **Razina za izvođenje**
 - Izvođenje AL operacije
 - Spremanje rezultata

Podjela naredaba po brzini izvođenja

- Trajanje operacija u svakoj razini je jedan period signala vremenskog vođenja CLOCK-a (uz pretpostavku da memorija nije spora)
- Naredbe procesora FRISC razlikuju se po načinu kako se izvode:
 - **Faza dohvata svih naredaba traje jedan period**
 - **Faza izvođenja također traje jedan period, ali kod nekih naredaba nije moguće preklapanje s fazom dohvata sljedeće naredbe**

Podjela naredaba po brzini izvođenja

- Kod jednostavnih naredaba moguće je preklapanje faze izvođenja sa fazom dohvata sljedeće naredbe.
 - Ove naredbe zovu se **jednociklusne** jer im efektivno vrijeme izvođenja u protočnoj strukturi iznosi jedan period (ciklus) CLOCK-a.
 - Ove naredbe efikasno koriste protočnu strukturu.
- U jednociklusne naredbe spadaju:
 - Aritmetičko-logičke naredbe
 - Registarske naredbe
 - Upravljačke naredbe kod kojih uvjet nije zadovoljen

Jednociklusne naredbe

- Izvođenje slijeda jednociklusnih naredaba:

	Dohvat	Izvođenje
T1	ADD	
T2	SUB	ADD
T3	XOR	SUB
T4	AND	XOR
T5		AND

Podjela naredaba po brzini izvođenja

- Kod složenijih naredaba nije moguće preklapanje faze izvođenja s fazom dohvata sljedeće naredbe
 - Takve naredbe zovu se **dvociklusne** naredbe jer im efektivno vrijeme izvođenja u protočnoj strukturi iznosi dva perioda (ciklusa) CLOCK-a.
 - Ove naredbe ne koriste efikasno protočnu strukturu
- U dvociklusne naredbe spadaju:
 - Memorijske naredbe
 - Upravljačke naredbe kod kojih je uvjet za izvođenje zadovoljen

Dvociklusne naredbe

- Izvođenje slijeda dvociklusihih naredaba:

	Dohvat	Izvođenje
T1	LOAD	
T2		LOAD
T3	JP	
T4		JP
T5	STORE	
T6		STORE

Jednoklusalne naredbe: dohvati

Prva polovica perioda:

- Postavljanje adrese na adresnu sabirnicu ($PC \rightarrow AR$)
- Aktiviranje signala rd^{**}

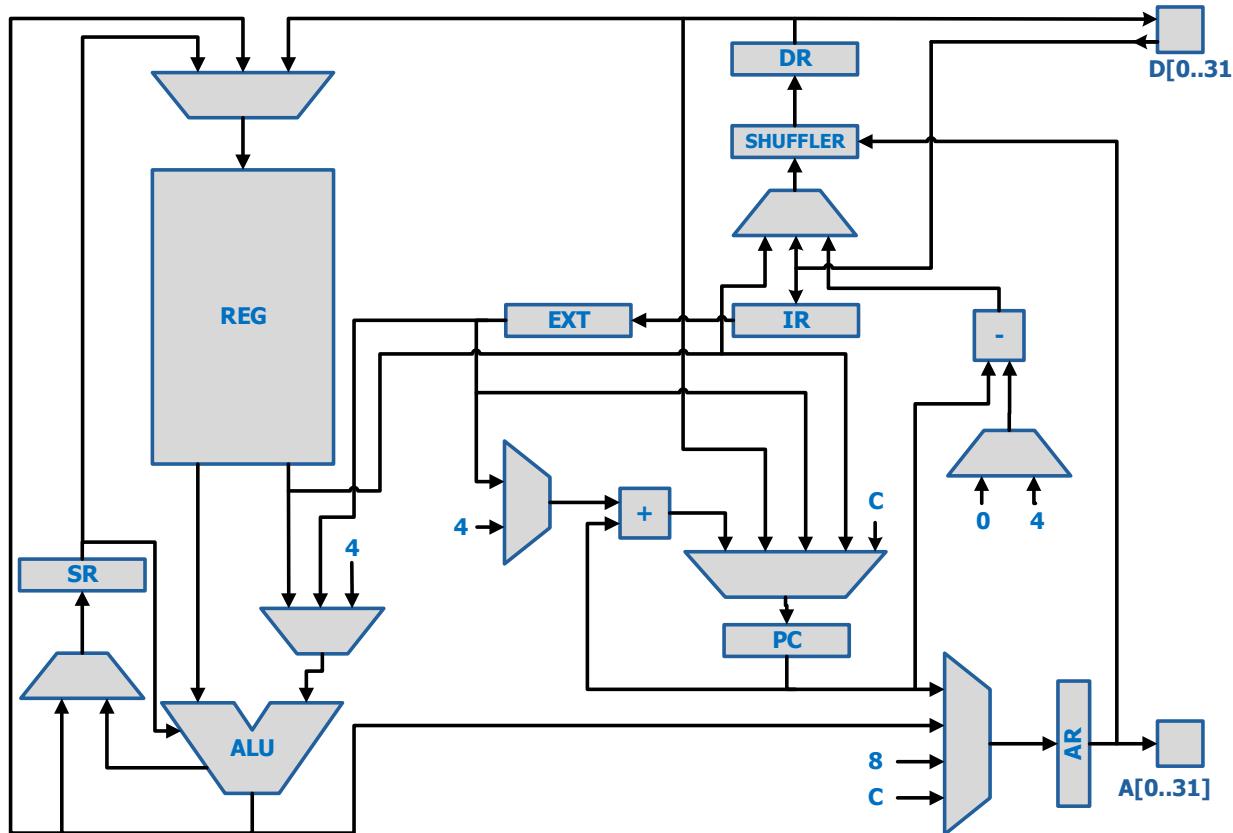
Druga polovica perioda *:

- Čitanje naredbe $(AR) \rightarrow IR$
- Deaktiviranje signala rd^{**}
- Dekodiranje naredbe dekodiranje
- Dohvat operanada i slanje u ALU $\text{operandi} \rightarrow ALU$
- Izbor i pokretanje ALU operacije
- Postavljanje PC za sljedeću naredbu $PC+4 \rightarrow PC$

*Ovi koraci izvode se u slučaju da je memorija dovoljno brza i da nije bio aktiviran WAIT signal

** aktiviranje i deaktiviranje signala rd i wr se neće više navoditi

Jednociklusne naredbe: dohvati



Dohvat

- Do trenutka dekodiranja sve naredbe imaju isti način dohvata naredbe

Jednociklusne naredbe: izvođenje

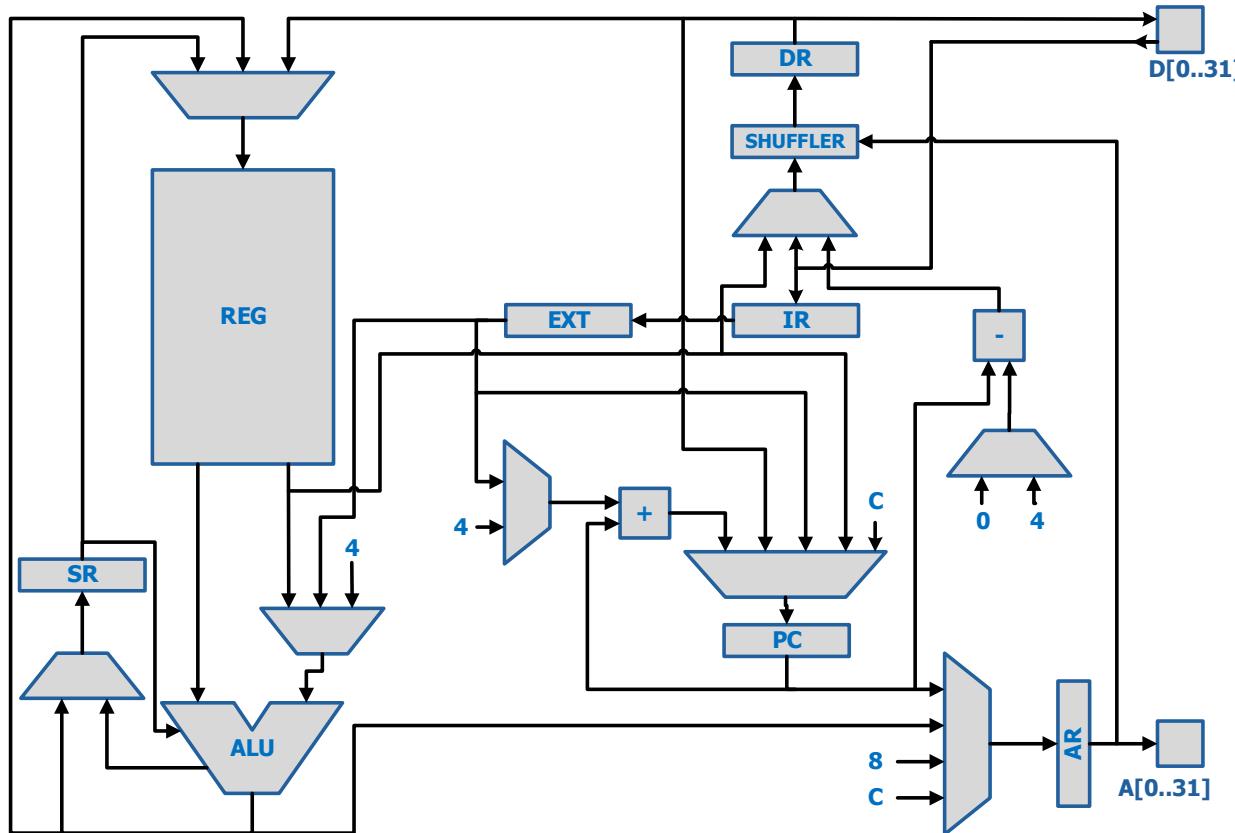
Prva polovica perioda:

- ALU završava operaciju i sprema se rezultat (osim kod CMP) ALU → Reg
- Spremaju se zastavice u SR

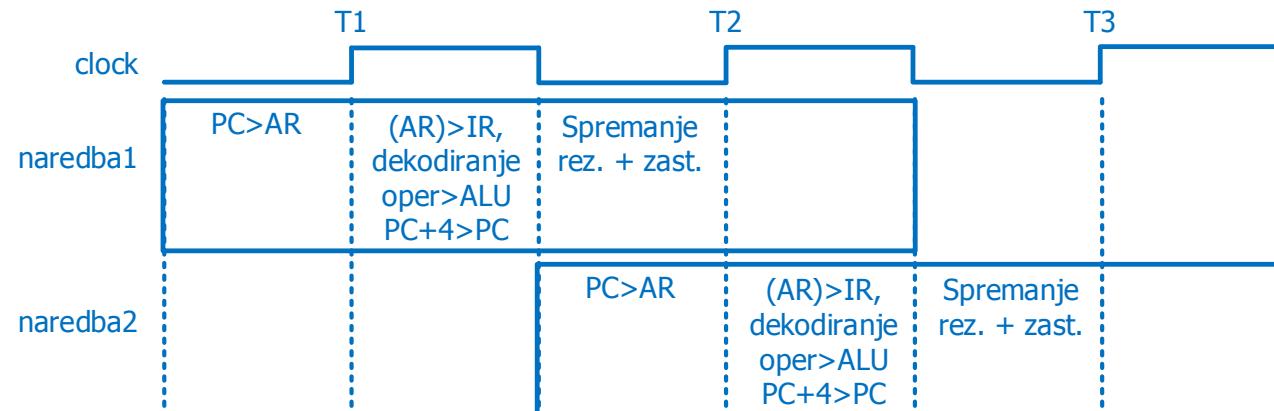
Druga polovica perioda:

- Nema aktivnosti vezano za izvođenje naredbe

Jednociklusne naredbe: izvođenje

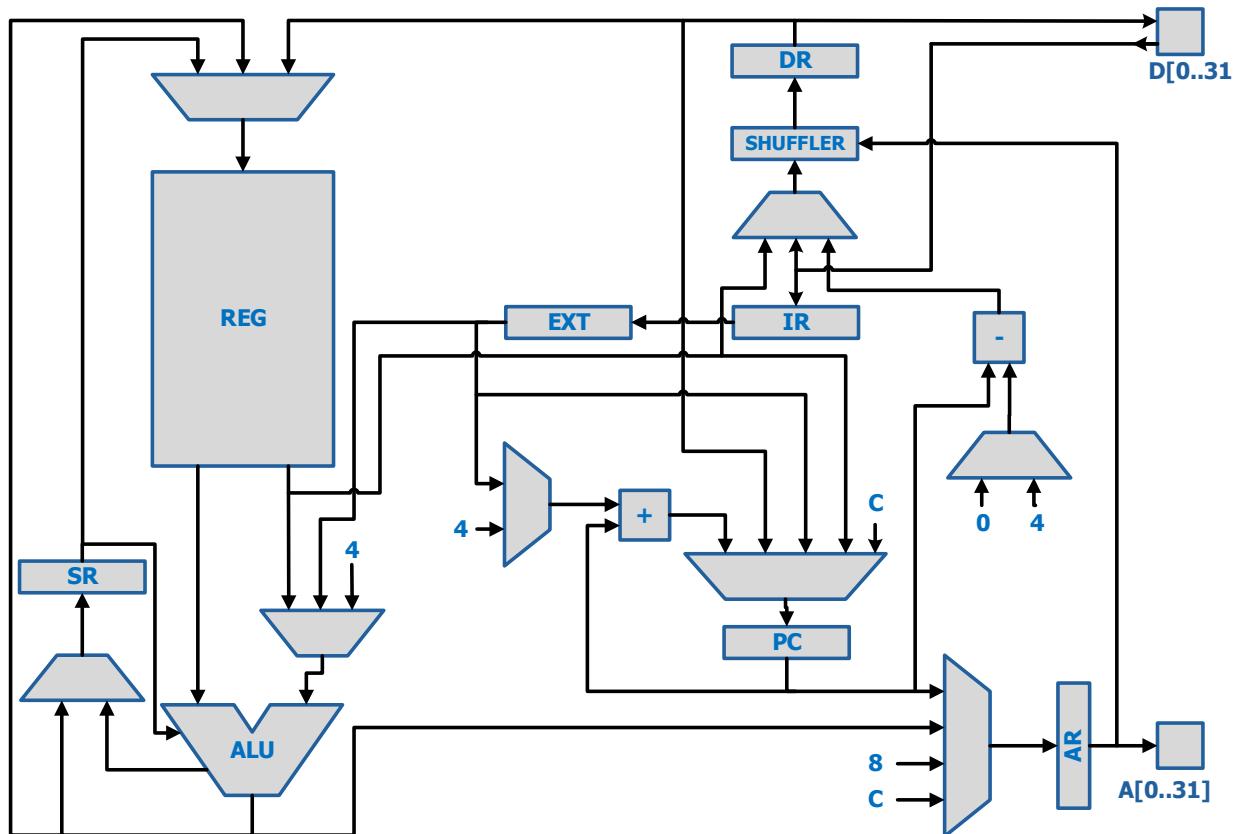


Izvođenje dvije AL naredbe u protočnoj strukturi



Jednociklusne naredbe izvođenje: Primjer

ADD R1,R2,R3
MOVE 100, R1



Hazardi

- Do sada objašnjeni rad protočne strukture za jednociklusne naredbe bio je **idealan**, no u stvarnom radu postoje situacije koje uzrokuju da protočna struktura djelomično gubi svoju efikasnost
- U nekim situacijama protočna struktura ne može obraditi sljedeću naredbu odmah u sljedećem periodu
- Takve situacije nazivaju se **hazardi**
- Postoje tri osnovna tipa hazarda:
 - **Strukturni**
 - **Upravljački**
 - **Podatkovni**

Strukturni hazard

- Strukturni hazard je pojava kad procesor u određenom trenutku ne može izvesti sve faze onih naredaba koje se nalaze u protočnoj strukturi, jer struktura (tj. sklopolje) procesora ne omogućuje istodobno izvođenje svih tih faza
- Jednostavan primjer strukturnog hazarda je izvođenje naredbe npr. LOAD ili npr. STORE kod procesora s Von Neumannovom arhitekturom. Struktura memorijskog sučelja (Von Neumannova arhitektura) ne dozvoljava istovremeno dva pristupa memoriji:
 - jedan pristup treba za izvođenje naredbe (npr. kod naredbe STORE za spremanje podatka)
 - drugi pristup je dohvat strojnog kôda sljedeće naredbe

Strukturni hazard

- Strukturni hazard rješava se tako da procesor odgodi izvođenje naredaba koje se nalaze u prethodnim razinama protočne strukture (to su naredbe koje su kasnije ušle u protočnu strukturu) dok se uzrok hazarda ne ukloni.
- Odgoda izvođenja naredaba u prethodnim razinama u literaturi se naziva **mjehurić (bubble)**. Slikovito se promatra kao da je u protočnu strukturu ušao mjehurić koji prolazi kroz razine i uzrokuje njihovu neaktivnost.
- Kada uzrok hazarda nestane, procesor nastavlja izvoditi naredbe na normalan način.

Naredbe FRISC-a i strukturni hazard

- Zbog Von Neumannove arhitekture memorijskog sučelja, sve memorijske naredbe procesora FRISC (LOAD, STORE, LOADH, STOREH, LOADB, STOREB, PUSH, POP) uzrokovat će strukturni hazard.
- Te naredbe imati će fazu izvođenja koja se ne može preklapati s fazom dohvata sljedeće naredbe te će njihovo izvođenje efektivno trajati dva perioda. Zato ćemo te naredbe svrstati u grupu **dvociklusnih** naredaba.

Izvođenje memoriskih naredaba

- **Memorijske naredbe LOAD i STORE:**
 - Naredba LOAD prilikom izvođenja mora pročitati podatak iz memorije, a naredba STORE ga mora upisati u memoriju (sve isto vrijedi i za naredbe LOADH,LOADB, STOREH, STOREB)
 - Pristup memoriji pri izvođenju LOAD/STORE **ne može** se odvijati istodobno s dohvatom sljedeće naredbe iz iste memorije
 - Zato se, nakon prepoznavanja naredbe LOAD/STORE, **onemogućuje** rad razine dohvata u sljedećem ciklusu (mjeehurić) te će biti aktivna samo razina izvođenja
 - Na kraju izvođenja se zato **omogućuje** rad razine za dohvat u sljedećem ciklusu
 - Međutim, za vrijeme tog (odgođenog) dohvata bit će neaktivna razina za izvođenje (mjeehurić prelazi iz razine dohvata u razinu izvođenja), jer nema dohvaćene naredbe koja bi se mogla izvoditi

Primjer

- Analizirajmo broj perioda potrebnih za izvođenje programa za zbrajanje dva broja:

LOAD R0, (100)

LOAD R1, (200)

ADD R0,R1,R2

STORE R2, (300)

	Dohvat	Izvođenje
T1	LOAD R0, (100)	
T2		LOAD R0, (100)
T3	LOAD R1, (200)	
T4		LOAD R1, (200)
T5	ADD R0,R1,R2	
T6	STORE R2,(300)	ADD R0,R1,R2
T7		STORE R2,(300)

- $T = 7$

Dvociklusne naredbe: dohvati STORE

Prva polovica perioda:

- Postavljanje adrese na adresnu sabirnicu PC → AR

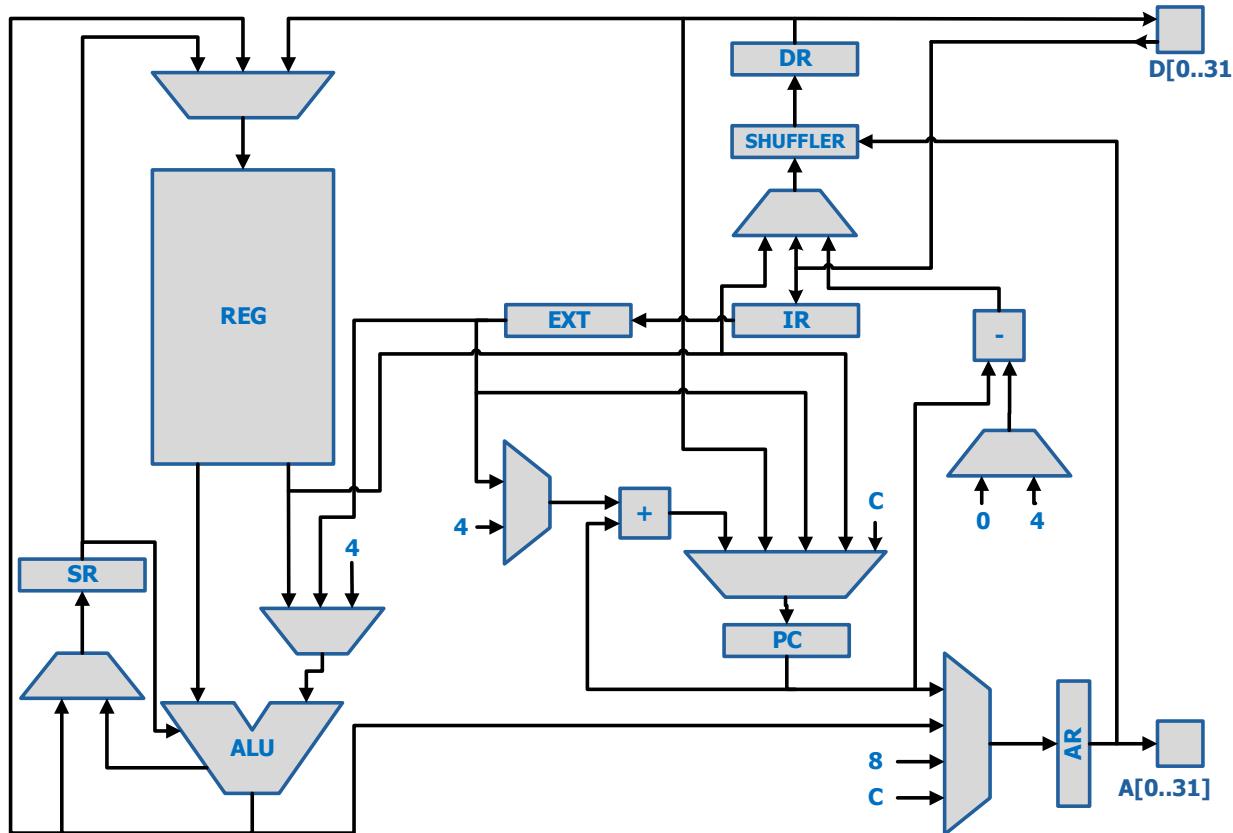
Druga polovica perioda *:

- Čitanje naredbe (AR) → IR
- Dekodiranje naredbe dekodiranje
- Dohvat adrese** i slanje prema ALU (ext → ALU) ili (Rx, ext → ALU)
- Prosljeđivanje adrese ili izračun adrese** (ALU proslj.) ili (ALU zbraja)
- Postavljanje PC za sljedeću naredbu PC+4 → PC
- onemogući dohvati u sljedećem ciklusu

*Ovi koraci izvode se u slučaju da je memorija dovoljno brza i da nije bio aktiviran WAIT signal

** ovisno o načinu adresiranja u naredbi

Dvociklusne naredbe: dohvat STORE



Dvociklusne naredbe: izvođenje STORE

Prva polovica perioda:

- Postavljanje adrese na adr.sabirnicu ALU_OUT → AR
- Podatak na sab. podataka (uz shuffle) REG_B → DR

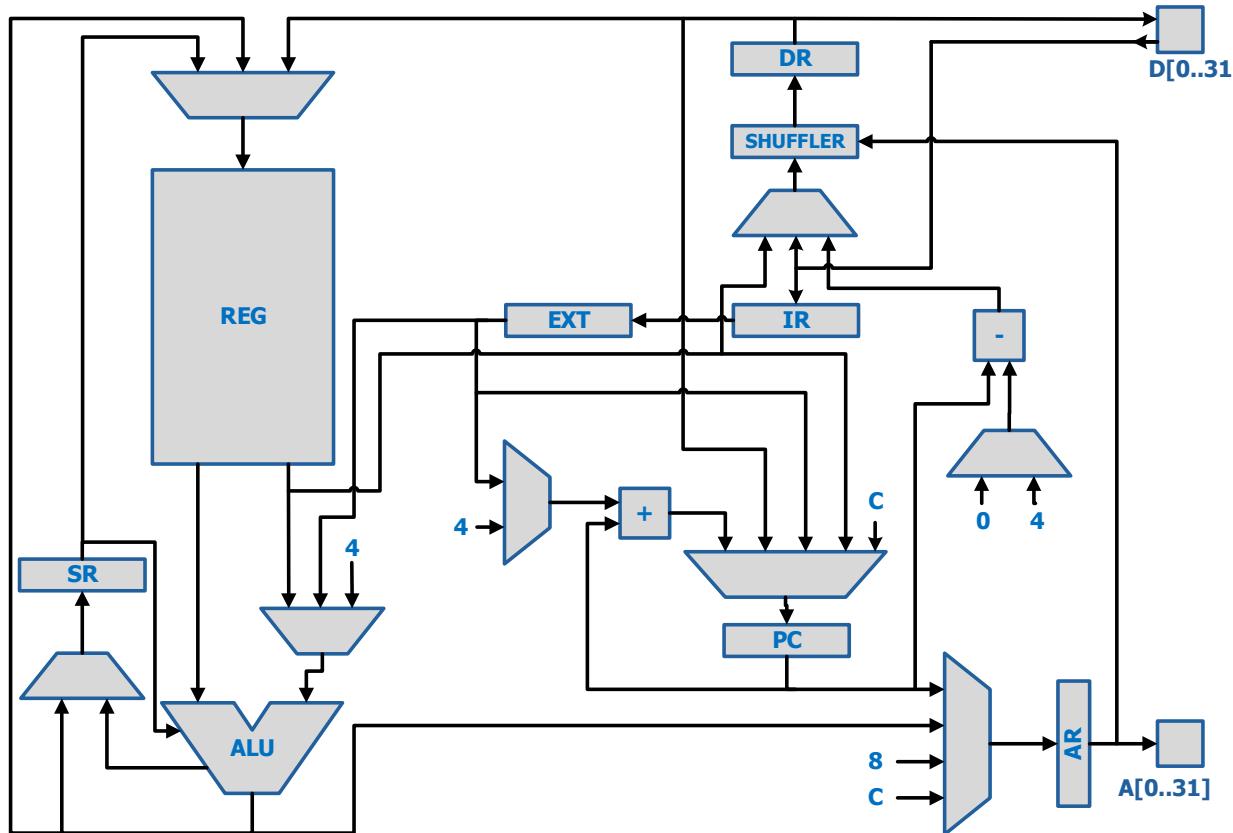
Druga polovica perioda *:

