

LDR I STR

Proširenja...

Naredbe load i store - nastavak

- Procesor ARM posjeduje dva tipa naredaba load/store:
 - **Prvi tip** naredaba može učitati ili upisati
 - 32-bitnu riječ ili
 - bajt bez predznaka

- Takve naredbe imaju općeniti format:

LDR|STR{<cond>}{B} Rd, <mode_2>

- gdje pojedine oznake znače:
 - {<cond>} polje uvjeta
 - {B} adresiranje bajta
 - Rd odredišni registar
 - <mode_2> opis načina adresiranja 2

- **Drugi tip** može učitati ili upisati
 - 16-bitnu poluriječ bez predznaka
 - poluriječ ili bajt te ih predznačno proširiti do širine riječi.

- Ovakve naredbe imaju općeniti format:

LDR|STR{<cond>}H|SH|SB Rd, <mode_3>

- gdje pojedine oznake znače:
 - {<cond>} polje uvjeta
 - H|SH|SB adresiranje poluriječi (H), predznačene poluriječi (SH) ili predznačenog bajta (SB)
 - Rd odredišni registar
 - <mode_3> opis načina adresiranja 3

Load / store

Ime naredbe	Engleski naziv
LDR	Load Word
LDRB	Load Unsigned Byte (zero extend)
LDRH	Load Unsigned Halfword (zero extend)
LDRSB	Load Signed Byte (sign extend)
LDRSH	Load Signed Halfword (sign extend)
STR	Store Word
STRB	Store Byte
STRH	Store Halfword

Load/Store

- Kako u naredbi (koja ima 32bita) definirati s koje adrese želimo pročitati/zapisati podatak
- Procesor mora imati mogućnost adresirati bilo koji podatak u memoriji
- Ako bi koristili samo slobodne bitove u strojnom kodu naredbe to bi nam bilo premalo
- OSNOVNA IDEJA:
 - Za adresiranje će se koristiti adresa koja se nalazi spremljena u nekom registru opće namjene (tako da imamo svih 32 bita na raspolaganju)

Load/store: Bazni registar

- Za sve naredbe load/store se adresa memorije izračunava korištenjem dva dijela:
 - vrijednost u nekom baznom registru
 - odmak (offset) u odnosu na vrijednost u baznom registru
- Bazni registar može biti bilo koji registar opće namjene*

* Ako se kao bazni registar izabere PC, tada se može postići relativno adresiranje u odnosu na trenutačnu poziciju u programu, te na taj način i izvedba programa koji su potpuno neovisni o položaju u memoriji. Kod direktnog kodiranja s PC kao baznim registrom (bez pomoći asemblera i korištenja labela) programer mora paziti na vrijednost PC-a u trenutku izvođenja naredbe

ODMAK ?

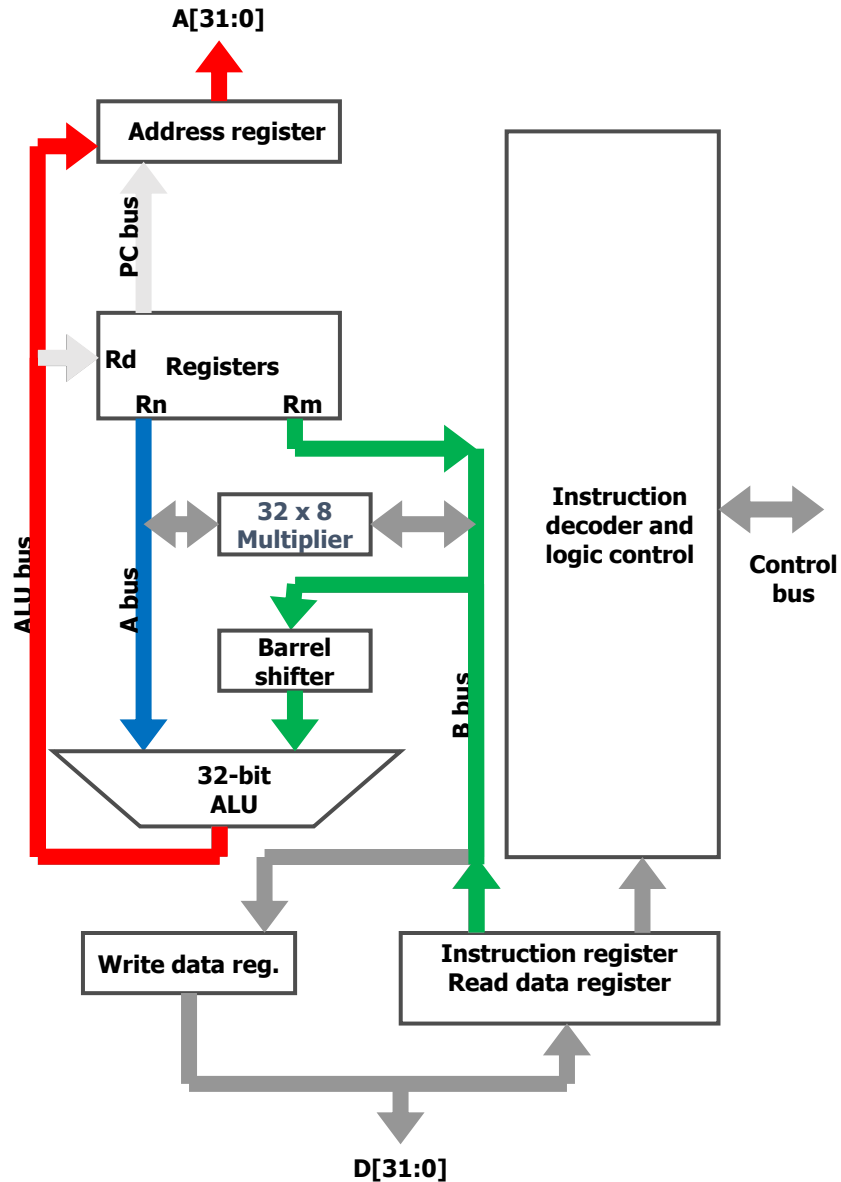
- Zašto za adresiranje koristiti bilo što osim samog baznog registra??
- Jedna od izvrsnih karakteristika Arm procesora
 - Izvedba ove dodatne mogućnosti ne košta ništa (logika već postoji zbog AL naredaba)
 - Poboljšava performanse
 - Efikasnije prevođenje nekih čestih programskih odsječaka

Load/store: Odmak (offset)

- Za pojedinu naredbu programer može izabrati jedan od tri formata odmaka:
 - U najjednostavnijem formatu odmak se definira kao **broj, odnosno neposredna vrijednost** (immediate) koji se izravno upisuje u kôd naredbe. Neposredna vrijednost zapisana je jednim bitom predznaka i iznosom odmaka od 12 ili 8 bitova (ovisno o naredbi). 12-bitnim odmakom se može adresirati memorijska lokacija odmaknuta za +/- 4095 mjesta, a 8-bitnim odmaknuta za +/- 255 mjesta u odnosu na vrijednost izabranog baznog registra
 - Odmak se može definirati i pomoću **vrijednosti iz registra opće namjene***
 - Treća mogućnost je definiranje odmaka pomoću **vrijednosti registra opće namjene* koja je još pomaknuta ulijevo ili udesno**

* osim registra PC

Podloga u mikroarhitekturi



Bazni registar

Odmak

Adresa

Load/store...

- Osim prethodno opisana tri načina za definiciju odmaka, procesor ARM omogućuje još tri različite inačice adresiranja memorije.
- Ove inačice zadaju treba li i kako mijenjati bazni registar tijekom izvođenja naredbe load/store. Ove tri inačice su:
- **Osnovni odmak**
 - Adresa se izračuna zbrajanjem ili oduzimanjem odmaka od baznog registra, a zatim se pristupi memoriji. **Vrijednost baznog registra ostaje nepromijenjena:**
$$\text{Reg} \Leftrightarrow \text{mem}[\text{BazniReg} + \text{Odmak}]$$

- **Predindeksiranje**

- Odmak se zbroji ili oduzme od baznog registra, a **izračunata vrijednost se spremi natrag u bazni registar**. Zatim se pristupi memoriji.

$\text{BazniReg} = \text{BazniReg} + \text{Odmak}$

$\text{Reg} \Leftrightarrow \text{mem}[\text{BazniReg}]$

- **Postindeksiranje**

- Memoriji se pristupi samo na temelju vrijednosti baznog registra. Tek nakon obavljenog prijenosa, odmak se zbroji ili oduzme od baznog registra, a **izračunata vrijednost se spremi natrag u bazni registar**.

$\text{Reg} \Leftrightarrow \text{mem}[\text{BazniReg}]$

$\text{BazniReg} = \text{BazniReg} + \text{Odmak}$

Load/store - Rekapitulacija

- Osnovni odmak:

$\text{Reg} \Leftarrow \text{mem}[\text{BazniReg} + \text{Odmak}]$

- Predindeksiranje:

$\text{BazniReg} = \text{BazniReg} + \text{Odmak}$

$\text{Reg} \Leftarrow \text{mem}[\text{BazniReg}]$

- Postindeksiranje:

$\text{Reg} \Leftarrow \text{mem}[\text{BazniReg}]$

$\text{BazniReg} = \text{BazniReg} + \text{Odmak}$

Pregled adresiranja u naredbama load/store

	Osnovni odmak	Predindeksiranje	Postindeksiranje
Neposredni	[R0, #4]	[R0, #4]!	[R0], #4
Registarski	[R7, -R3]	[R7, R3]!	[R7], R3
Registarski s pomakom	[R3, R5, LSL #2]	[R3,R5,LSL #2]!	[R3],R5,LSL #2

Primjer (8-bitno zbrajanje)

Treba napisati program za zbrajanje dva 8-bitna broja bez predznaka koji su zapisani u memoriji odmah iza programa. Rezultat treba spremiti u memoriju iza operanada.

