Arhitektura procesora FRISC

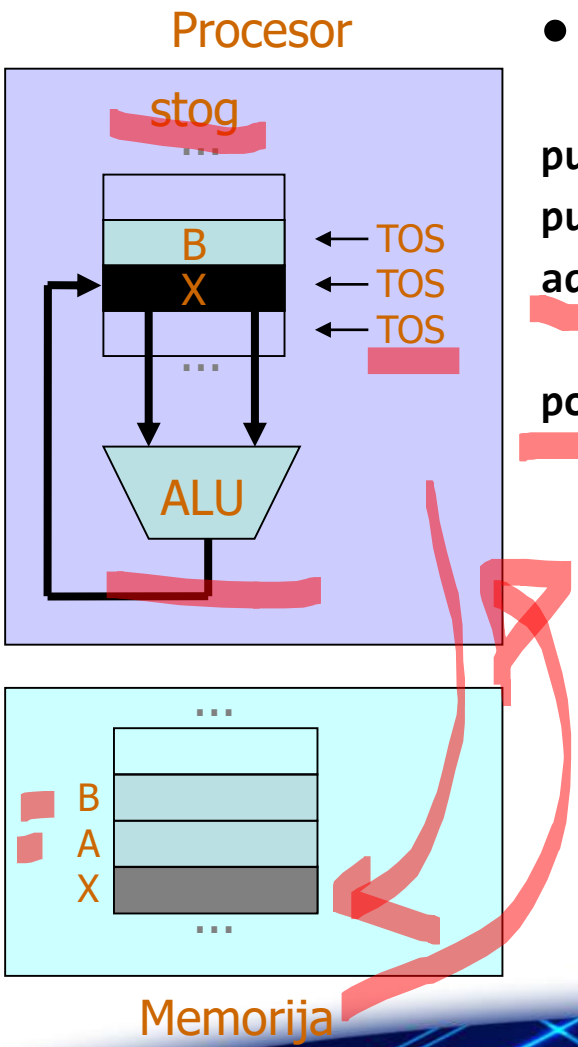
Arhitekture s obzirom na dohvat operanada

- Stogovne arhitekture
- Akumulatorske arhitekture
- Arhitekture registar-memorija
- Arhitekture registar-registar (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 eksplicitno gdje se nalaze operandi niti gdje se sprema rezultat
- Naredba za obradu podataka pristupa samo internom stogu

