

# ***Arhitektura procesora FRISC***

# **Poboljšana inačica procesora FRISC**

**(proširenje skupa naredaba i detaljnija  
arhitektura)**

# ***Poboljšana inačica procesora***

- Za proširenje skupa 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 precizniji način njihovog spajanja
  - Put podataka i upravljačku jedinicu
  - Priključke procesora
  - Spajanje procesora s memorijom i vanjskim jedinicama
  - Način komunikacije procesora s memorijom i vanjskim jedinicama

# Proširenja aritmetičko-logičkih naredaba

# ***Proširenja skupa naredaba***

---

- Do sada uvedene naredbe omogućuju izvođenje većine zadaća koje nam mogu zatrebati
- 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 AL-naredaba

- Pretpostavimo 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 komplicira:
  - Operandi više nisu samo registri, nego mogu biti i brojevi
- Jedno od pravila pri oblikovanju procesora:
  - Ubrzati rad onoga što se često koristi

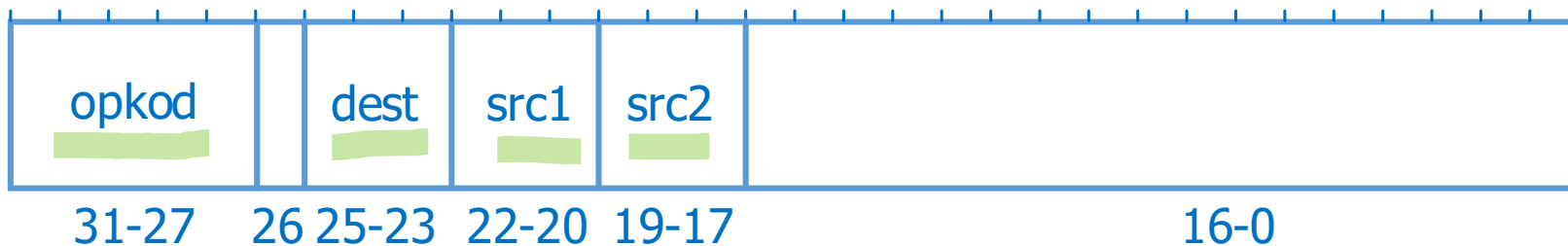


# Proširenja AL-naredaba

- 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:





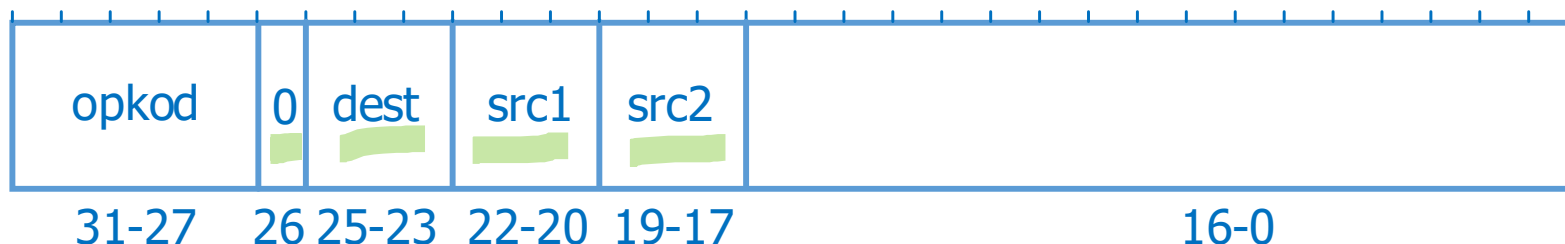
# Proširenja AL-naredaba

- 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

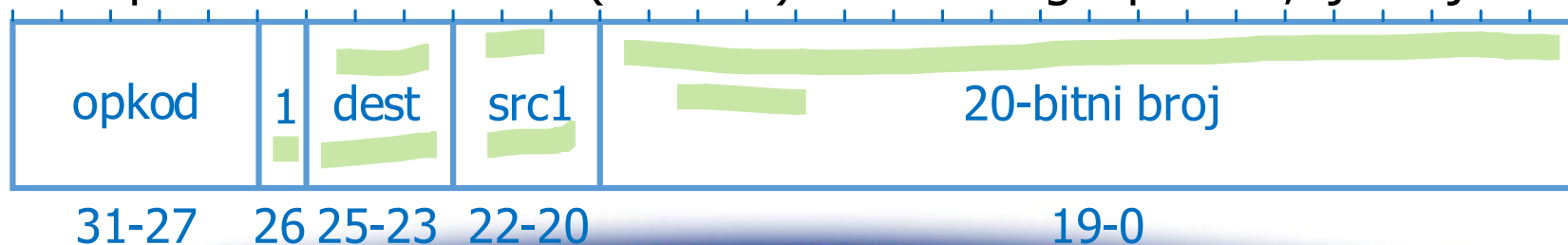


# Proširenja AL-naredaba

- 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 mjesta 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