- omogući dohvat u sljedećem ciklusu

*Ovi koraci izvode se u slučaju da je memorija dovoljno brza i da nije bio aktiviran WAIT signal

**Na WEB je stavljen ispravak opisa izvođenja naredbe
STORE iz knjige !!! (u knjizi je pogrešno ALU_OUT → AR
bilo napisano u ciklusu dohvata)**

Dvociklusne naredbe: izvođenje



Primjer: STOREB R1, (R2+3ABC)

Razina dohvata:

Prva polovina periode CLOCK-a:

PC → **AR**

Druga polovina periode CLOCK-a:

(AR) → **IR**,

dekodiranje

ext 3ABC i R2 → **ALU**

ALU: izvodi zbrajanje

PC +4 → **PC**

onemogući dohvat u sljedećem ciklusu

Razina izvođenja:

Prva polovina periode CLOCK-a:

ALU → **AR**

R1 → **DR**

Druga polovina periode CLOCK-a:

omogući dohvat u sljedećem ciklusu

Primjer: LOAD R0, (R1+300)

Razina dohvata:

Prva polovina periode CLOCK-a:

PC -> **AR**

Druga polovina periode CLOCK-a:

(AR) -> **IR**,

dekodiranje

ext 300 i R1 -> **ALU**

ALU: izvodi zbrajanje

PC +4 -> **PC**

onemogući dohvat u sljedećem ciklusu

Razina izvođenja:

Prva polovina periode CLOCK-a:

ALU -> **AR**

Druga polovina periode CLOCK-a:

(AR) -> **DR**

DR -> **R0**

omogući dohvat u sljedećem ciklusu

Dvociklusne naredbe: dohvati PUSH

(Dohvat je sličan kao i kod LOAD/STORE samo se za adresu uzima R7-4)

Prva polovica perioda:

- Postavljanje adrese na adresnu sabirnicu PC → AR

Druga polovica perioda *:

- | | |
|---|----------------|
| • Čitanje naredbe | (AR) → IR |
| • Dekodiranje naredbe | dekodiranje |
| • Dohvat adrese i slanje prema ALU | R7, 4 → ALU |
| • ALU | ALU oduzimanje |
| • Postavljanje PC za sljedeću naredbu | PC+4 → PC |
| • Onemogući dohvati u sljedećem ciklusu | |

*Ovi koraci izvode se u slučaju da je memorija dovoljno brza i da nije bio aktiviran WAIT signal

Dvociklusne naredbe: izvođenje PUSH

(Dohvat je sličan kao i kod LOAD/STORE samo se za adresu uzima R7-4)

Prva polovica perioda:

- Postavljanje adrese na adr.sabirnicu ALU_OUT → AR
- Osvježavanje SP ALU_OUT → R7
- Podatak na sab. podataka REG_B → DR

Druga polovica perioda *:

- omogući dohvati u sljedećem ciklusu

*Ovi koraci izvode se u slučaju da je memorija dovoljno brza i da nije bio aktiviran WAIT signal

Izvođenje memorijskih naredaba

- **Memorijske naredbe PUSH i POP:**
 - Prilikom izvođenja naredaba PUSH i POP također se pristupa memoriji kao i kod naredba LOAD i STORE, pa se naredbe izvode na sličan način
 - Dodatno, naredba PUSH smanjuje register SP, a POP povećava SP
- DZ: Proučiti iz knjige način izvođenja naredbe POP. Razmisliti koje su razlike između izvođenja LOAD I STORE, te koje su razlike između izvođenja POP i PUSH
- Napredno gradivo: proučiti tablicu izvođenja svih naredaba procesora FRISC (osim za prekide)

Upravljački hazard

- Drugi od tri ranije spomenuta hazarda je **upravljački hazard**. Ovaj hazard dešava se kad naredba koja se nalazi u protočnoj strukturi i spremna je za izvođenje nije naredba koja se u stvari treba izvesti
- Ovaj hazard događa se kod izvođenja naredaba grananja kad je procesor već učitao sljedeću naredbu i pripremio se za njeno izvođenje, ali zbog grananja program treba nastaviti s izvođenjem naredbe na nekoj drugoj adresi
- Zbog toga se ovaj hazard naziva još i hazardom grananja
- Naredbe koje uzrokuju ovaj hazard kod FRISC-a su upravljačke naredbe

Primjer

```
0      SUB R0,R0,R0
4      JP_NZ 14
8      ADD R0, 2, R0
C      JP_NZ 18
10     OR R0,R0,R0
14     AND R0,R0,R0
18     XOR R0,R0,R0
```

	Dohvat	Izvođenje
T1	SUB R0,R0,R0	
T2	JP_NZ 14	SUB R0,R0,R0
T3	ADD R0, 2, R0	JP_NZ 14
T4	JP_NZ 18	ADD R0, 2, R0
T5	OR R0,R0,R0	JP_NZ 18
T6	XOR R0,R0,R0	
T7	...	XOR R0,R0,R0

Uvjetna upravljačka naredba

- uvjetna upravljačka naredba ponaša se kao
 - dvociklusna ako je uvjet zadovoljen (jer dolazi do pojave hazarda i umeće se mjehurić u protočnu strukturu),
 - kada uvjet nije zadovoljen ponaša se kao jednociklusna

Izvođenje upravljačkih naredaba

- Specifičnosti pojedinih upravljačkih naredaba su:
 - Adresa skoka zadana je brojem ili registrom: zbrajanje nije potrebno za izračun adrese skoka (za razliku od memorijskih naredaba s indirektnim registarskim adresiranjem s odmakom)
 - Izuzetak je naredba JR u kojoj je zadana relativna adresa koja se pribraja registru PC, ali za to se ne koristi ALU, nego zasebni sklop za zbrajanje koji postoji uz registar PC
 - Naredbe CALL i RET su specifične po tome što osim promjene tijeka izvođenja zahtijevaju dodatni pristup memoriji, tj. stogu (po tome su one slične naredbama PUSH i POP)

Upravljačke naredbe: dohvati JP

Prva polovica perioda:

- Postavljanje adrese na adresnu sabirnicu PC → AR

Druga polovica perioda *:

- Čitanje naredbe (AR) → IR
- Dekodiranje naredbe dekodiranje
- Ispitivanje UVJETA
- Postavljanje PC za sljedeću naredbu PC+4 → PC
- **Ako je UVJET istinit onemogući dohvati u sljedećem ciklusu**

*Ovi koraci izvode se u slučaju da je memorija dovoljno brza i da nije bio aktiviran WAIT signal

Upravljačke naredbe: izvođenje JP

OVO SE IZVODI SAMO AKO JE UVJET BIO ISTINIT!

Prva polovica perioda:

- Nema aktivnosti

Druga polovica perioda:

- Adresa skoka u PC adr → PC
 - omogući dohvat u sljedećem ciklusu

Upravljačke naredbe: JR

- Dohvat i izvođenje isto kao i JP osim što se u periodu izvođenja

PC+ext → PC

Upravljačke naredbe: CALL, RET, HALT

- Ove naredbe također su jednociklusne ako je zadan uvjet za izvođenje i on nije istinit, a dvociklusne su ako su bezuvjetne ili je zadani uvjet istinit
- DZ: proučite iz knjige korake pri dohvatu i izvođenju naredbe CALL

Primjer

- Treba odrediti ukupno trajanje programa u ciklusima

```
START  ORG 0
      MOVE 50, R0
      SUB R0, 5, R0
      CALL_Z PRIPR
PETLJA SUB R0, 1, R0
        JR_NE PETLJA
        CALL_Z PRIPR
        HALT
PRIPR  ADD R0, 5, R0
      RET
```

Primjer

- Treba odrediti ukupno trajanje programa u ciklusima

START	ORG 0	0 pseudonaredba !!!
	MOVE 50, R0	1 x 1c
	SUB R0, 5, R0	1 x 1c (R0 = 4B = 75 ₁₀ !!!)
	CALL_Z PRIPR	1 x 1c (uvjet nije istinit!)
PETLJA	SUB R0, 1, R0	75 ₁₀ x 1c
	JR_NE PETLJA	74 ₁₀ x 2c + 1 x 1c = 149 ₁₀ c
	CALL_Z PRIPR	1 x 2c (uvjet istinit!)
	HALT	1 x 2c
PRIPR	ADD R0, 5, R0	1 x 1c
	RET	1 x 2c

UKUPNO: 3c + (75x 3c -1c) + (2c + 3c) + 2c = 234 c

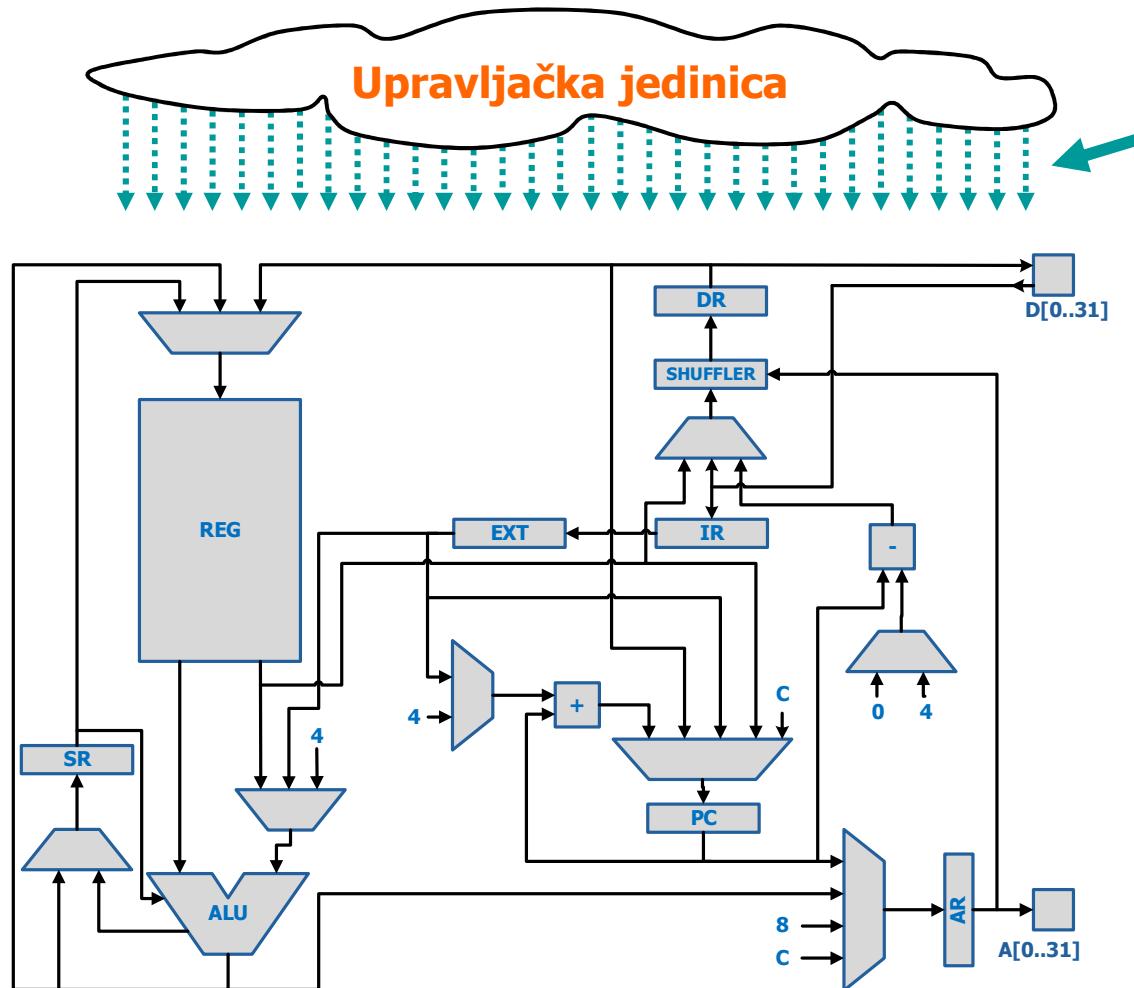
Podatkovni hazard

- **Podatkovni hazard** javlja se kod izvođenja naredaba u protočnoj strukturi kada se naredba ne može izvesti jer podaci potrebni za njeno izvođenje još nisu spremni
- Kod FRISC-a nema pojave podatkovnog hazarda pa ćemo primjer ovog hazarda proučiti kod procesora ARM

Cjelovit rad procesora

- U prethodnim predavanjima proučili smo arhitekturu puta podataka i proučili način izvođenja naredaba
- U nastavku ćemo ukratko objasniti na koji način funkcionira procesor kao cjelina, odnosno koja su načela upravljanja sa svim dijelovima puta podataka
- Prisjetimo se nakratko mikroarhitekture puta podataka procesora FRISC

Put podataka procesora FRISC



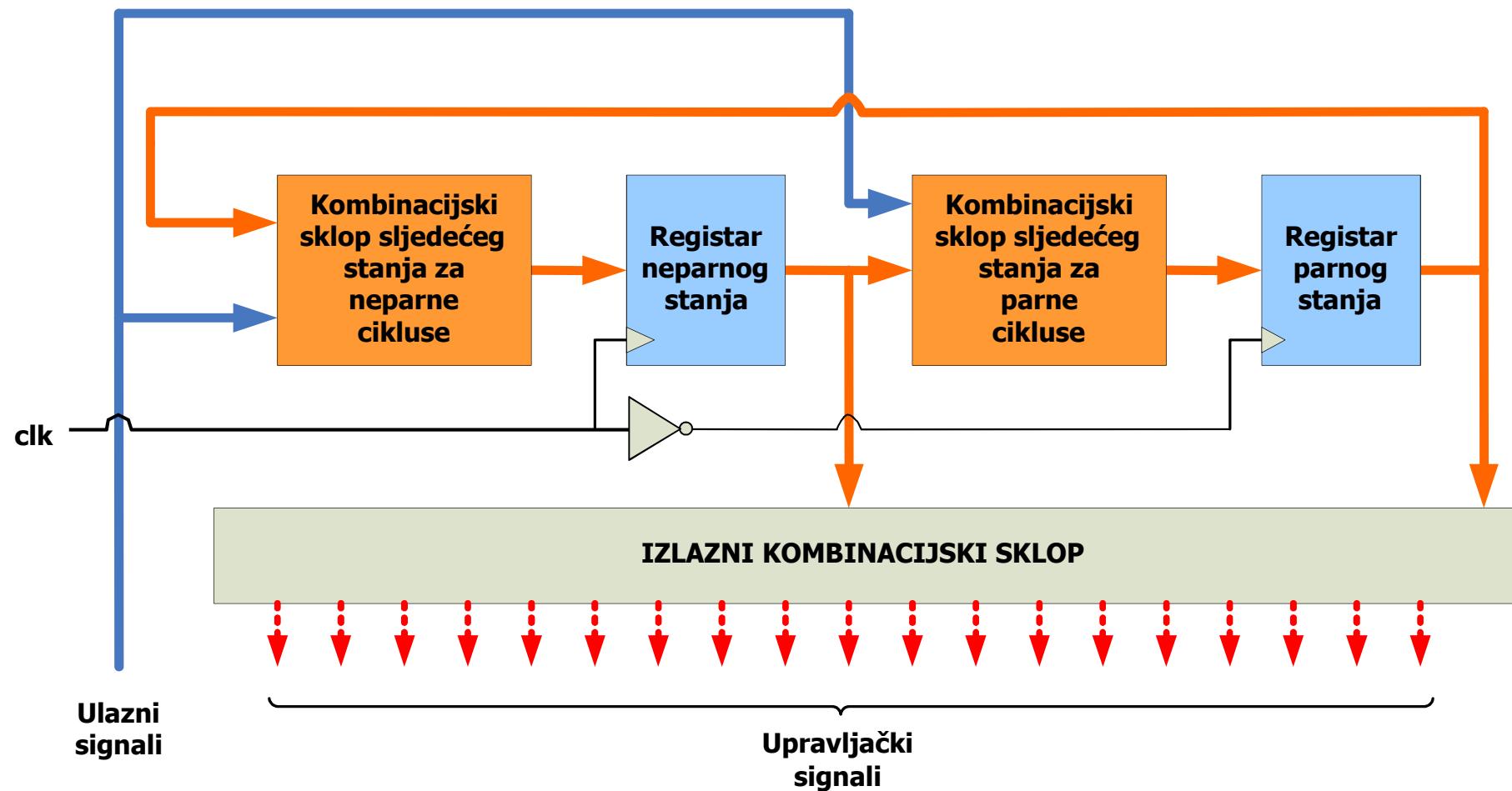
Kako generirati upravljačke signale potrebne za izvođenje opisanih naredaba ?

Upravljanje putom podataka

- Kao što ste naučili u "Digitalnoj", jednostavan način generiranja upravljačkih signala može se postići strojem s konačnim brojem stanja (finite state machine - FSM)
- Pri izvođenju naredaba treba generirati upravljačke signale na rastući i na padajući brid signala vremenskog vođenja clock
- Klasični FSM generira signale na jedan od bridova (rastući ili padajući) pa upravljačka jedinica FRISC-a koristi dvostruki FSM

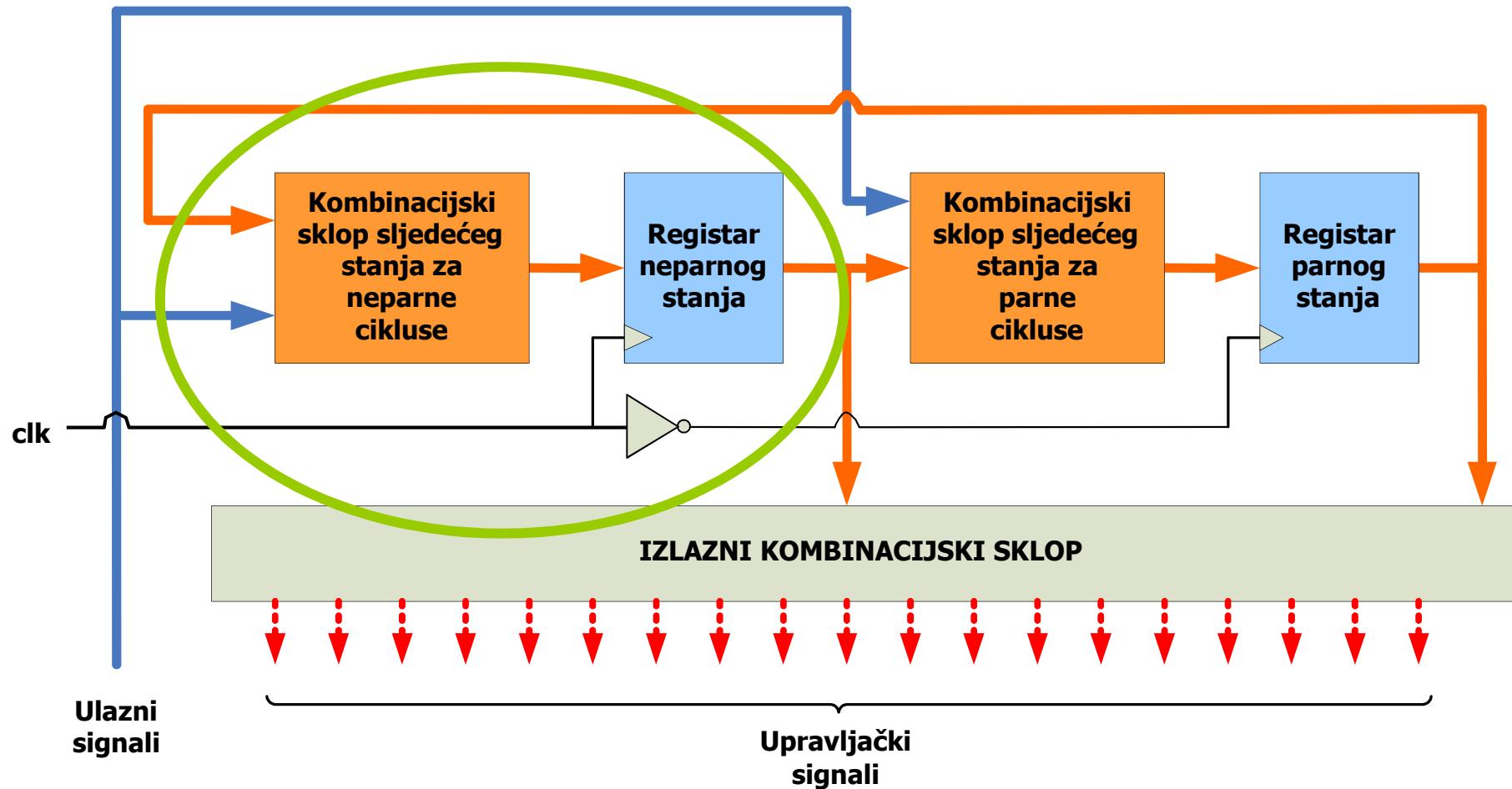
Upravljačka jedinica – stroj s konačnim brojem stanja