Stogovna arhitektura



- 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
```

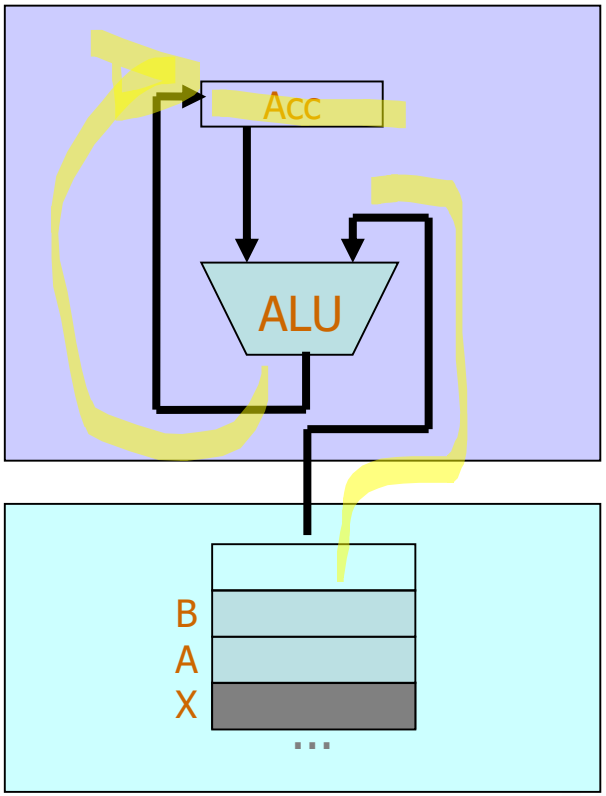
Stogovna arhitektura

- 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 jednostavni

- 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 !!!

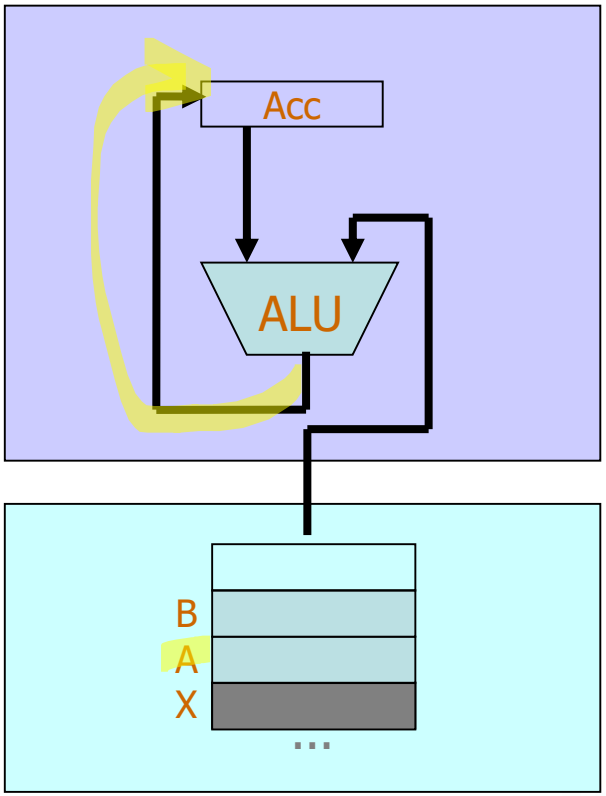
Akumulatorska arhitektura



- **Jedan operand je uvijek u posebnom registru** koji se naziva Akumulator (Acc)
- Ako postoji, **drugi operand se čita iz memorije**
- **Rezultat se sprema 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
```

