ARM

Arhitektura in
programiranje v zbirniku

ARM (Advanced RISC Machine) = RISC?

+ load/store arhitektura

+ cevovodna zgradba

+ reduciran nabor ukazov, vsi ukazi 32-bitni

+ ortogonalen registrski niz, vsi registri 32-bitni

- veliko načinov naslavljanja

- veliko formatov ukazov

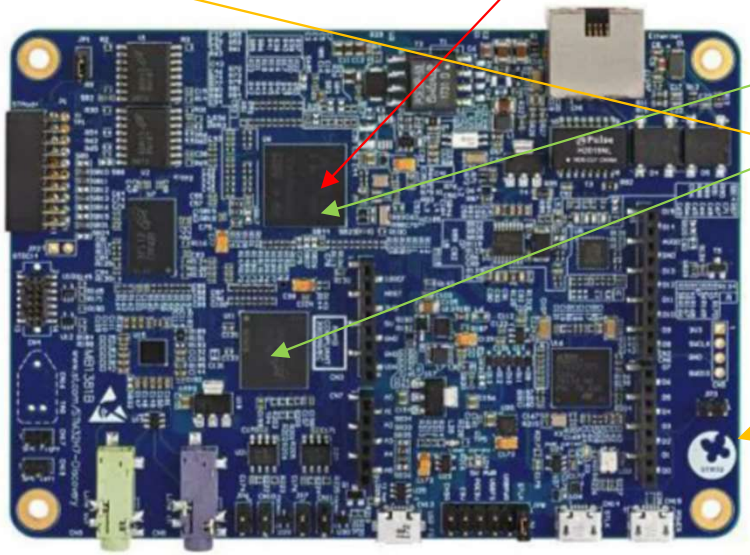
ARM (Advanced RISC Machine) = RISC?

- nekateri ukazi se izvajajo več kot en cikel (npr. *load/store multiple*) – obstaja nekaj kompleksnejših ukazov, kar omogoča manjšo velikost programov

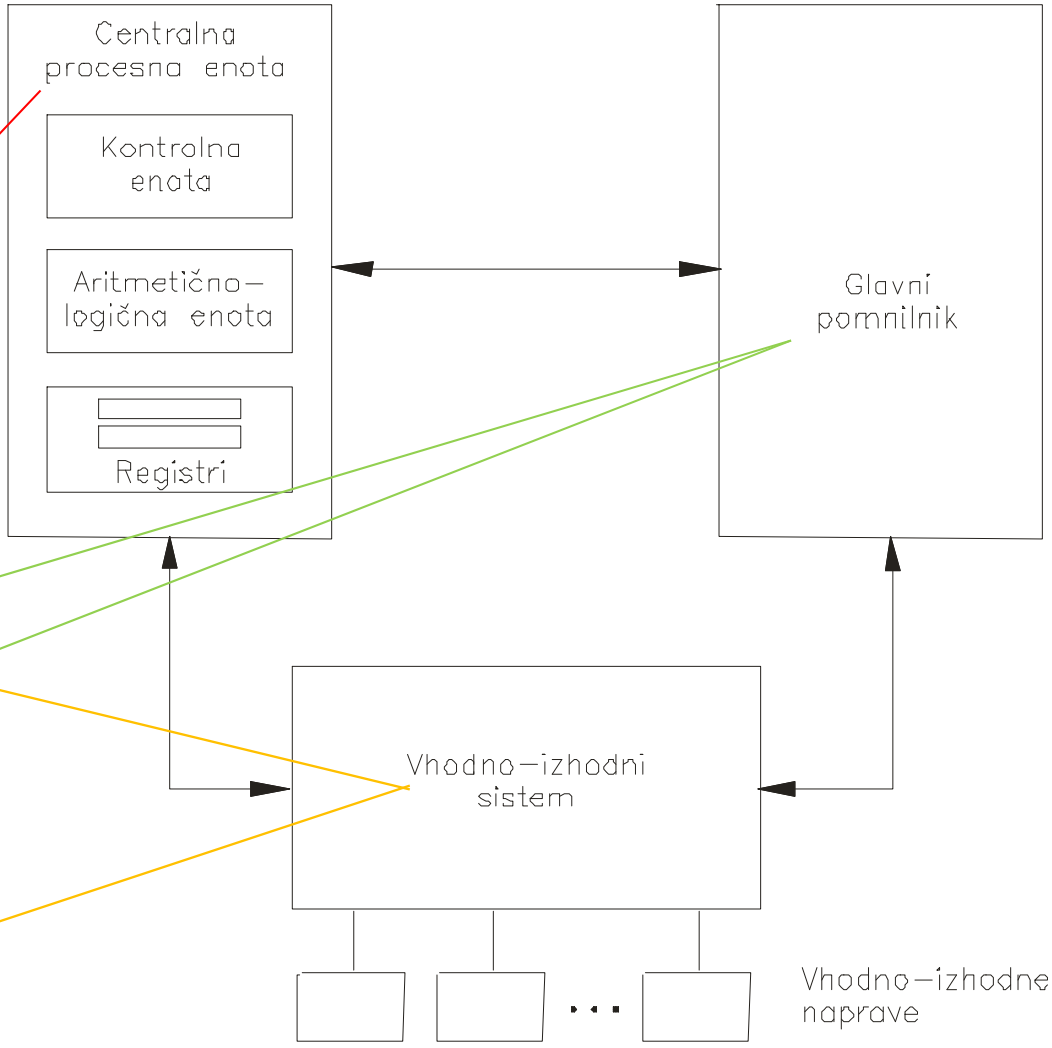
- dodaten 16-bitni nabor ukazov Thumb omogoča krajše programe.