i



Upravljačka jedinica – stroj s konačnim brojem stanja

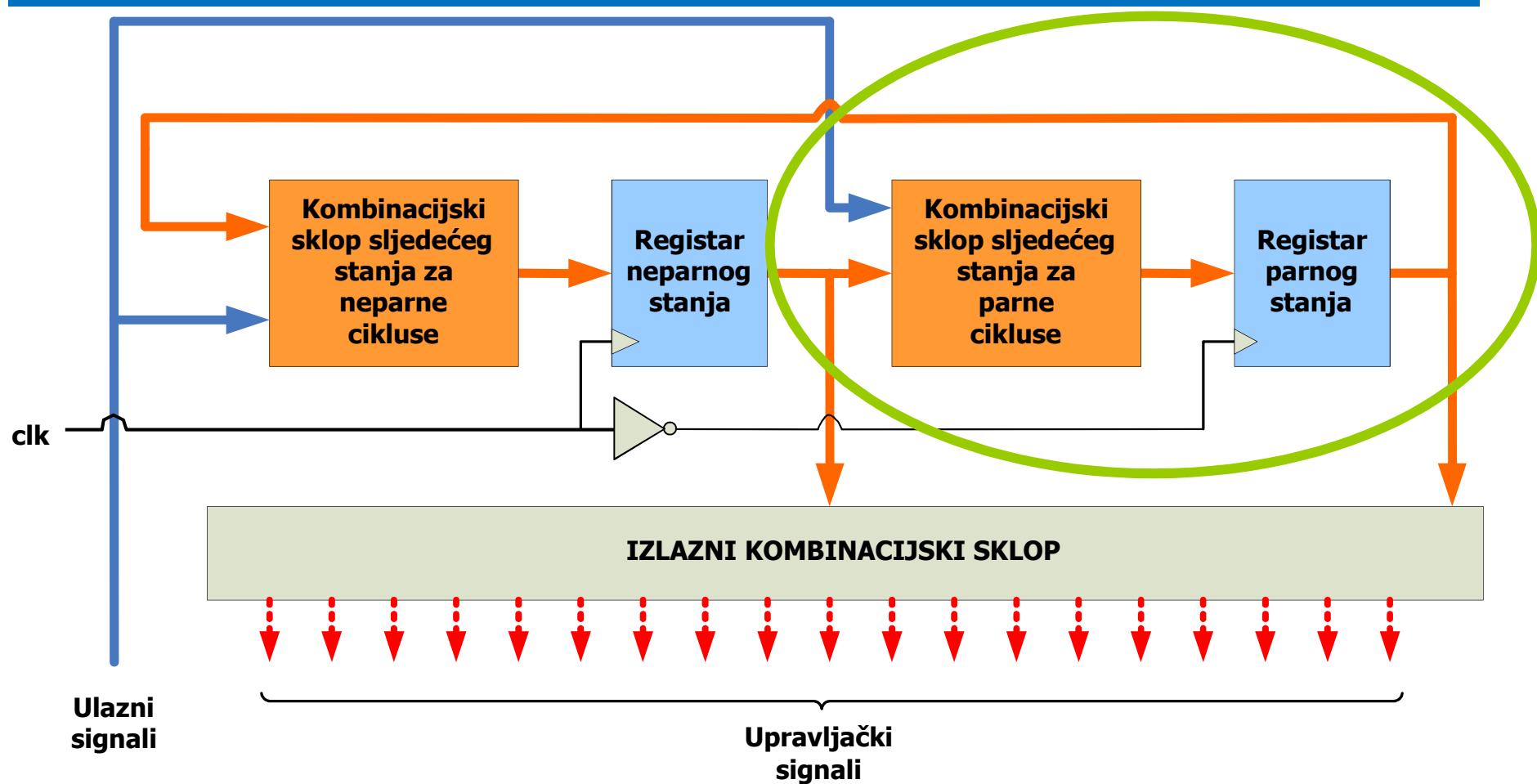
i



Prvi FSM zadužen je za generiranje svih signala u neparnim poluperiodima

Upravljačka jedinica – stroj s konačnim brojem stanja

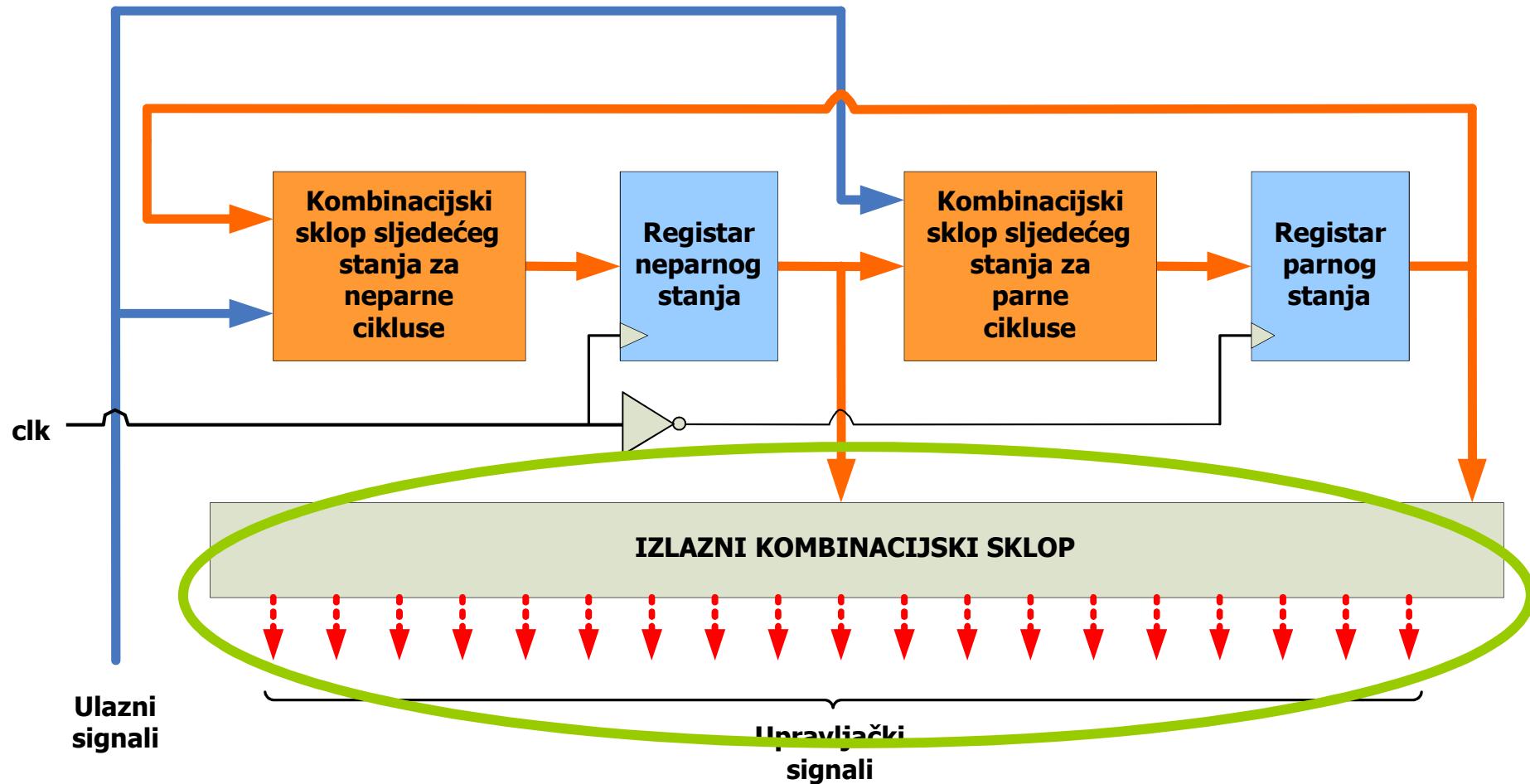
i



Drugi FSM zadužen je za generiranje svih signala u parnim poluperiodima

Upravljačka jedinica – stroj s konačnim brojem stanja

i



Upravljački signali generiraju se na temelju stanja oba stroja

Povezivanje računala s okolinom

UI uređaji

- Uređaji se nikada ne spajaju izravno na sabirnicu procesora (razlozi !!) već preko "ulaza i izlaza", tj. preko posebnih međusklopova namijenjenih upravo toj svrsi (oni služe kao posrednici)
- Ovakve međusklopove zovemo ulazno-izlaznim (UI) jedinicama (engl. IO unit) ili vanjskim jedinicama (skraćeno VJ)
- Zadaća UI jedinice je oslobođanje procesora od učestale komunikacije s uređajem i prilagodba načina komunikacije uređaja komunikaciji putem sabirnice računala



Povezivanje računala s okolinom

- Kakve sve mogu biti UI jedinice?
- Možemo ih podijeliti prema smjeru prijenosa:
 - ulazne
 - izlazne
 - dvosmjerne
- Možemo ih podijeliti prema namjeni:
 - za prijenos podataka
 - za brojanje impulsa
 - za mjerenje vremena
 - za generiranje impulsa
 - za AD i DA pretvorbu
 - specijalne namjene, itd.

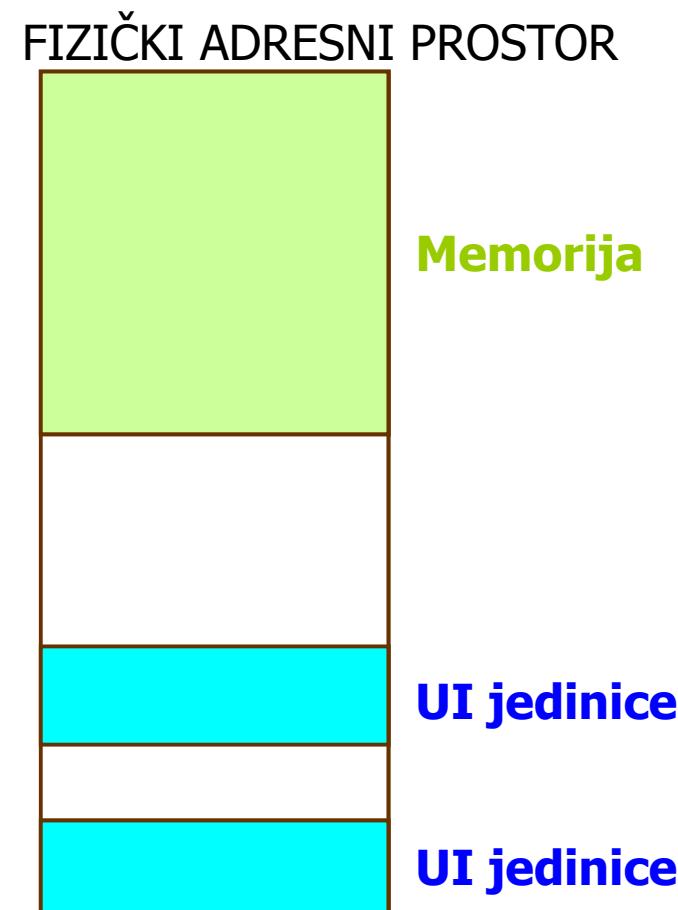
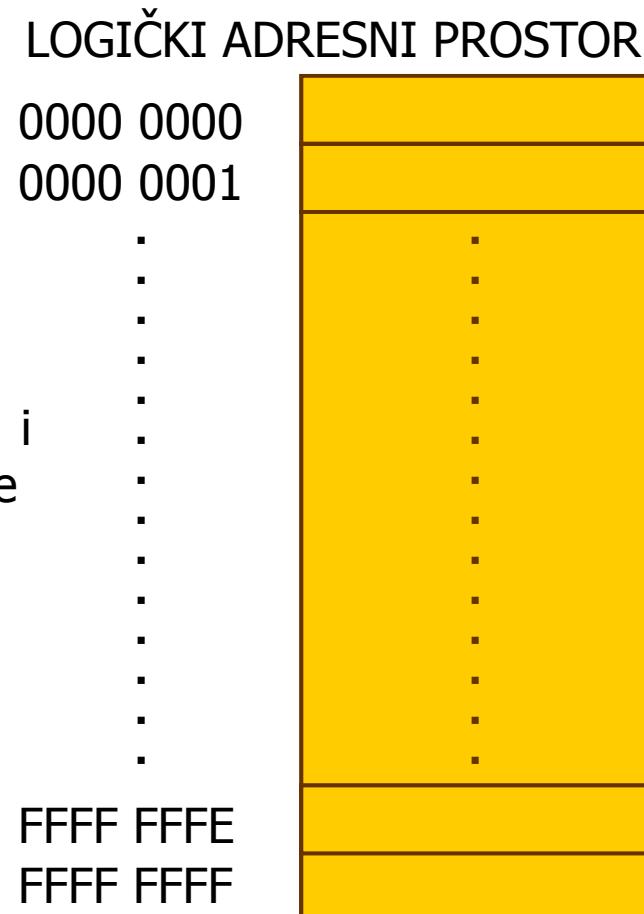
Komunikacija procesora s UI jedinicama

Načini adresiranja UI jedinica

- Načelno, **kod UI komunikacije postoje dvije operacije** kao i kod pristupanja memoriji:
 - čitanje
 - pisanje
- Na sabirnicu je spojen veći broj UI jedinica i procesor mora točno odabrati onu jedinicu s kojom želi komunicirati - to se radi pomoću adresne sabirnice
- Dva osnovna načina adresiranja UI jedinica su:
 - **memorijsko UI preslikavanje** (memory mapped IO)
 - **izdvojeno UI adresiranje** (isolated IO)

Memorijsko UI preslikavanje

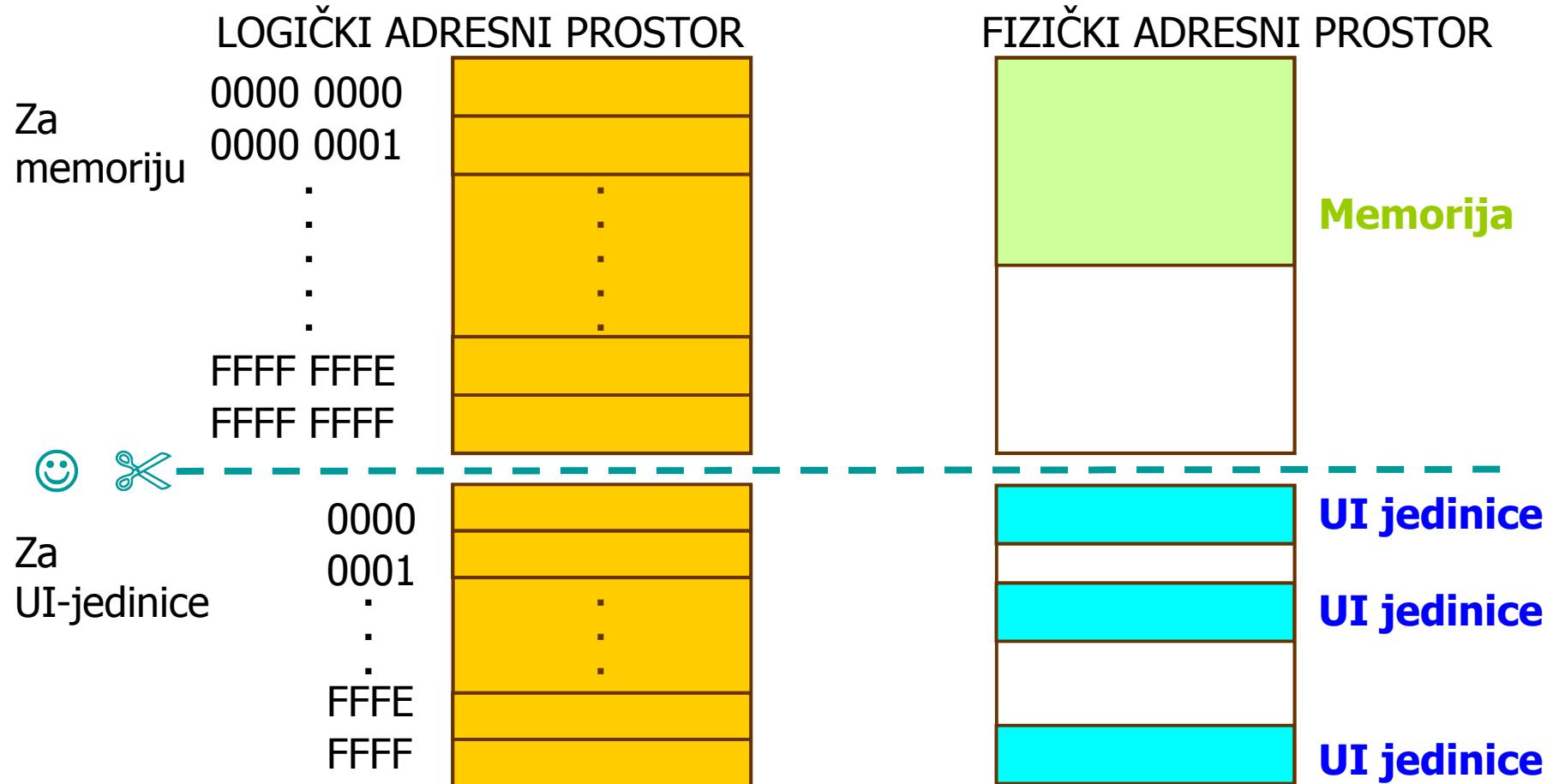
Za
memoriju i
UI-jedinice



Memorijsko UI preslikavanje

- **Memorijsko UI preslikavanje** ima sljedeće značajke:
 - procesor na jednak način pristupa memorijskim lokacijama i UI jedinicama
 - sabirnički protokoli čitanja i pisanja jednaki su za pristup memoriji i UI jedinicama

Izdvojeno UI adresiranje



Izdvojeno UI adresiranje

- **Izdvojeno UI adresiranje** ima sljedeće značajke
 - procesor na različite načine pristupa memoriji i UI jedinicama.
 - procesor ima posebne priključke kojima određuje je li neka adresa (npr. 10), koja se nalazi na adresnoj sabirnici, namijenjena memoriji ili UI jedinici

Usporedba

- Prednosti i nedostaci memorijskog i izdvojenog adresiranja

Komunikacija FRISC-a s UI jedinicama

- Za FRISC ćemo odabratи memorijsko preslikavanje:
 - bolje je prilagođeno RISC procesorima zbog veće jednostavnosti (samo jedan sabirnički protokol i nepotrebne dodatne naredbe)
 - memorijski adresni prostor je više nego dovoljan za naše potrebe pa možemo jedan njegov dio odvojiti za UI jedinice
 - komunikacija s UI jedinicama odvija se pomoću naredaba LOAD i STORE budući da su to jedine naredbe koje pristupaju memoriji
 - umjesto adresa memorijskih lokacija, u naredbama LOAD i STORE koristit će se adrese UI jedinica
 - **u zadatcima i na labosima po dogovoru*** uzimamo da je na FRISC spojena memorija na adresama 0 do FFFF (64 kB), a UI jedinice nalazit će se na adresama FFFF0000 do FFFFFFFF (sve možemo dohvatiti 20-bitnom adresom)

Vrste UI prijenosa podataka



- UI jedinice rade svojom brzinom koja je **različita** (obično manja) od brzine procesora
- Prije prenošenja podataka, obično se procesor i UI jedinica trebaju **sinkronizirati**, tj. uskladiti svoj rad da bi se prijenos podataka uspješno izveo
- S obzirom na to, prijenos se može dijeliti na:
 - **programski prijenos**
 - **sklopopovski prijenos**

Programski prijenos

- Glavne karakteristike **programskog prijenosa** su:
 - Prijenos se obavlja **pod upravljanjem programa**, tj. prijenos obavlja procesor (upravljan programom)
 - Svi podatci koji se prenose "prolaze kroz procesor"
 - Programski prijenos je **sporiji** od sklopoškog prijenosa
 - Procesor dio vremena troši na prijenos podataka što **usporava izvođenje ostalih poslova**
 - U nastavku ćemo vidjeti tri vrste programiranog prijenosa:
 - **bezuvjetni**
 - **uvjetni**
 - **prekidni**

Sklopoški prijenos

- Glavne karakteristike **sklopoškog prijenosa** su:
 - prijenos se obavlja pod upravljanjem specijaliziranog sklopolja, tj. **prijenos obavlja specijalna jedinica** (DMA kontroler), a procesor ne sudjeluje u prijenosu
 - podatci koji se prenose prolaze kroz specijalnu jedinicu, a ovisno o njenoj građi i organizaciji moguće je čak i da se prenose izravno između UI jedinice i memorije
 - prijenos se općenito odvija **velikim brzinama**
 - procesor ne troši vrijeme na prijenos podataka, ali **izvođenje programa je ipak usporeno** za vrijeme obavljanja prijenosa
 - na kraju ovog poglavlja ćemo vidjeti jednu vrstu sklopoškog prijenosa:
 - **izravni pristup memoriji (DMA)**

Bezuvjetni prijenos

Bezuvjetni prijenos

- **Bezuvjetni prijenos** je najjednostavnija vrsta prijenosa
- Glavna značajka je da se prije prijenosa **ne provjerava** je li UI jedinica spremna za prijenos podataka
 - nema sinkronizacije brzine rada između procesora i UI jedinice
- Dijagram toka je trivijalan i sastoji se samo od prijenosa:



- Prijenos je jedna naredba (eventualno više njih) za čitanje iz VJ ili za pisanje u VJ

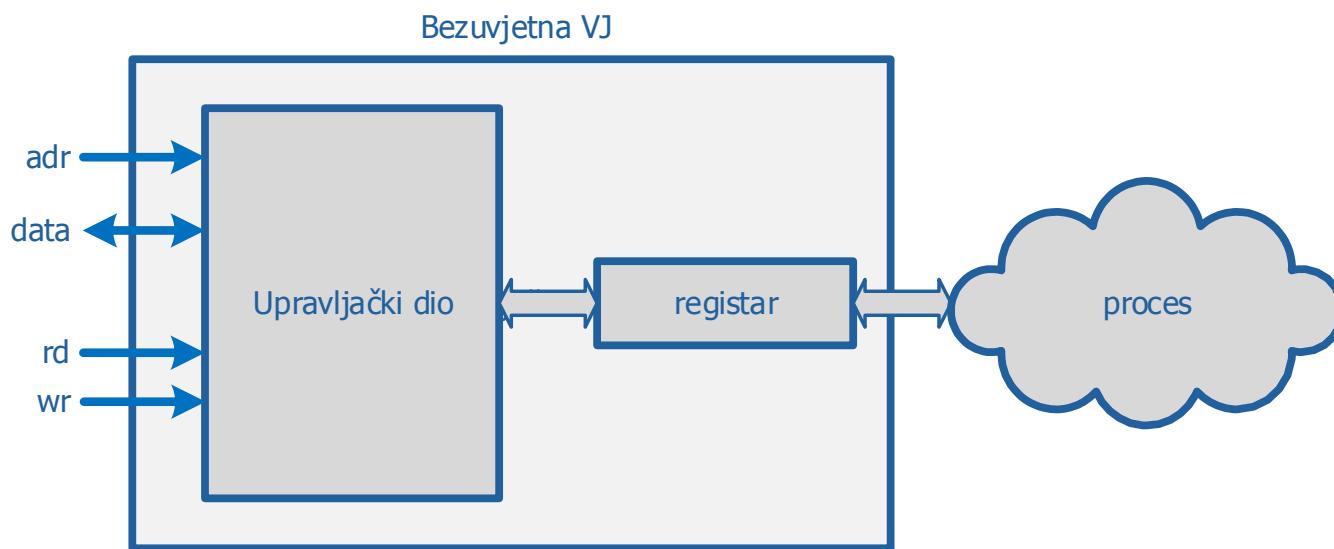
Bezuvjetni prijenos

- UI jedinica je sklopovski **najjednostavnija**
- **Najbrži** prijenos među programiranim prijenosima
- **Nedostatak je mogućnost da UI jedinica nije spremna za prijenos...**

Bezuvjetni prijenos

- Kad možemo koristiti bezuvjetni prijenos?
- Općenito:
 - kad nam nije bitna sinkronizacija
 - kad znamo da nećemo pristupati UI jedinici brže nego što ona radi
- Na primjer: želimo očitati trenutačnu vrijednost temperature u prostoriji preko temperaturnog senzora
- Na primjer: želimo svake sekunde osvježiti prikaz vremena na zaslonu (npr. na 7-segmentnom LCD-u)

Bezuvjetna VJ

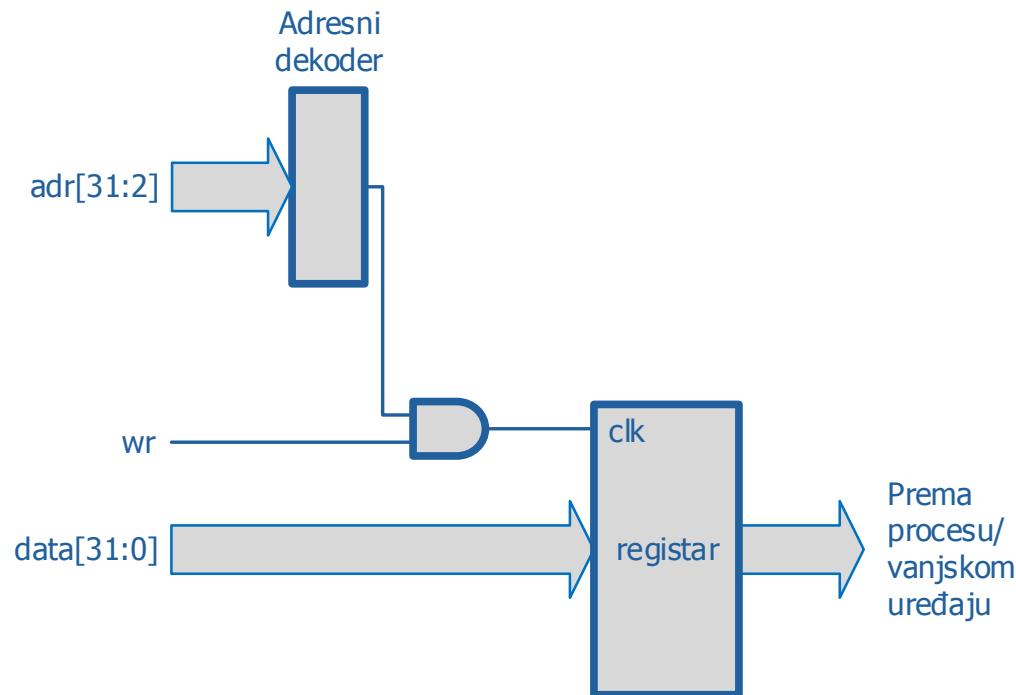


Osnovna građa bezuvjetne VJ

- Sve naše VJ zauzimat će određeni broj uzastopnih lokacija (jednu ili više njih)
- Zbog jednostavnosti, sve ove lokacije će biti 32-bitne pa ćemo s VJ komunicirati korištenjem naredaba LOAD i STORE
 - Dakle, svaka lokacija će zauzimati 4 adrese
- Bezuvjetna VJ zauzima samo jednu lokaciju

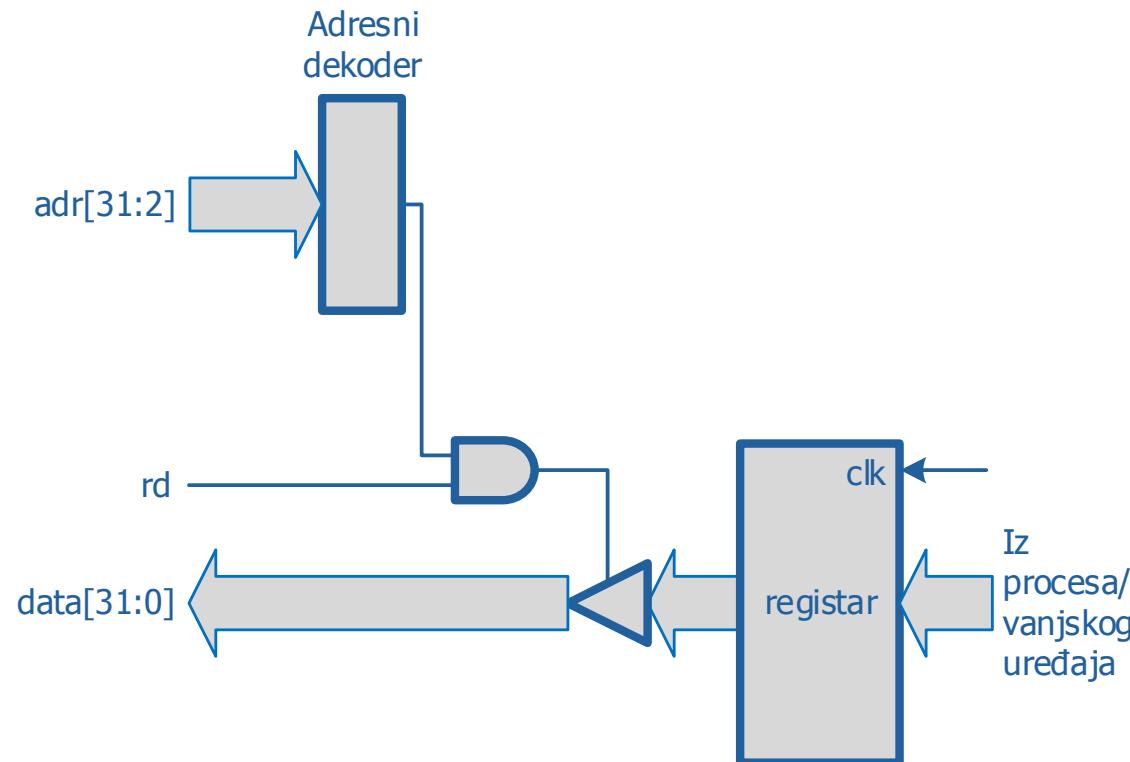
Bezuvjetna VJ

- Arhitektura jednostavne izvedbe bezuvjetne izlazne VJ



Bezuvjetna VJ

- Arhitektura jednostavne izvedbe bezuvjetne ulazne VJ



Bezuvjetni prijenos - Primjeri

Na bezuvjetnu izlaznu VJ0 na adresi FFFF0000 spojen je relej. Slanjem broja 0 relej se isključuje, a slanjem 1 se uključuje. Na bezuvjetnu ulaznu VJ1 na adresi FFFF 0010 spojena je tipka. Kad je tipka pritisnuta, s VJ se očitava stanje 1, a kad tipka nije pritisnuta, očitava se stanje 0.