;Program za 8 bitno zbrajanje dva broja bez predznaka, ver.1

```
LDRB R0, OP_A      ; prvi operand se učitava u R0 (za
                    ; signed se koristi LDRSB)

LDRB R1, OP_B      ; drugi operand se učitava u R1 (za
                    ; signed se koristi LDRSB)

ADDS R4, R0, R1    ; izvodi se zbrajanje i postavljaju se
                    ; zastavice (S)

STRB R4, REZ       ; rezultat se sprema u memoriju

SWI 123456
```

ORG 0x30

```
OP_A    DB 100      ; prvi operand (oktet)
OP_B    DB 20       ; drugi operand (oktet)
```

```
REZ     DB 1        ; rezultat
```

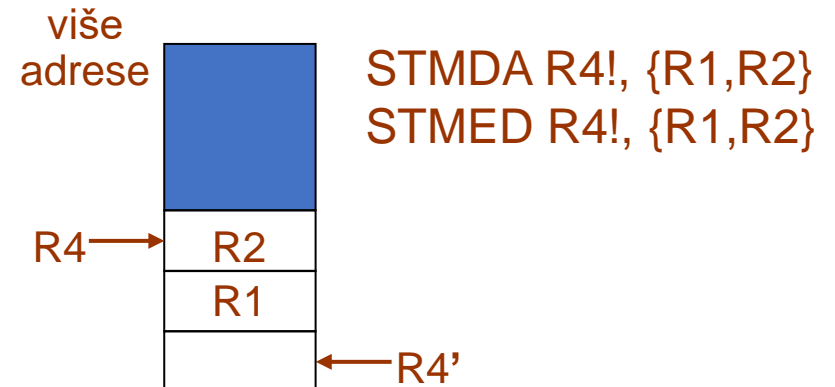
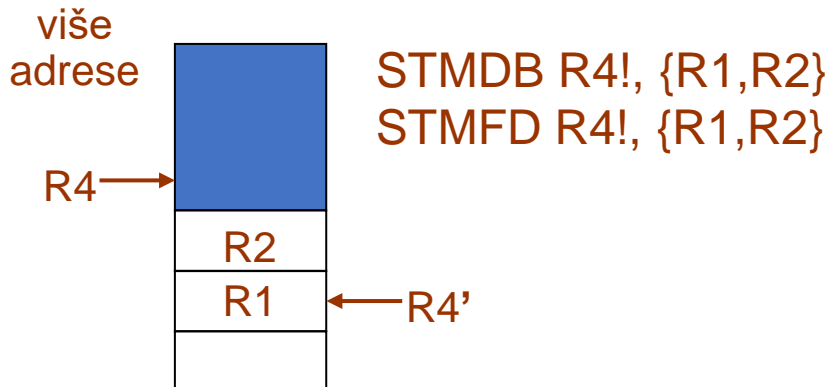
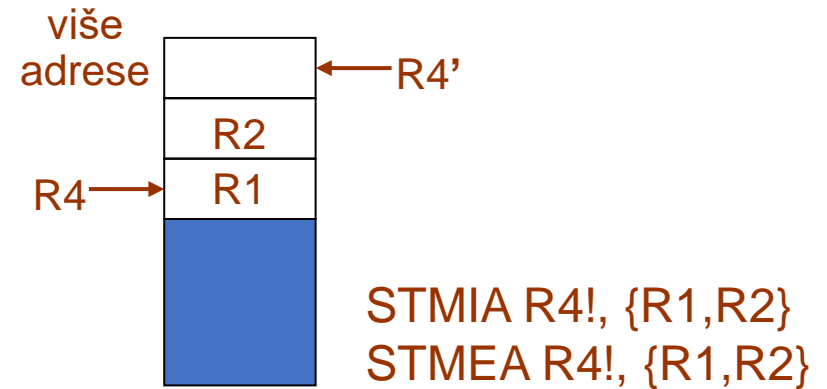
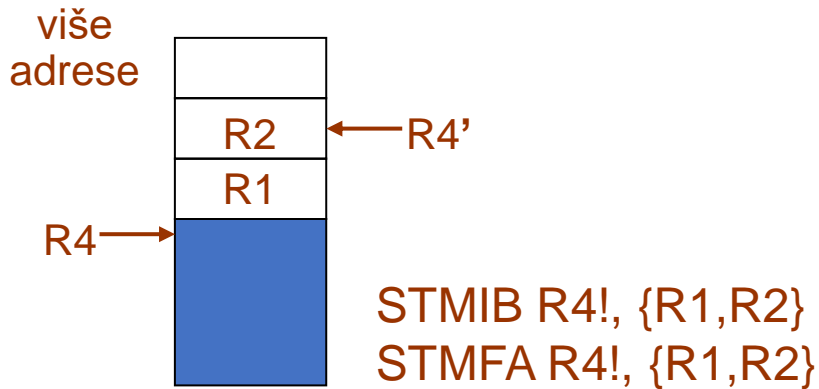
Naredbe Load Multiple i Store Multiple

- Pored osnovnih naredaba load i store, koje obavljaju prijenos podataka samo iz jednog registra, ARM ima dvije naredbe (Load Multiple i Store Multiple) koje programeru omogućuju da jednom naredbom obavi prijenos podataka između memorije i bilo kojeg podskupa registara ili čak svih registara.
- Osnovno ime naredbe je LDM i STM nakon čega se stavlja neki od nastavaka kojima se opisuje kako se sprema niz podataka (tj. niz registara) u memoriju

LDM/STM: načini adresiranja

- Pri pisanju (čitanju) više podataka u memoriju mora se definirati gdje će biti zapisan prvi te svaki sljedeći podatak.
- Da bi definirali gdje će se zapisati prvi podatak, moramo izabrati neki registar opće namjene koji pokazuje na početak memorijskog područja u koje će se upisivati podaci.
- Nakon toga moramo izabrati jednu od četiri kombinacije načina adresiranja niza: IB, IA, DB ili DA. Ove kratice znače:
 - IB - Increment Before (uvećaj prije)
 - IA - Increment After (uvećaj poslije)
 - DB - Decrement Before (smanji prije)
 - DA - Decrement After (smanji poslije)

LDM/STM: načini adresiranja



R4 = stanje prije naredbe

R4' = stanje poslije naredbe

Naredbe s normalnim nastavcima	Ekvivalentne naredbe za rad sa stogom
LDMIA	LDMFD
LDMIB	LDMED
LDMDA	LDMFA
LDMDB	LDMEA
STMIA	STMEA
STMIB	STMFA
STMDA	STMED
STMDB	STMFD

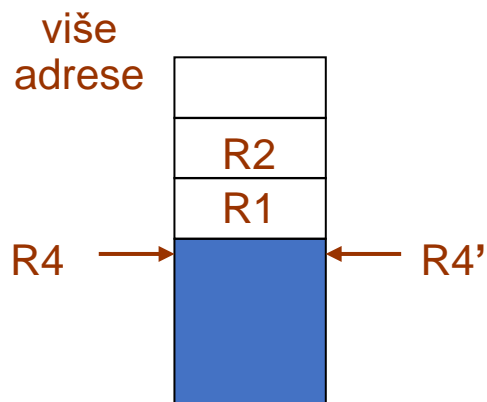
- Još jedna mogućnost koja se pruža programeru je automatsko pomicanje pokazivača, odnosno osvježavanje vrijednosti koja se nalazi u registru koji služi za adresiranje.
- Tako se na primjer naredbe

LDMIB R2, {R4,R8,R9}

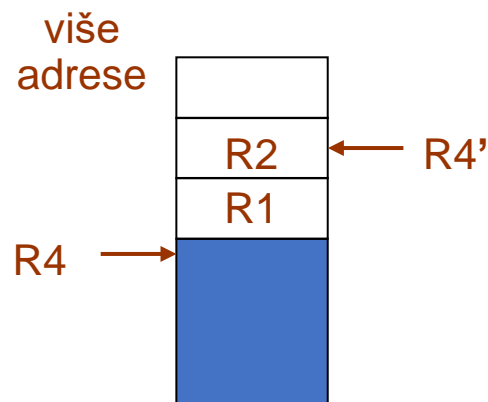
LDMIB R2!, {R4,R8,R9} ; iza R2 piše se uskličnik !

razlikuju po tome što će nakon izvođenja prve naredbe vrijednost registra R2 ostati **nepromijenjena**, dok će nakon druge naredbe registar R2 biti **promijenjen** (uvećan za 12_{10} , jer se R2 uvećava za 4 prije čitanja podatka za svaki registar iz niza)

LDM/STM korištenje



STMIB R4, {R1,R2}



STMIB R4!, {R1,R2}

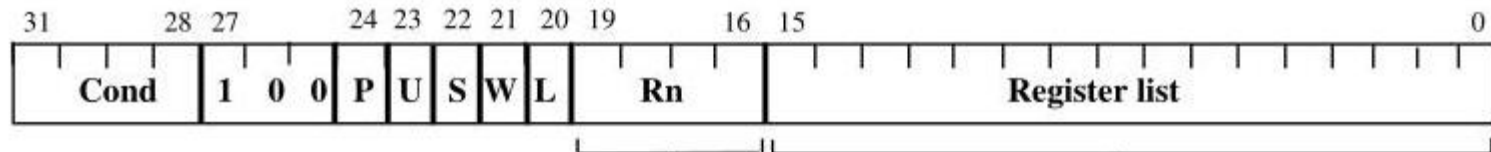
R4 = stanje prije naredbe

R4' = stanje poslije naredbe

LDM/STM PRAVILA!!!

- Bez obzira na izbor nekog od četiri adresiranja i bez obzira na redoslijed kojim su registri napisani u naredbi, **uvijek je na najnižoj adresi zapisan registar s najnižim brojem**
 - **LDMIA R4, {R1,R8,R3,R12}**
 - **LDMIA R4, {R12,R3,R8,R1}**
 - **LDMIA R4, {R1,R3,R8,R12}**
- **Bilo koja od ovih gore naredaba (ili neki drugi redoslijed) DAT ĆE ISTI KÔD NAREDBE I ISTI REZULTAT**

LDM/STM ... popis registara



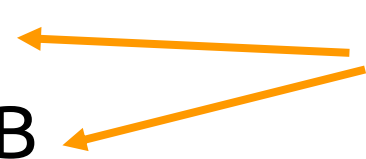
Base register

Each bit corresponds to a particular register. For example:

- Bit 0 set causes r0 to be transferred.
- Bit 0 unset causes r0 not to be transferred.

At least one register must be transferred as the list cannot be empty.

LDM/STM PRAVILA!!!

- Ako se podaci žele zapisati u memoriju naredbom STM, a zatim pročitati u istom redoslijedu naredbom LDM te ako se koriste nastavci za OBIČNO ADRESIRANJE (ne ekvivalenti za stog!!!), tada način adresiranja u naredbi LDM mora biti **INVERZAN** načinu adresiranja u naredbi STM
 - Primjer:
 - Zapisivanje STMIA
 - Čitanje LDMDB
- U naredbama se koristi inverzno adresiranje
IA ↔ DB
- 

LDM/STM PRAVILA!!!