- pogojno izvajanje ukazov – ukaz se izvede le, če je stanje zastavic ustrezno.

Osnovni model računalnika



STM32H750-DK

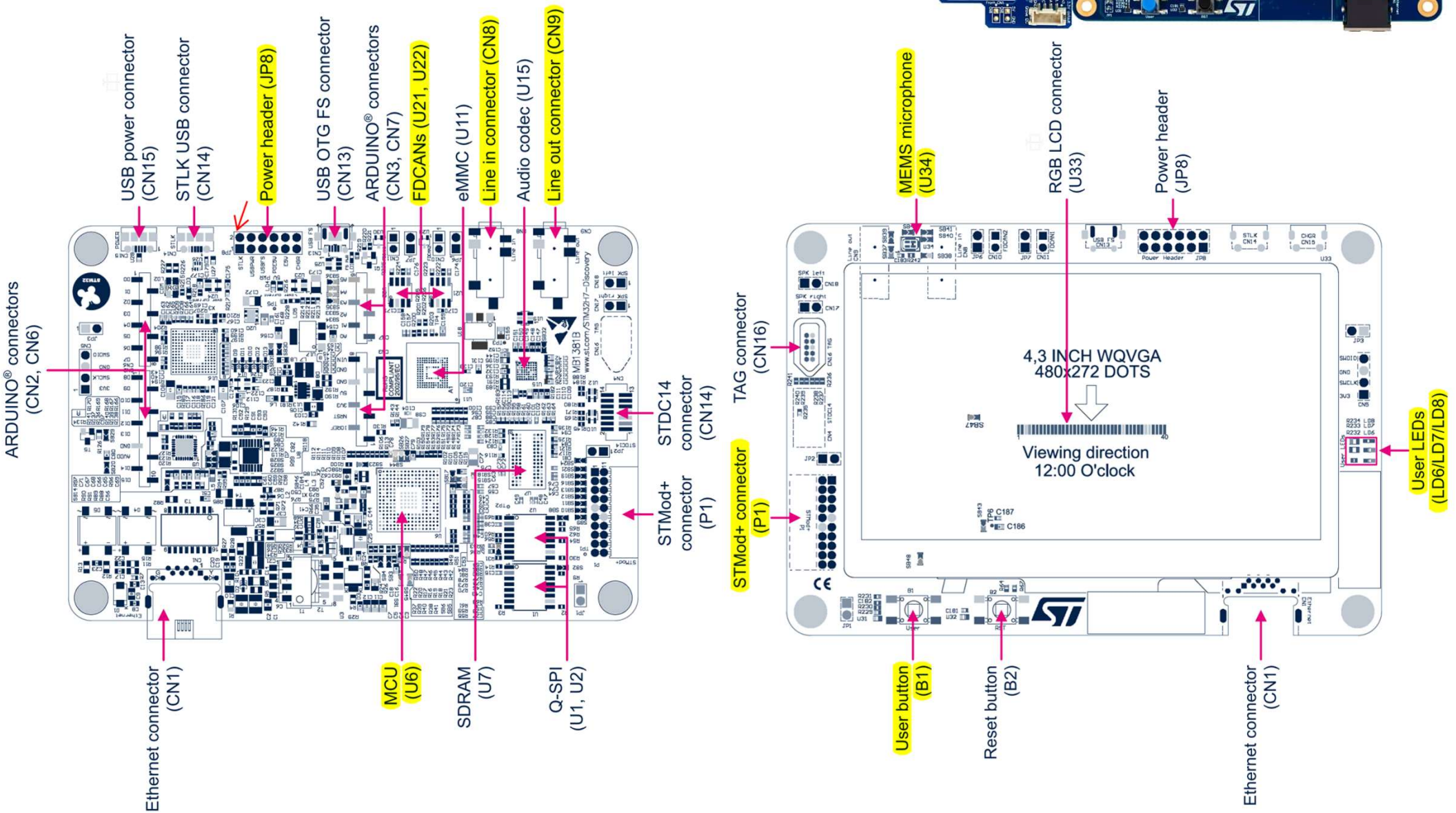


STM32H750B-DK Discovery razvojni sistem

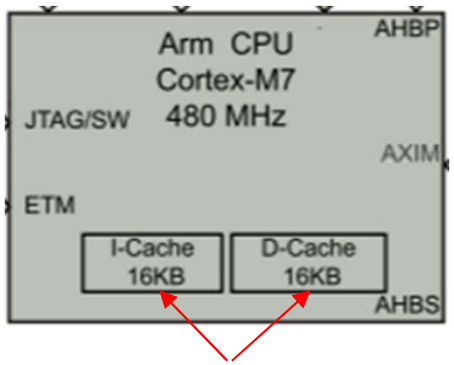
- Arm® Cortex® core-based microcontroller with 128 Kbytes (STM32H750XBH6) of Flash memory and 1 Mbyte of RAM, in TFBGA240+25 package
- 4.3” RGB interface LCD with touch panel connector
- Ethernet compliant with IEEE-802.3-2002, and POE
- USB OTG FS with Micro-AB connector
- SAI audio codec
- One ST-MEMS digital microphone
