Arhitektura procesora FRISC

Proširenje memorijskih naredaba

Proširenje memorijskih naredaba

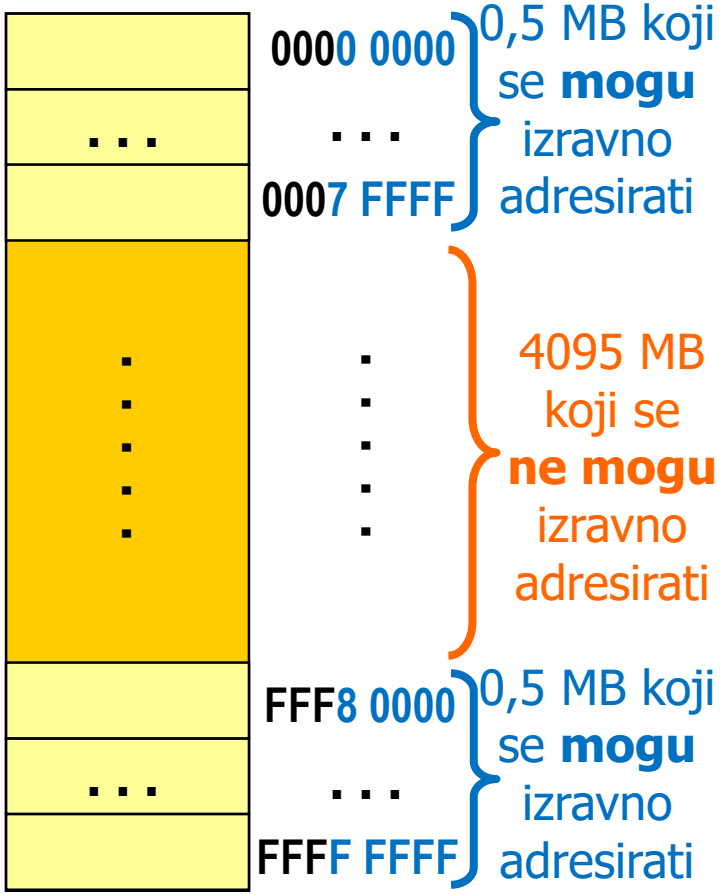
- Podsjetimo se da su memorijske naredbe LOAD i STORE imale zadavanje adrese **brojem**
- U strojnom kôdu je za adresu bilo na raspolaganju samo 20 bitova
- Budući da smo za adresnu sabirnicu odabrali širinu od 32 bita, moramo točno definirati vrijednost gornjih 12 bitova

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 2^{19} lokacija memorije (najnižih pola MB)
- Adresama FFF8 0000 do FFFF FFFF adresiramo najviših 2^{19} lokacija memorije (najviših pola MB),što ukupno daje 2^{20} lokacija (tj. jedan MB)
- Na ovaj način ne možemo adresirati svih 2^{32} 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 strojnom kodu naredbe kod adresiranja registrom 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 registar (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 memorijsku naredbu.

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

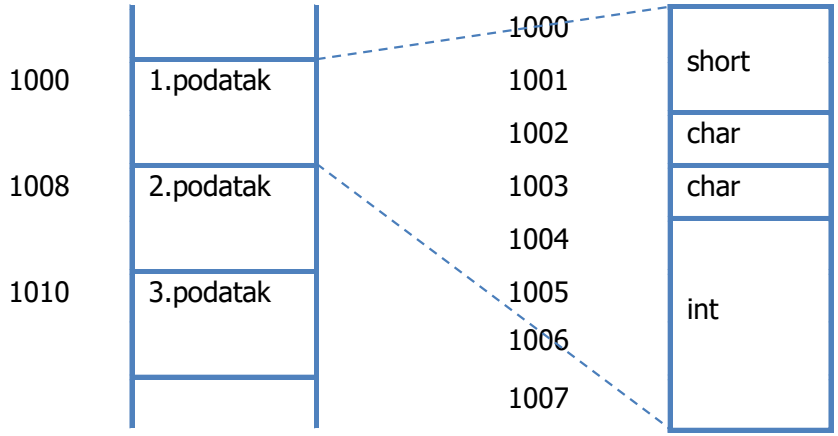
Proširenje memorijskih nar. - širina podatka

- U pristupanju bajtovima, riječima i poluriječima obično postoje 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 izbjeći 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 poluriječ 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 riječ 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 pamtiti 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 registar iz kojeg se uzima podatak koji se stavlja na stog
  - dest - opći registar u kojeg 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... za nove naredbe

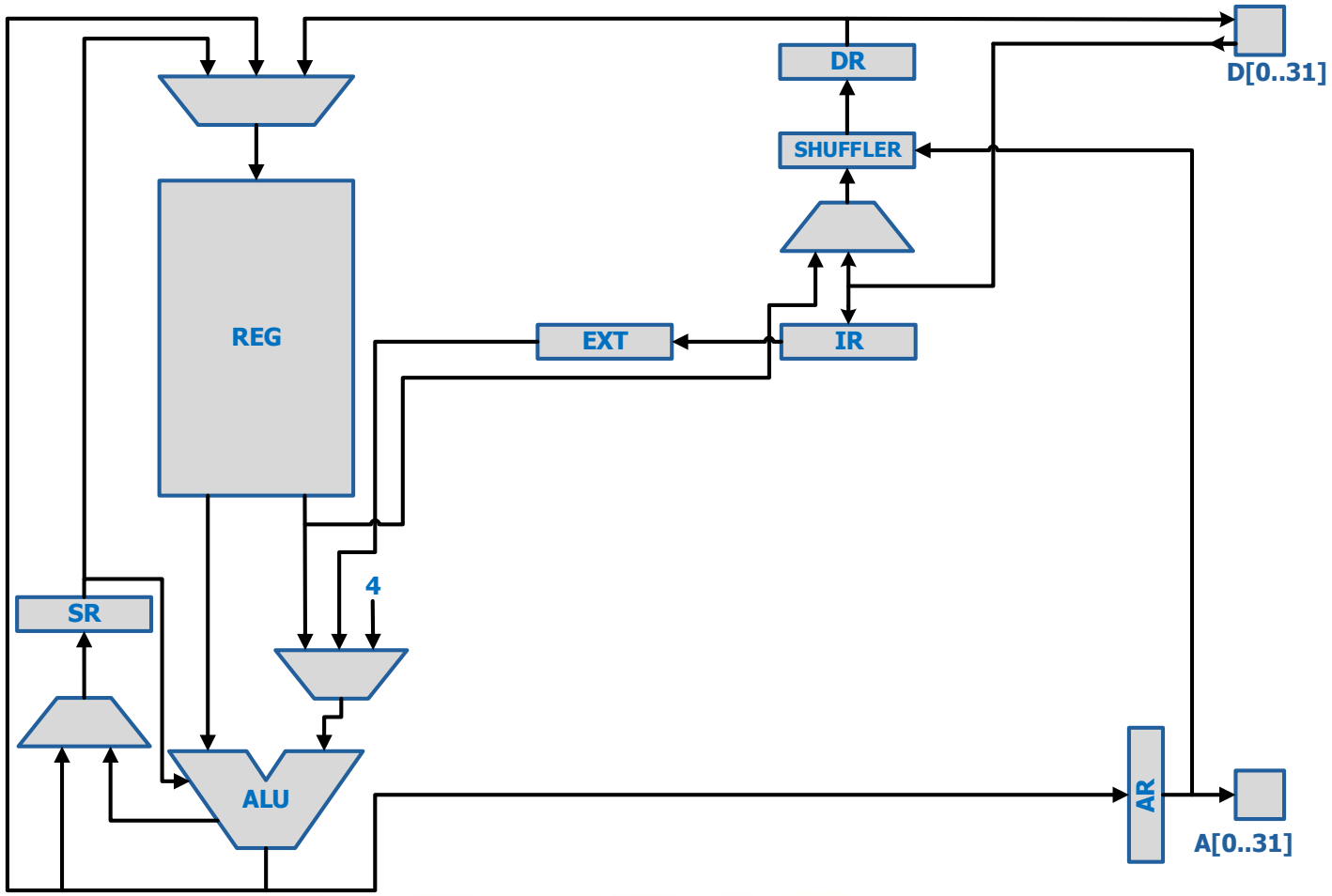
LOAD R1, (R4+20)
STORE R2, (R5-10)