- Ako se podaci žele zapisati u memoriju naredbom STM, a zatim pročitati u istom redoslijedu naredbom LDM te ako se koriste nastavci za RAD SA STOGOM, tada način adresiranja u naredbi LDM mora biti **ISTI** načinu adresiranja u naredbi STM

- Primjer:

- Zapisivanje
- Čitanje

STMFD
LDMFD

U obje naredbe
koristi se jednako
adresiranje FD

Primjeri korištenja LDM i STM ...



U donjim primjerima pretpostavljene vrijednosti prije izvođenja naredaba LDM/STM su:

r0=0 r1=1 r2=2 r3=3 r13=9000

a) STMIB r13, {r0,r1,r2,r3} ; vrijednosti se sprema, r13 ostane nepromijenjen

Nakon izvođenja gornje naredbe, stanje u memoriji je (heksadekadski, little-endian):

0x00009000: 10 00 FF E7 00 00 00 00 01 00 00 00 02 00 00 00

0x00009010: 03 00 00 00 00 E8 00 E8 10 00 FF E7 00 E8 00 E8

r13=9000

b) STMIB r13, {r3,r1,r0,r2} ; redoslijed pisanja registara ne utječe na redoslijed spremanja

0x00009000: 10 00 FF E7 00 00 00 00 01 00 00 00 02 00 00 00

0x00009010: 03 00 00 00 00 E8 00 E8 10 00 FF E7 00 E8 00 E8

r13=9000

c) STMIB r13!, {r0,r3} ; osvježava se r13

0x00009000: 10 00 FF E7 00 00 00 00 03 00 00 00 E8 00 E8 00

r13=9008

... Primjeri korištenja LDM i STM ...



d) **STMDB r13!, {r0,r1,r2,r3}** ; primjer korištenja DB

0x00008FF0: 00 00 00 00 01 00 00 00 02 00 00 00 03 00 00 00

0x00009000: 10 00 FF E7 00 E8 00 E8 10 00 FF E7 00 E8 00 E8

r13 = 8FF0

e) **STMDA r13!, {r0,r1,r2,r3}** ; primjer korištenja DA i kasnijeg obnavljanja registara

;0x00008FF0 00 E8 00 E8 00 00 00 00 01 00 00 00 02 00 00 00

;0x00009000 03 00 00 00 00 E8 00 E8 10 00 FF E7 00 E8 00 E8

;r13 = 8FF0

Da bi u registre r0, r1, r2 i r3 učitali iste podatke koje smo iz njih spremili u memoriju pomoću naredbe STMDA, moramo upotrijebiti naredbu LDM s inverznim nastavkom IB:

LDMIB r13!, {r0,r1,r2,r3}



- f) **STMFA r13!, {r0,r1,r2,r3}** ; primjer spremanja registara pomoću STM i kasnijeg
; obnavljanja registara naredbom LDM uz korištenje
; nastavaka za STOG

0x00009000: 00 E8 00 E8 00 00 00 00 01 00 00 00 02 00 00 00

0x00009010: 03 00 00 00 00 E8 00 E8 10 00 FF E7 00 E8 00 E8

r13 = 9010

Da bi u registre r0, r1, r2 i r3 učitali iste podatke koje smo iz njih spremili u memoriju pomoću naredbe STMFA, moramo upotrijebiti naredbu LDM s istim nastavkom FA:

LDMFA r13!, {r0,r1,r2,r3} ; za adresiranje stoga nastavci moraju biti isti

Načini adresiranja

- Procesor ARM omogućuje različita adresiranja. Oblici adresiranja svrstani su u nekoliko načina (engl. Addressing Modes) i u tablicama u Prilogu su označeni brojevima 1 do 4

Načini adresiranja 1

- Načini adresiranja 1 su adresiranja u “širem smislu” jer se u njima ne dohvaća podatak iz memorije (ne koristi se adresa)
- Koriste se za **definiranje jednog od operandada u naredbama za obradu podataka**. Osnovni oblik naredaba koje koriste taj način adresiranja izgleda ovako:

`<opcode>{<cond>}{S} <Rd>, <Rn>, <Operand2>`

- Drugi operand `<Operand2>` može se “adresirati” na 11 različitih načina koji tvore načine adresiranja 1

Načini adresiranja 1

- Već smo prije naučili da drugi operand može biti
 - Neposredna vrijednost
 - Vrijednost iz registra
 - Vrijednost iz registra s pomakom
- Sve ove mogućnosti zasnivaju se na arhitekturi puta podataka prema ALU

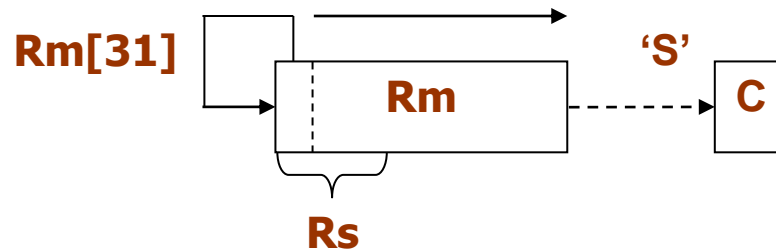
Načini adresiranja 1

Adresiranje	Sintaksa adresiranja
Neposredno	#<immediate>
Registarsko	<Rm>
Registarsko s neposrednim logičkim pomakom ulijevo (ASL je sinonim za LSL)	<Rm>,LSL #<shift_imm> <Rm>,ASL #<shift_imm>
Registarsko s registarskim logičkim pomakom ulijevo (ASL je sinonim za LSL)	<Rm>,LSL <Rs> <Rm>,ASL <Rs>
Registarsko s neposrednim logičkim pomakom udesno	<Rm>,LSR #<shift_imm>
Registarsko s registarskim logičkim pomakom udesno	<Rm>,LSR <Rs>
Registarsko s neposrednim aritmetičkim pomakom udesno	<Rm>,ASR #<shift_imm>
Registarsko s registarskim aritmetičkim pomakom udesno	<Rm>,ASR <Rs>
Registarsko s neposrednim rotiranjem udesno	<Rm>,ROR #<shift_imm>
Registarsko s registarskim rotiranjem udesno	<Rm>,ROR <Rs>
Registarsko s proširenim rotiranjem udesno	<Rm>,RRX

- Načine pisanja neposrednih vrijednosti smo ranije objasnili
- Korištenje registra kao drugog operanda ne treba posebno objašnjavati
- Proučimo koje mogućnosti postoje ako se za drugi operand izabere registar s pomakom
- Napomena: način izračuna vrijednost registra s pomakom prikazana je u šalabahteru radi vašeg lakšeg snalaženja

Registarsko i registarsko s pomakom

- Ako se nastavak 'S' doda naredbi koja inače ne mijenja zastavicu C, tada se C puni bitom označenim strelicom
 - Primjer: ... <Rm>, ASR <Rs>



- Registar **Rm** se ne mijenja, nego njegova pomaknuta vrijednost služi samo kao operand
- Iznos pomaka zadaje se registrom ili neposredno (brojem)
- Ostale pomake i rotiranja proučite iz tablice u Prilogu

Načini adresiranja 1

- Pogledajte tablicu za npr. naredbu MOV:

MOV{cond}{S} Rd, <Operand2>

- U tablici “Operand2” izaberete neki od načina adresiranja
- Napomena!!! **Ne postoji** naredba npr. LSL R0, #2 (dešava se na ispitima). Treba koristiti naredbu MOV (u ovom primjeru: MOV R0, R0, LSL #2).

Načini adresiranja 2,3,4

- Načini adresiranja 2, 3 i 4 koriste se u pojedinim naredbama Load i Store:
 - načini adresiranja 2: naredbe Load i Store word ili unsigned byte
 - načini adresiranja 3: naredbe Load i Store halfword ili signed byte
 - načini adresiranja 4: naredbe Load i Store Multiple

Načini adresiranja 2

- Karakteristični su za naredbe load i store kojima se premješta riječ ili nepredznačeni bajt
- Postoji devet mogućih kombinacija adresiranja s obzirom na vrstu odmaka te vrstu indeksiranja
- Odmak se zadaje neposrednim podatkom, registrom ili pomaknutim registrom*. Ako je izabran pomak registra, tada se vrsta pomaka (LSL, LSR, ASR, ROR, RRX) zadaje kao u načinu adresiranja 1, no uz ograničenje da se iznos pomaka zadaje samo neposredno

* Odmak se ne mora napisati i tada se podrazumijeva da je 0 (zadan neposredno)

Načini adresiranja 2

Adresiranje	Sintaksa adresiranja
Neposredni odmak	[<Rn>, #+/-<offset_12>]
Registarski odmak	[<Rn>, +/-<Rm>]
Registarski odmak s pomakom	[<Rn>, +/-<Rm>, LSL #<shift_imm>] [<Rn>, +/-<Rm>, LSR #<shift_imm>] [<Rn>, +/-<Rm>, ASR #<shift_imm>] [<Rn>, +/-<Rm>, ROR #<shift_imm>] [<Rn>, +/-<Rm>, RRX]
Neposredni predindeksirani	[<Rn>, #+/-<offset_12>]!
Registarski predindeksirani	[<Rn>, +/-<Rm>]!
Registarski predindeksirani odmak s pomakom	[<Rn>, +/-<Rm>, LSL #<shift_imm>]! [<Rn>, +/-<Rm>, LSR #<shift_imm>]! [<Rn>, +/-<Rm>, ASR #<shift_imm>]! [<Rn>, +/-<Rm>, ROR #<shift_imm>]! [<Rn>, +/-<Rm>, RRX]!
Neposredni postindeksirani	[<Rn>], #+/-<offset_12>
Registarski postindeksirani	[<Rn>], +/-<Rm>
Registarski postindeksirani odmak s pomakom	[<Rn>], +/-<Rm>, LSL #<shift_imm> [<Rn>], +/-<Rm>, LSR #<shift_imm> [<Rn>], +/-<Rm>, ASR #<shift_imm> [<Rn>], +/-<Rm>, ROR #<shift_imm> [<Rn>], +/-<Rm>, RRX

Načini adresiranja 2

- Pogledajte tablicu za npr. naredbu LDR:

LDR{cond} Rd, <mode2>

- U tablici “Način adresiranja 2” izaberete neki od načina adresiranja i upotrijebite ga u naredbi. Na primjer:

LDRHI R0, [R3, -R5, ROR #22]!