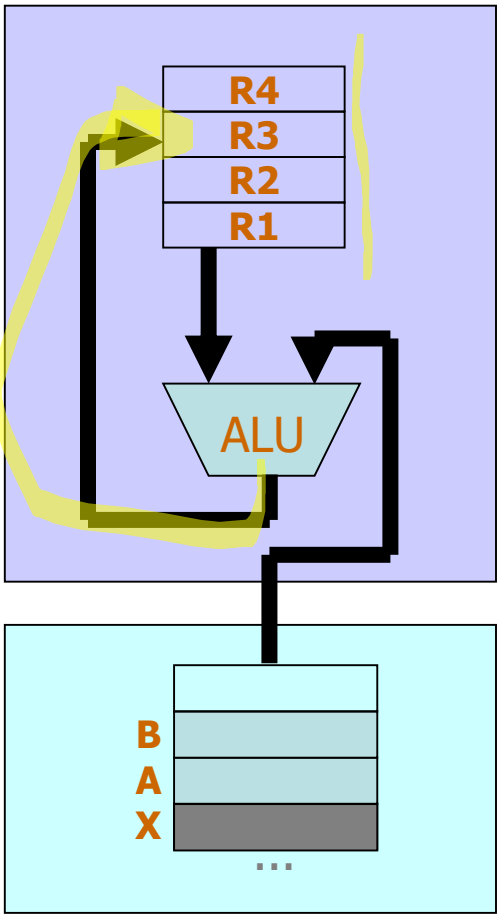
Akumulatorska arhitektura

- 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 jednostavni

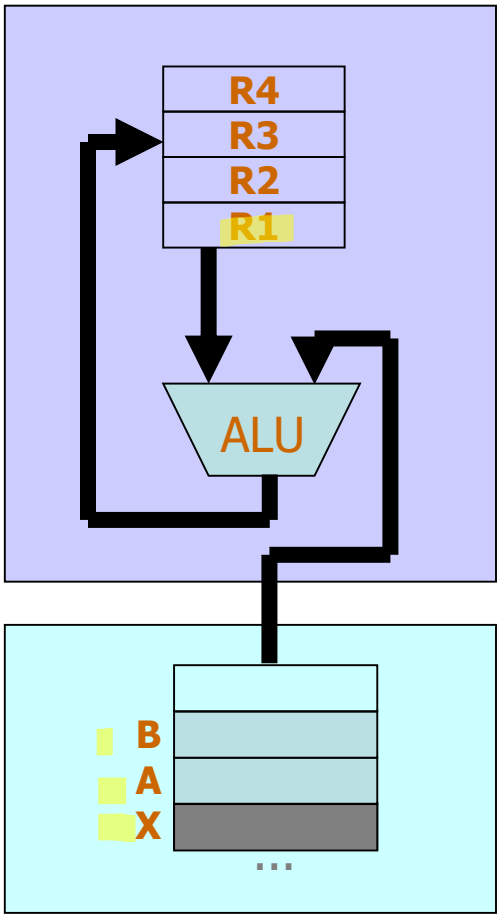
- Loše strane ove arhitekture:
  - Međurezultati (osim zadnjeg) se ne mogu koristiti već sve mora biti pohranjeno u memoriju
  - Jako velik broj pristupa memoriji !!!
  - Prevođenje nije efikasno

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



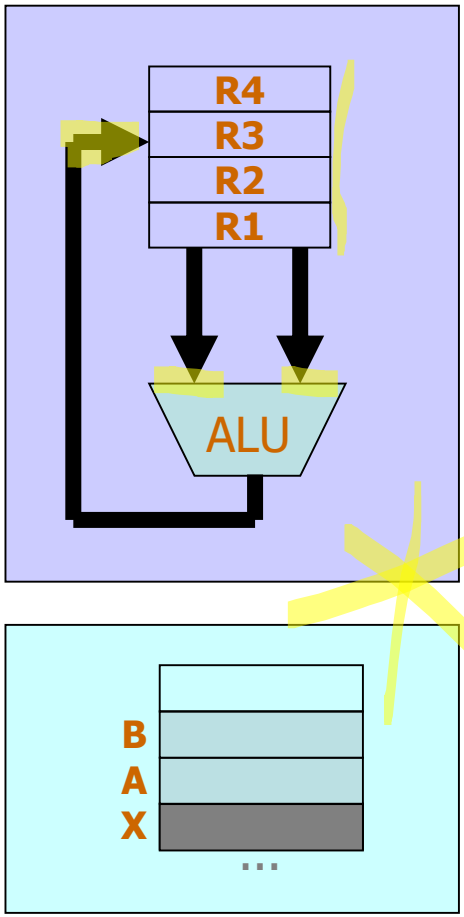
Primjer: računanje izraza x=a+b

```
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-memorija

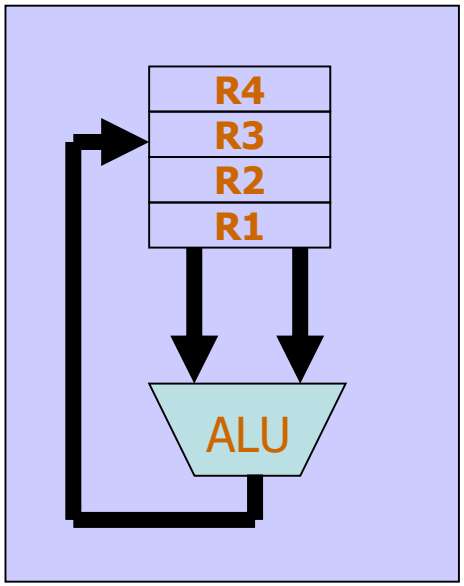
- Karakteristike arhitekture registar-memorija su:
  - Varijable se mogu čuvati u registrima opće namjene (što je više registara to je manje potrebno komunicirati s memorijom, a posljedica je brži pristup podatcima)
  - Prevoditelji efikasnije prevode programe jer su naredbe moćnije i podatci mogu biti u registrima
  - Naredbe veće i sporije
- Prednosti:
  - Jednostavan pristup podatcima u memoriji
- Nedostatci
  - Dodatan pristup memoriji pri izvođenju naredaba, vrijeme izvođenja varira ovisno o naredbi i operandima

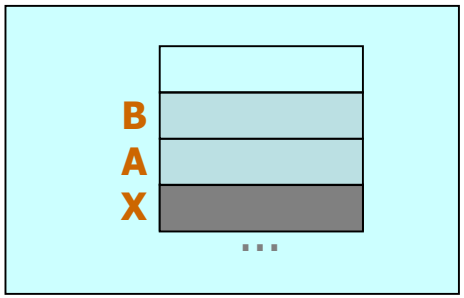
Arhitektura registar-registar (load-store)



- **Oba operanda su u registrima opće namjene**
- **Rezultat se sprema 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
```