LOAD R0, (R5) //+0

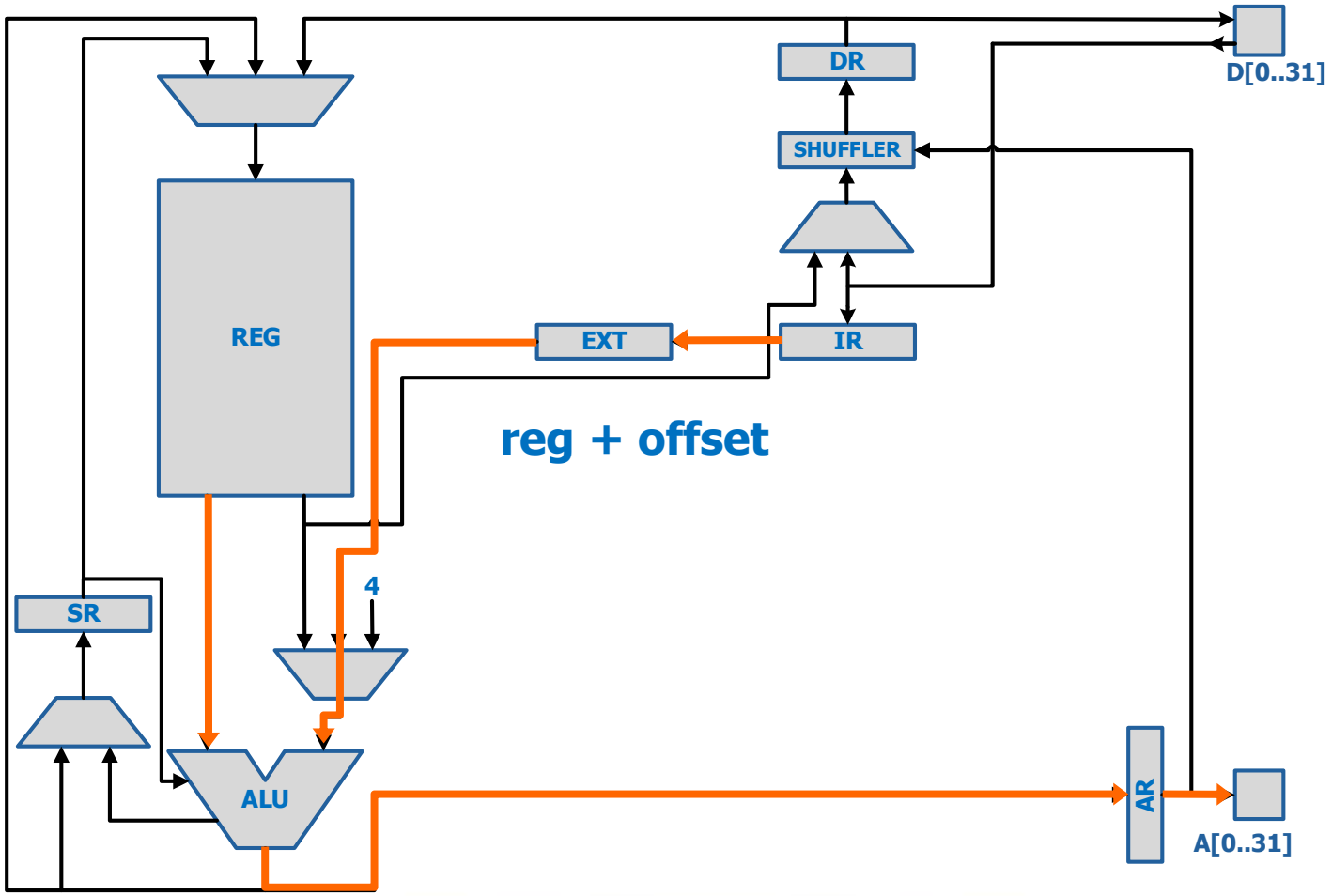
LOADB
STOREB
LOADH
STOREH

PUSH src:
R7-4 -> R7
src -> (R7)

POP dest:
(R7) -> dest
R7+4 -> R7



Put podataka (adresiranje LOAD/STORE)



LOAD R1, (R4+20)
STORE R2, (R5-10)

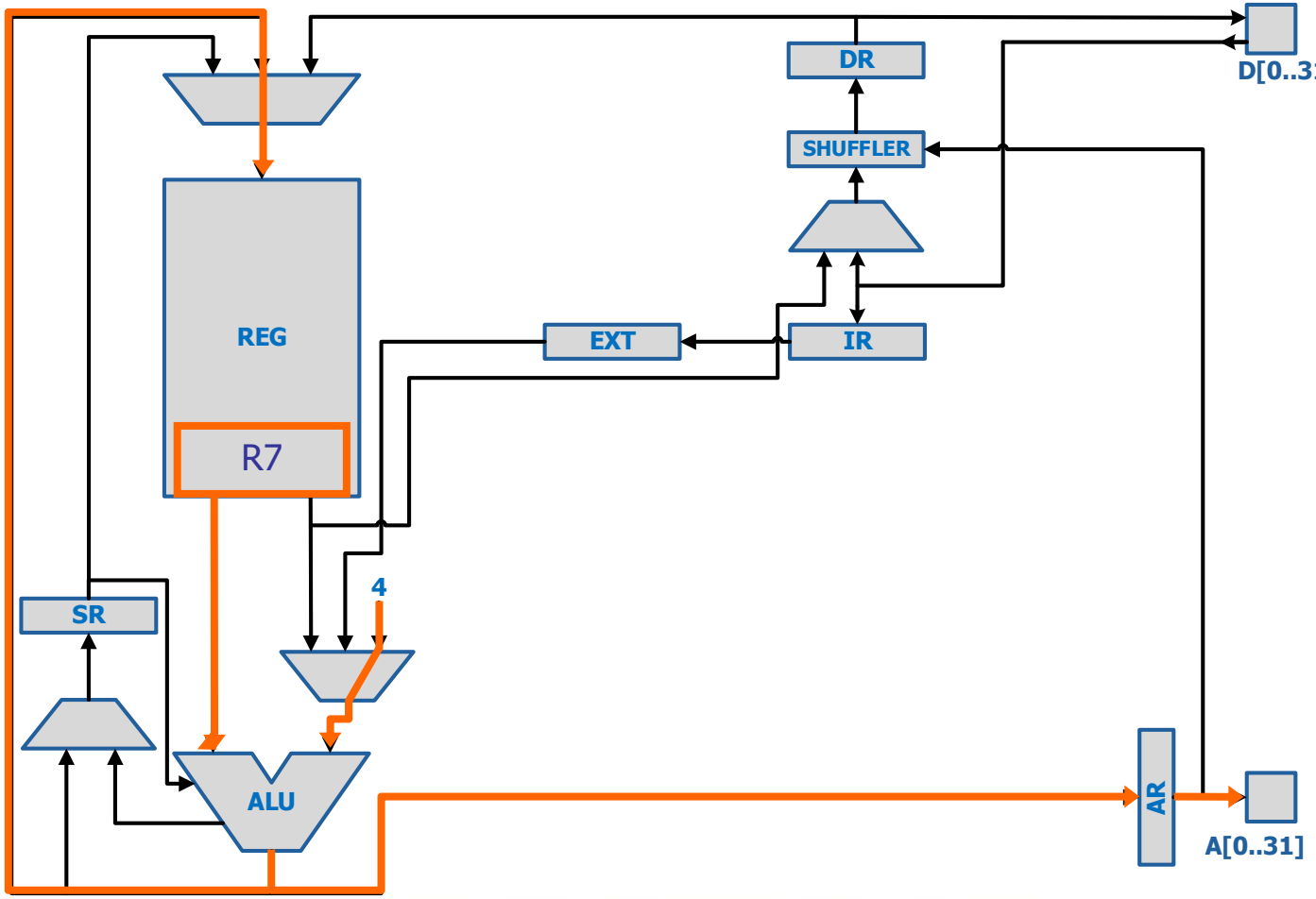
LOAD R0, (R5) //+0

LOADB
STOREB
LOADH
STOREH

PUSH src:
R7-4 -> R7
src -> (R7)

POP dest:
(R7) -> dest
R7+4 -> R7

Put podataka (adresiranje u PUSH/POP)



LOAD R1, (R4+20)
STORE R2, (R5-10)