Napisati program koji treba ispitivati je li tipka pritisnuta i samo tada držati relej uključen. Građa VJ0 i VJ1 je kao u prethodnom opisu.

Bezuvjetni prijenos - Primjeri

```
RELEJ EQU 0FFFF0000 ; Definiranje naziva VJ
```

```
TIPKA EQU 0FFFF0010
```

```
; Glavni program
```

```
PETLJA LOAD R0, (TIPKA)
```

```
    OR     R0, R0, R0
```

```
    JR_Z ISKLJUCI
```

```
UKLJUCI MOVE 1, R0
```

```
    STORE R0, (RELEJ)
```

```
    JR     PETLJA
```

```
ISKLJUCI MOVE 0, R0
```

```
    STORE R0, (RELEJ)
```

```
    JR     PETLJA
```

Bezuvjetni prijenos - Primjeri

Na bezuvjetnu ulaznu vanjsku jedinicu na adresi FFFF0000 spojen je digitalni termometar s kojeg se može očitati trenutna vrijednost temperature. Vanjska jedinica građena je kao što je pokazano na prethodnim slikama.

Program mora svakih 5 minuta očitati temperaturu, izračunati srednju vrijednost svih dotadašnjih temperatura i spremiti je u lokaciju SR_TEMP. Nakon tri sata treba zaustaviti procesor. Pretpostavite da već postoji potprogram za dijeljenje (DIJELI) kojem su parametri u fiksnim memorijskim lokacijama SUMA i BROJ_OCITANJA a rezultat se vraća registrom R0.

Pretpostavka je da se vrijednost temperature vraća u nižih 8 bitova, a gornjih 24 bita su nule, te da procesor radi sa signalom vremenskog vođenja clock frekvencije 10 MHz.

```

; Definiranje adrese vanjske jedinice
TERMOMET EQU 0FFFF0000

; Glavni program

MOVE 10000, R7
; Inicijalizacija varijabli
MOVE 0, R0
STORE R0, (SUMA)
STORE R0, (BROJ_OCITANJA)
STORE R0, (SR_TEMP)
MOVE %D 36, R1 ; brojač za 3 sata: 36*5 min = 3 sata
PETLJA LOAD R0, (TERMOMET) ; Učitaj temperaturu
CALL RACUNAJ_SREDNJIU ; Izračunaj i spremi
STORE R0, (SR_TEMP) ; srednju temperat.
CALL KASNI_5_MINUTA
SUB R1, 1, R1 ; Istečela 3 sata?
JR_NZ PETLJA
HALT

```

```
; Potprogram za izračun sredine:  
RACUNAJ_SREDNU ; Parametar R0: nova temperatura  
; Rezultat u R0  
PUSH R1  
LOAD R1, (SUMA)  
ADD R0, R1, R1  
STORE R1, (SUMA)  
LOAD R1, (BROJ_OCITANJA)  
ADD R1, 1, R1  
STORE R1, (BROJ_OCITANJA)  
CALL DIJELI  
POP R1  
RET
```

```
; Potprogram za (približno) 5-minutno kašnjenje  
; Nema parametara ni rezultata
```

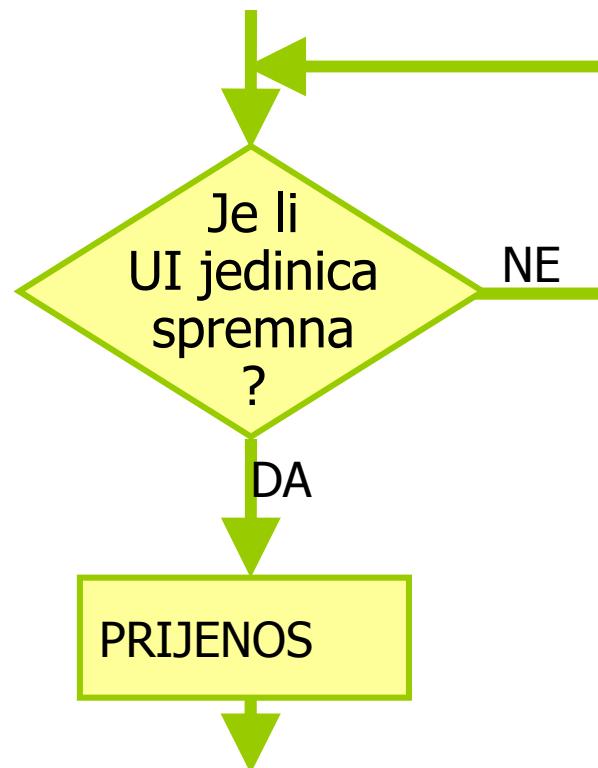
KASNI_5_MINUTA

```
PUSH  R0  
  
LOAD  R0, (KONST)      ; 109 prolazaka  
  
LOOP   SUB   R0, 1, R0      ; 1 takt  
        JR_NZ LOOP          ; 2 takta  
  
POP    R0  
  
RET  
  
KONST  DW  %D 1000000000  
  
; Ovaj potprogram napisan je za CLOCK frekvencije 10 MHz:  
; 5 min = 300 s = 300*10*106 taktova = 3*109 taktova  
;  
; Varijable  
  
SUMA      DW 0  
  
BROJ_OCITANJA DW 0  
  
SR_TEMP    DW 0
```

Uvjetni prijenos

Uvjetni prijenos

- **Uvjetni prijenos** rješava probleme gubitka i uvišestručenja podataka kod bezuvjetnog prijenosa
- Glavna značajka je da se prije prijenosa **uvijek provjerava** je li VJ spremna za prijenos podataka



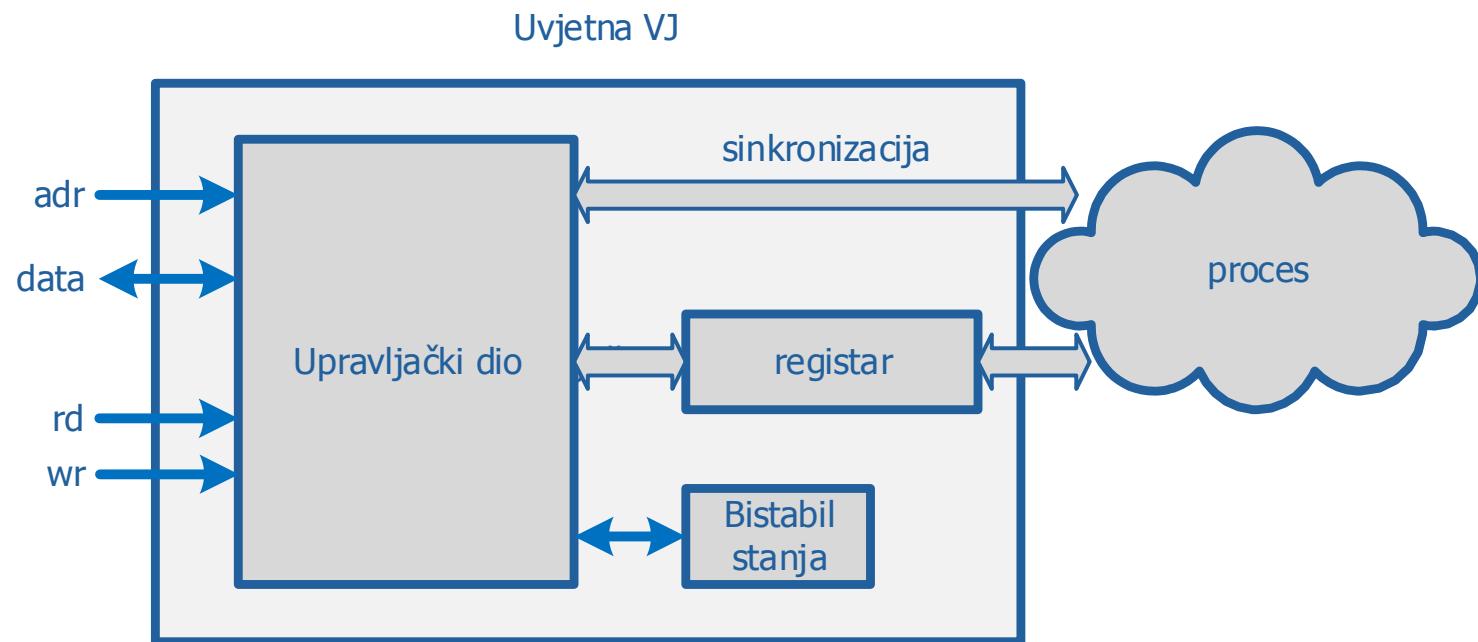
Uvjetni prijenos

- Glavni **nedostatak** uvjetnog prijenosa:
 - Budući da je procesor tipično puno brži od vanjskih uređaja, može se dogoditi da procesor puno vremena gubi na čekanje da VJ postane spremna
- Uvjetna VJ je sklopovski složenija od bezuvjetne VJ:
 - VJ mora imati stanje spremnosti koje se pamti u **bistabilu stanja** (u tzv. status-bistabilu)
 - VJ mora imati dodatne **sinkronizacijske priključke** za povezivanje s vanjskim uređajem

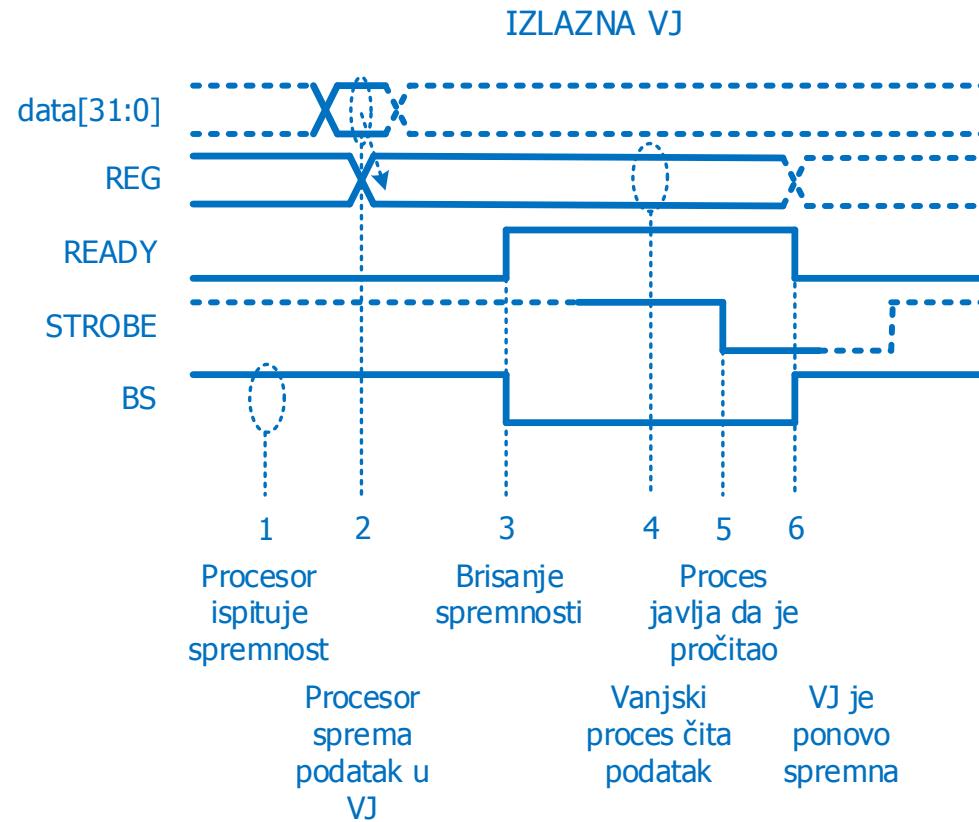
Uvjetni prijenos

- Kad možemo koristiti uvjetni prijenos?
- Općenito:
 - kad nam je bitno da nema gubitaka/uvišestručenja podataka (tj. bitna nam je sinkronizacija)
 - kad ne znamo kojom brzinom ćemo pristupati VJ pa se može dogoditi da ona još nije spremna
- Na primjer: želimo slati znakove na pisač (naravno, pri tome je bitno da svi znakovi budu primljeni i ispisani)
- Na primjer: želimo čitati niz podataka koji od nekuda (npr. s mreže ili iz tipkovnice) pristižu na VJ i koje sve želimo bez gubitaka pohraniti u memoriju

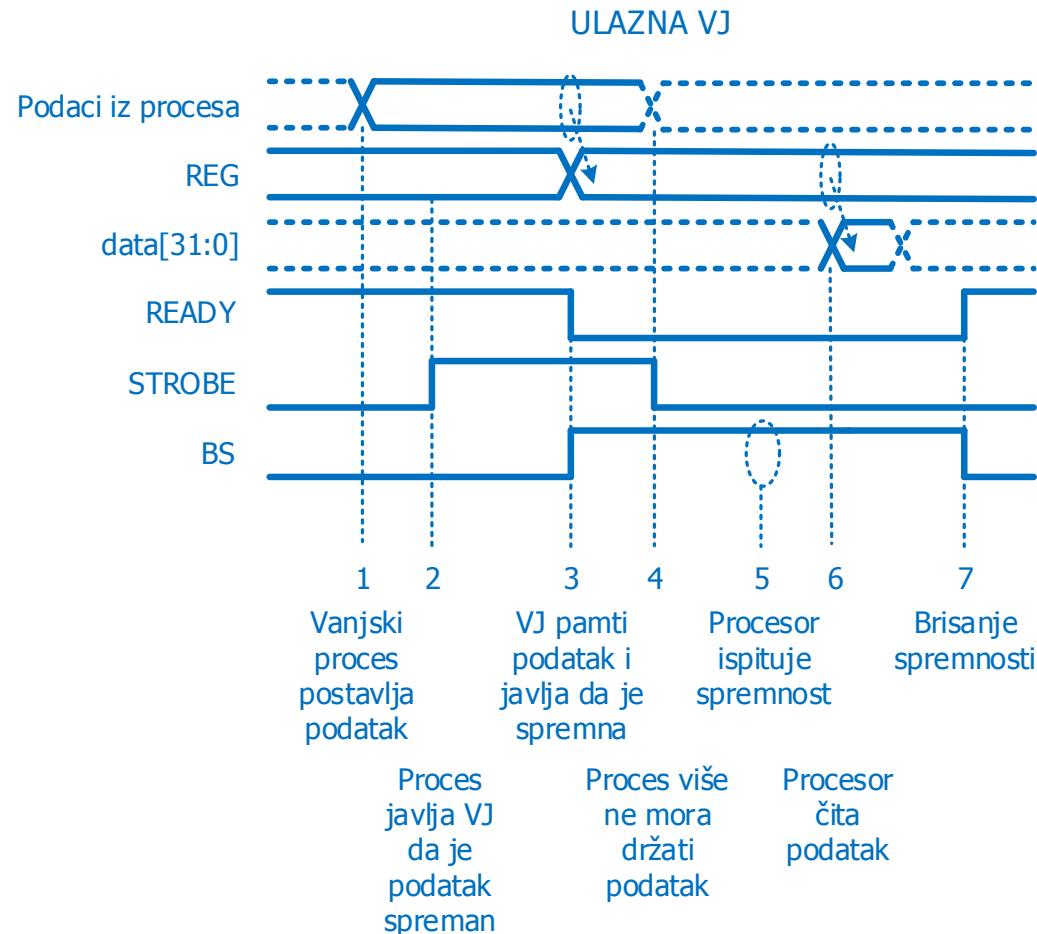
Osnovna građa uvjetne UI jedinice



Vremenski dijagram uvjetne izlazne komunikacije



Vremenski dijagram uvjetne ulazne komunikacije



Uvjetni prijenos - rekapitulacija

- Iz prethodnih opisa vidi se da je uvijek vrijedi:
 - Dok je VJ spremna za komunikaciju s procesorom, nije spremna za komunikaciju s vanjskim procesom (i obrnuto)
- Bistabil stanja služi za sinkronizaciju između VJ i procesora:
 - Procesor programski ispituje b.stanja da bi utvrdio spremnost VJ
 - VJ postavlja bistabil stanja kad postane spremna
 - Bistabil stanja briše se (programska ili automatska) nakon obavljenog prijenosa
- **Brisanjem bistabila stanja, omogućuje se nastavak komunikacije s vanjskim procesom**
- Sinkronizacijski priključci služe za sinkronizaciju između VJ i vanjskog procesa:
 - Vanjski proces sklopovski ispituje sinkronizacijske priključke da bi utvrdio spremnost VJ

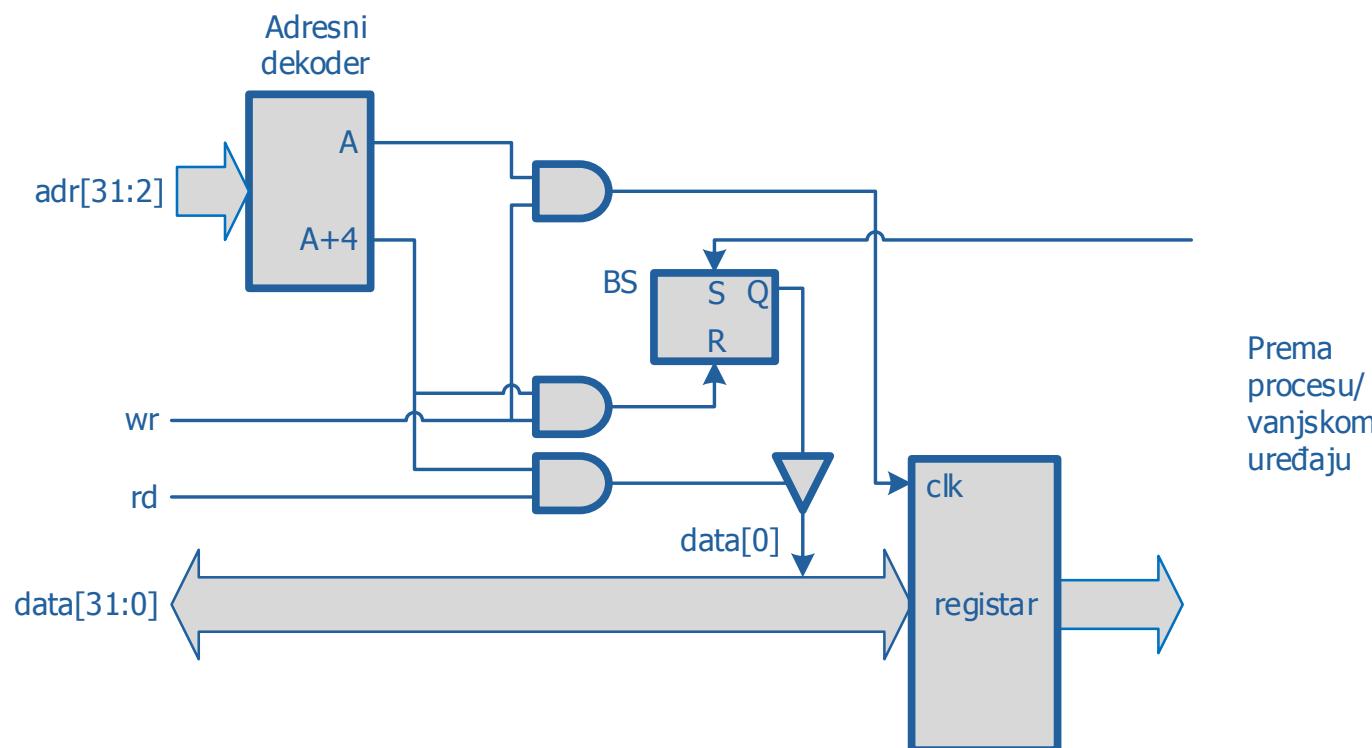
Osnovna građa uvjetne UI jedinice

- Ovakva vanjska jedinica **zauzimat će dvije uzastopne 32-bitne lokacije**
 - Na prvoj lokaciji se čita ili piše podatak
 - Na drugoj lokaciji se pristupa bistabilu stanja
- Na lokaciji za bistabil stanja može se:
 - Pročitati trenutačni sadržaj bistabila (ispitivanje stanja)
 - Obrisati bistabil (operacijom upisa bilo kojeg podatka - poslani podatak se zanemaruje) *

* Za naše uvjetne UI jedinice ćemo pretpostavljati da se bistabil stanja uvijek mora brisati programski (tj. pretpostavljamo da se to ne radi automatski)

Arhitektura izlazne uvjetne VJ

- Pojednostavljeni prikaz upravljačkog dijela izlazne uvjetne VJ



Uvjetni prijenos - Primjeri

- Napišite program koji će na uvjetnu vanjsku jedinicu (građenu kao što je upravo opisano) koja se nalazi na adresi 0FFFF0000 poslati 200 32-bitnih podataka koji se nalaze u memoriji od lokacije PODACI. Nakon što su svi podaci poslani, treba zaustaviti procesor.

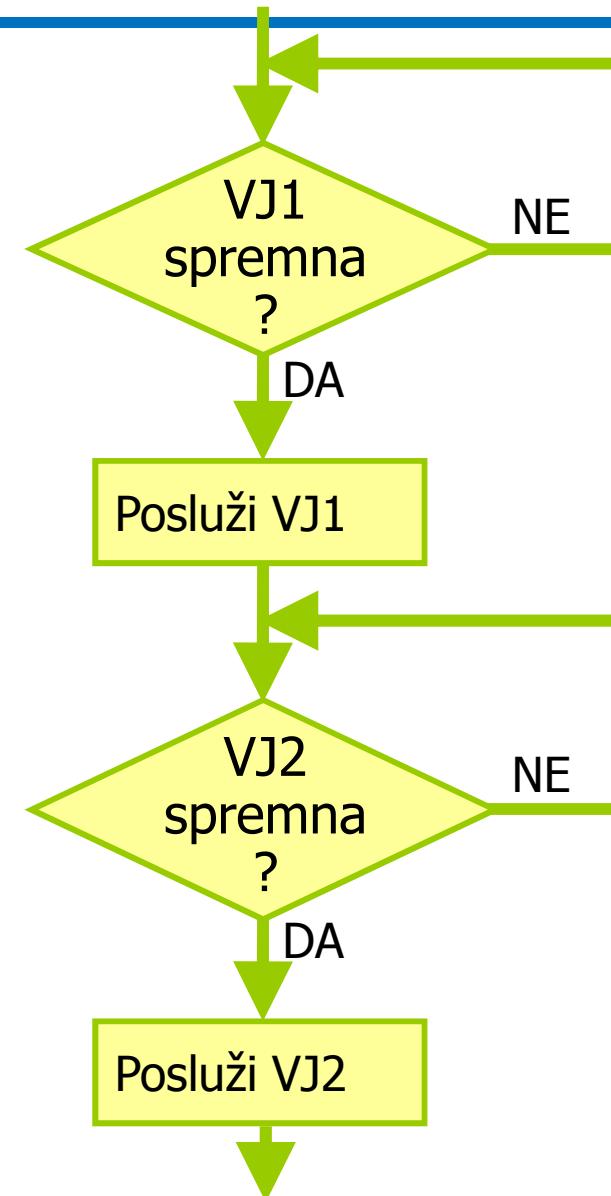
```
; Definiranje adresa vanjske jedinice
REG EQU      0FFFF0000
BS EQU      0FFFF0004

; Glavni program
; Inicijalizacija varijabli
MOVE    PODACI, R0 ; adresa podataka
MOVE    %D 200, R1 ; brojač petlje
CEKAJ   LOAD  R2, (BS); čitaj stanje iz VJ
        OR     R2, R2, R2 ; postavi zastavice
        JR_Z  CEKAJ       ; ispitaj stanje VJ
SPREMNA LOAD  R2, (R0) ; čitaj podatak iz mem.
        STORE R2, (REG); šalji podatak u VJ
        STORE R2, (BS); briši stanje VJ
        ADD    R0, 4, R0
        SUB    R1, 1, R1
        JR_NZ CEKAJ
HALT

PODACI DW 12, 5452, 331A3, ...
```

Posluživanje više uvjetnih vanjskih jedinica

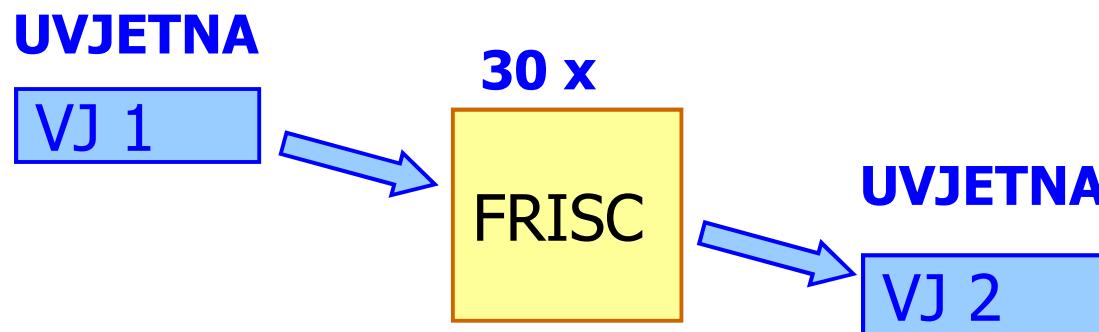
- Ako su **jedinice zavisne**, onda moramo za svaku **čekati** da postane spremna
- Zavisne jedinice znače da je redoslijed njihovog posluživanja bitan
- Na slici je primjer gdje se podatak primljen od VJ1 prenosi na VJ2



Uvjetni prijenos – dvije VJ

Na FRISC su spojene dvije uvjetne vanjske jedinice: ulazna vj1 na adresi FFFF1000 te izlazna vj2 adresi FFFF2000.