Arhitektura registar-registar (load-store)

- Karakteristike registarsko-registarske arhitekture su (isto kao kod registarsko-memorijske):
  - Varijable se mogu čuvati u registrima opće namjene
  - Prevoditelji efikasnije prevode programe
  - Naredbe veće i sporije
- Prednosti:
  - Brzo izvođenje, jednostavan format naredaba, jednostavno generiranje kôda, jednostavnost protočne strukture, uniformno vrijeme izvođenja
- Nedostatci:
  - Veći broj naredaba u programu zbog zasebnih učitavanja podataka iz memorije

Usporedba prethodnih 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 registarskih arhitektura (operacije s podatcima koji su bilo gdje na stogu)

- većina današnjih procesora ima registarsku arhitekturu (reg-mem ili reg-reg) među kojima su više zastupljene reg-reg (load-store) arhitekture

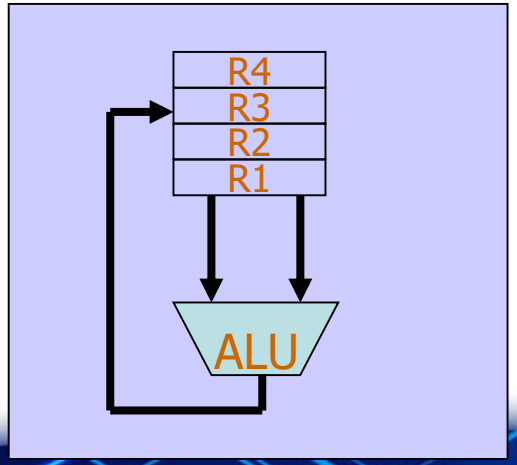
Primjeri navedenih arhitektura

- Stogovna:
  - WISC CPU/16, MISC M17, picoJava
- Akumulatorska
  - EDSAC
- Registarsko-memorijska
  - Motorola 68000 i Intel 80x86 (imaju karakeristike reg-mem i reg-reg)
- Registarsko-registarska (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 sklopovski 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...

CISC

- 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 programskih 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 programskih 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

CISC

- 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 “sklopovski”)
  - Skupo projektiranje i visoka cijena
- Primjeri CISC procesora: Intel 80x86, Motorola 68000

RISC

- 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 sklopovsku izvedbu.

RISC

- 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

RISC

- 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 registra 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
- Zato biramo **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 ***