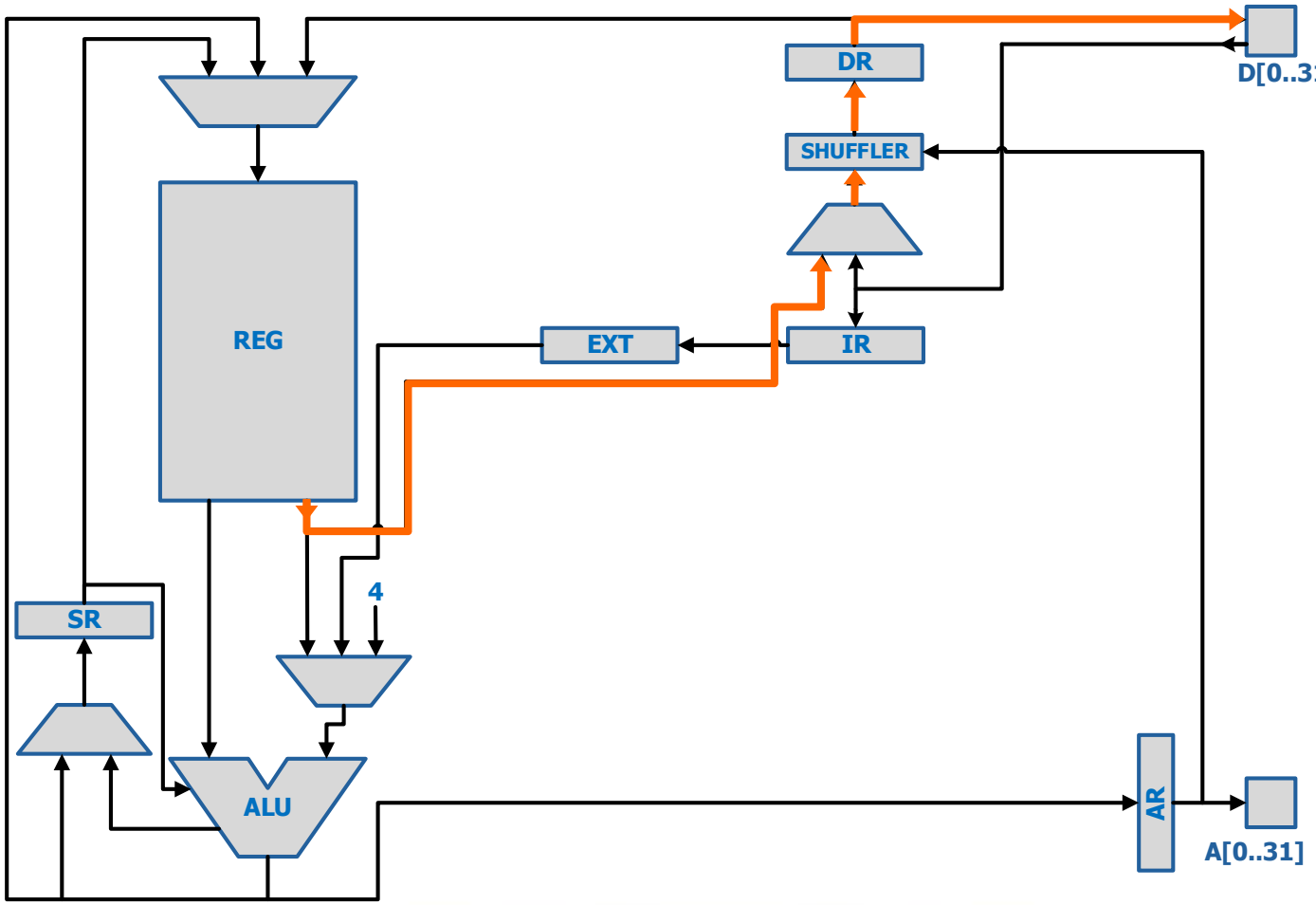
LOAD R0, (R5) //+0

LOADB
STOREB
LOADH
STOREH

PUSH src:
R7-4 -> R7
src -> (R7)

POP dest:
(R7) -> dest
R7+4 -> R7

Put podataka (podatci za STORE i PUSH)



LOAD R1, (R4+20)
STORE R2, (R5-10)

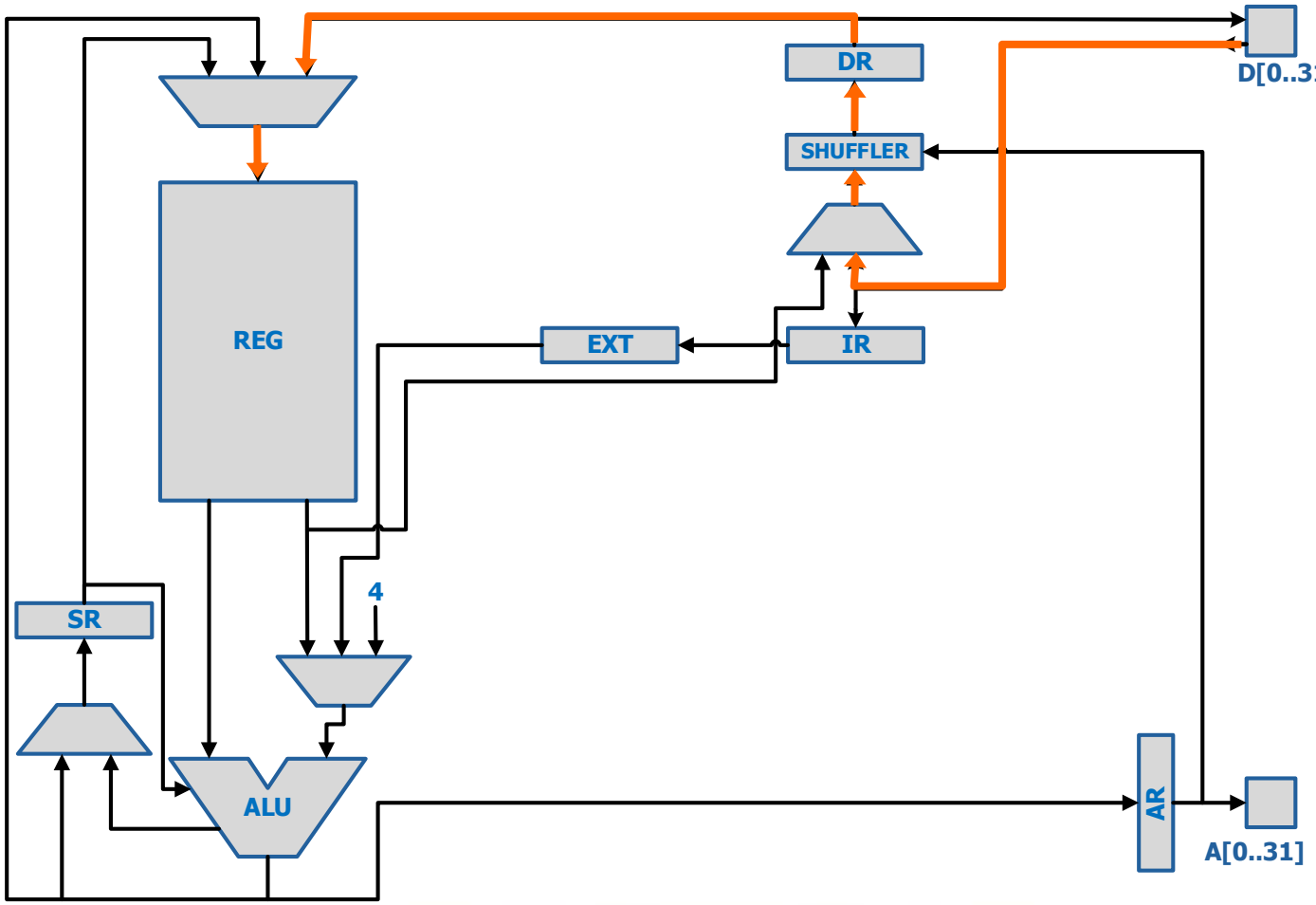
LOAD R0, (R5) //+0

LOADB
STOREB
LOADH
STOREH

PUSH src:
R7-4 -> R7
src -> (R7)

POP dest:
(R7) -> dest
R7+4 -> R7

Put podataka (podatci za LOAD i POP)



LOAD R1, (R4+20)
STORE R2, (R5-10)

LOAD R0, (R5) //+0

LOADB
STOREB
LOADH
STOREH

PUSH src:
R7-4 -> R7
src -> (R7)

POP dest:
(R7) -> dest
R7+4 -> R7

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 registar koji će svojim 32-bitnim sadržajem zadati adresu skoka na bilo kojoj memorijskoj lokaciji u prostoru od 4 GB
  - Novi način zadavanja adrese piše se ovako:

    `JP (addreg)`

    gdje je **`adrreg`** jedan od registara opće namjene

Proširenje upravljačkih naredaba - primjeri

Primjer skoka na 32-bitnu adresu:

Treba skočiti na adresu 200000.