FRISC treba prenijeti 30 podataka sa VJ1 na VJ2 nakon čega nastavlja s izvođenjem glavnog programa.



```

; Definiranje adresa vanjskih jedinica
PRIMI_1 EQU 0FFFF1000
BS_1     EQU 0FFFF1004
SALJI_2 EQU 0FFFF2000
BS_2     EQU 0FFFF2004
; Glavni program
        MOVE 30, R3          ; brojač podataka
CEKAJ_1 LOAD  R0, (BS_1)
        OR    R0, R0, R0
        JR_Z CEKAJ_1          ; Čekaj da VJ1 postane spremna
        LOAD  R1, (PRIMI_1)    ; Posluži VJ1
        STORE R0, (BS_1)

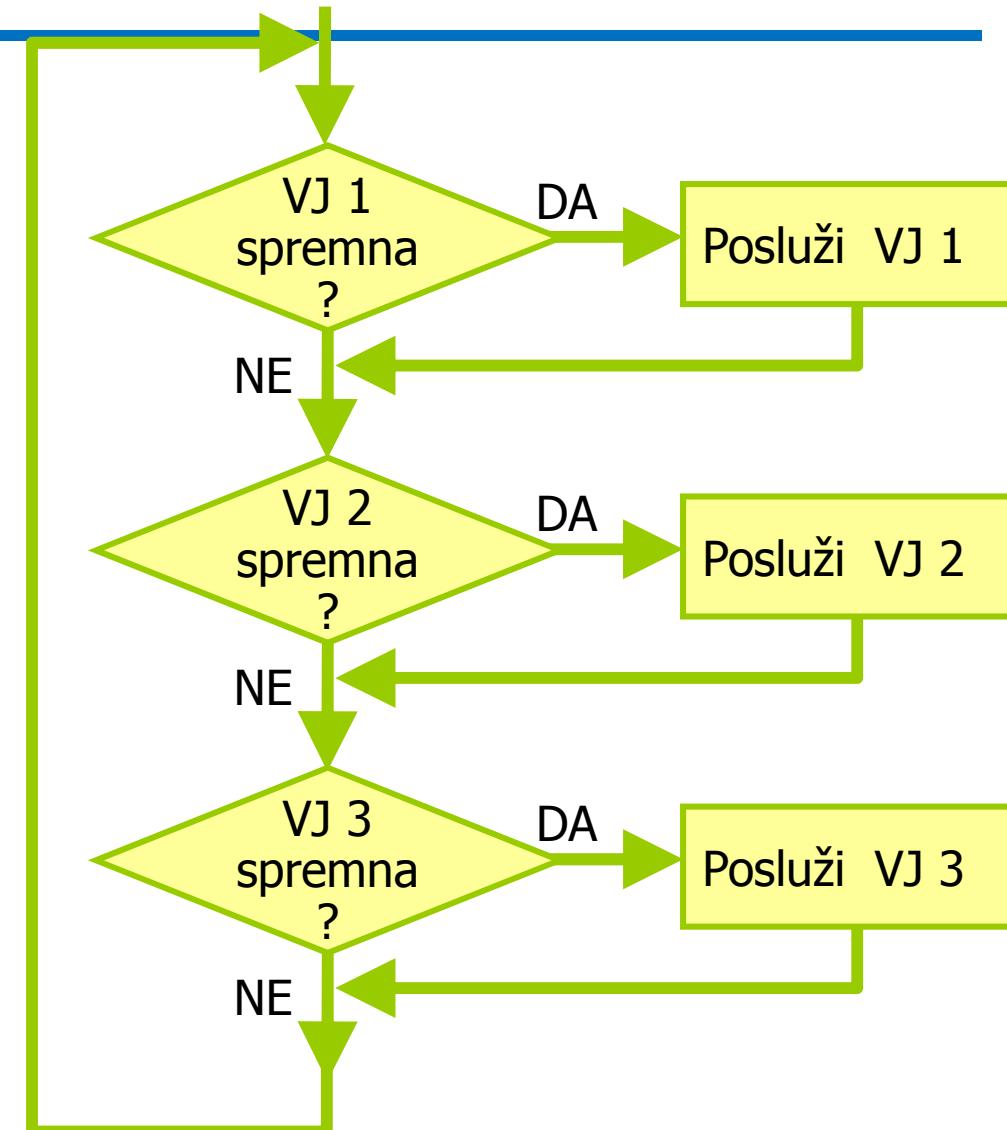
CEKAJ_2 LOAD  R0, (BS_2)
        OR    R0, R0, R0
        JR_Z CEKAJ_2          ; Čekaj da VJ2 postane spremna
        STORE R1, (SALJI_2)    ; Posluži VJ2
        STORE R0, (BS_2)

        SUB   R3, 1, R3
        JR_NZ CEKAJ_1          ; Ispitaj je li preneseno svih 30 podataka

```

Posluživanje više uvjetnih vanjskih jedinica

- Ako imamo **više nezavisnih uvjetnih vanjskih jedinica**, onda možemo primijeniti postupak **prozivanja** (eng. polling)
- Nezavisne jedinice znače da jedna ne ovisi o drugoj, tj. da redoslijed posluživanja nije bitan
- **Prozivanje umanjuje nedostatak uvjetnog prijenosa:** čekanje na spremnost pojedine vanjske jedinice
 - Veća je vjerojatnost da će jedna od nekoliko jedinica postati spremna
- Nakon posluživanja jedne jedinice prelazi se na sljedeću jedinicu
 - Ne bi bilo dobro vratiti se na ispitivanje prve zbog mogućeg "izgladnjivanja" zadnjih jedinica u lancu ispitivanja

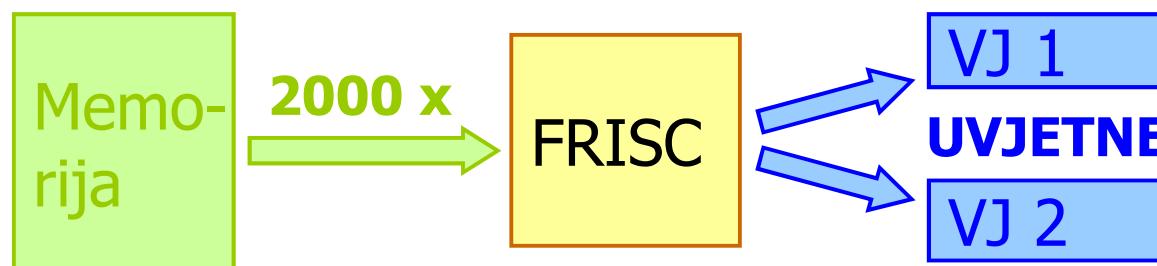


Uvjetni prijenos – primjer

Na FRISC su spojene dvije uvjetne vanjske jedinice: vj1 na adresi FFFF1000 i vj2 adresi FFFF2000.

Obje jedinice rade neovisno i FRISC im šalje 2000 podataka smještenih od adrese PODACI. Kako koja jedinica postane spremna tako joj se pošalje podatak (to znači da brža jedinica prima više podataka).

Kada je završen prijenos svih 2000 podataka, treba zaustaviti procesor.



```
; Definiranje adresa vanjskih jedinica
```

```
SALJI_1    EQU    0FFFF1000
```

```
BS_1       EQU    0FFFF1004
```

```
SALJI_2    EQU    0FFFF2000
```

```
BS_2       EQU    0FFFF2004
```

```
; Glavni program
```

```
MOVE      10000, R7
```

```
MOVE      2000,  R1      ; brojač podataka
```

```
MOVE      PODACI, R2      ; adresa podataka
```

```
; POSTUPAK PROZIVANJA
```

```
PROZIVAJ  LOAD     R0, (BS_1)
```

```
OR       R0, R0, R0
```

```
CALL_NZ SALJI_VJ1
```

```
OR       R1, R1, R1
```

```
JR_Z    KRAJ
```

```
LOAD     R0, (BS_2)
```

```
OR       R0, R0, R0
```

```
CALL_NZ SALJI_VJ2
```

```
OR       R1, R1, R1
```

```
JR_NZ   PROZIVAJ
```

```
KRAJ    HALT
```

```
SALJI_VJ1 LOAD R0, (R2) ; podatak iz memorije
    STORE R0, (SALJI_1) ; šalji podatak
    STORE R0, (BS_1) ; briši spremnost
    ADD   R2, 4, R2 ; pomakni pokazivač
    SUB   R1, 1, R1 ; smanji brojač
    RET
```

; potprogram za posluživanje vj2, isto kao i za VJ1

```
SALJI_VJ2 LOAD R0, (R2)
    STORE R0, (SALJI_2)
    STORE R0, (BS_2)
    ADD   R2, 4, R2
    SUB   R1, 1, R1
    RET
```

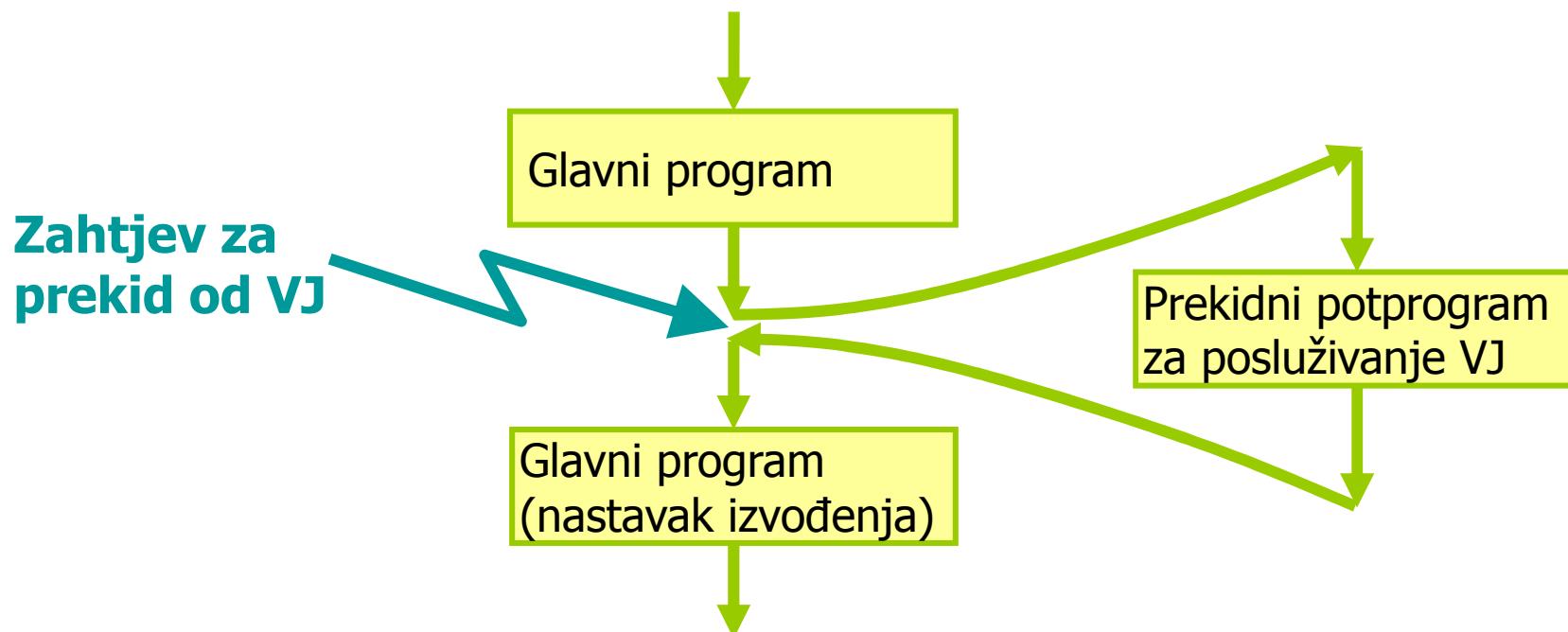
Posluživanje više VJ - Komentar

- Načelno, kod posluživanja više vanjskih jedinica treba odrediti redoslijede čitanja/pisanja u ovisnosti o samim jedinicama:
 - Uvjetnim VJ treba uvijek ispitati spremnost i poslužiti ih čim se ustanovi da su spremne
 - Ako su uvjetne VJ nezavisne, onda ih se proziva
 - Ako su uvjetne VJ zavisne, onda se mora čekati da postanu spremne
- Bezuvjetne VJ treba čitati/pisati tek kad zatreba.
 - Nema smisla npr. pročitati podatak s bezuvjetne, a onda čekati da uvjetna postane spremna kako bi joj poslali taj podatak.
 - Treba pokušati raditi s "najsvježijim" podatcima: npr. čekati da uvjetna postane spremna i tek onda pročitati podatak s bezuvjetne i odmah ga poslati uvjetnoj

Prekidni prijenos

Prekidni prijenos

- Glavna značajka prekidnog prijenosa je da **UI jedinica samostalno dojavljuje** svoju spremnost procesoru koji za to vrijeme normalno izvodi neki program
 - Spremnost se dojavljuje zahtjevom za prekid (engl. interrupt request)



Iz dijagrama toka vidi se da zahtjev za prekidom (ili kraće prekid) može doći u bilo kojem trenutku izvođenja glavnog programa

Prekidni prijenos

- **Prekidni prijenos** rješava:
 - problem gubitka i uvišeSTRUČENJA podataka (koji postoji kod bezuvjetnog prijenosa)
 - gubitka vremena na čekanje spremnosti (koji postoji kod uvjetnog prijenosa)
- Prekidni prijenos je učinkovitiji od uvjetnog (u smislu količine dodatnog posla kojeg procesor u jedinici vremena može obaviti uz komunikaciju s VJ), ali ipak nije tako učinkovit kao bezuvjetni (zato što se vrijeme se troši na prihvaćanje zahtjeva za prekid, odlazak u prekidni potprogram i povratak iz prekidnog potprograma)
- Prekidna jedinica građena je slično uvjetnoj, ali je ipak nešto složenija
- Prekidni prijenos koristimo u istim slučajevima kad i uvjetni, ali kad nam je važno da procesor može izvoditi neki program bez usporenja zbog čekanja spremnosti VJ

Prekidni sustavi procesora

- Prekidni sustavi tako se razlikuju od procesora do procesora
 - zato nećemo objašnjavati sve moguće varijante prekidnih sustava
 - orijentirat ćemo se na konkretni prekidni sustav procesora FRISC (i kasnije procesora ARM)
- Prekidni sustav definira sljedeće:
 - koliko prekidnih priključaka procesor ima i koji su im prioriteti
 - kako procesor potvrđuje UI jedinici da je prihvatio zahtjev za prekid
 - kako se određuje adresa prekidnog potprograma
 - može li se prekidni potprogram ponovno prekinuti i kako
 - kako se prepoznae UI jedinica koja je izazvala prekid
 - kako se procesoru može dozvoliti ili zabraniti prihvatanje prekida
 - kako jedinica zna da je njen posluživanje dovršeno
 - kako se obavlja poziv i povratak iz prekidnog potprograma

Prekidni sustavi procesora

- Načelno ponašanje procesora s obzirom na prekide:
 1. Procesor **izvodi program**, a VJ postavlja **zahtjev za prekid**
 2. Procesor izvodi trenutnu naredbu **do kraja**, tj. ispituje ima li zahtjeva za prekid tek na kraju izvođenja naredbe
 3. Ako je u procesoru dozvoljeno prihvatanje postavljenog prekida, onda procesor **prihvaca prekid**, a u suprotnom nastavlja s radom
 4. Prihvatanje prekida sastoji se od:
 1. Procesor **zabranjuje** prihvatanje dalnjih prekida (osim eventualno prekida jačeg prioriteta ako ih podržava)
 2. Procesor **određuje adresu prekidnog potprograma**
 3. Procesor **pohranjuje register PC**, a često i **register stanja** (može pohranjivati i druge registre)
 4. Procesor **skače u prekidni potprogram**

Prekidni sustavi procesora

- Načelno ponašanje prekidnog potprograma (skraćeno p.p.):
 1. **Sprema se kontekst** (sve registre koje potprogram mijenja, a nisu automatski spremjeni prilikom prihvaćanja prekida)
 2. **Otkriva se uzročnik prekida** (ako ih ima više), tj. otkriva se koja VJ je izazvala prekid *
 3. **Dojavljuje se VJ da je njen prekid prihvaćen** (VJ mora ukloniti zahtjev za prekid) *
 4. **Poslužuje se VJ**
 5. **Obnavljanje konteksta**
 6. **Ponovno dozvoljavanje prekida ****
 7. **Dojavljuje se VJ da je njezin prekid obrađen** (VJ može nastaviti s radom, ponovno postati spremna i zahtijevati prekid) **, ***
 8. **Izlazak iz prekidnog potprograma** i povratak u glavni program na mjesto gdje je bio prekinut

* ovisno o procesoru može se izvesti sklopovski

** ovisno o procesoru može se izvesti sklopovski prilikom koraka 8

*** ovaj korak ne postoji kod svih procesora

Prekidni sustav procesora FRISC

Prekidni sustav FRISC-a

- Prekidni priključci FRISC-a su na sabirnici int[1:0]
 - int[0] - maskirajući prekid (označava se i s INT)
 - int[1] - nemaskirajući prekid (označava se i s NMI)
- **Maskirajući prekid** (maskable interrupt) možemo programski zabraniti ili onemogućiti (maskirati). Ovdje se radi o dozvoljavanju ili zabranjivanju **prihvaćanja prekida** od strane procesora (ne o dozvoljavanju ili zabranjivanju postavljanja zahtjeva od strane VJ)
- **Nemaskirajući prekid** (nonmaskable interrupt) ne možemo ga zabraniti
- Nemaskirajući prekid je **višeg prioriteta** od maskirajućeg
- Maskirajući prekid je **inicijalno zabranjen** (nemaskirajući je, naravno, dozvoljen)

Prekidni sustav FRISC-a

- Prekid se maskira pomoću prekidne zastavice GIE (global interrupt enable) u registru stanja SR:



- **GIE**
 - 0: prihvatanje maskirajućeg prekida zabranjeno
 - 1: prihvatanje maskirajućeg prekida dozvoljeno

Prekidni sustav FRISC-a

- FRISC ima zastavicu **IIF** (internal interrupt flag) koja nije u registru SR:
 - ova zastavica nije dostupna programeru, a koristi se kod nemaskirajućeg prekida NMI
 - početno stanje IIF je 1
 - dok se ne obrađuje NMI, IIF je u stanju 1
 - Čim se prihvati NMI, IIF se automatski prebacuje u stanje 0
 - IIF se automatski vraća u stanje 1 po povratku iz NMI
 - dok se obrađuje NMI (na temelju stanja IFF=0):
 - novi zahtjev NMI se ne prihvata
 - zahtjevi sa INT se ne prihvataju

Prekidni sustav FRISC-a

- **Ispitivanje prekida kod FRISC-a:**
 - Postojanje prekida ispituje se na kraju perioda CLOCK-a (može se reći i na padajući brid CLOCK-a)
 - Naredba koja je u razini izvođenja se izvodi do kraja
 - Ispitivanje i prihvaćanje prekida ovisi o stanju zastavica i trenutačnim zahtjevima za prekid:
 - Ako je IIF=0, prekidi se ne prihvaćaju
 - U suprotnom, ako je NMI prisutan, on se prihvaca, a ako NMI nije prisutan, onda se ispituju maskirajući prekidi
 - Ako je GIE=0, maskirajući prekid se ne prihvaca
 - U suprotnom se maskirajući prekid prihvaca

Prekidni sustav FRISC-a

- **Prihvaćanje nemaskirajućeg prekida kod FRISC-a:**
 - Briše se IIF (zabranjivanje svih dalnjih prekida)
 - Sprema se PC na stog
 - Skok u prekidni potprogram na adresi C_{16} ($C_{16} \rightarrow PC$)
- Komentari:
 - Dojava VJ da je prihvaćen zahtjev za prekid obavlja se programski
 - Prekidni potprogram mora biti uvijek na memorijskoj adresi 12_{10} (tj. $0C_{16}$)

Prekidni sustav FRISC-a

- **Prihvaćanje maskirajućeg prekida kod FRISC-a:**
 - Briše se GIE (zabranjivanje dalnjih maskirajućih prekida)
 - Sprema se PC na stog
 - Dohvat **adrese** prekidnog potprograma (tzv. prekidnog vektora) s memorijске lokacije na adresi 8 i skok u prekidni potprogram, tj. $(8) \rightarrow \text{PC}$
- Komentari:
 - dojava o prihvaćanju prekida programski u prekidnom potprogramu
 - Prekidni vektor omogućuje postavljanje prekidnog potprograma na bilo koju adresu u memoriji, ali zahtijeva jedan ciklus čitanja više za dohvat prekidnog vektora. Prekidni vektor mora biti zapisan na adresi 8

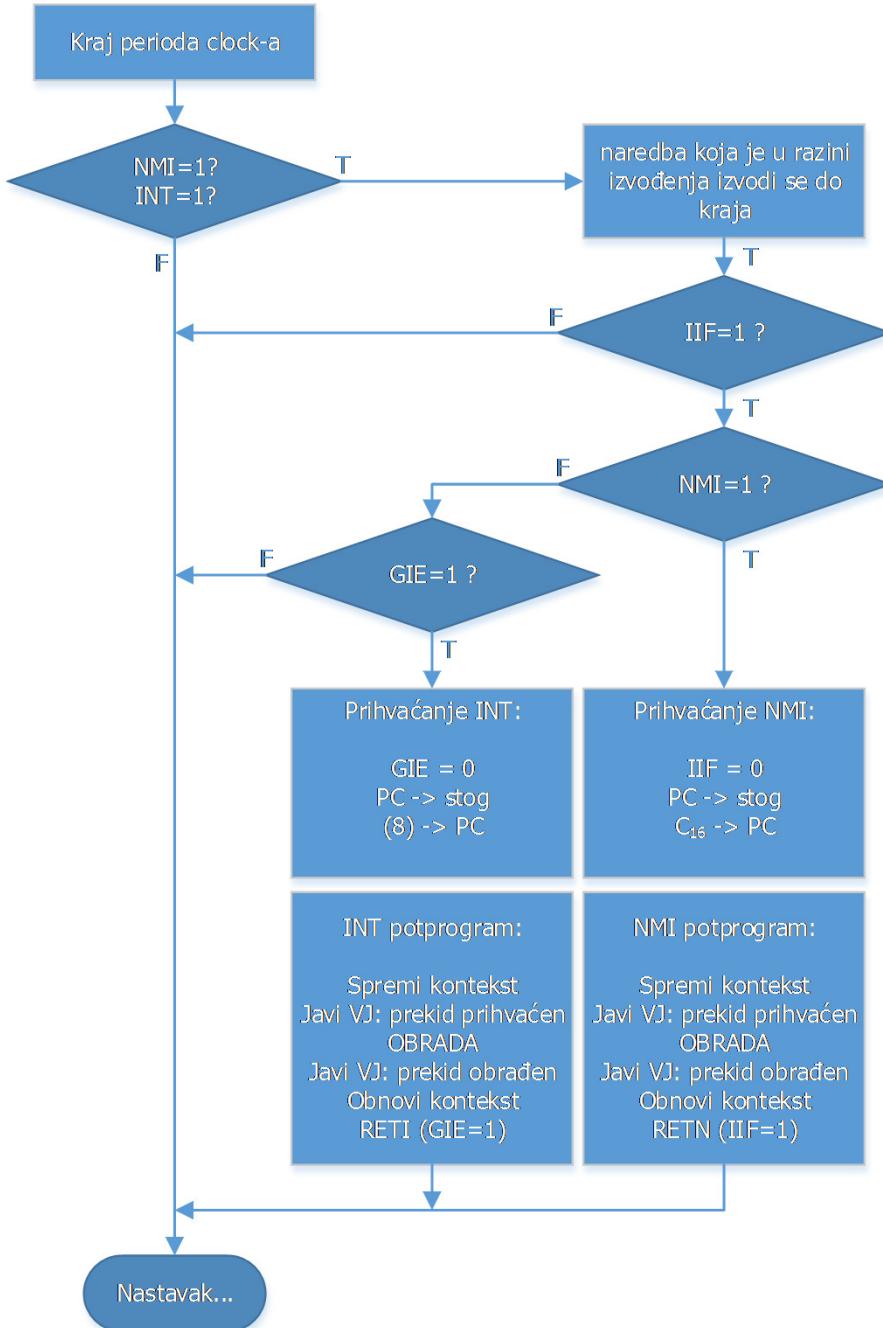
Prekidni potprogram FRISC-a

1. Sprema se kontekst (registre koje potprogram mijenja)
2. Otkriva se uzročnik prekida (ako ih ima više), tj. otkriva se koja VJ je izazvala prekid
3. Dojavljuje se VJ da je njen prekid prihvaćen
4. Poslužuje se VJ
5. Obnavljanje konteksta
6. Dojava VJ da je njezin prekid obrađen
7. Povratak iz potprograma s dozvoljavanjem prekida
 - naredbom RETI za maskirajući
 - naredbom RETN za nemaskirajući