Načini adresiranja 3

- Šest formata adresa koje se koriste kod ostalih naredaba load i store (za poluriječ i predznačeni bajt)
- Ovih šest formata adresa slični su formatima iz načina adresiranja 2. Razlike su:
 - odmak se može zadati neposrednom vrijednošću ili registrom (ne može se zadati registar s pomakom)
 - neposredna vrijednost odmaka se zapisuje samo sa osam bitova

Načini adresiranja 3

Adresiranje	Sintaksa adresiranja
Neposredni odmak	[<Rn>, #+/-<offset_8>]
Registarski odmak	[<Rn>, +/-<Rm>]
Neposredni preindeksirani odmak	[<Rn>, #+/-<offset_8>]!
Registarski preindeksirani odmak	[<Rn>, +/-<Rm>]!
Neposredni postindeksirani odmak	[<Rn>], #+/-<offset_8>
Registarski postindeksirani odmak	[<Rn>], +/-<Rm>

Načini adresiranja 3

- Pogledajte tablicu za npr. naredbu LDR:

LDR{cond}SB Rd, <mode3>

- u tablici “Način adresiranja 3” izaberete neki od načina adresiranja i upotrijebite ga u naredbi. Na primjer:

LDRGTSB R0, [R3], #20

Načini adresiranja 4

- Naredbama load i store multiple moguće je pročitati ili upisati sadržaj jednog, sadržaj više ili čak sadržaje svih registara u memoriju, odnosno u niz uzastopnih memorijskih lokacija.
- Registar s najmanjim brojem je uvijek zapisan na najmanju memorijsku adresu, a registar s najvećim brojem na najveću.
- Na isti način, kod čitanja: u registar s najmanjim brojem učitava se podatak s najmanje adrese, a u registar s najvećim brojem učitava se podatak s najveće adrese.
- IA, IB, DA, DB (FD, FA, ED, EA)

Načini adresiranja 4

- Pogledajte tablicu za npr. naredbu LDM:

LDM{cond} <a_mode4L> Rd{!}, <reglist-pc>

- U tablici “Način adresiranja <a_mode4L> izaberete neki od načina adresiranja i upotrijebite ga u naredbi. Npr.

LDMGTIA R0!, {R4,R5,R6}

- Analogno vrijedi za naredbe STM i tablicu “Način adresiranja <a_mode4S>.
- Napomena: <reglist-pc> u opisu naredbe znači da se u popis registara ne smije staviti PC

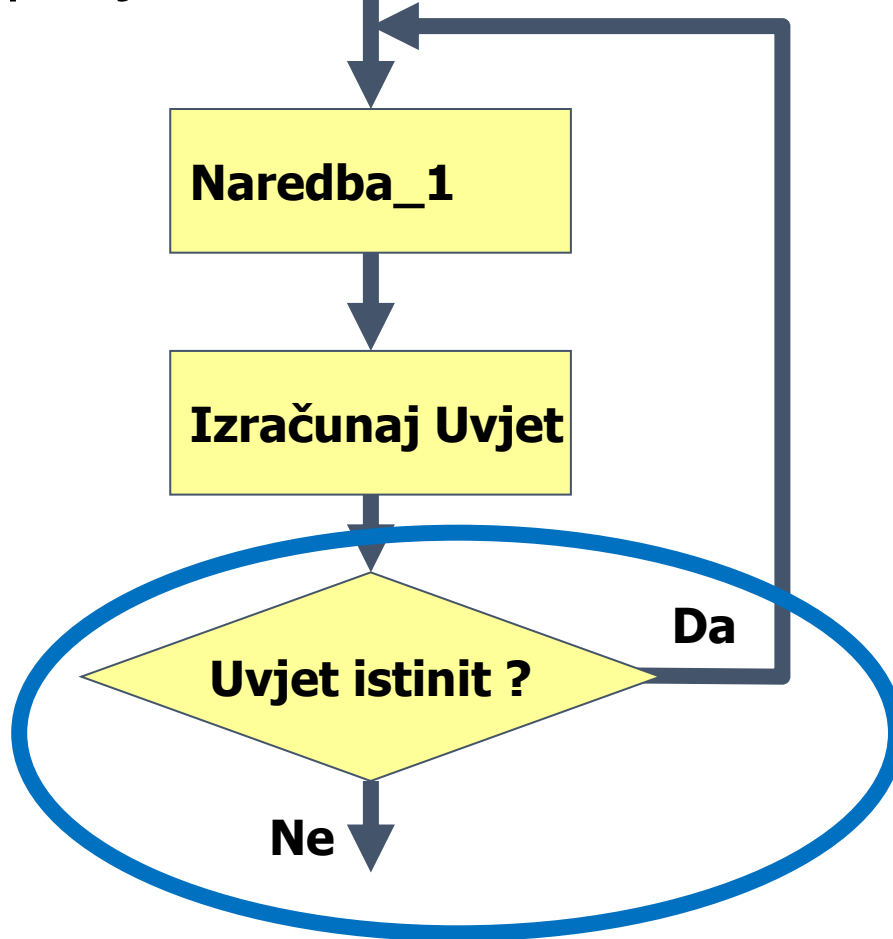
Upravljačke naredbe

- Za sada imamo:
 - Aritmetičko-logičke naredbe (ADD, SUB, AND, ...)
 - Memorijske naredbe (LDR, STR)
- 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:
 $(a \text{ OR } 00001111) \text{ XOR } (b \text{ AND } 00111100)$
 - Izračunavanje logičkih izraza: $a \text{ AND } b \text{ XOR true}$
 - Sve kombinacije gore navedenih izraza gdje se naredbe izvode **sljedno** jedna iza druge

Upravljačke naredbe

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

do
Naredba_1
While (Uvjet)



**kako
ostvariti
uvjetni
skok**

?

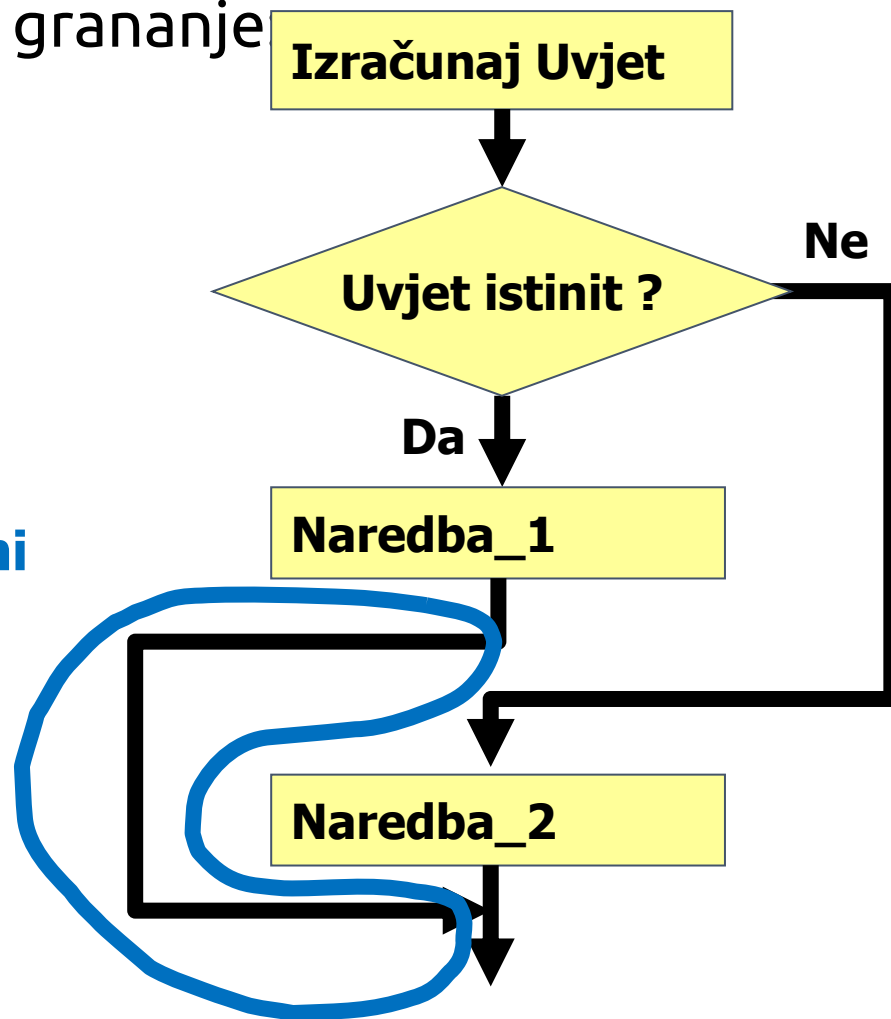
Upravljačke naredbe

- **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

Naredba B (Branch)

B adresa

Bxx adresa

- bezuvjetno grananje (skok) ili uvjetno grananje (xx = uvjet) na memorijsku adresu
- Adresa se mora nalaziti 32 MB ispred ili iza trenutne naredbe (relativan skok u odnosu na sadržaj PC)
- Drugi način grananja je da se u registar PC izravno stavi neka vrijednost, čime se skok ne ograničava na udaljenost od 32 MB, već se može skočiti na bilo koju adresu u adresnom području (tj. može se skočiti bilo gdje unutar adresnog prostora)

Naredba B (Branch)

- 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)

Upravljačke naredbe

- Promotrimo li malo bolje, vidimo da je naredba bezuvjetnog skoka samo **specijalni slučaj** uvjetnog skoka pri čemu je uvjet uvijek 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)
- Kako se mogu usporediti dvije vrijednosti:
 1. prvo se izvede AL-operacija (najčešće je to oduzimanje)
 2. nakon toga se ispituju zastavice (Podsjetnik: zastavice su bistabili koji se postavljaju na temelju ALU-operacije)

Upravljačke naredbe - uvjeti

- Napravimo popis svih uvjeta koji bi nam mogli trebati u naredbi B:
 - 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



Naredbe grananja - uvjeti

Mnemonički naziv	Puni naziv (engleski)	Ispitivano stanje zastavica
EQ	Equal	Z
NE	Not equal	!Z
CS/HS	Carry set/unsigned higher or same	C
CC/LO	Carry clear/unsigned lower	!C
MI	Minus/negative	N
PL	Plus/positive or zero	!N
VS	Overflow	V
VC	No overflow	!V
HI	Unsigned higher	C and !Z
LS	Unsigned lower or same	!C or Z
GE	Signed greater than or equal	N == V
LT	Signed less than	N != V
GT	Signed greater than	!Z and N == V
LE	Signed less than or equal	Z or N != V
AL	Always (unconditional)	-
(NV)	See Condition code 0b1111	-

Primjeri naredaba za grananje

B	labela	; bezuvjetni skok
BCC	labela	; uvjetni skok (carry clear)
BEQ	labela	; uvjetni skok (equal)
MOV	PC, #0	; R15 = 0, tj. skoči na adresu 0
MOV	PC, R3	; R15 = R3, skok na bilo koju 32-bitnu adresu ; koja je zadana registrom R3

Primjer uvjetnog grananja - naredba if:

Zbrojiti registre R0 i R1 i rezultat staviti u R2. Ako nema prijenosa, onda ne treba napraviti ništa. Ako ima prijenosa (tj. zbroj u NBC-u je izašao iz opsega), onda treba pobrisati R2.

```
1)  R2 = R0 + R1;  
     if( R2 nije ispravan) {  
         R2 = 0;  
     }
```

```
3)  R2 = R0 + R1;  
     if( R2 je ispravan)  
         goto dalje;  
     R2 = 0;  
     dalje: ...
```

```
2)  R2 = R0 + R1;  
     if( R2 je ispravan) {  
     } else {  
         R2 = 0;  
     }
```

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.