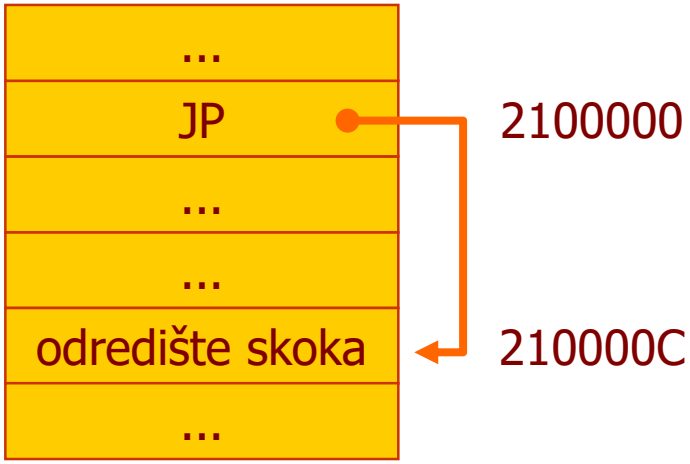
```
    LOAD R0, (A_200000)   ;;; Priprema adrese u R0
    JP  (R0)              ;;; Skok "na R0"
    ...

;;; 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 funkcionira:
  - možemo s bilo kojeg mjesta 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 registra)
- Pogledajmo kada je ovakav način skakanja naročito nepraktičan i koje je moguće rješenje...

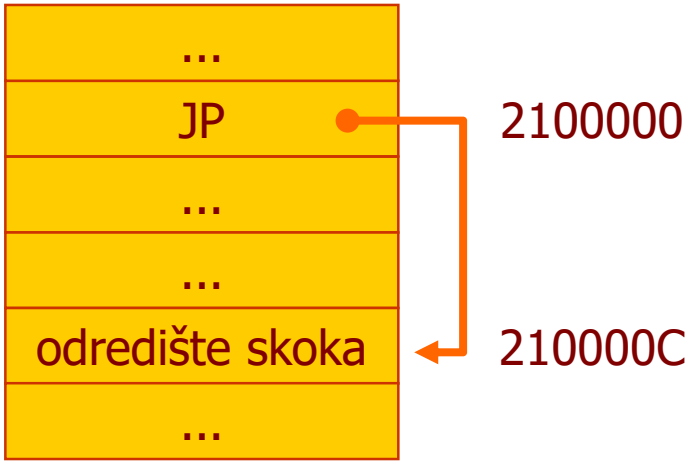
Proširenje upravljačkih nar. - relativni skok

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



- Pretpostavimo da na procesor spojimo npr. 256MB memorije
- Pretpostavimo 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

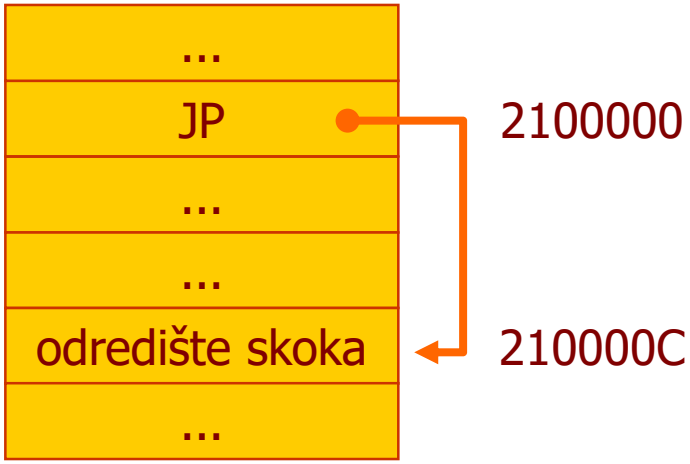
Proširenje upravljačkih nar. - relativni skok



- 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

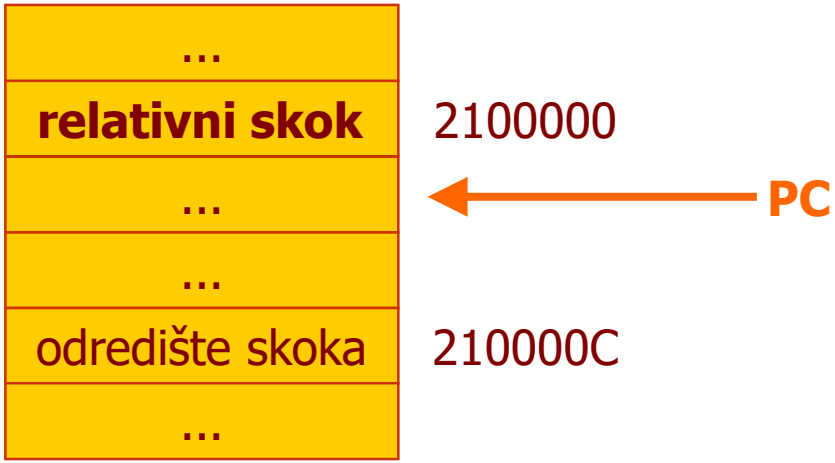
. . . . .

Proširenje upravljačkih nar. - relativni skok



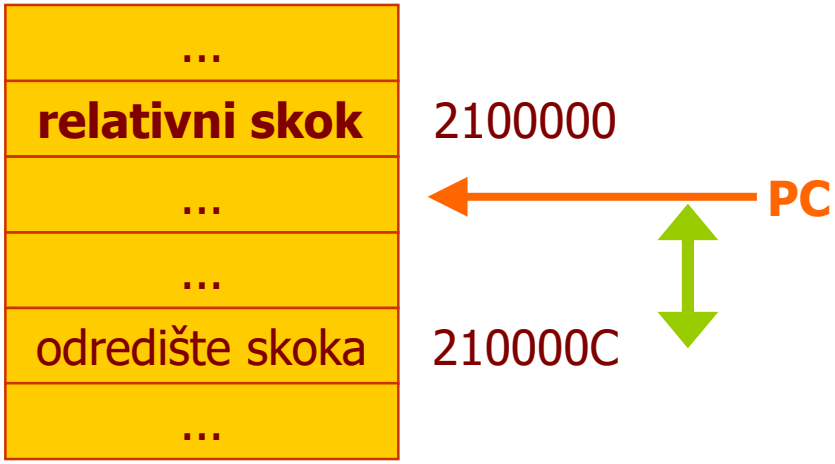
- Dodatna nepraktičnost: ne možemo koristiti labelu za skok (labele povećavaju čitljivost)
- Također treba znati adresu na koju želimo skočiti (Znamo li je? Teorijski znamo, ali u praksi NE - zato što obično ne vodimo računa na kojim adresama se nalaze naredbe)
- **Postoji li bolje rješenje?**

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

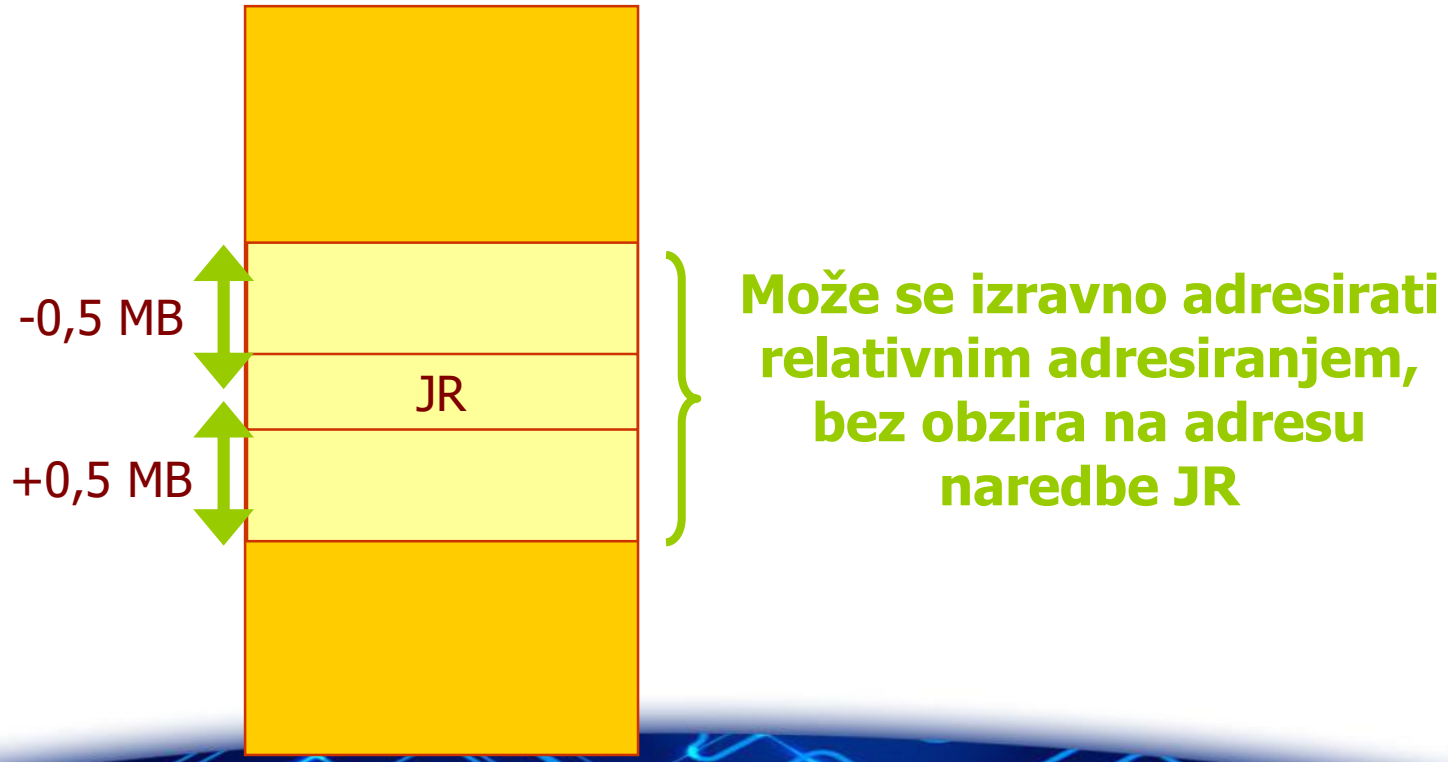
Proširenje upravljačkih nar. - relativni skok

- **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:

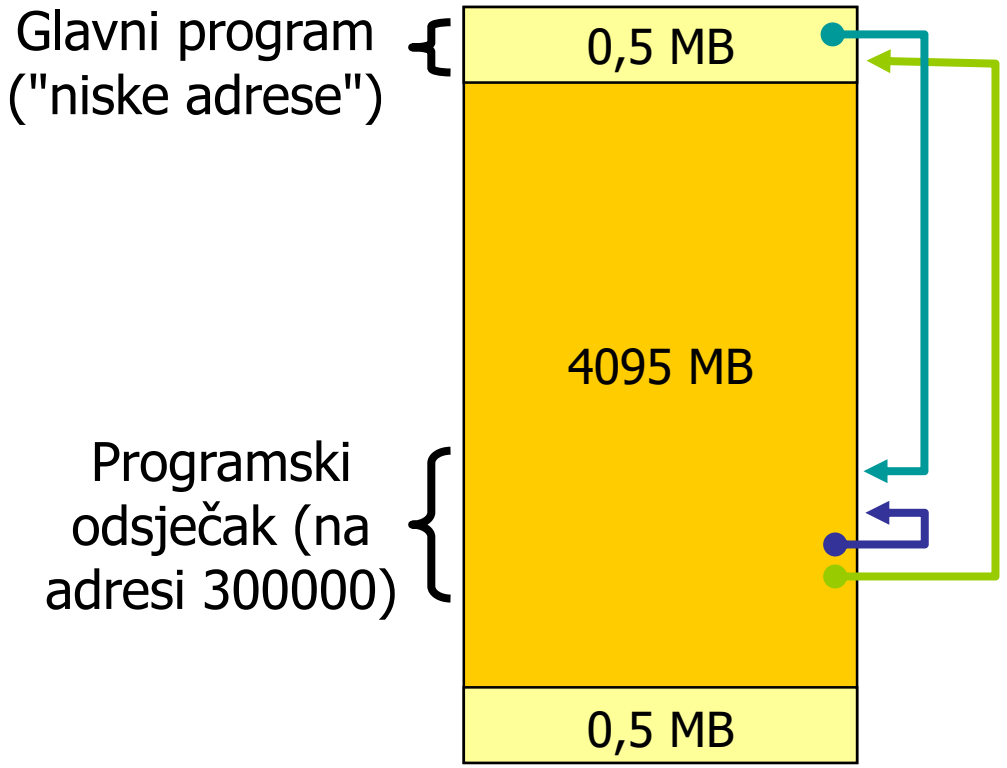


Proširenje upravljačkih naredaba - primjer

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 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
```