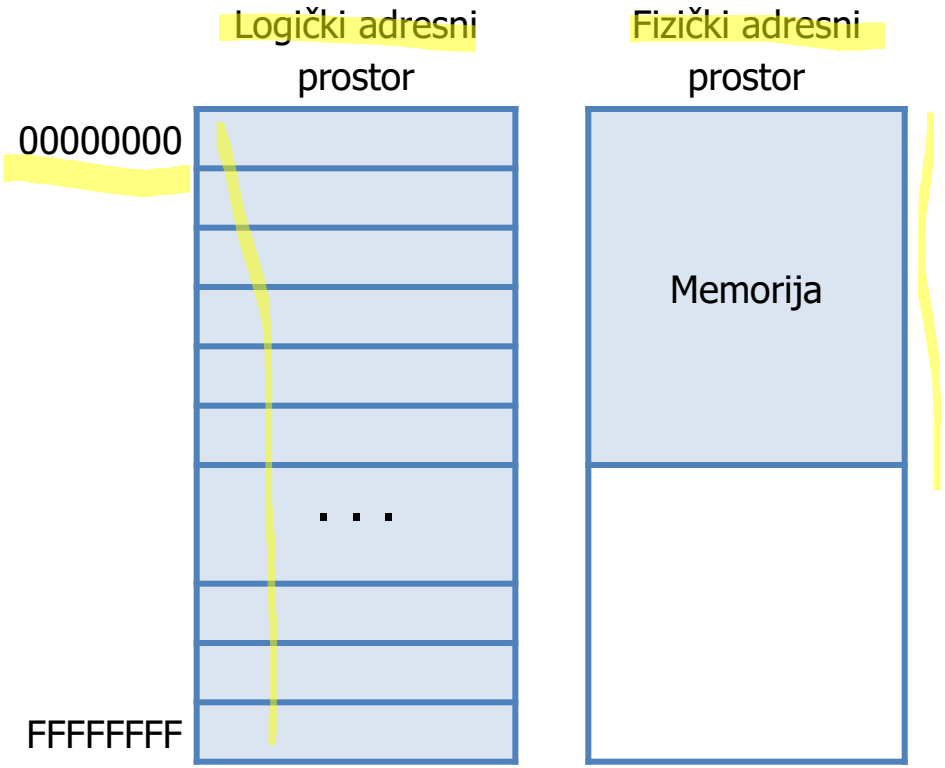
- Širinu memorijske riječi odabiremo da bude jedan bajt (8b), ali zbog veće efikasnosti ćemo moći pročitati odjednom 4 bajta (32b) što nam dozvoljava širina podatkovne sabirnice

* 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).