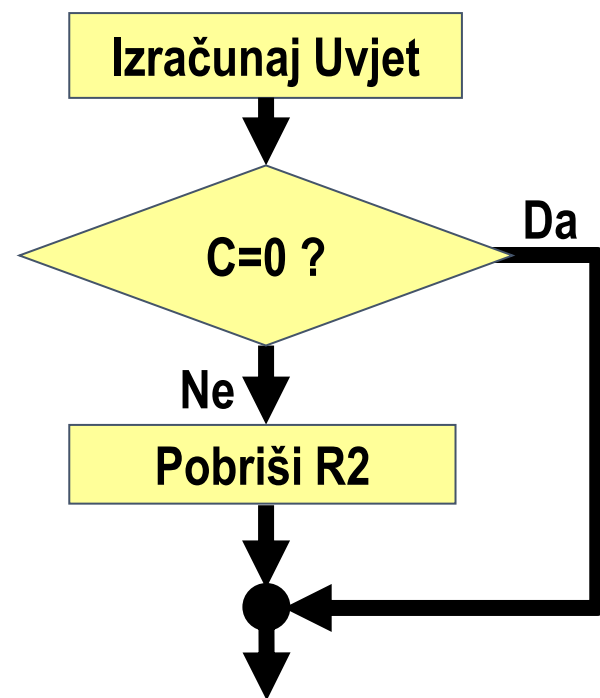
Rješenje:

```
ADDS R2,R0,R1 ; AL-operacija
```

```
BCC DALJE ; ispitivanje  
           ; zastavica  
           ; i skok
```

```
SUB R2,R2,R2 ; briši R2
```

```
DALJE ... ; nastavak  
          ; programa
```



Upravljačke naredbe - primjeri

Svaki uvjet može se napisati i na "obrnut" način. U praksi uvjet "okrećemo" tako da dobijemo ili kraći ili nama razumljiviji program ili oboje 😊.

- Prethodni program napisan s "obrnutim" uvjetom izgledao bi ovako:

ADDS R2,R0,R1

BCS BRISI

B DALJE

BRISI SUB R2,R2,R2

DALJE ...

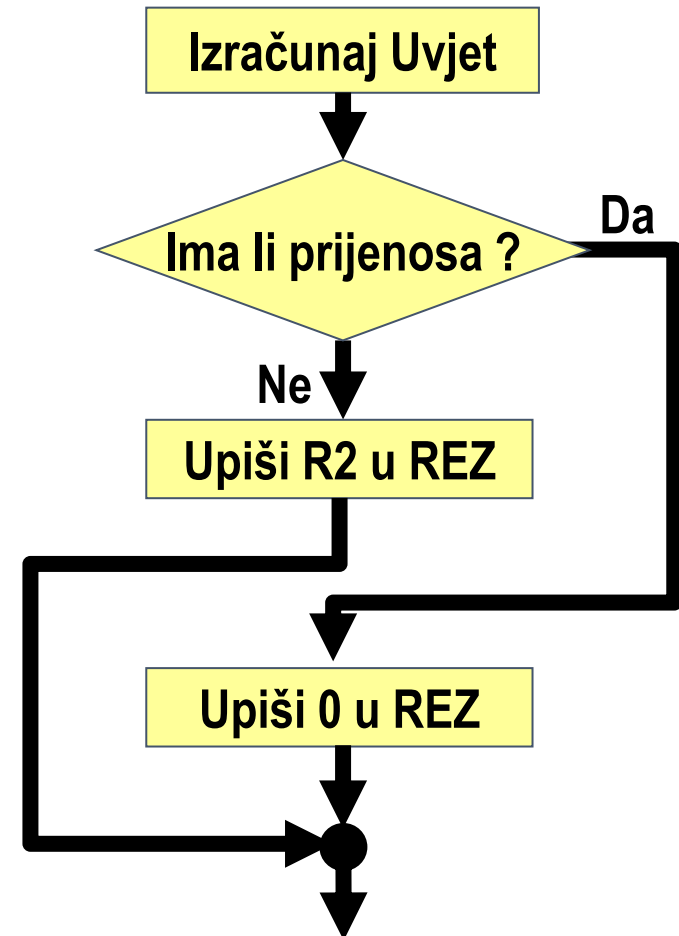
Upravljačke naredbe - primjeri

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

```
ADDS R2,R0,R1
BCS BRISI
PISI STR R2,REZ      ; upiši R2 u REZ
B DALJE

BRISI SUB R3,R3,R3    ; obriši
STR R3,REZ
DALJE ...

REZ DW 0
```



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

Dodatno ćemo vidjeti i petlju s brojačem.

```

LDR R0, 100      ; R0 = A
LDR R1, 200      ; R1 = B
ORR R2, R0, R0   ; R2 = Brojac := A
MOV R3, #0       ; R3 = Umnozak := 0
EOR R4, R4, R4   ; R4 = 0

```

```

PETLJA  CMP R2, #0      ;ili npr SUBS R2,R2,#0
        BEQ  KRAJ

```

```

        ADD R3, R3, R1
        SUB R2, R2, #1
        B   PETLJA

```

```

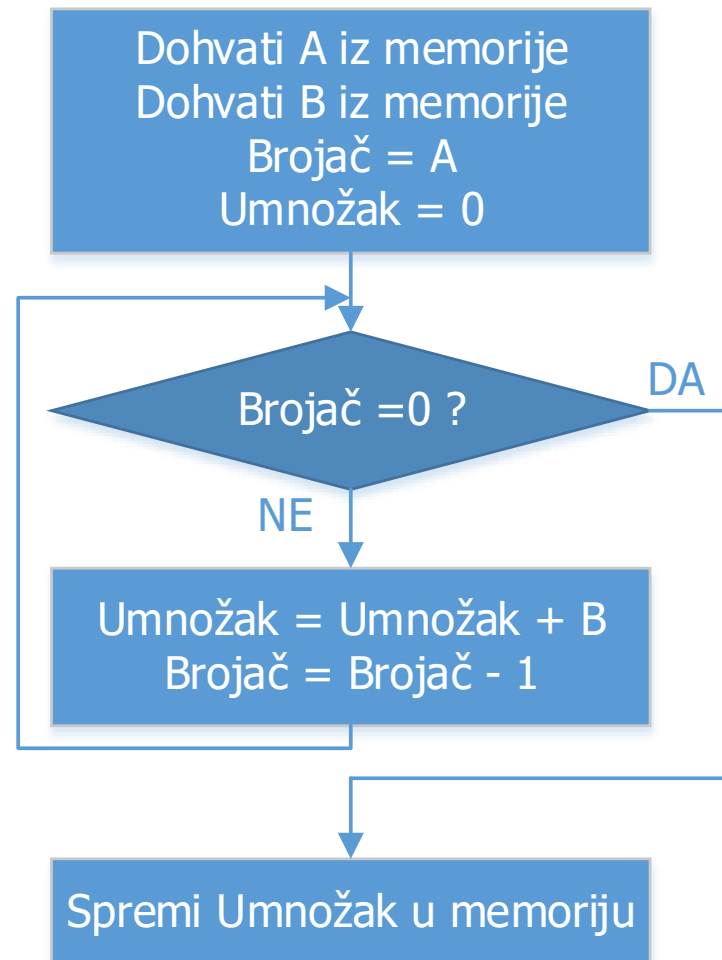
KRAJ    STR R3, 300

```

```

ORG 100
DW 5
ORG 200
DW 4

```



Uspoređivanje brojeva

- Arm 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 usporede 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" operandada 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 HS (unsigned Higher or Same)
- U naredbi CMP pišemo brojeve u istom redoslijedu kao u uvjetu $R5 \geq R2$:

CMP R5, R2

BHS 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
	BHS	UVECAJ_R7
NISTA	B	DALJE
UVECAJ_R7	ADD	R7, R7, #7
DALJE	...	

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

	CMP	R5, R2
	BLO	DALJE
UVECAJ_R7	ADD	R7, R7, #7
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:

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

```
CMP      R0, R1
BLT      R0_MANJI
```

```
R0_VECI_JEDNAK  MOV      R2, R1      ; R1 je manji
                  B        KRAJ
```

```
R0_MANJI        MOV      R2, R0      ; R0 je manji
```

```
KRAJ
```

Uspoređivanje brojeva

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

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

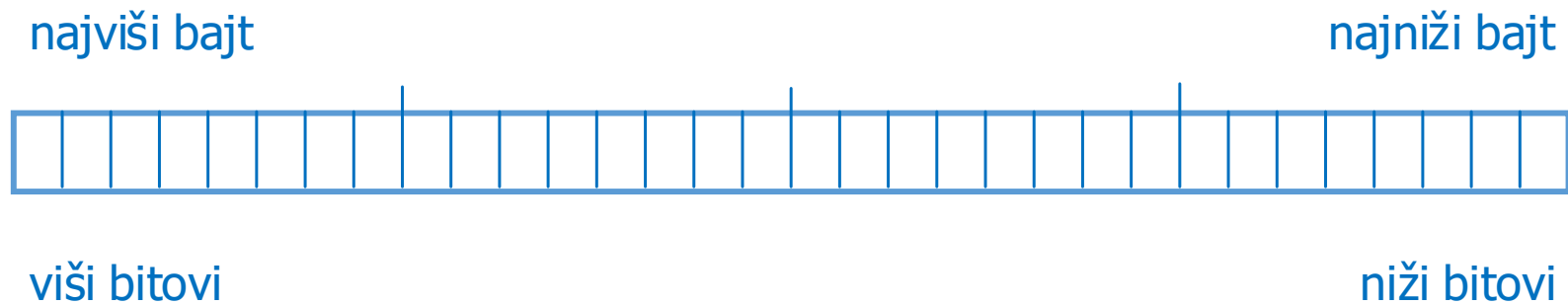
```
ORRS    R0,R6,R6 ; broj ispitaj i stavi u R0
BEQ     KRAJ      ; ako je 0, u R0 je rezultat
BMI     NEG       ; ispitaj predznak
POZ     MOV       R0, #1
        B         KRAJ
NEG     MVN       R0,#0

KRAJ
```

- 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 pojmova vezanih za bitove u podatku:
- Paritet / Parnost
- Maska



Rad s bitovima - Postavljanje bitova

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

```
LDR    R1, MASKA
ORR     R0, R1, R0
SWI     123456
```

```
MASKA   DW      0b1111    ; ostali bitovi su 0
```

Ili jednostavnije (i bolje):

```
ORR     R0, R0, 0b1111
SWI     123456
```

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.