* Koraci 5. i 6. mogu se izvesti i u obratnom redoslijedu

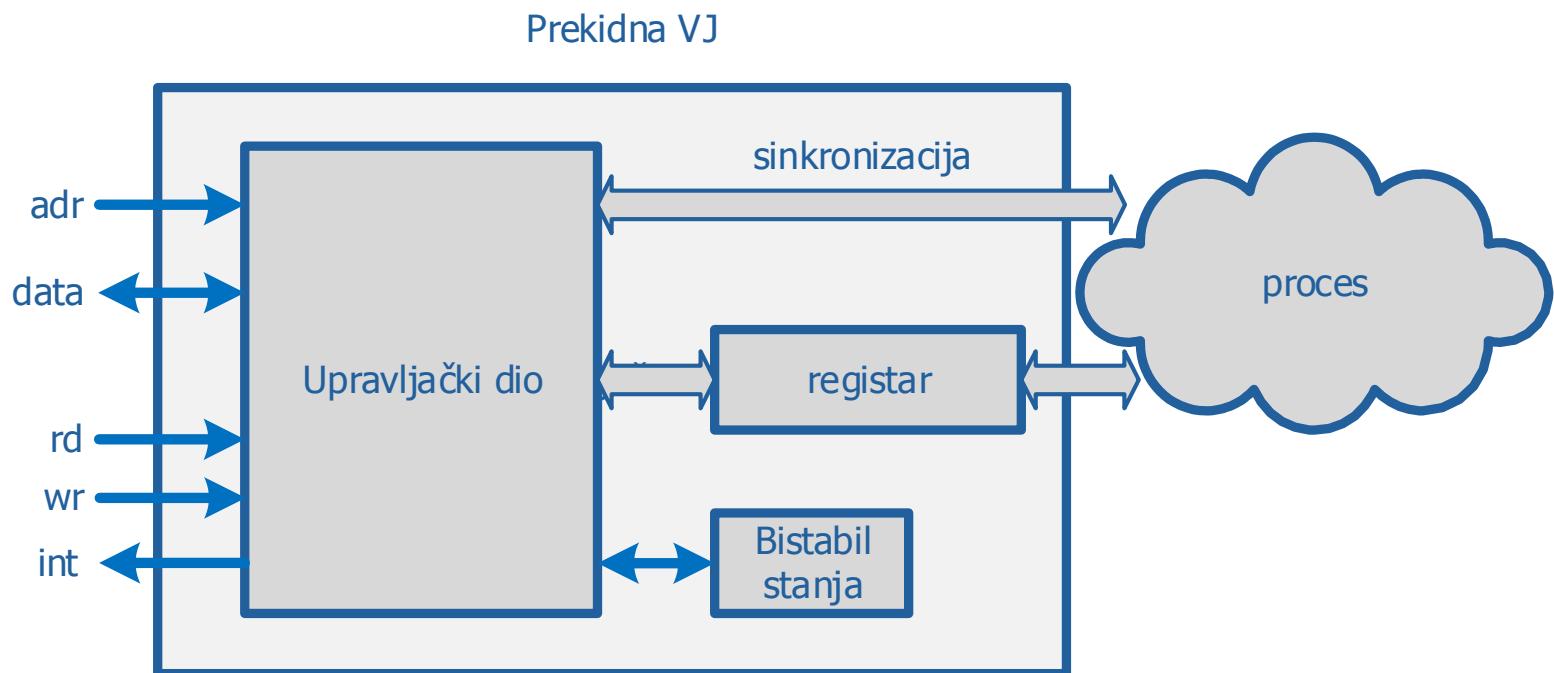
Prekidni sustav FRISC-a

- Naredbe za povratak iz potprograma rade kao i običan RET, ali dodatno dozvoljavaju prekid koji je FRISC bio automatski zabranio kod prihvatanja prekida (drugim riječima, obnavljaju stanje prekidne zastavice GIE odnosno IIF):
 - Za **maskirajući prekid** je prije prihvatanja prekida vrijedilo GIE=1, a u trenutku prihvatanja maskirajućeg prekida se GIE automatski obriše
 - RETI (**R****E****T**urn from **m**askable **I**nterrupt) obnavlja stanje GIE=1
 - Za **nemaskirajući prekid** je prije prihvatanja prekida vrijedilo IIF=1, a u trenutku prihvatanja nemaskirajućeg prekida se IIF automatski obriše
 - RETN (**R****E****T**urn from **N**on**m**askable **i**nterrupt) obnavlja stanje IIF=1



Osnovna građa prekidne UI jedinice

- Najjednostavnija građa opće prekidne UI jedinice može se prikazati sljedećom blok shemom (sve je slično kao kod uvjetne UI jedinice)



Osnovna građa prekidne UI jedinice

- Prekidna VJ može biti spremna ili nespremna kao i uvjetna VJ:
 - Uvjetna VJ je "pasivna": procesor treba ispitivati spremnost što znači da je cijeli tijek prijenosa pod upravljanjem programa
 - Prekidna VJ je "aktivna": kad postane spremna, sama od procesora zahtjeva posluživanje, postavljajući zahtjev za prekid.
- Prekidnoj VJ se programski može zabraniti ili dozvoliti da postavlja prekid kad postane spremna (to je **različito** od dozvoljavanja i zabranjivanja prihvaćanja prekida u procesoru)
 - Obično VJ kojoj se zabrani postavljanje prekida i dalje normalno radi te je se može posluživati kao uvjetnu VJ
- Zabranjivanje postavljanja prekida ima učinak zaustavljanja VJ u prekidnom načinu rada

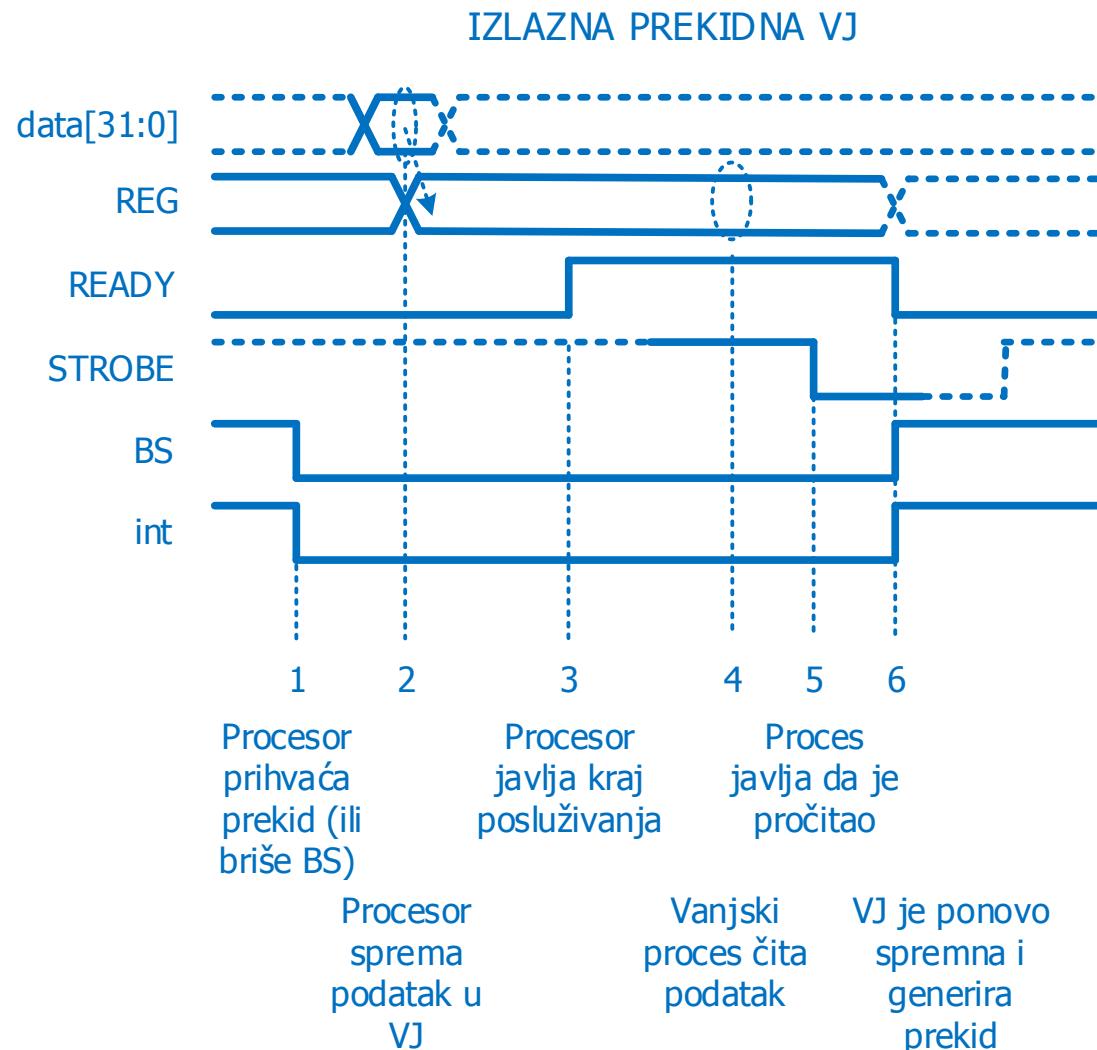
Osnovna građa prekidne UI jedinice

- Nakon što bistabil stanja postane 1 (kad VJ postane spremna), automatski se postavlja zahtjev za prekid (uz pretpostavku da je dozvoljeno postavljanje prekida - u suprotnom bistabil stanja ne utječe na stanje prekidnog priključka)
- Brisanje bistabila stanja:
 - **uklanja zahtjev za prekid** pa ima ulogu dojave o prihvaćanju zahtjeva za prekid (radi se na početku prekidnog potprograma)
 - **ne omogućava nastavak komunikacije s vanjskim procesom** (za razliku od brisanja bistabila stanja kod uvjetne jedinice)
- Nastavak komunikacije s vanjskim procesom moguć je tek nakon što se VJ dojavi da je njen prekid obrađen
 - to znači da tek nakon toga VJ može ponovno postati spremna i postaviti novi prekid (radi se na kraju prekidnog potprograma)

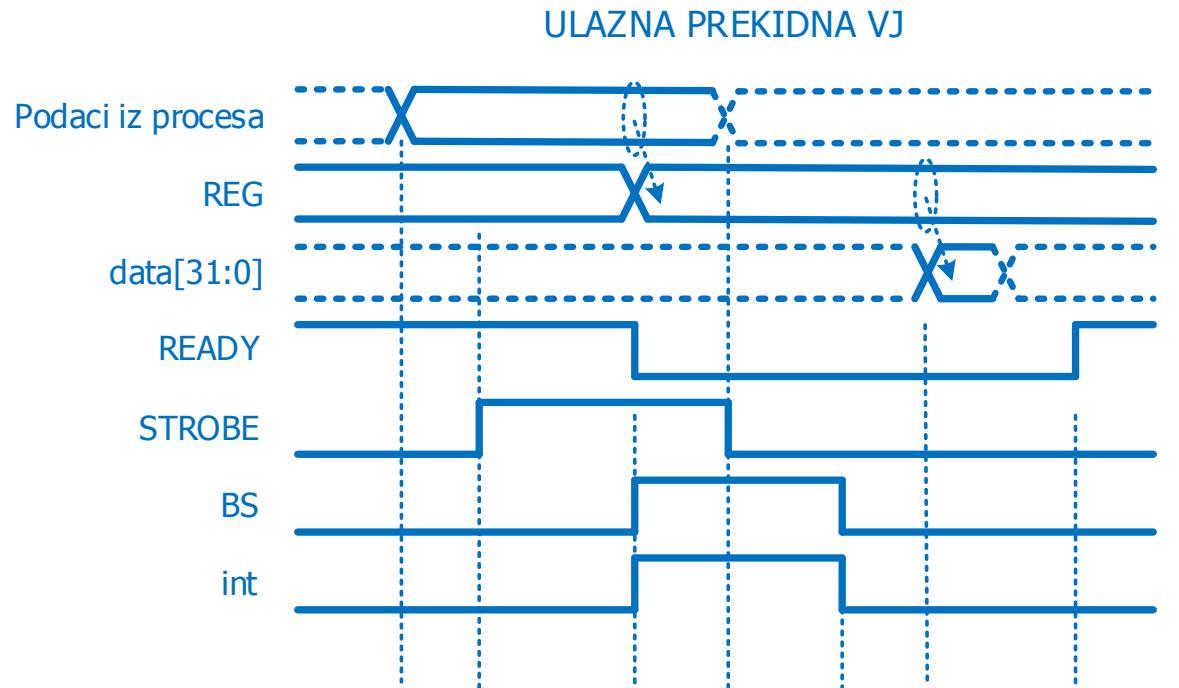
Osnovna građa prekidne UI jedinice

- Prekidna VJ zauzimat će **četiri uzastopne 32-bitne lokacije** (*
kod konkretnih VJ raspored može biti drugačiji):
 - Na **prvoj lokaciji** se čita ili piše podatak
 - Na **drugoj lokaciji** se pristupa bistabilu stanja:
 - Čita se trenutačni sadržaj bistabila (ispitivanje stanja)
 - Briše se bistabil operacijom upisa bilo kojeg podatka (poslani podatak se zanemaruje)
 - Na **trećoj lokaciji** se upisom bilo kojeg podatka (poslani podatak se zanemaruje) dojavljuje da je prekid obrađen
 - Pomoću **četvrte lokacije** upravlja se postavljanjem prekida:
 - Upis 0 zabranjuje, a upis 1 dozvoljava postavljanje zahtjeva za prekid
 - Čitanje vraća trenutačnu o(ne)mogućenost postavljanja prekida
 - Inicijalno ćemo pretpostaviti da je dozvoljeno postavljanje zahtjeva za prekid

Vrem. dijagram za izlaznu prekidnu VJ



Vrem. dijagram za ulaznu prekidnu VJ



1	2	3	4	5	6	7
Vanjski proces postavlja podatak		VJ pamti podatak i generira prekid		Procesor prihvata prekid (ili briše BS)		Procesor javlja kraj posluživanja
Proces javlja VJ da je podatak spreman		Proces više ne mora držati podatak		Procesor čita podatak		

Prekidni prijenos - Primjeri

FRISC treba primiti 100_{16} podataka od prekidne VJ spojene na NMI. Adresa VJ je FFFF3000. Primljene podatke treba spremati u memorijski blok podataka na adresi 1000, samo ako su pozitivni.

Nakon primitka svih podataka treba zaustaviti rad VJ i rad programa.

```

VJ_DATA EQU 0FFFF3000
VJ_STAT EQU 0FFFF3004
VJ_IEND EQU 0FFFF3008
VJ_STOP EQU 0FFFF300C

        ORG 0
        MOVE 10000, R7 ; početak izvođenja
        JP    GLAVNI      ; skoči na početak glavnog programa

        ORG 0C          ; adresa p.p. za NMI
        PUSH R0          ; spremi kontekst
        PUSH R1
        MOVE SR,R0
        PUSH R0
        STORE R0,(VJ_STAT) ; briši BS (dojavi prihvaćanje prekida)
        LOAD R0,(ADR_PODAT)
        LOAD R1,(VJ_DATA) ; primi podatak i...
        OR    R1,R1,R1
        JR_M NEMOJ       ;...ako je pozitivan
SPREMI STORE R1,(R0)      ;...spremi ga u blok...
        ADD   R0,4,R0
        STORE R0,(ADR_PODAT)

```

```
NEMOJ    LOAD  R0,(BROJAC) ; provjera brojača...
          SUB   R0,1,R0      ;... primljenih podataka
          STORE R0,(BROJAC)
          JR_NZ VAN           ;ima jos podataka->VAN

STOP     STORE R0,(VJ_STOP) ;zaustavi VJ i procesor
          MOVE  1,R0
          STORE R0,(PROC_HALT)

VAN      STORE R0,(VJ_IEND) ; dojavi kraj posluživanja

          POP   R0            ; obnova konteksta
          MOVE R0,SR
          POP   R1
          POP   R0
          RETN             ; povratak i IIF=1

PROC_HALT DW  0           ; oznaka za glavni program
                           ; 0 = nastavi rad, 1 = zaustavi procesor

BROJAC   DW  100 ; brojač prenesenih podataka
ADR_PODAT DW  1000 ; adresa za spremanje u blok
```

GLAVNI

```
PETLJA    LOAD R0,(PROC_HALT) ; "koristan posao"  
          OR    R0,R0,R0  
          JR_Z PETLJA           ; nastavi ako je 0  
  
HALT
```

Prekidni prijenos - Primjeri

FRISC treba poslati 100_{16} 16-bitnih podataka iz bloka memorije na adresi 1000 na prekidnu VJ na adresi FFFF0000. VJ je spojena na INT. Nakon prijenosa cijelog bloka treba zaustaviti rad prekidne VJ, a glavni program treba nastaviti s radom.

```
SEND EQU 0FFFF0000
IACK EQU 0FFFF0004
IEND EQU 0FFFF0008
STOP EQU 0FFFF000C
```

```
ORG 0
MOVE 10000, R7 ; početak izvođenja
JP GLAVNI ; preskakanje vektora

; PREKIDNI VEKTOR na adresi 8
ORG 8
DW 200 ; adresa p.p.

; GLAVNI PROGRAM
GLAVNI MOVE 1000, R0 ; adresa podataka
STORE R0, (PODATAK)
MOVE 100, R0 ; brojač podataka
STORE R0, (BROJAC)
; DOZVOLI PREKID NA INT0
MOVE %B 10000, SR

PETLJA JP PETLJA ; "koristan posao"
```

```
; PREKIDNI POTPROGRAM NA ADRESI 200

ORG    200
PUSH   R0          ; spremanje konteksta
PUSH   R1
PUSH   R2
MOVE   SR,R0
PUSH   R0

STORE  R0, (IACK)   ; prihvaćen prekid

LOAD   R0, (BROJAC) ; dohvati varijabli
LOAD   R1, (PODATAK)

LOADH  R2, (R1)      ; čitanje iz memorije
STORE  R2, (SEND)    ; i slanje na VJ

ADD    R1, 2, R1      ; pomicanje pokazivača
STORE  R1, (PODATAK)
SUB    R0, 1, R0      ; smanjenje brojača
STORE  R0, (BROJAC)
JR_NZ  IMA_JOS
```

ZADNJI MOVE R0 ; ako je zadnji podatak
STORE R0, (STOP) ; zaustavi VJ

IMA_JOS POP R0 ; obnavljanje konteksta
MOVE R0, SR
POP R2
POP R1
POP R0

STORE R0, (IEND) ; kraj posluživanja
RETI

BROJAC DW 0 ; varijable za p.p.
PODATAK DW 0

; Podaci iz memorije koji se šalju na VJ
ORG 1000
DH 12, 4, 456A, 1, 0AB, 2, 885, ...

Prekidni prijenos - Primjeri

- Komentari:
- **U kontekst prekidnog potprograma ulazi i SR***
(osim u rijetkim slučajevima kad se SR ne mijenja u prekidnom potprogramu)
- **Za obične potprograme SR ne ulazi u kontekst**, jer pozivatelj može prepostaviti da će potprogram promijeniti SR i zato pozivatelj nikada nema u SR-u neko stanje koje će mu trebati nakon povratka iz potprograma (Ako pozivatelju običnog potprograma treba stanje iz SR-a, onda ga pozivatelj treba spremiti).

* Neki procesori automatski spremaju statusni registar prilikom prihvatanja prekida

Posluživanje više prekidnih VJ

- Do sada smo vidjeli samo najjednostavniji slučaj kad je na procesor spojena jedna prekidna VJ
- Kad postoji više prekidnih VJ, one se mogu posluživati sa ili bez gniježđenja (engl. nesting):
 - **bez gniježđenja** prekidnih potprograma:
 - **dok se poslužuje jedna VJ, drugi prekidi se ne prihvataju**
 - jednostavniji slučaj
 - **sa gniježđenjem** prekidnih potprograma:
 - **dok se poslužuje jedna VJ, može se prihvatiti drugi prekid (većeg prioriteta)**
 - komplikiraniji slučaj

Posluživanje više prekidnih VJ

- Svim vanjskim jedinicama treba **dodijeliti različite prioritete**, što se može napraviti:
 - programski (**FRISC za maskirajuće**)
 - sklopovski
 - sam procesor ima više prekidnih priključaka s različitim prioritetima (**FRISC: NMI je prioritetniji od INT**)
 - prioritetni lanac vanjskih jedinica (daisy-chain)
 - jedinica za kontrolu prioriteta (programmable interrupt controller ili priority interrupt controller)
- **Prioriteti imaju dvojaku ulogu:**
 - kod istovremenih prekida određuje se kojoj VJ se prihvata prekid (služi i za grijevanje prekida i kad nema grijevanja)
 - za vrijeme obrade jednog prekida određuje hoće li se prihvatiti novi prekid (služi samo za grijevanje prekida)

Posluživanje više prekidnih VJ

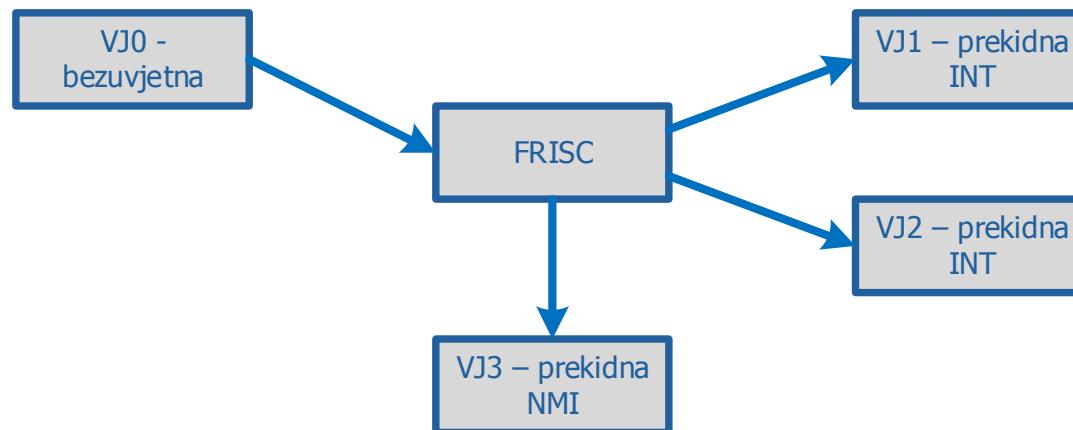
- Bez obzira kako se poslužuju, **uvijek treba odrediti uzročnike prekida** o čemu ovisi koju VJ ćemo poslužiti
- Ovisno o prekidnom sustavu procesora, moguća su različita rješenja:
 - VJ sklopovalski utječe na odabir adrese prekidnog potprograma čime se automatski određuje uzročnik prekida
 - Adresa prekidnog potprograma bira se na temelju ulaznog prekidnog priključka čiji prekid je prihvaćen
(FRISC: NMI ima različitu adresu p.p. od INT)
 - Programski se određuje koja jedinica je izazvala prekid
(FRISC: ispitivanjem BS)

Posluživanje više prekidnih VJ

- Ispitivanje uzročnika prekida kod prekida:
 - **ispituje se spremnost VJ** (bistabil stanja)
 - ovo **ne treba miješati s ispitivanjem uvjetnih VJ**, jer se ovdje samo jednom ispita spremnost, tj. nema čekanja da VJ postane spremna niti se obavlja prozivanje
 - **redoslijed ispitivanja definira prioritete VJ:**
 - jedinice koje se prije ispituju imaju veći prioritet

Prekidni prijenos - Primjer

Na FRISC su spojene vj0, vj1, vj2 i vj3 na adresama FFFF0000, FFFF1000, FFFF2000 i FFFF3000. Vj0 je ulazna bezuvjetna, vj1 i vj2 izlazne prekidne jedinice spojene na INT (ne mogu se međusobno prekidati), a vj3 je izlazna prekidna jedinica spojena na NMI. Procesor šalje podatke s vj0 na vj1 i vj2 i broji koliko je podataka poslao. Kad vj3 zatraži prekid, treba joj poslati broj do tada prenesenih podataka.