apsolutno adresiranje je nemoguće jer je adresa 300000 šira od 20 bitova, a relativno je nemoguće jer je skok predug

>>>>

```
; 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
```

>>>>

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

```
; 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 apsolutna adresa se može koristiti jer je odredišna adresa malena

Proširenje upravljačkih nar. - potprogrami

- S običnim skokovima i ostalim naredbama možemo programirati svaki algoritam
- Međutim, 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 registar 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 sprema 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 → SP`**
  - **`PC → (SP)`**
  - **`ADRESA → PC`**

} `PC → stog`

---

* 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) → PC**
  - **SP + 4 → SP**

} **stog → PC**

Proširenje upravljačkih nar. - potprogrami

- Zašto se povratna adresa sprema na stog?
- Zato jer je stog promjenjive veličine, a to omogućava jednostavno gniježđenje potprograma do potrebne dubine, kao i rekurzivno pozivanje potprograma
- Spremanje na fiksnu memorijsku 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 sprema na novo mjesto, ovisno o trenutačnoj popunjenosti 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

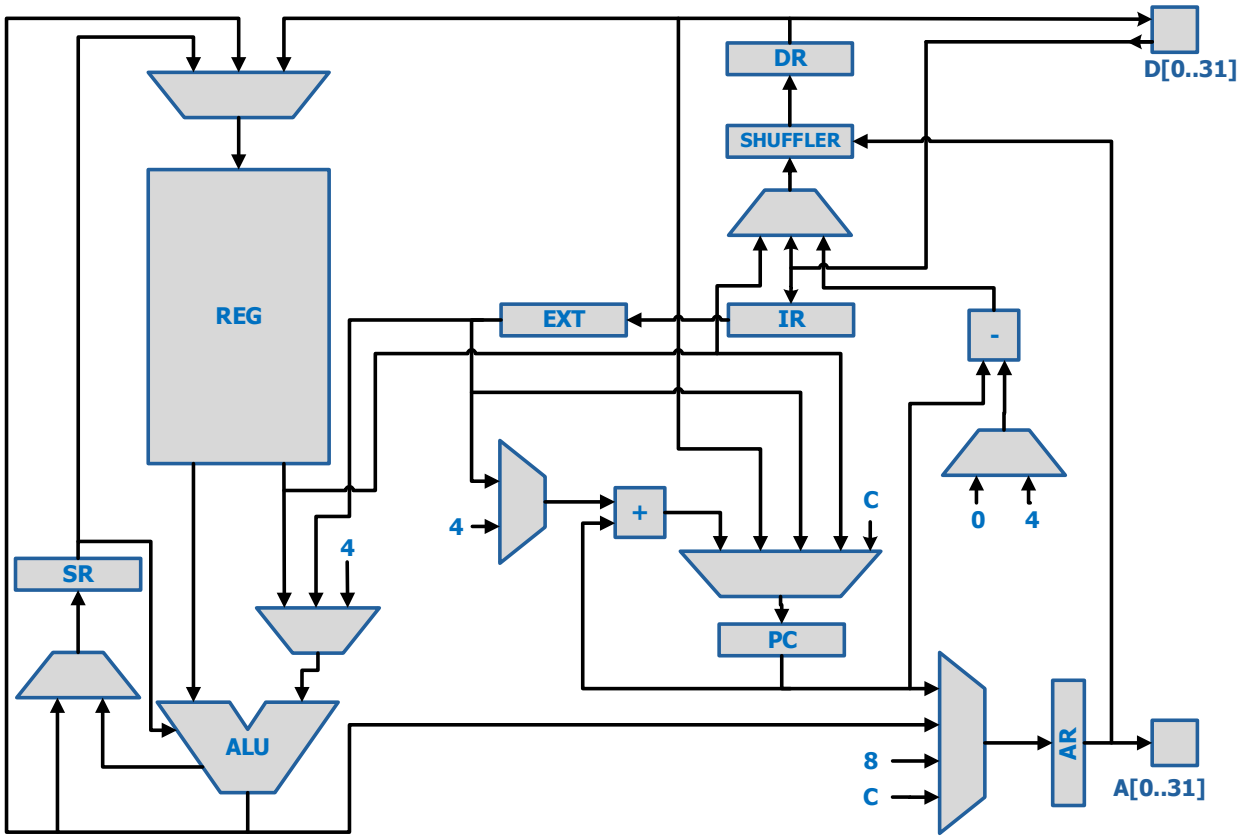
- Na kraju, dodajmo još jednu naredbu koja postoji u mnogim procesorima
- To je naredba za **zaustavljanje rada procesora** i naziva se **HALT**
- Budući da ova naredba prekida normalni slijed izvođenja, svrstat ćemo je u upravljačke (iako nije naredba skoka)
- Zbog pravilnosti će se i naredba HALT moći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