; izračunavamo:  $R0 := (R1 + 55) - (R2 \text{ xor } 4ABC)$

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*

; izračunavamo:  $R0 := (R1 + 55) - (R2 \text{ xor } 4ABC)$

```
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

# 20-bitni brojevi u naredbama

---

Podsjetnik:

- **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

# 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., **ali oni moraju imati u gornjih 13 bita ili sve jedinice ili sve nule.**
  - Asemblerski prevoditelj provjerava ispravnost brojeva tako da prvo svaki napisani broj **pretvori u 32-bitni zapis**:
    - 32-bitni NBC ako je broj pozitivan
    - 32-bitni 2's komplement zapis ako je broj negativan
  - Ako su u dobivenom 32-bitnom zapisu gornjih 13 bitova isti, onda je **broj ispravan** i u strojni kôd se upisuje najnižih 20 bitova 32-bitnog zapisa
  - Ako gornjih 13 bitova nisu isti, onda je **broj pogrešno napisan** (poruka: wrong number)

# ***20-bitni brojevi u naredbama***

---

Domaća zadaća:

proučiti dokument  
„Proširivanje 20 na 32 bita”



# ***Proširenja AL-naredaba - pomaci i rotacije***

---

- U asemblerskom programiranju često treba obaviti različite 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 samo za jedan bit ili za željeni broj bitova
- Teorijski bi mogli odabrati samo dvije operacije pomaka/rotiranja, a ostale ostvariti programski.
  - To bi bilo u skladu s idejom "jednostavnog procesora"