PRIMI0 EQU 0FFFF0000

SALJI1 EQU 0FFFF1000

BS1 EQU 0FFFF1004

POSLUZEN1 EQU 0FFFF1008

SALJI2 EQU 0FFFF2000

BS2 EQU 0FFFF2004

POSLUZEN2 EQU 0FFFF2008

SALJI3 EQU 0FFFF3000

BS3 EQU 0FFFF3004

POSLUZEN3 EQU 0FFFF3008

ORG 0

MOVE 10000, SP

JP GLAVNI

; prekidni vektor za maskirajući prek.

ORG 8

DW 100

```
; prekidni potprogram za
; nemaskirajući prekid na adresi 0C
ORG    0C

PUSH    R0
STORE  R0, (BS3) ; prihvaćen prekid
LOAD    R0, (BROJAC) ; broj poslanih
STORE  R0, (SALJI3) ; pošalji na vj3
POP    R0
STORE  R0, (POSLUZEN3) ; dojava kraja

RETN

; Glavni program

GLAVNI ; dozvoli prekid na INT0
MOVE   %B 10000, SR
; "koristan posao"
PETLJA JR    PETLJA

; brojač poslanih podataka
BROJAC DW    0
```

```
ORG 100

PUSH R0          ; spremanje
MOVE SR, R0      ; konteksta
PUSH R0

ISPITAJ LOAD R0, (BS1) ; otkrivanje
AND   R0, 1, R0  ; uzročnika
JR_NZ P_VJ1      ; prekida
JR    P_VJ2

VAN   POP  R0
MOVE R0, SR      ; obnova
POP   R0          ; konteksta

RETI
```

P_VJ1 ; dio za posluživanje vj1

STORE R0, (BS1) ; prihvaćen prekid

LOAD R0, (PRIMI0) ; čitaj bezuvjetnu vj0

STORE R0, (SALJI1) ; šalji na vj1

LOAD R0, (BROJAC) ; povećaj

ADD R0, 1, R0 ; brojač poslanih

STORE R0, (BROJAC) ; podataka

STORE R0, (POSLUZEN1) ; kraj posluživanja

JR VAN ; povratak

P_VJ2 ; dio za posluživanje vj2

; analogno kao i P_VJ1

...

Prekidni prijenos - Primjer

- Komentar:
- Vj1 i vj2 se ne mogu međusobno prekidati, ali u prekidnom potprogramu se prvo ispituje vj1 pa će ona biti prioritetnija od vj2 u smislu da će kod istovremenog prekida prva biti poslužena vj1.
- Kod istovremenog prekida dešava se sljedeće:
- Prihvaća se prekid čime se automatski zabrani prihvaćanje dalnjih prekida. U p.p.-u se ustanovi da je vj1 izazvala prekid te se obrađuje njen prekid. Cijelo to vrijeme vj2 zahtjeva prekid, ali je prihvaćanje prekida u procesoru zabranjeno i prekid od vj2 se ne prihvata: kažemo da je prekid "na čekanju" (tzv. pending interrupt).
- Nakon povratka iz prekidnog potprograma od vj1, doći će do omogućavanja prekida (naredba RETI). Tada će prekid od vj2 konačno biti prihvacen te će se skočiti u prekidni potprogram koji će tada poslužiti vj2.

FRISC VJ

Sklop FRISC-CT

Sklop FRISC-CT

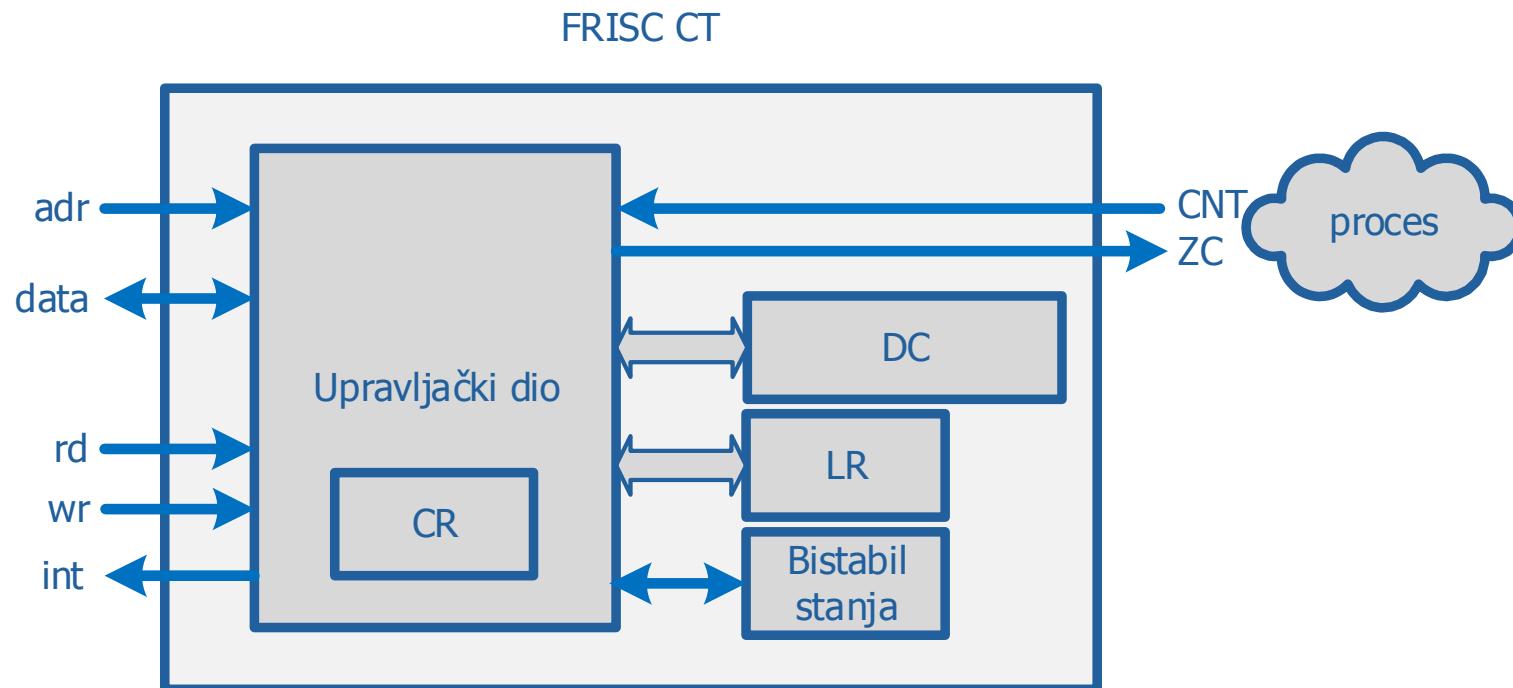
- Sklop FRISC-CT, ili kraće CT (kratica od Counter Timer)
služi za brojenje impulsa i mjerjenje vremena
 - Za razliku od dosadašnjih VJ, CT ne prenosi podatke
 - Slični sklopovi postoje i za komercijalne procesore
- CT **oslobađa procesor** od nepotrebnih čekanja ili čestih posluživanja prekida

Sklop FRISC-CT

- Za razliku od do sada pokazanih vanjskih jedinica koje su bile općenite i imale su nepromjenjivu zadaću, CT je sklop koji se **može "programirati" (točnije: konfigurirati)**
- Na taj način se ponašanje i funkcija vanjske jedinice prilagođavaju konkretnoj zadaći koju treba napraviti
- Programiranje sklopa se ostvaruje **slanjem posebnih upravljačkih riječi na određene adrese** na kojima se sklop nalazi

Blok-shema CT-a

- Sučelje za spajanje s FRISC-om, isto je kao za prekidne jedinice
- DC broji prema nuli i ima 16 bita



Adrese CT-a

- CT zauzima četiri uzastopne 32-bitne lokacije:

Adresa	Pisanje	Čitanje
PA	Upis CR	Čitanje CR
PA + 4	upis u LR (i DC)	Trenutna vrijednost DC
PA + 8	dojava prihvaćanja prekida (tj. brisanje BS)	čitanje BS
PA+ 12₁₀	dojava o kraju posluživanja prekida	-

Upravljačka riječ - CR

bitovi 31 – 3	bit 2	bit 1	bit 0
-	VRSTA INT	INT	STOP / START
	0 – maskirajući 1 – nemaskirajući	0 – ne postavlja prekid 1 – postavlja prekid	0 – brojilo je zaustavljeno 1 – brojilo broji

Ostale adrese

- **CT registar punjenja brojača LR (engl. Load Register)**
- Na adresi PA+4 pristupa se 16-bitnom LR registru. Pri pisanju u ovaj registar zapisati će se nižih 16 bitova podatka dok će viši dio podatka biti zanemaren. Pri čitanju s ove adrese pročitati će se trenutna vrijednost brojila (DC).
- **CT bistabil stanja BS**
- Na adresi PA+8 pristupa se CT jednobitnom bistabilu stanja BS. Kada procesor inicira naredbu pisanja na ovu adresu, podatak koji procesor šalje se zanemaruje a CT briše BS (BS=0). Pri prekidnom prijenosu, ovime procesor potvrđuje prihvatanje zahtjeva za prekid.
- Čitanjem ove adrese, na najnižem bitu podatka čita se trenutna vrijednost BS.
- **CT dojava kraja posluživanja prekida**
- Preko četvrte adrese koju zauzima CT (PA+12₁₀) procesor pisanjem bilo kojeg podatka (podatak koji procesor šalje se zanemaruje) procesor javlja CT šklopu da je posluživanje obrade njegovog prekida završeno. Citanje s ove adrese nije definirano.

Kad DC=0

U trenutku kad brojilo dođe do nule CT izvodi sljedeće:

- CT postavlja BS=1 i generira prekid (ako je konfiguracijom omogućeno)
- Na izlazu ZC (engl. Zero Count) CT generira jedan pozitivan impuls
- CT automatski kopira vrijednost iz LR u DC, čime brojilo može ponovo početi brojati nove impulse

CT Primjeri

Na CT-ov priključak CNT spojen je izlaz iz stroja koji za svaki proizvedeni vijak generira impuls. Računalo mora u lokaciji BR_PAK prebrajati proizvedene pakete od po 200 vijaka. Treba riješiti zadatak CT-om (na adresi FFFF0000) tako da:

- a) CT radi u uvjetnom načinu,
- b) CT radi u prekidnom načinu i spojen je na INT.

Uvjetni način rada:

```
CTCR    EQU    0FFFF0000
CTLR    EQU    0FFFF0004
CTSTAT  EQU    0FFFF0008
CTEND   EQU    0FFFF000C
```

```
ORG    0
; GLAVNI PROGRAM

; INICIJALIZACIJA CT-a
GLAVNI MOVE    %D 200, R0
          STORE   R0, (CTLR)

          MOVE    1, R0 ; brojilo broji
          STORE   R0, (CTCR)
; ISPITIVANJE CT-a
PETLJA LOAD    R0, (CTSTAT) ;čekanje spremnosti
          AND     R0, 1, R0      ;tj. čekanje da se
          JR_Z   PETLJA        ;proizvede paket
```

```
STORE R0, (CTSTAT) ;brisanje spremnosti
LOAD  R0, (BR_PAK)
ADD   R0, 1, R0      ;povećaj brojač
STORE R0, (BR_PAK)

STORE R0, (CTEND) ;kraj posluživanja
JR    PETLJA

BR_PAK DW 0
```

Prekidni način rada:

CTCR	EQU	0FFFF0000
CTCR	EQU	0FFFF0004
CTIACK	EQU	0FFFF0008
CTIEND	EQU	0FFFF000C

```
ORG    0
MOVE   10000, R7
JP     GLAVNI
; PREKIDNI VEKTOR
ORG    8
DW     1000
GLAVNI ; GLAVNI PROGRAM
; INICIJALIZACIJA CT-a
MOVE   %D 200, R0          ; postavljanje brojača
STORE  R0, (CTLR)

; KONTR. RIJEĆ
MOVE   %B 11, R0          ; INT + brojilo broji
STORE  R0, (CTCR)
MOVE   %B 10000, SR ; OMOGUĆI PREKID
```

```
; "KORISTAN POSAO"

PETLJA    JR      PETLJA
ORG      1000  ; PREKIDNI POTPROGRAM

PUSH     R0
MOVE     SR, R0      spremanje konteksta
PUSH     R0
STORE   R0, (CTIACK) obriši spremnost

LOAD     R0, (BR_PAK) povećaj
ADD      R0, 1, R0    brojač
STORE   R0, (BR_PAK) paketa

POP     R0
MOVE     R0, SR      obnova konteksta
POP     R0
STORE   R0, (CTIEND) dojava kraja
RETI

BR_PAK  DW      0      ; BROJAČ PAKETA
```

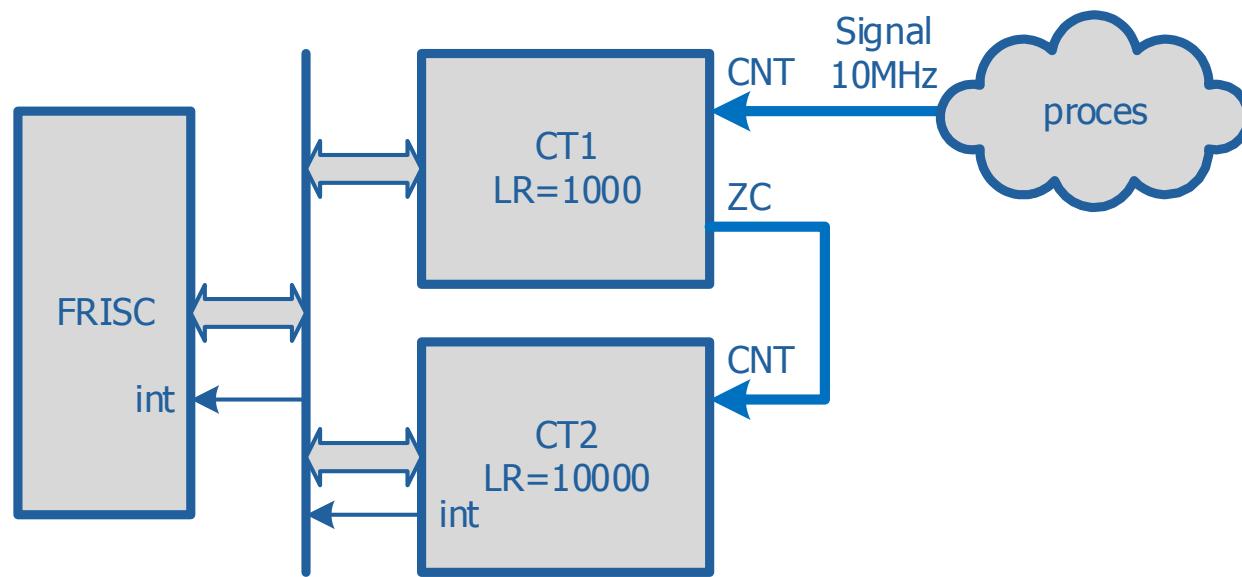
Napomena:

Iako je rješenje s prekidom, dulje, izvodi se puno efikasnije, jer uvjetno posluživanje troši gotovo svo vrijeme samo na ispitivanje spremnosti CT-a.

CT Primjeri

FRISC treba svake sekunde izvesti potprogram POTP. Vremensko kašnjenje treba ostvariti pomoću sklopova CT, a ne programskom petljom za kašnjenje. Pretpostavka je da program POTP već postoji i da njegovo izvođenje sigurno traje kraće od jedne sekunde. Signal koji se dovodi na CT ima frekvenciju 10 MHz.

Prijedlog rješenja



CTCR1 EQU **0FFFF1000**

CTLR1 EQU **0FFFF1004**

CTCR2 EQU **0FFFF2000**

CTLR2 EQU **0FFFF2004**

CTIACK2 EQU **0FFFF2008**

CTIEND2 EQU **0FFFF200C**

ORG 0

MOVE 10000, R7

JP GLAVNI

ORG 8

DW 1000 ; prekidni vektor

```

GLAVNI ; inicijaliziraj CT1
        MOVE  %D 1000, R0      ; LR1=1000
        STORE R0, (CTLR1)
        MOVE  1, R0      ; CR1: bez INT
        STORE R0, (CTCR1)

; inicijaliziraj CT2
        MOVE  %D 10000, R0     ; LR2=10000
        STORE R0, (CTLR2)
        MOVE  %B 11, R0      ; CR2: postavlja INT
        STORE R0, (CTCR2)

        MOVE  %B 10000, SR  ; omogući prekid INT0

; prekidni potprogram - izvodi se svake sek.
; kontekst se ne sprema jer se ne mijenjaju registri
        ORG   1000
        STORE R0, (CTIACK2) ; potvrda prekida
        CALL   POTP  ; poziv zadanog potprograma
        STORE R0, (CTIEND2) ; potvrda kraja
        RETI

```

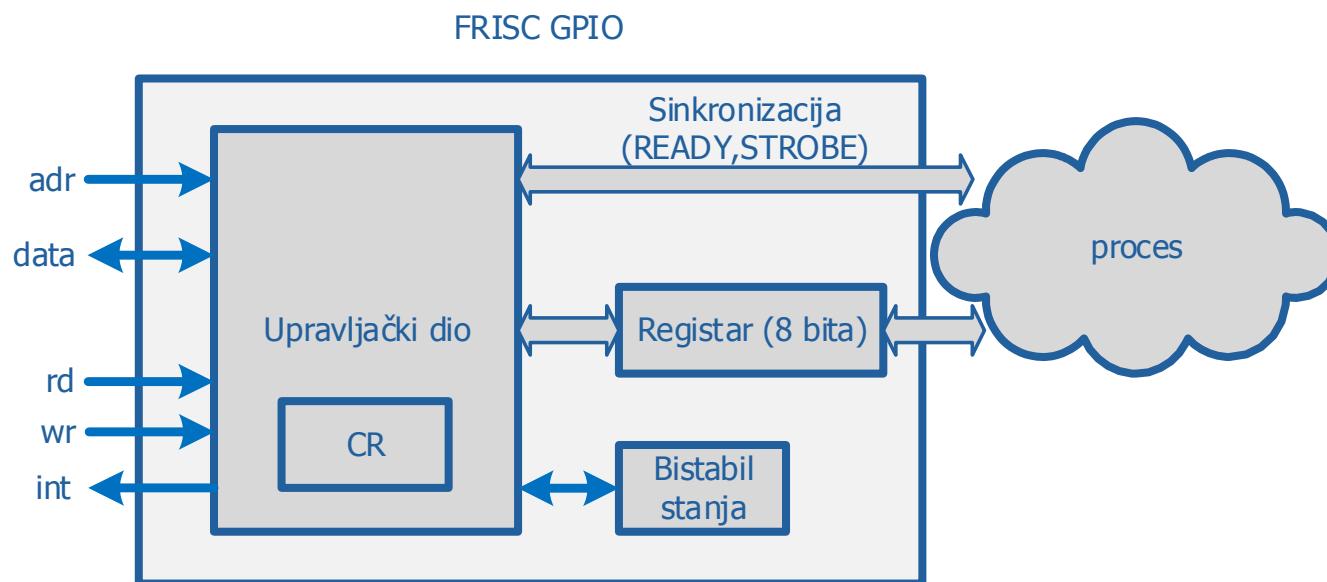
Sklop FRISC-PIO

Sklop FRISC-GPIO

- Sklop FRISC-GPIO, ili kraće GPIO (kratica od General Purpose Input-Output) je sklop koji **služi za paralelni prijenos podataka** (8 bitnih)
 - Posrednik između procesora i procesa (djelomično sličan općim VJ kojima smo do sada prenosili podatke)
 - Slični sklopovi postoje i za komercijalne procesore
- GPIO može raditi u 4 načina rada od kojih dva omogućavaju sinkroni prijenos podataka a dva asinkroni:
 - ulazni (ulazni, sinkroni)
 - ispitivanje bitova (ulazni, asinkroni)
 - izlazni (izlazni, sinkroni)
 - postavljanje bitova (izlazni, asinkroni)

Blok-shema GPIO

- Sučelje za spajanje s FRISC-om, isto je kao za prekidne jedinice



Adresa	Pisanje	Čitanje
PA	upravljačka riječ CR	upravljačka riječ CR
PA + 4	upis podatka u DR	čitanje DR
PA + 8	dojava prihvaćanja prekida (tj. brisanje BS)	čitanje BS
PA + 12₁₀	dojava o kraju posluživanja prekida	-

31 – 24	23 – 16	15 – 8	7 – 5	4	3	2	1 – 0
-	ACTIVE	MASK	-	AND/OR	VRSTA INT	INT	MODE
0 – aktivna je 0 1 – aktivna je 1			0 – OR 1 – AND		0 – maskirajući 1 – nemaskirajući	0 – ne postavlja prekid 1 – postavlja prekid	00 – izlazni način 01 – ulazni način 10 – postavljanje bitova 11 – ispitivanje bitova

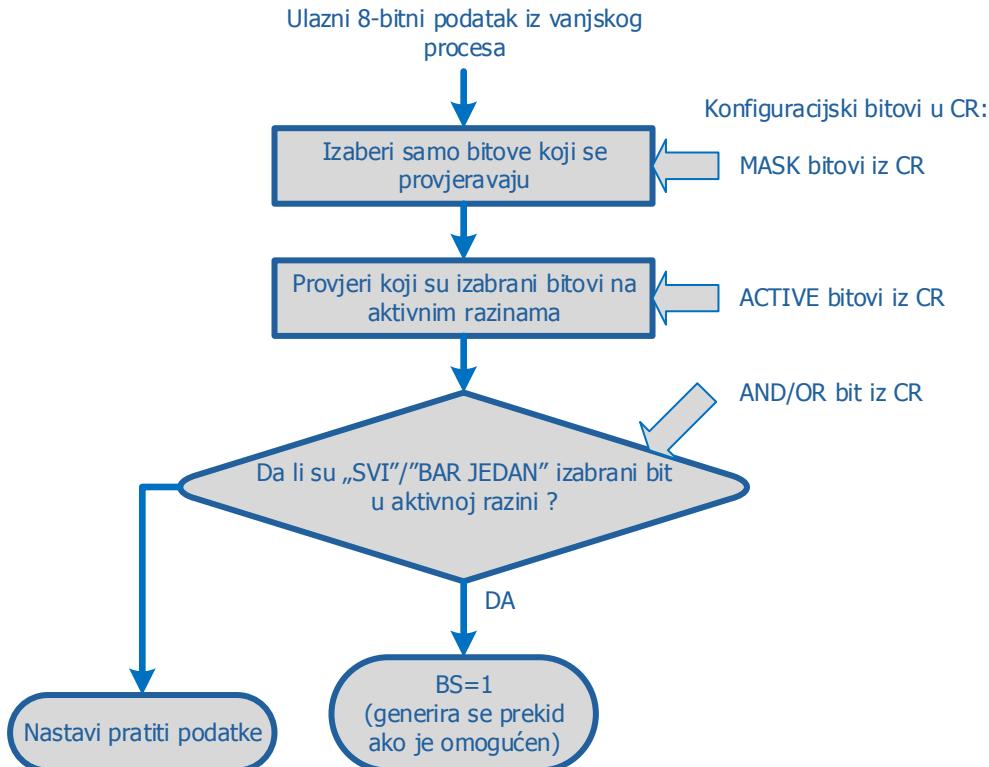
Načini rada

- Izlazni
 - prijenos je sinkroniziran upravljačkim signalima READY i STROBE
 - Istovjetno već ranije opisanoj izlaznoj prekidnoj VJ
- Ulazni
 - prijenos je sinkroniziran upravljačkim signalima READY i STROBE
 - Istovjetno već ranije opisanoj ulaznoj prekidnoj VJ

Načini rada

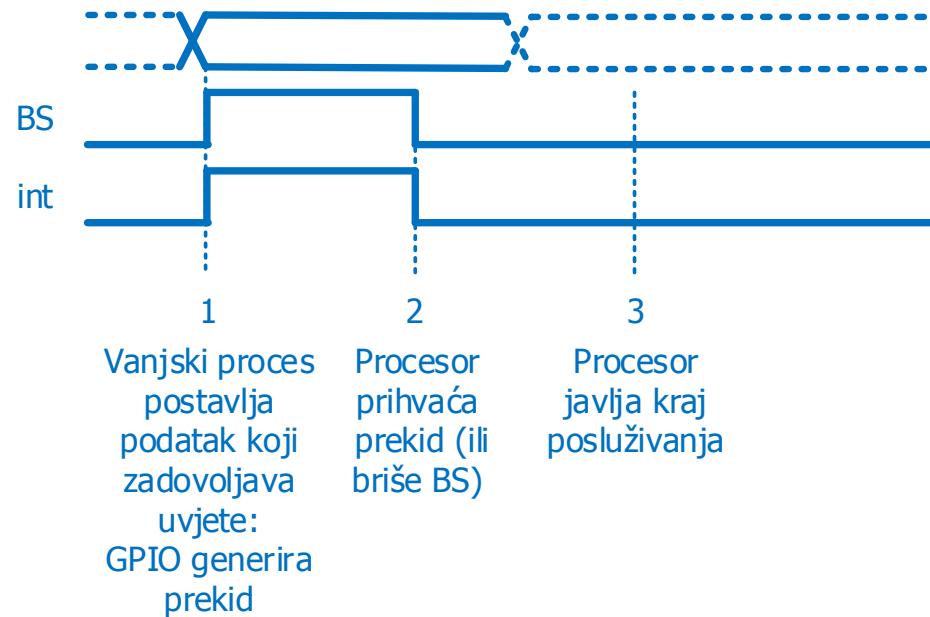
- Postavljanje bitova
 - ne koriste se sinkronizacijski signali, bistabil stanja ni prekid
 - GPIO u ovom načinu radi kao bezuvjetna izlazna VJ
- Ispitivanje bitova
 - ne koriste se sinkronizacijski signali
 - GPIO u ovom načinu radi kao bezuvjetna ulazna VJ
 - Dodatna funkcija: automatska provjera stanja na ulazima i postavljanje BS i generiranje prekida

Ispitivanje bitova



31 – 24	23 – 16	15 – 8	7 – 5	4	3	2	1 – 0
-	ACTIVE	MASK	-	AND/OR	VRSTA INT	INT	MODE
0 – aktivna je 0 1 – aktivna je 1				0 – OR 1 – AND	0 – maskirajući 1 – nemaskirajući	0 – ne postavlja prekid 1 – postavlja prekid	00 – izlazni način 01 – ulazni način 10 – postavljanje bitova 11 – ispitivanje bitova

Ispitivanje bitova



GPIO Primjeri

Na pisač koji je spojen na GPIO treba poslati 80_{16} znakova veličine jedan oktet, smještenih u memoriji od lokacije ZNAKOVI. GPIO radi u prekidnom načinu, a adresa mu je FFFF0000. Pretpostavka je da nema drugih izvora prekida, a GPIO je spojen na int[0]. Pretpostavka je da pisač ima linije za rukovanje kompatibilne sa READY i STROBE.