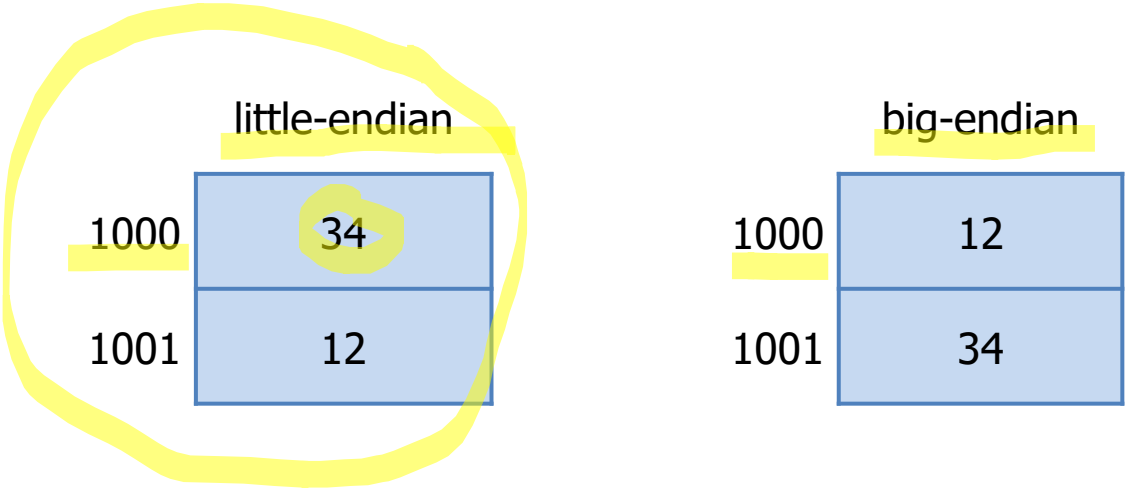
Odluka: Širina sabirnica

- 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



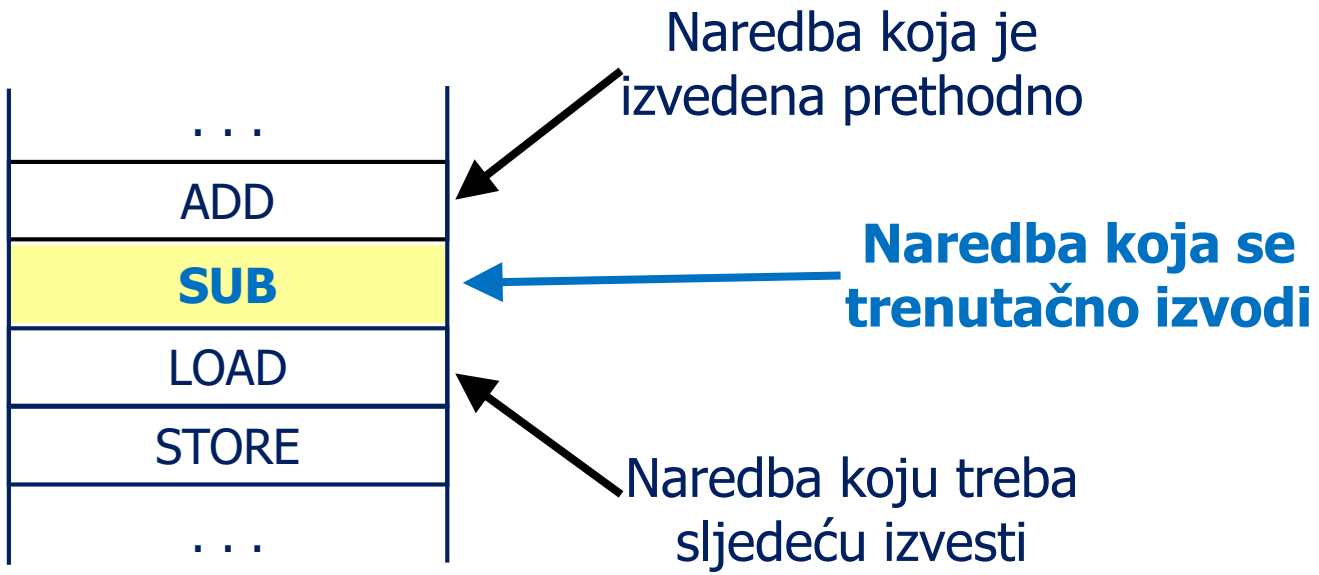
Redoslijed zapisa podataka u mem.



Zapis podatka $1234_{(16)}$ u memoriji

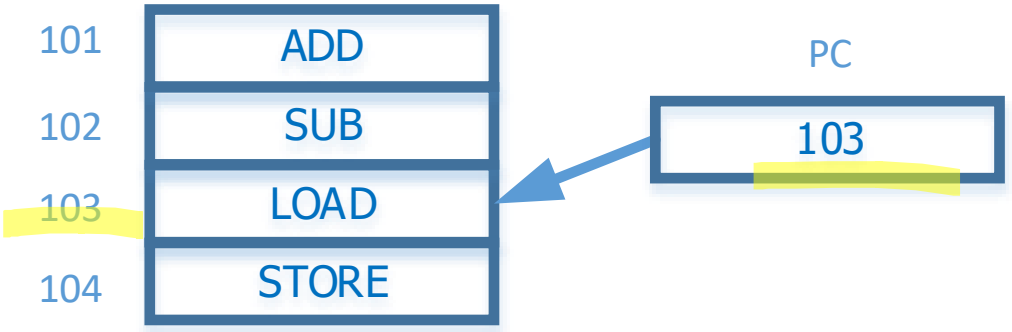
Programsko brojilo

- 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 "znati" 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 broji)



* Za sada zanemarimo širinu naredaba i točne adrese

Programsko brojilo

- Naš registar PC bit će širok 32 bita, jer smo za adresnu sabirnicu odabrali istu širinu, a logično je da se registrom PC može adresirati cijeli memorijski prostor

- Registar 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 procesorske riječi

32

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
  - Memorijske 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 memorijske 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. ...

Ukratko: 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 EKSKLUZIVNI 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 *operanda*** 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  registar, (adresa)
STORE registar, (adresa)
```

  - Naredba LOAD "puni registar", tj. čita sadržaj četiriju memorijskih lokacija počevši od zadane *adrese* i stavlja ih u *registar*

  - Naredba STORE "sprema registar", 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 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