# ***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 sprema 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 podacima 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 podacima 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

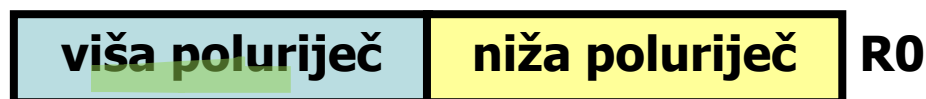
SHL, SHR, ASHR, ROTL, ROTR:

C=izlazni bit, V=0, Z=nula, N=predznak

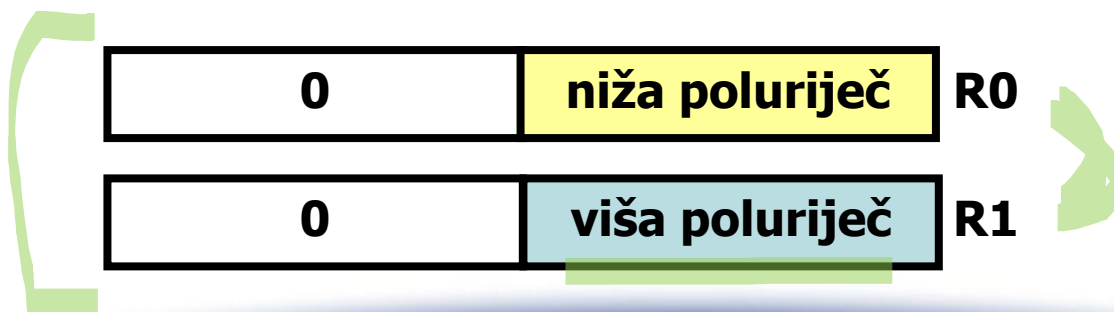
- Jedina novost kod postavljanja zastavica je u naredbama pomaka i rotacije gdje se u zastavicu C upisuje izlazni bit od zadnjeg koraka pomaka/rotacije.

# 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, 0FFFF, R0 ; više bitove R0 stavljamo u 0

AND R1, 0FFFF, R1 ; više bitove R1 stavljamo u 0

CMP R1, R0 ; usporedba

JP\_UGT 100 ; viša p.r. > niža p.r.  $\Rightarrow$  "goto 100"

JP 200 ; viša p.r.  $\leq$  niža p.r.  $\Rightarrow$  "goto 200"

# ***Proširenja AL-naredaba - put podataka***

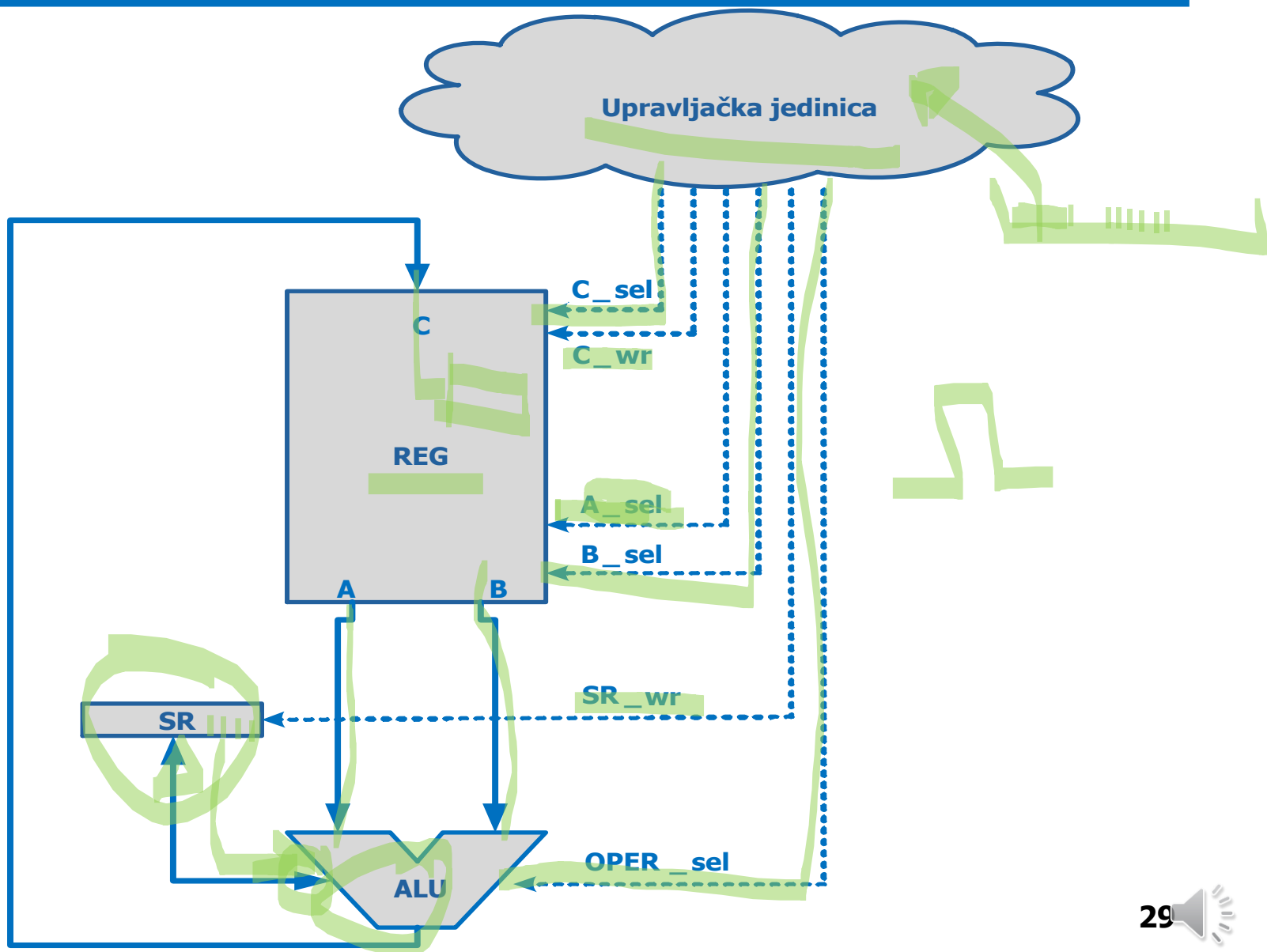
---

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

>>>>

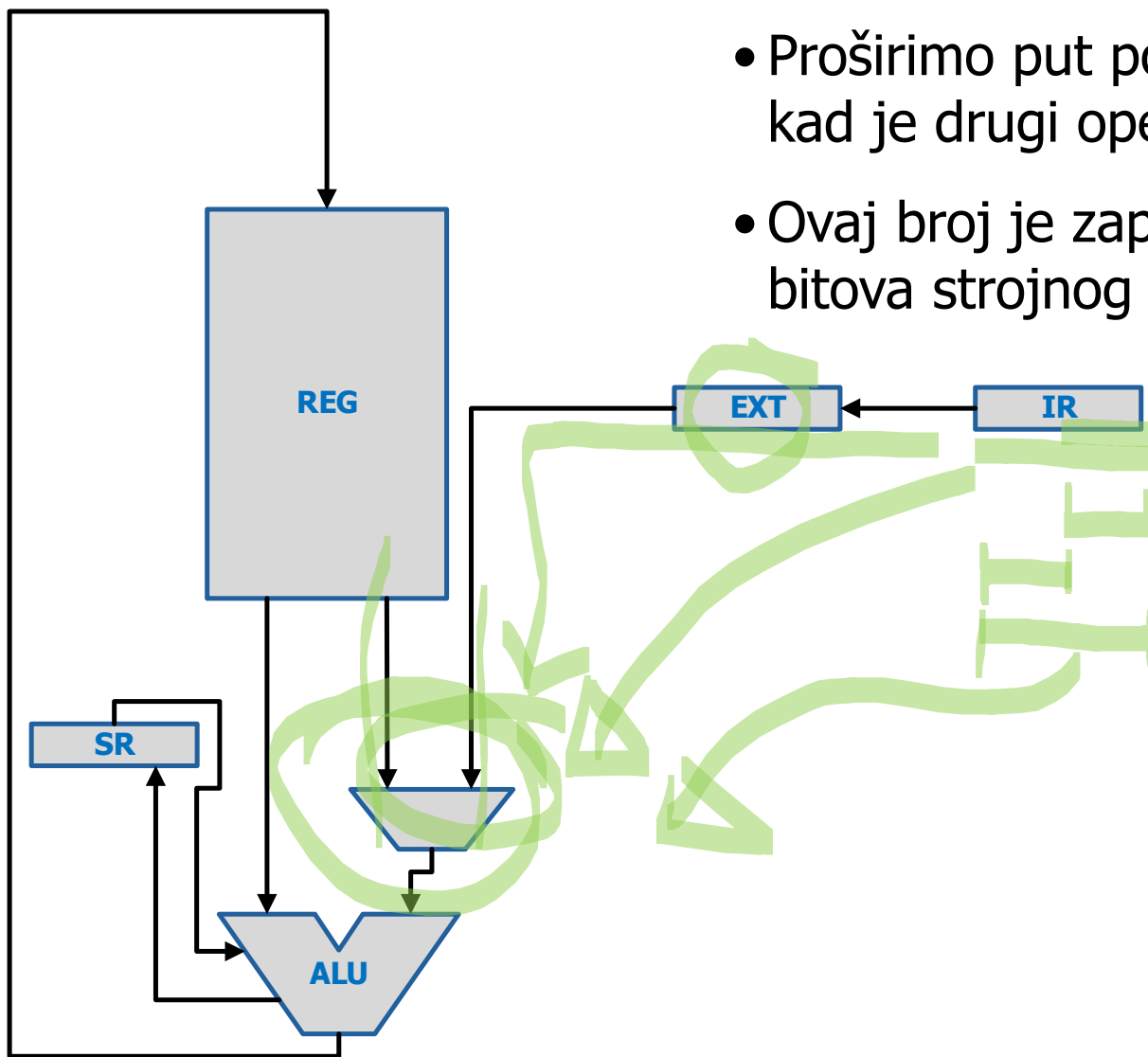


# Upravljanje putom podataka za AL naredbe



# 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 registarskih naredaba

# Uvođenje registarskih naredaba


- Dosta česta zadaća koju treba obaviti je punjenje broja (konstante) u registar, na primjer:
  - pri inicijalizaciji brojača za petlju