>>>>

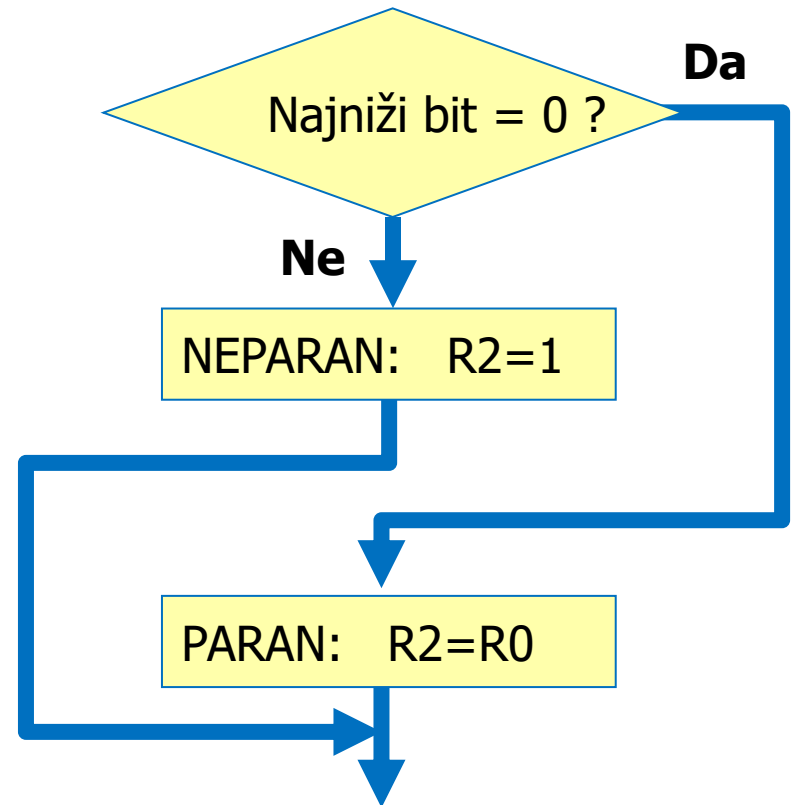
Rad s bitovima - Ispitivanje bitova

```
ANDS    R2, R0, #1    ; ispitaj najniži bit  
BEQ      PARAN
```

```
NEPARAN  MOV     R2, #1  
B        KRAJ
```

```
PARAN    MOV     R2, R0
```

```
KRAJ     SWI 123456
```



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 nuli:
 - 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.

```
LDR    R1, MASKA
ANDS   R1, R0, R1
BNE    IMA_1
```

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

```
IMA_1  SWI 123456        ; u isp. bitovima ima 1
```

```
MASKA  DW 0b11000000000000000000000000000000000000000011
```

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 nuli:
 - Ako rezultat=0, onda su svi ispitivani bitovi u jedinicama
 - Ako rezultat \neq 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.

```
LDR    R1, MASKA
```

```
ORR     R1, R0, R1
```

```
MVN     R2, #0
```

```
EORS    R1, R1, R2    ; komplementiranje
```

```
BNE     KRAJ
```

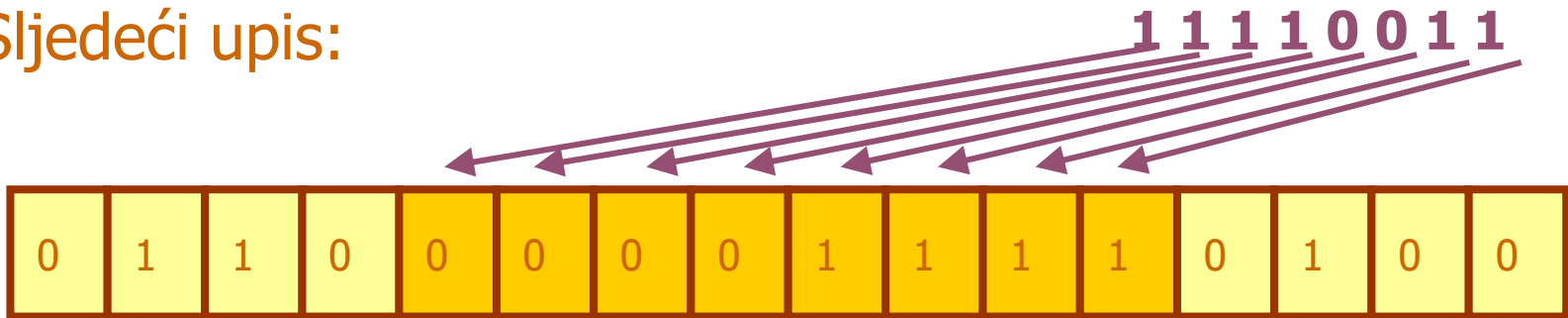
```
MOV     R0, #0        ; u isp. bitovima su sve 1
```

```
KRAJ    SWI 123456     ;
```

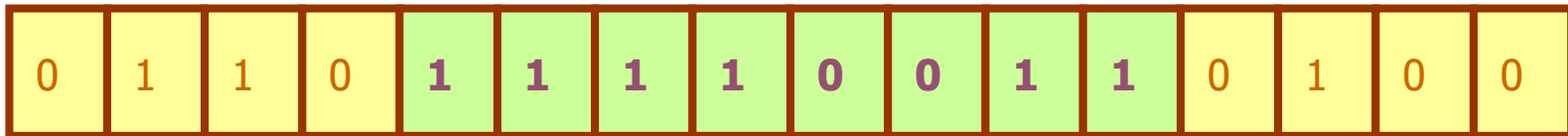
```
MASKA   DW 0b101111111111111100000001111111100001
```

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
Poravnati podatak:	00yyyy00		
OR:	xxyyyyxx		



The diagram illustrates the bit-wise OR operation. Two arrows point from the 'Maskirani podatak' (0000yyyy) and 'Maskirani registar' (xx0000xx) to the 'OR' result (xxyyyyxx). The alignment of the data is shown by the 'Poravnati podatak' (00yyyy00) row, which is shifted to the right relative to the original data.



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

```
LDR R0, POD_R0
```

```
LDR R7, POD_R7
```

```
; brisanje bitova 2 do 10 u R0
```

```
LDR R1, MASKA0
```

```
AND R0, R0, R1
```

```
; brisanje svih bitova osim 20 do 28 u R7
```

```
LDR R1, MASKA7
```

```
AND R7, R7, R1
```

```
MOV R7, R7, ROR #18 ; poravnanje rotacijom
```

```
ORR R0, R7, R0 ; upis u R0
```

```
SWI 123456
```

```
POD_R0 DW 0b01010101010101010101010101010101010101
```

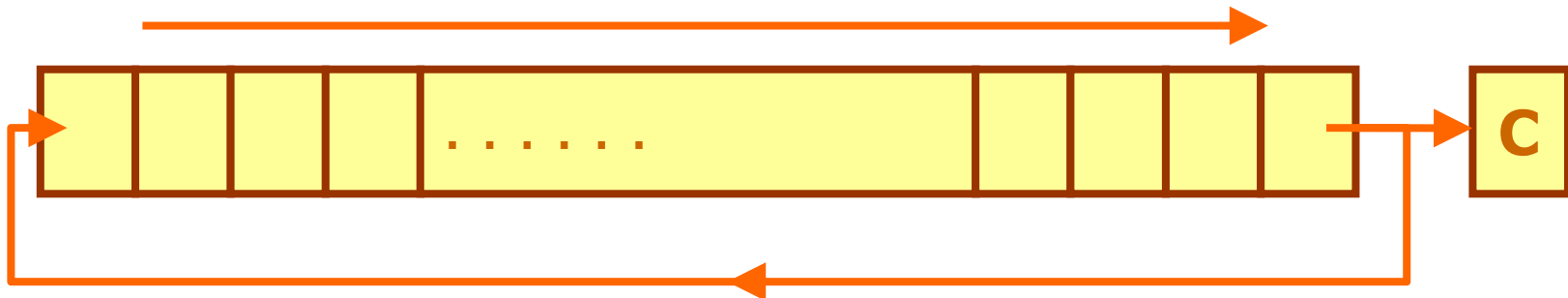
```
POD_R7 DW 0b10101010101010101010101010101010101010
```

```
MASKA0 DW 0b111111111111111111111111000000000011
```

```
MASKA7 DW 0b00011111111111110000000000000000000000
```

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

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

```
MOV R0, #0b11001010 ; ulazni podatak
```

```
MOV R1, #0           ; R1 = brojač nula
```

```
MOV R2, #6           ; R2 = brojač za petlju
```

```
MOV R0, R0, ROR #3   ; "izbaci" bitove 0 do 2
```

```
LOOP MOVS R0, R0, RRX           ; može i npr movs r0,r0,ror #1
```

```
BCS JEDAN
```

```
ADD R1, R1, #1
```

```
JEDAN SUBS R2,R2,#1
```

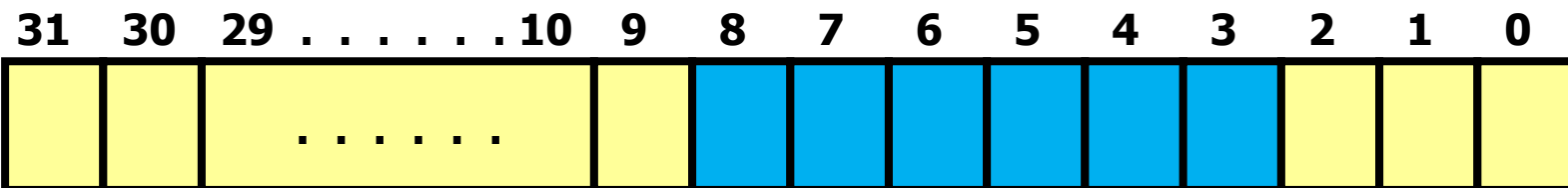
```
BNE LOOP
```

```
STR R1, NULE
```

```
SWI 123456
```

```
ORG 0x100
```

```
NULE DW 0
```