PIO će raditi u izlaznom načinu., jer će s pisačem biti potrebna sinkronizacija.

PIOC EQU 0FFFF0000

PIOD EQU 0FFFF0004

PIOIACK EQU 0FFFF0008

PIOIEND EQU 0FFFF000C

ORG 0

MOVE 10000, R7

JP GLAVNI

ORG 8 ; prekidni vektor

DW 500

; glavni program

; inicijalizacija sklopa PIO

GLAVNI MOVE %B 0100, R0 ; 0(maskirajući)1(prekid)00(izlazni)

STORE R0, (PIOC) ; pošalji u OCR

MOVE %B 010000, SR ; dozvoli INT0

PETLJA JR PETLJA ; "koristan posao"

BROJAC DW 0 ; brojač poslanih znakova

ZNAKOVI DB ... ; 80 znakova za slanje

```
ORG 500 ; prekidni potprogram  
PUSH R0  
PUSH R1  
PUSH R2  
MOVE SR, R0  
PUSH R0  
  
STORE R0, (PIOIACK) ; potvrda prekida  
  
LOAD R1, (BROJAC) ; dohvati brojača  
MOVE ZNAKOVI, R0 ; dohvati početne adrese  
  
ADD R0, R1, R0 ; računanje adrese znaka  
  
LOADB R2, (R0) ; dohvati znak iz mem.  
STORE R2, (PIOD) ; slanje znaka na PIO  
ADD R1, 1, R1 ; povećanje brojača  
STORE R1, (BROJAC)  
  
CMP R1, %D 80 ; je li poslan  
JR_NE JOS ; zadnji znak ?
```

KRAJ ; zabrani PIO-u da dalje zahtijeva prekide

```
MOVE 00, R0  
STORE R0, (PIOC)
```

JOS ; ima još znakova za slanje

```
POP R0  
MOVE R0, SR  
POP R2  
POP R1  
POP R0  
STORE R0, (PIOIEND) ; dojava kraja posluživ.  
RETI
```

GPIO Primjeri

Na prvi GPIO na ulazne bitove PIOD3-PIOD7 spojeno je 5 senzora (aktivna razina im je niska). Svaki puta kad se svih 5 senzora aktiviraju, FRISC treba bezuvjetno poslati procesu podatak iz bloka memorije s početnom adresom BLOK. Proces je spojen na drugi GPIO. Prvi GPIO radi u prekidnom načinu.

Nakon slanja 10 podataka, treba zabraniti daljnje generiranje prekida od strane GPIO1 i nastaviti izvođenje glavnog programa.

Odaberimo adrese za GPIO1 i GPIO2: FFFF1000 i FFFF2000. GPIO1 ćemo spojiti na INT i programirati da radi u načinu ispitivanja bitova, a GPIO2 u načinu postavljanja bitova.

```
PIOC1      EQU      0FFFF1000  
PIOD1      EQU      0FFFF1004  
PIOIACK1   EQU      0FFFF1008  
PIOIEND1   EQU      0FFFF100C  
PIOC2      EQU      0FFFF2000  
PIOD2      EQU      0FFFF2004
```

```
ORG      0  
MOVE    10000, R7  
JP      GLAVNI
```

```
ORG      8      ; prekidni vektor  
DW      500  
GLAVNI  MOVE    BLOK, R0          ; adresu podataka  
        STORE  R0, (PODAT)       ; stavi u PODAT  
  
        MOVE    %D 10, R0         ; broj podataka  
        STORE  R0, (BROJAC)       ; stavi u BROJAC
```

```
; inicijalizacija sklopa PIO1
MOVE %B 111100000010111, R0
STORE R0, (PIOC1) ; pošalji u CR
MOVE %B 010, R0
STORE R0, (PIOC2) ; pošalji u CR

MOVE %B 10000, SR ; dozvoli INT
PETLJA JR PETLJA ; "koristan posao"

PODAT DW 0
BROJAC DW 0

BLOK DW 3, 1, 5, 7, 3, 9, 2, 6, 5, 4
; prekidni potprogram
ORG 500

PUSH R0 ; spremi kontekst
PUSH R1
MOVE SR, R0
PUSH R0
```

STORE R0, (PIOIACK1); dojavi prihvat na PIO1

LOAD R0, (PODAT) ; dohvati adresu podatka
LOAD R1, (R0) ; dohvati podatak
STORE R1, (PIOD2) ; šalji podatak na PIO2
ADD R0, 4, R0 ; pomakni adresu na
STORE R0, (PODAT) ; sljedeći podatak

LOAD R0, (BROJAC) ; dohvati
SUB R0, 1, R0 ; i smanji
STORE R0, (BROJAC) ; brojač
JR_NZ JOS ; ima li još podataka

KRAJ ; zabrani prekide na PIO1

MOVE %B 011, R0
STORE R0, (PIOC1) ; pošalji u CR

JOS POP R0 ; obnovi kontekst
MOVE R0, SR
POP R1
POP R0
STORE R0, (PIOIEND1) ; dojava kraja posluž.
RETI

DMA

Sklopoški UI prijenos - DMA

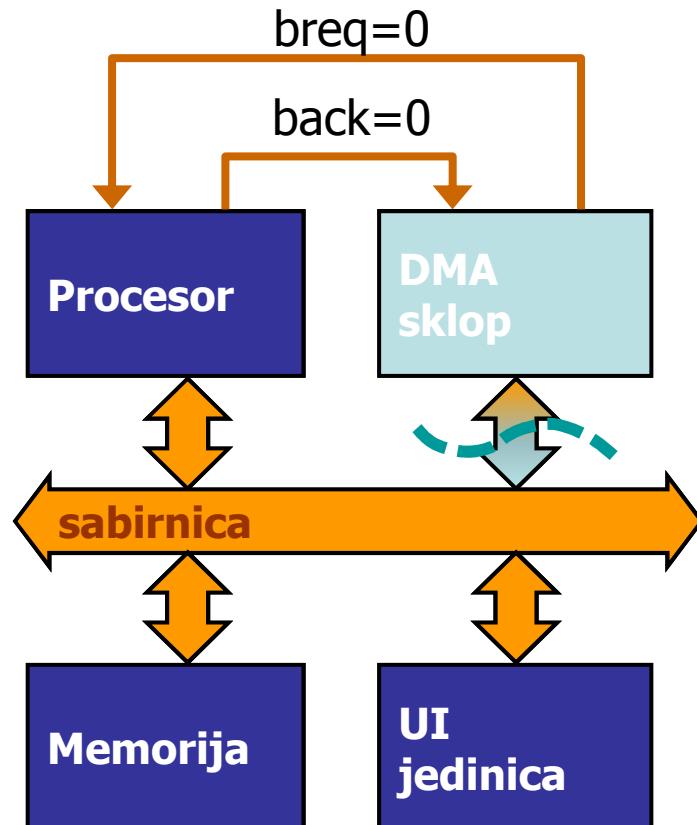
DMA prijenos

- Izravni pristup memoriji ili DMA (Direct Memory Access) je **sklo povski** ulazno-izlazni prijenos
- Prijenos ne obavlja procesor, nego posebna DMA-jedinica (ili DMA-sklop, engl. DMA controller)
- Velika brzina prijenosa (podatci ne prolaze kroz procesor)
- Moguć je prijenos između memorije i UI jedinice ili neka druga kombinacija izvora i odredišta podataka

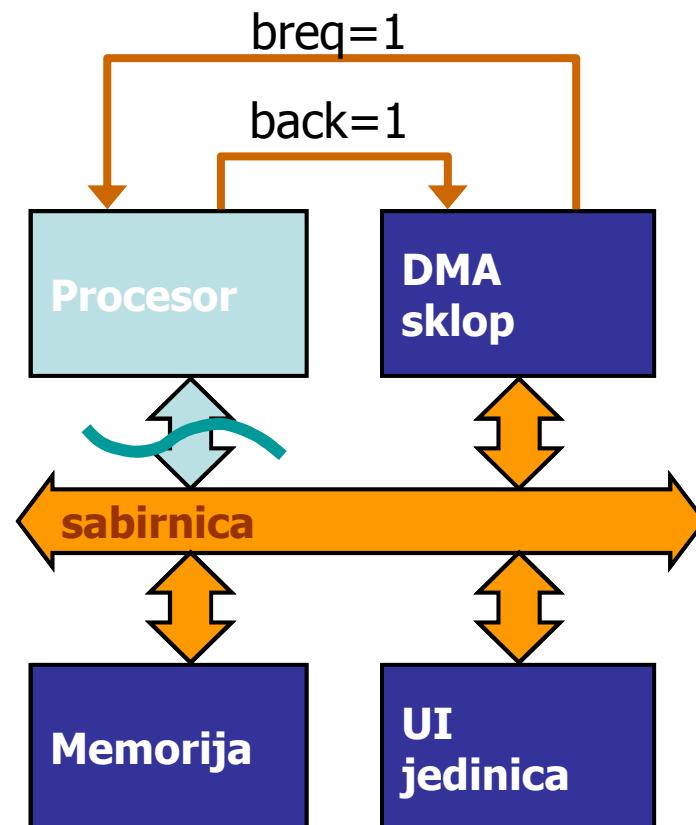
DMA prijenos

- Osnovna ideja:
 - Procesor inicijalizira DMA-sklop, kako bi ovaj znao koliko podataka treba prenijeti, gdje su izvor i odredište podataka itd.
 - DMA-sklop preuzme upravljanje nad sabirnicom i obavlja prijenos
 - Procesor je za vrijeme prijenosa neaktivan
 - Procesor i DMA-sklop moraju se međusobno sinkronizirati kako bi u svakom trenutku samo jedan od njih upravljao sabirnicom: za to se koriste linijama BREQ i BACK
 - Pod upravljanjem sabirnicom misli se na iniciranje sabirničkih transakcija čitanja i pisanja (npr. upravljanje adresnom sabirnicom, rd, wr, itd.)

DMA - Shema



Procesor upravlja
sabirnicom*



DMA-sklop upravlja
sabirnicom

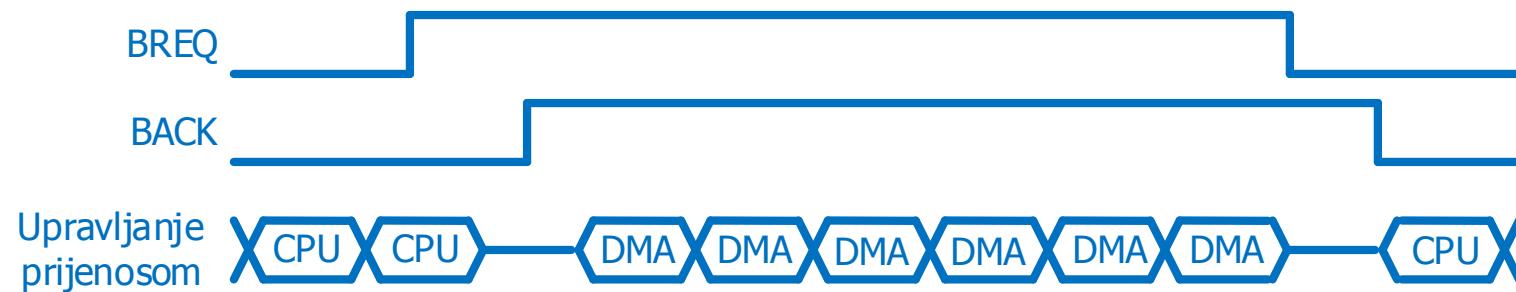
*Slika simbolično prikazuje stanje na sabirnici. DMA sklop ne upravlja sabirnicom, ali mu procesor može normalno pristupati kao i drugim UI jedinicama i memoriji

DMA - Vrste prijenosa

- Vrste DMA prijenosa:
 - Zaustavljanje procesora (engl. continuous, halting)
 - Krađa ciklusa (engl. cycle stealing, word-at-a-time)
 - Blokovski (engl. burst)

DMA - Zaustavljanje procesora

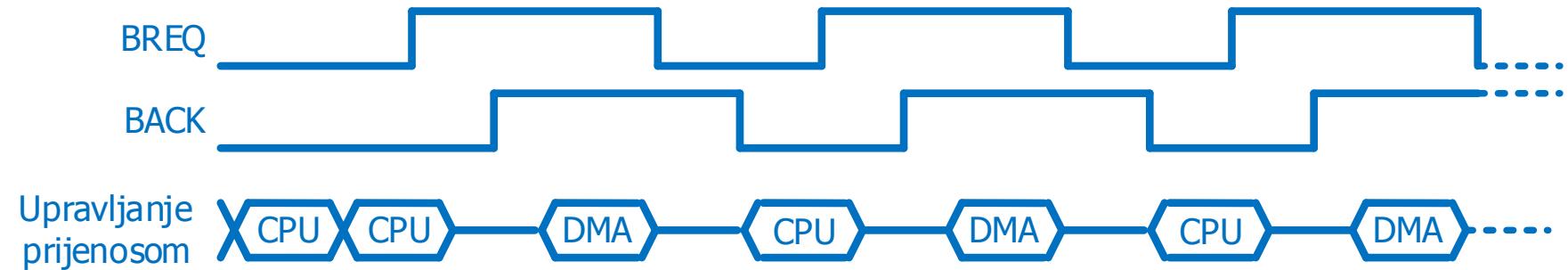
- **breq** je prioritetniji od svih zahtjeva za prekid



DMA - Krađa ciklusa

- DMA kradom ciklusa odvija se ovako:
 - DMA-sklop preuzme upravljanje nad sabirnicom
 - DMA-sklop prenese **jedan podatak**
 - Upravljanje nad sabirnicom se vraća procesoru
 - Gornja tri koraka se ponavljaju dok se ne prenesu svi podatci
- Prednost: procesor se usporava, ali ipak izvodi glavni program
- Nedostatak: sporije od zaustavljanja procesora (dio vremena troši se na sinkronizaciju oko upravljanja sabirnicom)

DMA - Krađa ciklusa



DMA - Blokovski prijenos

- DMA blokovskim prijenosom odvija se ovako:
 - DMA-sklop preuzme upravljanje nad sabirnicom
 - DMA-sklop prenese **nekoliko podataka (tj. blok)**
 - Upravljanje nad sabirnicom se vraća procesoru
 - Gornja tri koraka se ponavljaju dok se ne prenesu svi podatci
- Kompromis između zaustavljanja procesora i krađe ciklusa
- Zaustavljanje procesora može se promatrati kao blokovski prijenos kod kojeg je veličina bloka jednaka svim podatcima
- Krađa ciklusa može se promatrati kao blokovski prijenos kod kojeg je u bloku samo jedan podatak

DMA-sklopovi

- DMA-sklopovi mogu imati različite mogućnosti:
 - prijenos podataka između različitih izvora i odredišta
 - memorija → VJ
 - memorija → memorija (kopiranje bloka, inicijalizacija bloka)
 - VJ → memorija
 - VJ → VJ
 - traženje podataka u bloku ili kombinirani prijenos s traženjem
 - odabir načina prijenosa
 - dojava kraja prijenosa
 - postavljanje spremnosti
 - prekid
 - generiranje impulsa (mogu se prebrajati npr. CT-om)
 - itd.

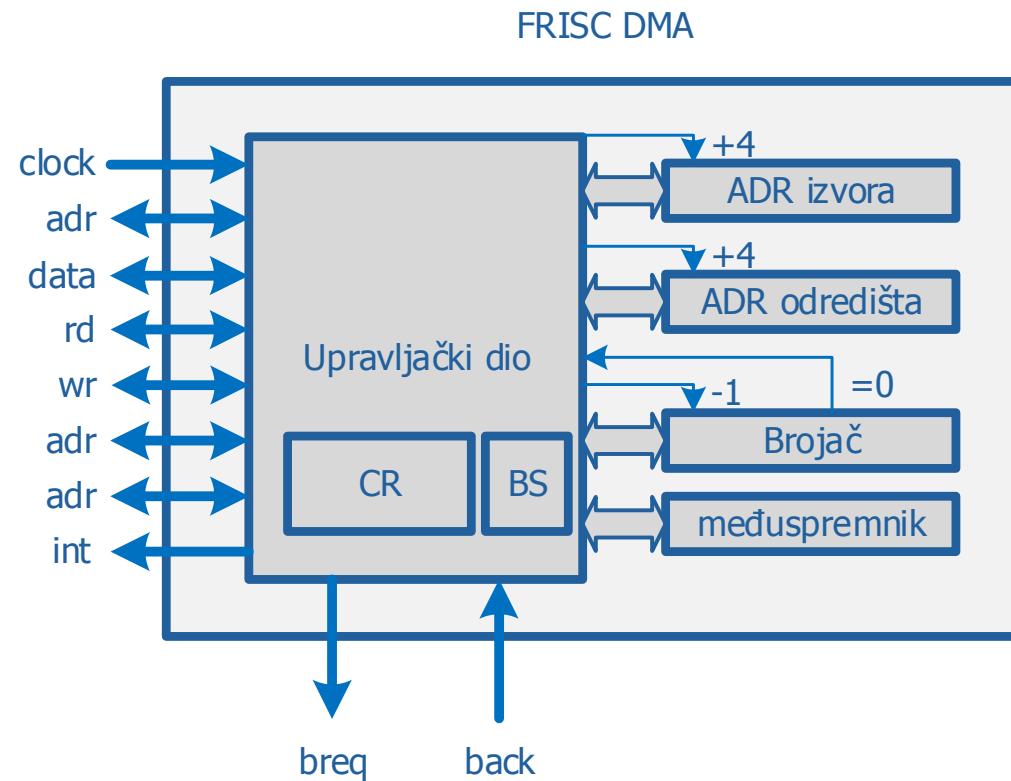
Sklop FRISC-DMA



Sklop FRISC-DMA

- Sklop FRISC-DMA ima sljedeće mogućnosti:
 - Prijenos 32-bitnih podataka
 - Odabir vrste prijenosa:
 - zaustavljanje procesora
 - krađa ciklusa
 - Zbog memorijskog preslikavanja, tj. jednakog pristupa memoriji i UI jedinicama, izvor i odredište mogu biti bilo koja kombinacija ovih komponenata
 - Pristup UI jedinicama obavlja se bez provjere (bezuvjetno) što znači da UI jedinica mora biti sposobna primati/slati podatke brzinom DMA-prijenosu
 - Dojava kraja prijenosa obavlja se postavljanjem spremnosti i prekidom

FRISC-DMA



Programski pogled

adresa	Pisanje	čitanje
PA	upis adrese izvora	čitanje adrese izvora
PA + 4	upis adrese odredišta	čitanje adrese odredišta
PA + 8	upis u brojač podataka	čitanje brojača podataka
PA + 12₁₀	upis upravljačke riječi CR	Čitanje CR
PA + 16₁₀	pokretanje prijenosa	-
PA + 20₁₀	potvrda prihvata prekida (tj. brisanje bistabila stanja)	čitanje bistabila stanja BS

CR

bitovi 31 – 4		bit 3	bit 2	bit 1	bit 0
-	DESTINATION	SOURCE	MODE		INT
	0 – memorija 1 – vanjska jedinica	0 – memorija 1 – vanjska jedinica	0 procesora 1 – krađa ciklusa	– zaustavljanje 1 – postavlja prekid	0 – ne postavlja prekid 1 – postavlja prekid

Inicijalizacija sklopa FRISC-DMA

- Prilikom inicijalizacije treba zadati (u bilo kojem redoslijedu):
 - broj podataka
 - adresu izvora
 - adresu odredišta
 - upravljačku riječ
- Nakon toga treba pokrenuti DMA-prijenos upisom na adresu PA+16 (odmah nakon toga počinje DMA-prijenos)
 - ako je odabrano zaustavljanje procesora, dio programa iza naredbe za pokretanje DMA-prijenosa neće se izvoditi sve dok se prijenos ne završi
 - ako je odabrana krađa ciklusa, dio programa iza naredbe za pokretanje se izvodi, ali usporeno

DMA Primjeri

Treba prenijeti 1000_{10} podataka iz VJ na adresi FFFF3330 u memoriju od adrese 4000. Adresa DMA-sklopa je FFFF0000. Prije ili za vrijeme prijenosa može se izvoditi potprogram POTPR. Kad je prijenos gotov, treba pozvati potprogram GOTOVO te nastaviti s radom glavnog programa (pretpostavka je da su ovi potprogrami već napisani i da postoje u memoriji). Prijenos treba obaviti zaustavljanjem procesora.

```
DMA_SRC    EQU 0FFFF0000
DMA_DEST   EQU 0FFFF0004
DMA_SIZE   EQU 0FFFF0008
DMA_CTRL   EQU 0FFFF000C
DMA_START  EQU 0FFFF0010
DMA_BS     EQU 0FFFF0014
```

```
INIT      MOVE 1000, SP
          ; inicijalizacija DMA-sklopa
          MOVE 0FFFF3330, R0      ; upis adrese
          STORE R0, (DMA_SRC)     ; izvora

          MOVE 4000, R0           ; upis adrese
          STORE R0, (DMA_DEST)    ; odredišta

          MOVE %D 1000, R0         ; upis broja
          STORE R0, (DMA_SIZE)    ; podataka

          ; Upis upravljačke riječi:
          MOVE %B 0100, R0
          STORE R0, (DMA_CTRL)
```

```
; pokretanje DMA-prijenosu
GLAVNI    STORE R0, (DMA_START)

; dio programa koji će se izvesti tek nakon
; završenog DMA-prijenosu
STORE R0, (DMA_ACK) ; brisanje BS DMA

CALL POTPR
CALL GOTOVO
...           ; nastavak glavnog programa
```

DMA Primjeri

- Prethodni zadatak treba riješiti krađom ciklusa.

```
DMA_SRC    EQU 0FFFF0000
DMA_DEST   EQU 0FFFF0004
DMA_SIZE   EQU 0FFFF0008
DMA_CTRL   EQU 0FFFF000C
DMA_START  EQU 0FFFF0010
DMA_BS     EQU 0FFFF0014
```

```
INIT      MOVE 1000, SP
          ; inicijalizacija DMA-sklopa
          MOVE 0FFF3330, R0      ; upis adrese
          STORE R0, (DMA_SRC)    ; izvora

          MOVE 4000, R0          ; upis adrese
          STORE R0, (DMA_DEST)   ; odredišta

          MOVE %D 1000, R0        ; upis broja
          STORE R0, (DMA_SIZE)   ; podataka

          ; Upis upravljačke riječi:
          MOVE %B 0110, R0
          STORE R0, (DMA_CTRL)
```

```
; pokretanje DMA-prijenosu
GLAVNI STORE R0, (DMA_START)
; dio programa koji se izvodi usporeno i istodobno
; s DMA-prijenosom

CALL POTPR

; dio programa koji se mora izvesti NAKON DMA-prijenosu

CEKAJ LOAD R0, (DMA_BS) ; učitaj spremnost DMA-sklopa
AND R0, 1, R0
JR_Z CEKAJ ; čekaj završetak prijenosa
STORE R0, (DMA_ACK) ; brisanje BS DMA
CALL GOTVO
... ; nastavak glavnog programa
```

DMA - Brzina prijenosa

- Približna usporedba bezuvjetnog prijenosa i DMA za N podataka (npr. iz VJ u memoriju):

	Naredba	trajanje
PETLJA	LOAD R0, (VJ_READ)	2
	STORE R0, (R2)	2
	ADD R2, 4, R2	1
	SUB R3, 1, R3	1
	JR_NZ PETLJA	2

- Vidimo da za N podataka programu treba približno $N*8$ perioda (stvarno mu treba jedan period manje jer kod zadnjeg prolaska kroz petlju JR traje samo jedan period, ali je to zanemarivo u odnosu na $N*8$).
- U slučaju prijenosa podataka korištenjem FRISC-DMA, sustavu će trebati $N*2$ perioda (DMA čita podatak preko sabirnice (prvi period) i onda ga zapisuje (drugi period)).
- Kod mnogih stvarnih VJ čest je slučaj da je DMA dio VJ pa DMA ne treba komunicirati sa VJ preko procesorske sabirnice te je za jedan podatak potreban samo jedan ciklus !!!