```
HALT_uvjet
HALT
```

Proširenje upravljačkih nar. - put podataka

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

Put podataka...



PC+4

JP adr

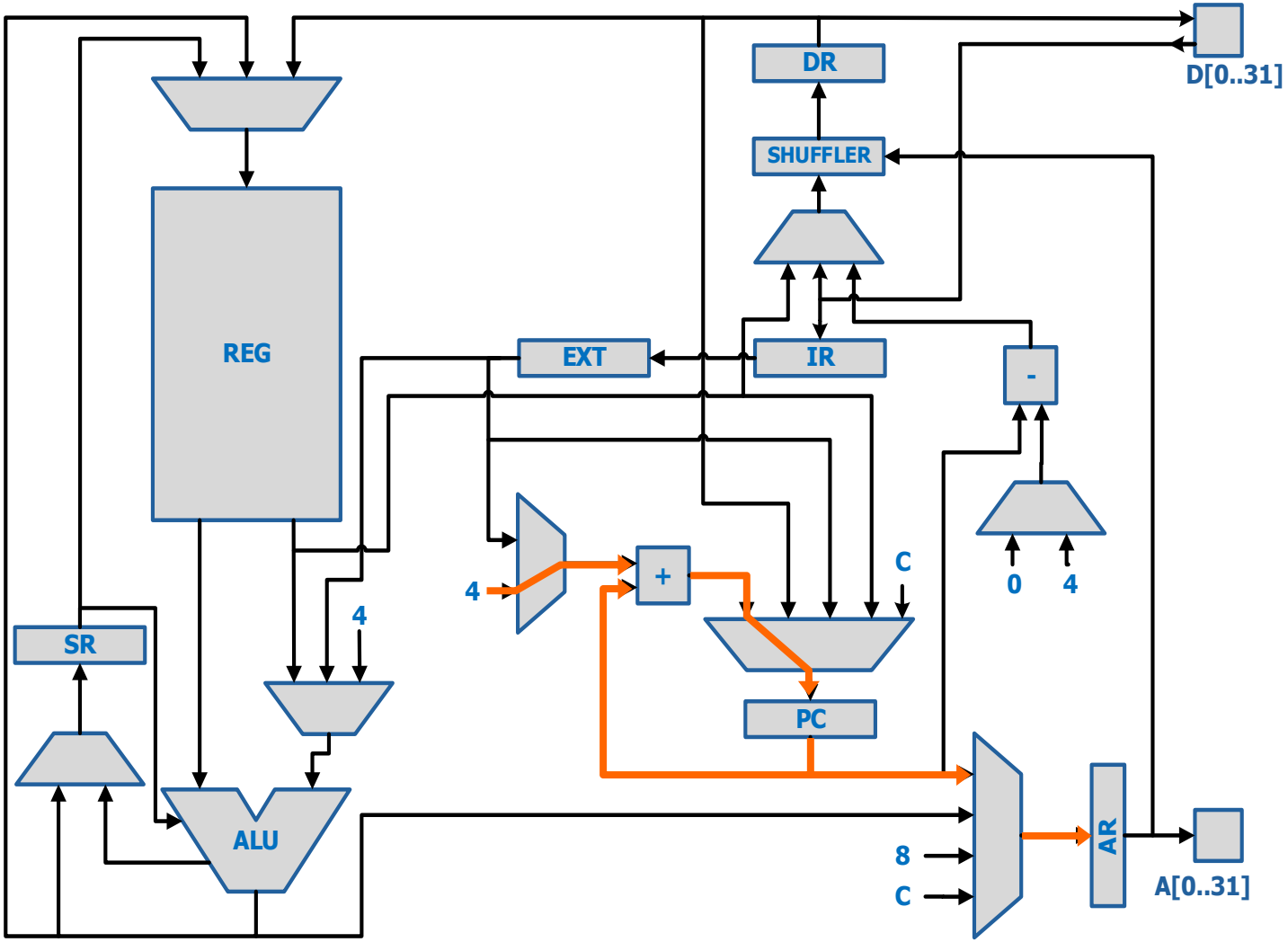
JP (reg)

JR

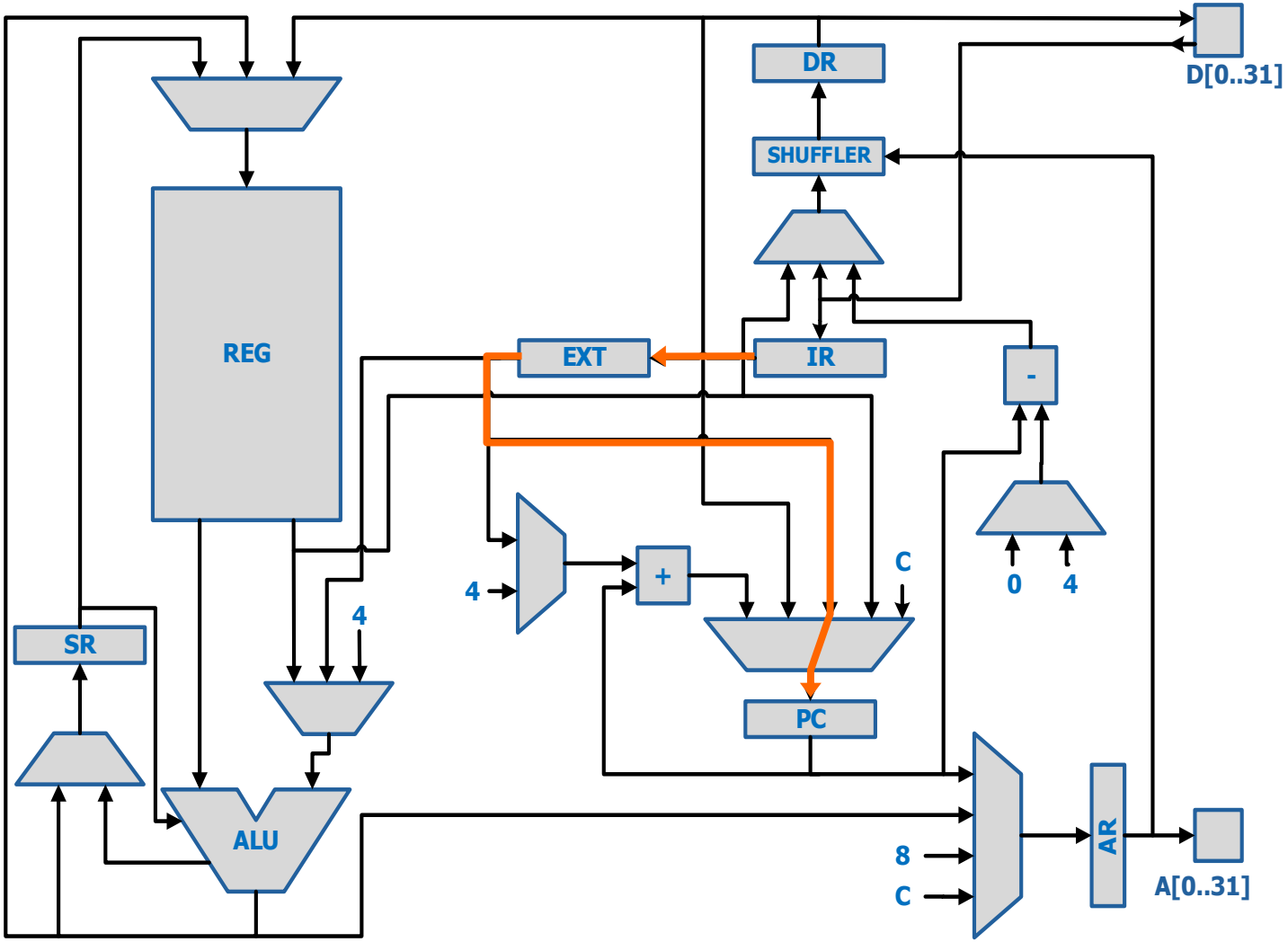
CALL

RET

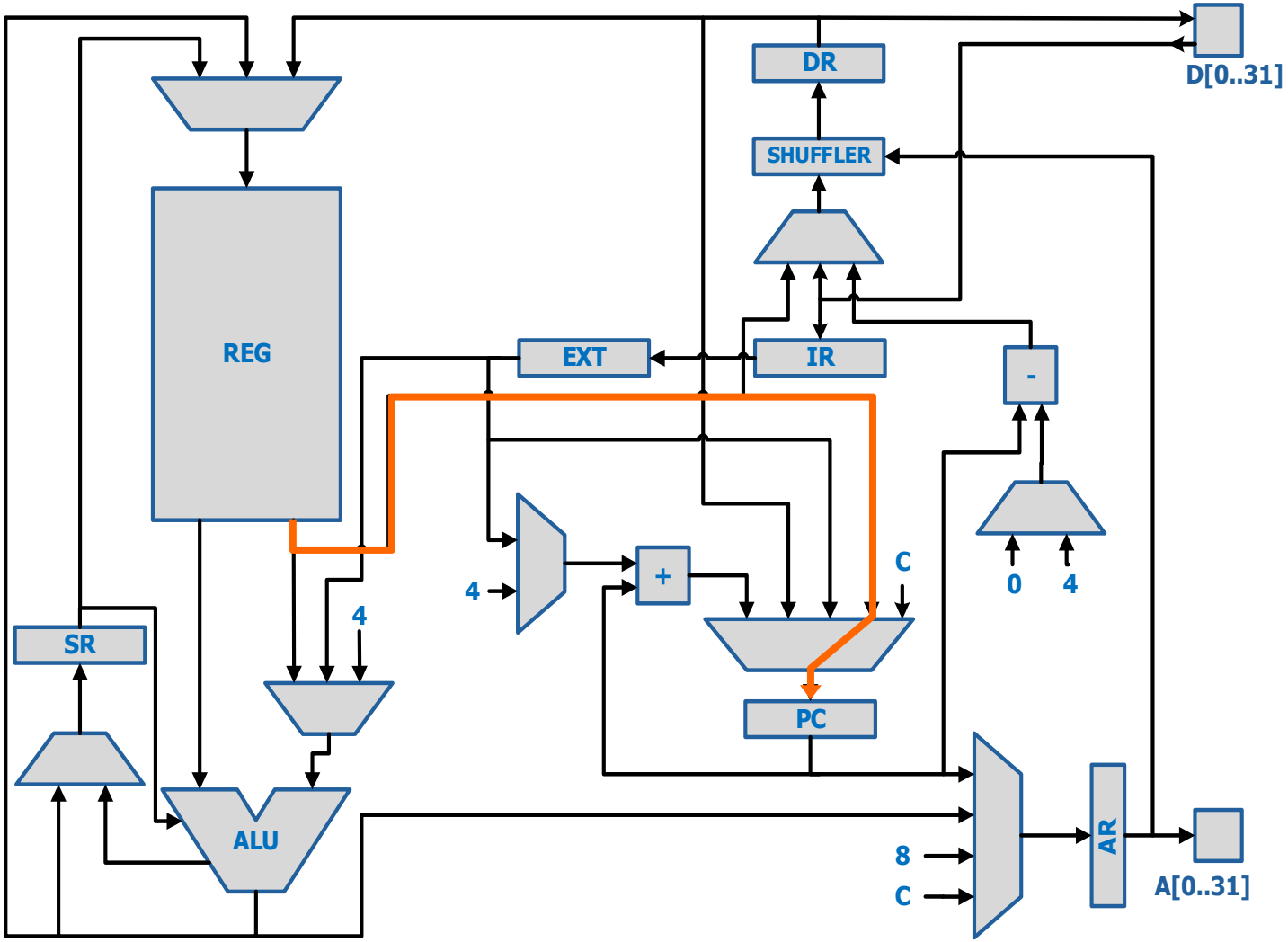
Put podataka (dohvat naredbe)



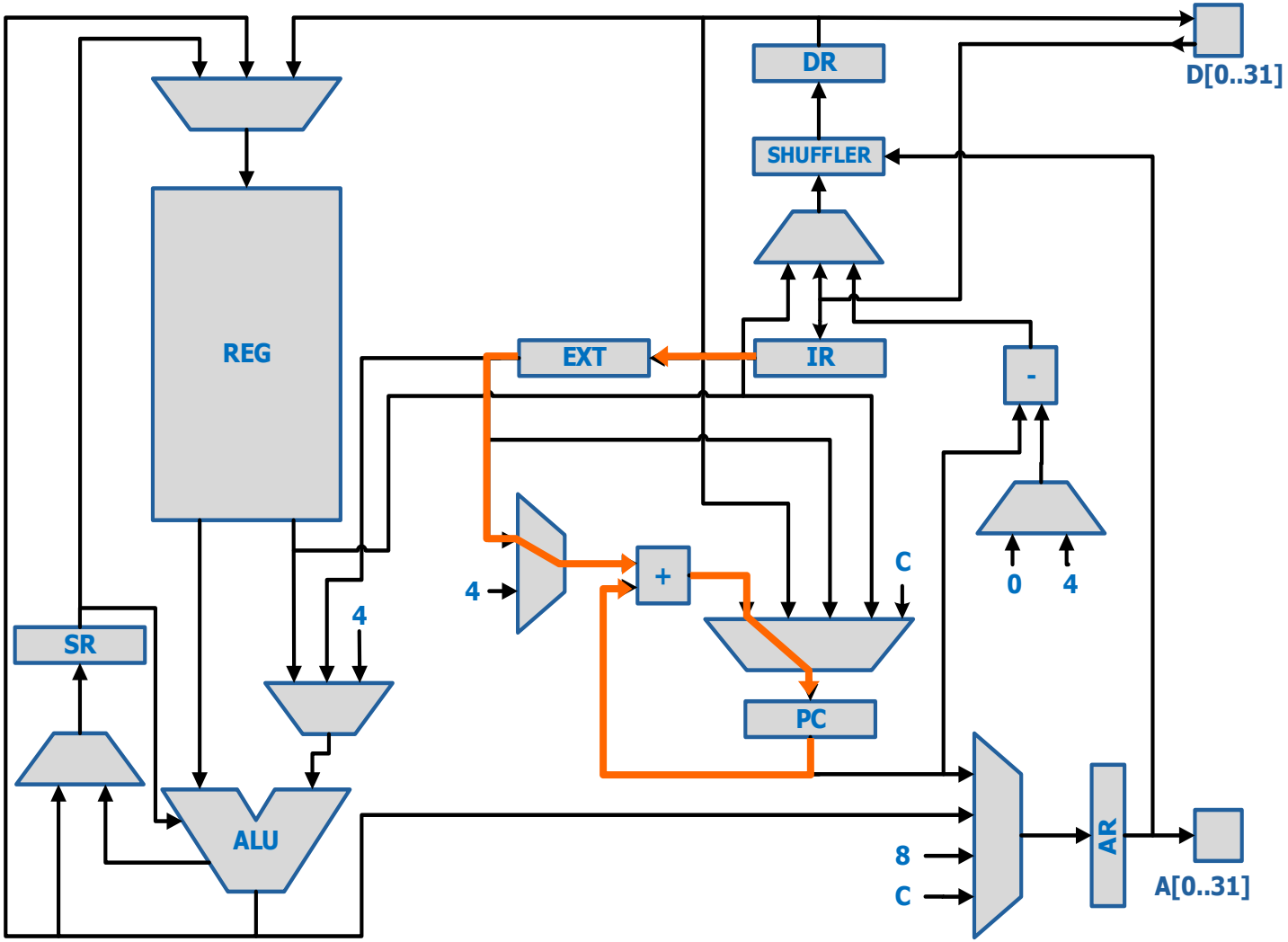
Put podataka (JP adr)



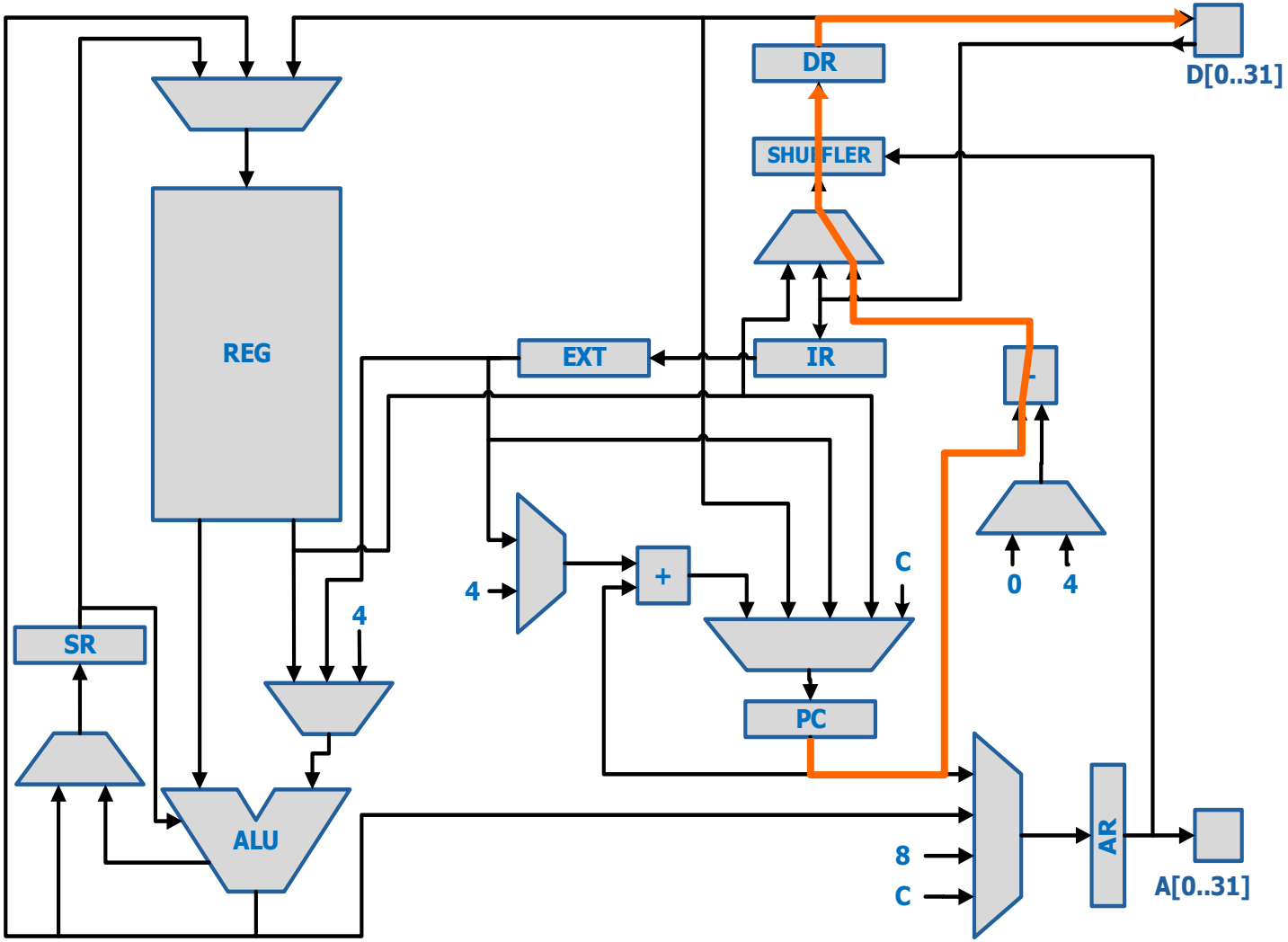
Put podataka (JP (Reg))