Višestruka preciznost

- Dijelovi računala (memorijske lokacije, registri, ALU, sabirnice) su ograničeni na određen broj bita
- Npr. Arm ima 32-bitnu arhitekturu (tj. riječ mu ima 32 bita) pa može normalno raditi s podacima te širine
- Ako treba raditi s brojevima većeg opsega ili kakvim drugim podacima 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

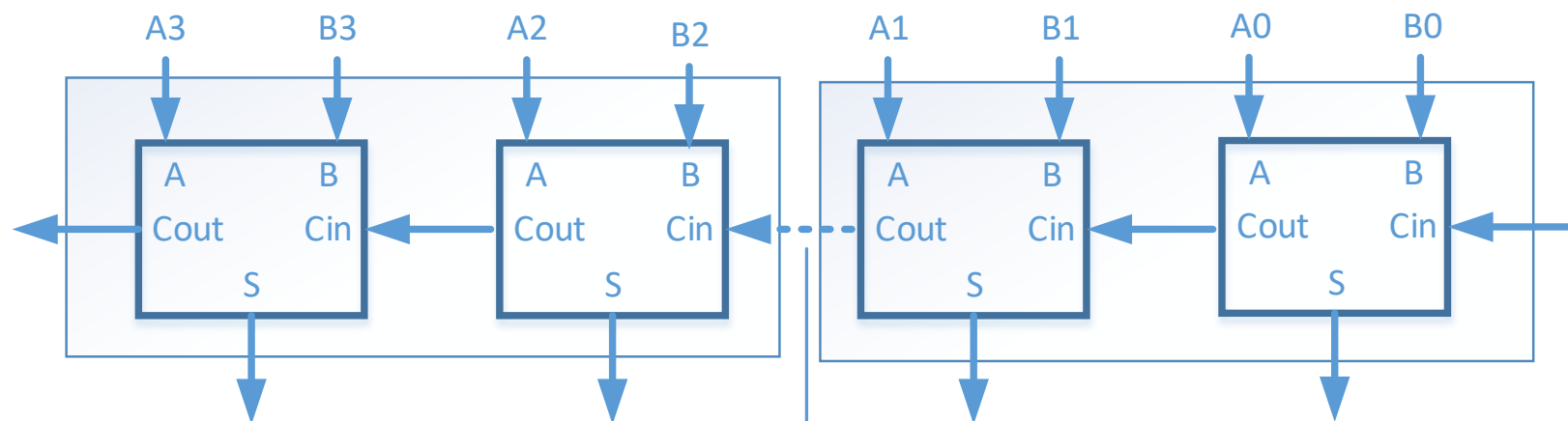
Višestruka preciznost

- **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 Arm 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 registra
- 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



Veza koju treba
programski
ostvariti

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 spremljen u zastavici C, onda naredba ADC zapravo radi ovako:

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

- Sklopovski se naredba ADC izvodi tako da se na ulaz Cin od najnižeg potpunog zbrajala, 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 sprema na RL i RH.

; ZBROJI NIŽE DIJELOVE

LDR R0, AL

LDR R1, BL

ADDS R2, R0, R1

STR R2, RL

; ZBROJI VIŠE DIJELOVE

LDR R0, AH

LDR R1, BH

ADC R2, R0, R1

STR R2, RH

SWI 123456



; OPERANDI

AL DW 0x0A3541E21

AH DW 0x942F075F

BL DW 0x936104A7

BH DW 0x017F3784

; 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 međuprijenosi, koje bi trebalo prenositi između viših i nižih riječi podataka

Višestruka preciznost - Pomaci i rotacije

- Sklopovski ostvareni pomaci i rotacije prenose bitove između pojedinih riječi pa to također treba napraviti i u programu
- Redoslijed operacije 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 komplicirano...

Višestruka preciznost - Pomaci i rotacije

- Npr neka je 64 bitni podatak u registrima R1,R0
- Prvo pomaknemo ulijevo R0. Izlazni bit je u zastavici C. Pomaknemo R1 ulijevo i upišemo C u najniži bit od R1:

```
MOVS    R0,R0,LSL #1
MOV     R1,R1,LSL #1
BCC     DALJE
ORR     R1,R1,#1
```

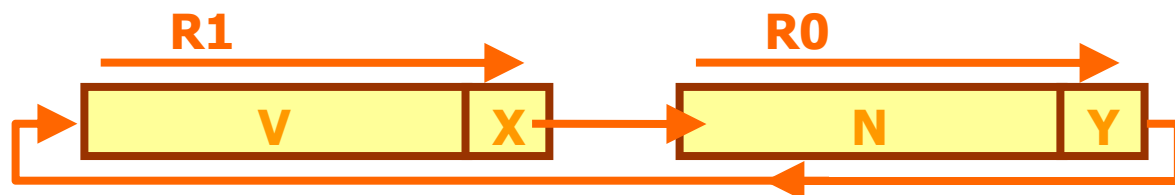
DALJE

- Drugo rješenje

```
MOVS    R0,R0,LSL #1
MOV     R1,R1,LSL #1
ADC     R1,R1,#0
```

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:



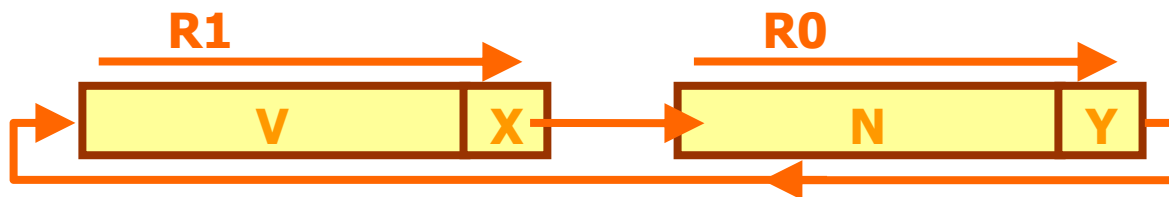
Vidimo da su bitovi V i N na ispravnom mjestu, ali bitovi X i Y moraju zamijeniti mjesta

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
```

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

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

```
EOR    R0, R0, R3
EOR    R1, R1, R3
```

```
; Nezavisna rotacija R0 i R1
```

```
MOV     R0, R0, ROR #1
MOV     R1, R1, ROR #1
```

AArch32 uvjetno izvođenje naredaba

- AArch32 ima strojne kodove fiksne širine 32 bita
- Formati strojnih kodova dosta ovise o pojedinoj naredbi (vidi primjere na sljedećem slajdu)

Primjeri formata naredaba

Naredba

Load/Store Multiple

B, BL

Aritmetičko logička,
neposredni pomak

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Uvjet				1	0	0	P	U	S	W	L	Rn				Popis registara															
Uvjet				1	0	1	L	24-bitni odmak																							
Uvjet				0	0	0	Op kod				S	Rn				Rd				Iznos pomaka				po- mak	0	Rm					

↑
Polje uvjeta

Polje uvjeta (condition field)

- Većina ARM-ovih naredaba **može se izvoditi uvjetno**. Te naredbe u strojnom kodu imaju **polje uvjeta** (condition field) u kojem se zadaje uvjet koji mora biti zadovoljen da bi se naredba izvela.
- Svi uvjeti temelje se na zastavicama stanja u CPSR.
- Ako uvjet nije zadovoljen, umjesto naredbe izvest će se NOP (NOP je naredba koja ne izvodi ništa - kratica od No Operation).
- Uvjetno izvođenje omogućuje programeru izvedbu programa bez korištenja naredbi grananja, čime se može ubrzati izvođenje nekog kratkog dijela programa

Uvjeti (isti kao i kod Branch)

Opcode [31:28]	Mnemonički naziv	Puni naziv (engleski)	Stanje zastavica
0000	EQ	Equal	Z
0001	NE	Not equal	!Z
0010	CS/HS	Carry set/unsigned higher or same	C
0011	CC/LO	Carry clear/unsigned lower	!C
0100	MI	Minus/negative	N
0101	PL	Plus/positive or zero	!N
0110	VS	Overflow	V
0111	VC	No overflow	!V
1000	HI	Unsigned higher	C and !Z
1001	LS	Unsigned lower or same	!C or Z
1010	GE	Signed greater than or equal	N == V
1011	LT	Signed less than	N != V
1100	GT	Signed greater than	!Z and N == V
1101	LE	Signed less than or equal	Z or N != V
1110	AL	Always (unconditional)	-
1111	(NV)	See Condition code 0b1111	-

Primjer uvjetnog izvođenja

- Primjer od ranije (pomak u lijevo 64bitnog broja):

```
MOVS    R0,R0,LSL #1
MOV     R1,R1,LSL #1
BCC     DALJE
ORR     R1,R1,#1
```

DALJE

```
MOVS    R0,R0,LSL #1
MOV     R1,R1,LSL #1
ORRCS   R1,R1,#1
```

Primjer uvjetnog izvođenja niza naredaba

1. način - korištenjem naredbe uvjetnog grananja:

```
CMP R0, #0  
BNE DALJE  
MOV R1, #1    ; uvjetni dio koda  
MOV R2, #2    ; uvjetni dio koda  
MOV R3, #3    ; uvjetni dio koda  
DALJE  
...
```

Ovisno o protočnoj strukturi i ovisno koliko često je ispitivani uvjet istinit, može se odrediti koji način pisanja je efikasniji u pojedinoj situaciji.

2. način - korištenjem uvjetnog izvođenja naredaba:

```
CMP R0, #0  
MOVEQ R1, #1    ; uvjetni dio koda  
MOVEQ R2, #2    ; uvjetni dio koda  
MOVEQ R3, #3    ; uvjetni dio koda
```

Primjer: množenje dva 16-bitna broja

Treba pomnožiti dva 16-bitna broja metodom pomaka. Operandi su spremljeni od adrese 100, a 32-bitni rezultat treba staviti iza operanada. Multiplikand može biti u zapisu NBC ili 2'k, a multiplikator može biti samo u zapisu NBC.

Načelo množenja na primjeru dva 4-bitna broja:

multiplikator

$$\begin{array}{r} 0011 * 1101 \\ \hline 00000011 \\ 00001100 \\ + 00011000 \\ \hline 00100111 \end{array}$$

Primjer programa za množenje



; Program za množenje 16-bitnih brojeva metodom pomaka

MOV R4, #0x100	;postavlja u registar R4 adresu podataka 100
LDRH R5, [R4], #2	;multiplikator se učitava u R5
LDRSH R6, [R4], #2	;multiplikand se učitava u R6 (predznak sačuvan)
MOV R7, #0	;čisti se registar za spremanje rezultata (R7)

PETLJA

MOVS R5, R5, LSR #1	;pomak multiplikatora za jedan bit udesno, ;najniži bit otići će u C
ADDCS R7, R7, R6	;ako je zastavica C=1, R6 se dodaje privr. rezultatu
MOV R6, R6, LSL #1	;pomak R6 za jedan bit ulijevo ;(priprema za mogući sljedeći korak)
CMP R5, #0	;provjerava je li multiplikator različit od nule
BNE PETLJA	;ako je, onda se petlja ponavlja

STR R7, [R4]	;spremanje rezultata
--------------	----------------------

SWI 123456

ORG 0x100

DH 5, -1

Primjer dijeljenja metodom pomaka



- Dijeljenje A/B (djeljenik A , djelitelj B) odvija se u dva koraka:
- **Prvi korak:** poravnanje brojeva – traženje najvećeg višekratnika djelitelja sadržanog u djeljeniku
 - Djelitelj B pomičemo u lijevo sve dok vrijedi $A \geq 2B$ i $MSB(B)=0$ te pamtimo **broj_pomaka**
 - Ponavlja se dok $A \geq 2B$ zato da pronađemo **najveći** višekratnik
 - Prije ispitivanja $A \geq 2B$ provjeravamo da li je $MSB(B)=1$
 - Ako $MSB(B)=1$ tada smo sigurni da je to najveći višekratnik jer bi množenjem s 2 dobili broj koji je veći od opsega brojeva koji se mogu prikazati u 32 bita što bi sigurno bilo veće od djeljenika
 - U slučaju $MSB(B)=1$ moramo preskočiti ispitivanje $A \geq 2B$ jer bi usporedba mogla dovesti do krivog rezultata (B pomaknuto u lijevo za jedan bit prelazi 32 bita registra)
- **Drugi korak:** Nakon što od djeljenika oduzmemo najveći višekratnik djelitelja, postupak se ponavlja **broj_pomaka** puta (kao dijeljenje na papiru - vidi sljedeći slajd)

Postupak...



```
1101101001 : 1010 = 1
- 1010000000
0011101001 : 1010 = 10
- 1010000000
0011101001 : 1010 = 101
- 10100000
0001001001 : 1010 = 1010
- 1010000
0001001001 : 1010 = 10101
- 101000
0000100001 : 1010 = 101011
- 10100
0000001101 : 1010 = 1010111
- 1010
0000000011 (ostatak)
```

max. 6 pomaka djelitelja u lijevo, oduzmemo od djeljenika i stavimo 1 u rez
(5) Djelitelj za 1 u desno, rez 1 u lijevo
- Ne može se oduzeti, dodamo 0 u rez
(4) Djelitelj za 1 u desno, rez 1 u lijevo
- Može se oduzeti, dodamo 1 u rez
(3) Djelitelj za 1 u desno, rez 1 u lijevo
- Ne može se oduzeti, dodamo 0 u rez
(2) Djelitelj za 1 u desno, rez 1 u lijevo
- Može se oduzeti, dodamo 1 u rez
(1) Djelitelj za 1 u desno, rez 1 u lijevo
- Može se oduzeti, dodamo 1 u rez
(0) Djelitelj za 1 u desno, rez 1 u lijevo
- Može se oduzeti, dodamo 1 u rez

Broj_pomaka = 0 -> KRAJ POSTUPKA

Primjer programa za dijeljenje



; Program za 32-bitno dijeljenje metodom pomaka

```
MOV  R4, #0x100      ; postavlja u registar R4 adresu podataka 100
LDR  R5, [R4], #4     ; djeljenik (A) se učitava u R5
LDR  R6, [R4], #4     ; djelitelj (B) se učitava u R6
MOV  R7, #0           ; čisti se registar za spremanje rezultata (R7)
MOV  R8, #0           ; brojač inicijalnih pomaka
CMP  R6, R5           ; ako je B>A, nema potrebe za dijeljenjem
BHI  KRAJ
```

PORAVNAJ ;inicijalni korak: pronalaženje najvećeg višekratnika djelitelja

```
ANDS  R3, R6, #0x80000000 ; provjerava je li MSB djelitelja jednak 1
BNE  DIV      ; ako je, poravnavanje je nepotrebno
CMP  R5, R6, LSL #1      ; provjerava je li A >= 2*B
MOVHS R6, R6, LSL #1     ; ako je, pomiče B ulijevo
ADDHS R8, R8, #1         ; povećava brojač pomaka
BHI  PORAVNAJ           ; samo ako je A>2*B, poravnavanje se nastavlja
```

Primjer programa za dijeljenje (2. dio)



DIV

CMP R5, R6	; uspoređuje trenutni ostatak i B
SUBHS R5, R5, R6	; ako je ostatak \geq B, onda ga umanji za B
ADDHS R7, R7, #1	; i poveća rezultat za 1
CMP R8, #0	; ako je brojač pomaka > 0 nastavi, inače KRAJ
MOVHI R6, R6, LSR #1	; pomakne B udesno,
MOVHI R7, R7, LSL #1	; a rezultat ulijevo
SUBHI R8, R8, #1	; umanji brojač pomaka
BHI DIV	; i ponovi petlju

KRAJ

STR R5, [R4], #4	; spremanje ostatka
STR R7, [R4]	; spremanje rezultata
SWI 123456	

ORG 0x100

DW 0b1101101001 ; dec 873

DW 0b1010 ; dec 10

Još jedan primjer....



```
*****
```

```
; Program koji niz podataka (terminiran nulom) koji se nalazi u memoriji  
; na adresi 200 ispisuje u obrnutom redoslijedu na istu adresu. (obrni_niz_stog)
```

```
*****
```

```
MOV    R13, #0x400    ; inicijalizacija stoga  
MOV    R1, #0x200     ; učitavanje početne adrese  
MOV    R0, #0         ; brojač duljine niza
```

```
PETLJA LDR    R2, [R1], #4    ; petlja kojom čitamo niz  
        CMP    R2, #0        ; ako je nula, niz je gotov  
        BEQ    ISPIS  
        STMFD  SP!, {R2}     ; spremanje na stog  
        ADD    R0, R0, #1    ; povećaj brojač  
        B      PETLJA        ; idi dalje
```

>>>>

<<<<

ISPIS MOV R1, #0x200 ; učitavanje početne adrese

POM CMP R0, #0
 BEQ KRAJ
 LDMFD SP!, {R2} ; čitanje sa stoga
 STR R2, [R1], #4
 SUB R0, R0, #1 ; smanjujemo brojač
 B POM

KRAJ SWI 123456 ; Kraj

ORG 0x200

DW 1,2,3,4,5,0

Ispitivanje specijalnog broja...



```
*****
```

```
; Ispitivanje je li broj specijalan  
; specNum.s
```

```
*****
```

```
; Napisati program koji ispituje je li troznamenkasti dekadski broj 'xyz' zapisan  
; kao niz ASCII znamenaka (tzv. string) "xyz\0" specijalan broj za kojeg vrijedi:  
;  $xyz = xy * xy - z * z$ , gdje su x y i z znamenke stotice, desetice i jedinice.  
; Primjer takvog broja su brojevi:  $100 = 10*10 - 0*0$  i  $147 = 14*14 - 7*7$   
; Broj zapisan kao niz ASCII znamenki sa završnim null-znakom nalazi se na  
; adresi 1000. Ukoliko broj zadovoljava gornji uvjet, tada je potrebno u registar  
; r1 staviti sve jedinice, a ako uvjet nije ispunjen, tada u r1 treba staviti sve  
; nule.
```

```
*****
```

; ovaj odsječak vadi znamenke iz ASCII zapisa: u r1 će se nalaziti znamenka
; stotica, u r2 će se nalaziti znamenka desetica, a u r3 će biti znamenka jedinica

```
main    MOV r0, #0x1000
        LDRB r1, [r0], #1
        LDRB r2, [r0], #1
        LDRB r3, [r0]
        SUB r1, r1, #48          ; znamenka stotica (48 je ASCII znak od 0)
        SUB r2, r2, #48          ; desetice
        SUB r3, r3, #48          ; jedinice
```

; množenje s konstantom 100 ostvareno je kombinacijom naredaba ADD i MOV
; s pomakom, što je efikasnije od korištenja naredbe MUL (multiply)

```
ADD r5, r1, r1, LSL #3          ; r5 = r1+8*r1 = 9*r1
ADD r5, r5, r1, LSL #4          ; r5 = r5+16*r1 = 9*r1+16*r1 = 25*r1
MOV r5, r5, LSL #2              ; konačno r5 = 4*r5 = 4*25*r1 = 100 * r1
```

>>>>

<<<<

; množenje s konstantom 10 pomoću kombinacije naredaba ADD i MOV s pomakom

MOV r6, r2

ADD r6, r6, r6, LSL #2

MOV r6, r6, LSL #1 ; r6 = 10 * r2

ADD r7, r5, r6

ADD r7, r7, r3 ; konačan broj u binarnom zapisu, potreban za ispitivanje
; uvjeta zadatka $xyz = xy * xy - z * z$

; generiranje broja xy a nakon toga i broja $xy * xy$ te $z * z$

ADD r1, r1, r1, LSL #2

MOV r1, r1, LSL #1

ADD r1, r1, r2

; u r1 se nalazi broj xy

MUL r4, r1, r1

; u r4 se nalazi broj $xy * xy$

MUL r5, r3, r3

; u r5 se nalazi broj $z * z$

>>>>

<<<<

```
SUB  r4, r4, r5          ; r1 = xy*xy - z*z
CMP  r4, r7
MVNEQ r1, #0             ; sve jedinice u R1
MOVNE r1, #0             ; sve nule u R1

KRAJ  SWI 123456          ; Kraj programa

ORG 0x1000
; neki ASCII podaci ...
DSTR "147"
```

Komentari:

Množenje kombinacijom ALU-operacija može se ostvariti kad množimo s konstantom.

Obično se može ostvariti na više načina. Na primjer, množenje sa 100:

$$100 = ((1+8)+16)*4 \quad (\text{iz primjera: 3 naredbe})$$

$$100 = 128-32+4 \quad (3 \text{ naredbe})$$

$$100 = (1+32)*3+1 \quad (3 \text{ naredbe})$$

$$100 = 64+32+4 \quad (3 \text{ naredbe})$$

$$100 = (1+16)*3*2-2 \quad (4 \text{ naredbe})$$