  - pri inicijalizaciji 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 registarskih 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 registarskih 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 registarskih 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 registarskih 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 registarskih naredaba

- Registar SR je 5-bitni (to još ne znamo, ali tako će ispasti 😊) pa moramo definirati kako MOVE barata s podacima različitih širina ( $R_i$  označuje opći registar R0-R7):
  - **MOVE podatak, SR** → nakon predznačnog proširenja podatka na 32 bita, odbacuje se gornjih 27 bita i u SR se puni samo najnižih 5 bita podatka
  - **MOVE  $R_i$ , SR** → u SR se puni najnižih 5-bita iz registra  $R_i$
  - **MOVE SR,  $R_i$**  → SR se puni u najniže bitove od  $R_i$ , a viši bitovi se pune nulama

# Uvođenje registarskih 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 apsolutnog iznosa.  
; "Mali brojevi" stanu u strojni kod (19+1 bit).

MOVE 12345, R0 ; 12345 ima 17+1 bit

MOVE 0FFFF1234, R1 ; isto kao -EDCC, a  
; EDCC ima 16+1 bit

MOVE -100A, R2 ; 100A ima 13+1 bit

>>>>

# Uvođenje registarskih 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 ima 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
```

# ***Uvođenje registarskih nar. - strojni kôd***

---

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

# *Registarske naredbe – put podataka*