Put podataka (JR)



Put podataka (CALL, PC → stog)

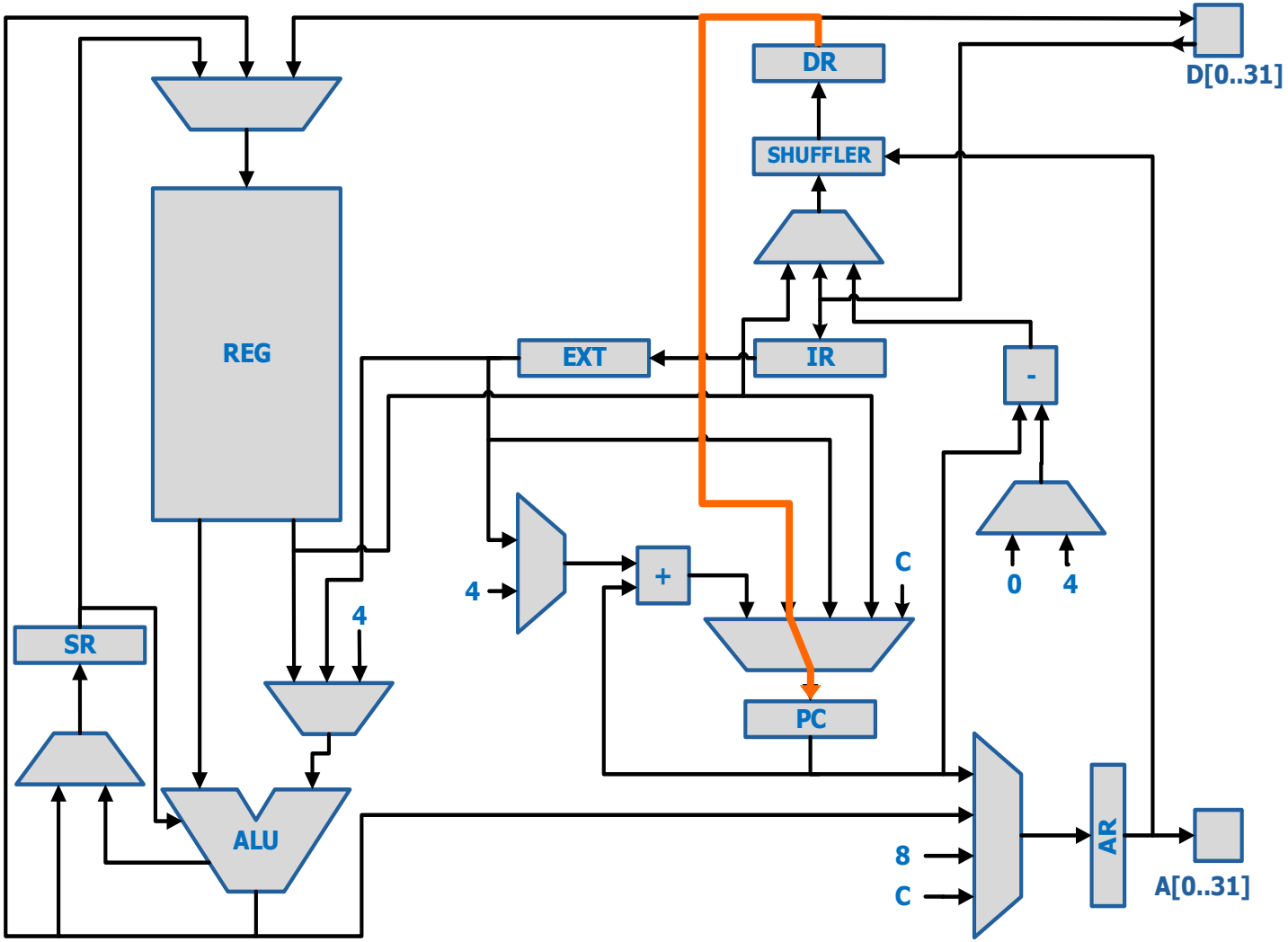


1) PC → stog

(Adresa je kao u PUSH)

2) PC se puni kao u JP

Put podataka (RET, stog → PC)



1) Stog → PC

(Adresa je kao u POP)

Put podataka procesora FRISC

- Ovime smo kompletirali naredbeni skup procesora FRISC i definirali dijelove puta podataka potrebne za izvođenje tih naredaba

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

- 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 prevođenja, taj podatak je zapisan u strojnom kôdu.
  - Podatak predstavlja broj s kojim se izvodi operacija, a ne adresu u memoriji
  - `MOVE  200, R3`
  - `ADD  R0, 12, R2`

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)`
  - `LOAD  R5, (R0)      // odmak je 0`

- Implicitno adresiranje (implied addressing)
  - iz same naredbe se zna gdje se nalazi podatak ili odredište skoka i sl.
  - `PUSH  R5      // iako nije eksplicitno zadano,`
    `// naredba radi sa stogom`