- 2 x 512-Mbit Quad-SPI NOR Flash memory
- 128-Mbit SDRAM
- 4-Gbyte on-board eMMC
- 1 user and reset push-button
- Fanout daughterboard
- 2 x FDCANs
- Board connectors:
  - USB FS Micro-AB connectors
  - ST-LINK Micro-B USB connector
  - USB power Micro-B connector
  - Ethernet RJ45
  - Stereo headset jack including analog microphone input
  - Audio header for external speakers
  - Arduino™ Uno V3 expansion connectors
  - STMod+

https://www.st.com/en/evaluation-tools/stm32h750b-dk.html

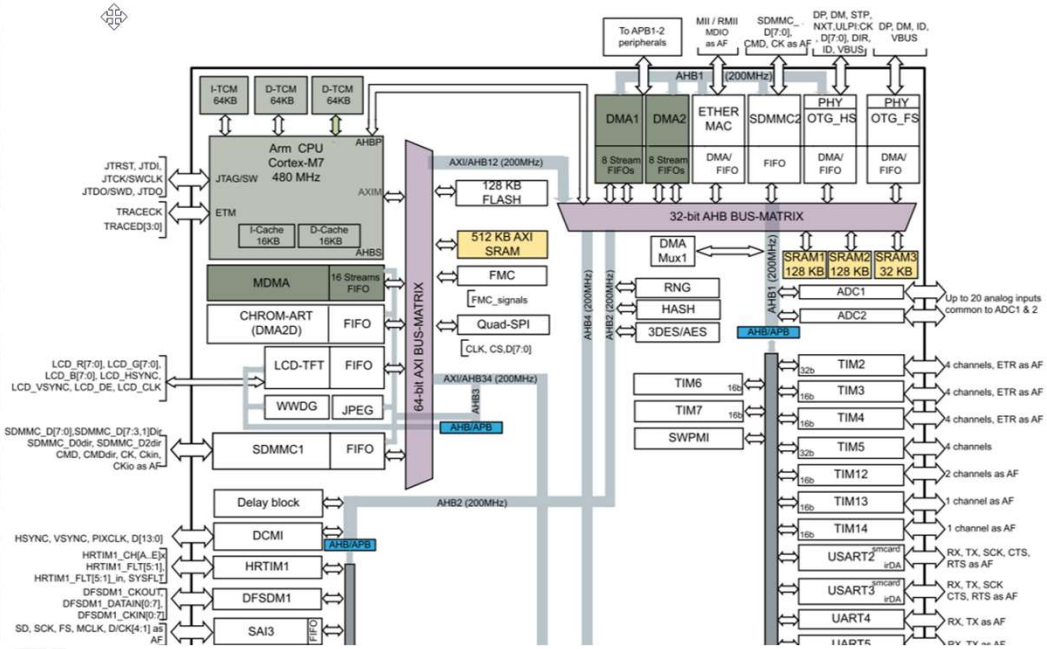
STM32H750B-DK Discovery razvojni sistem



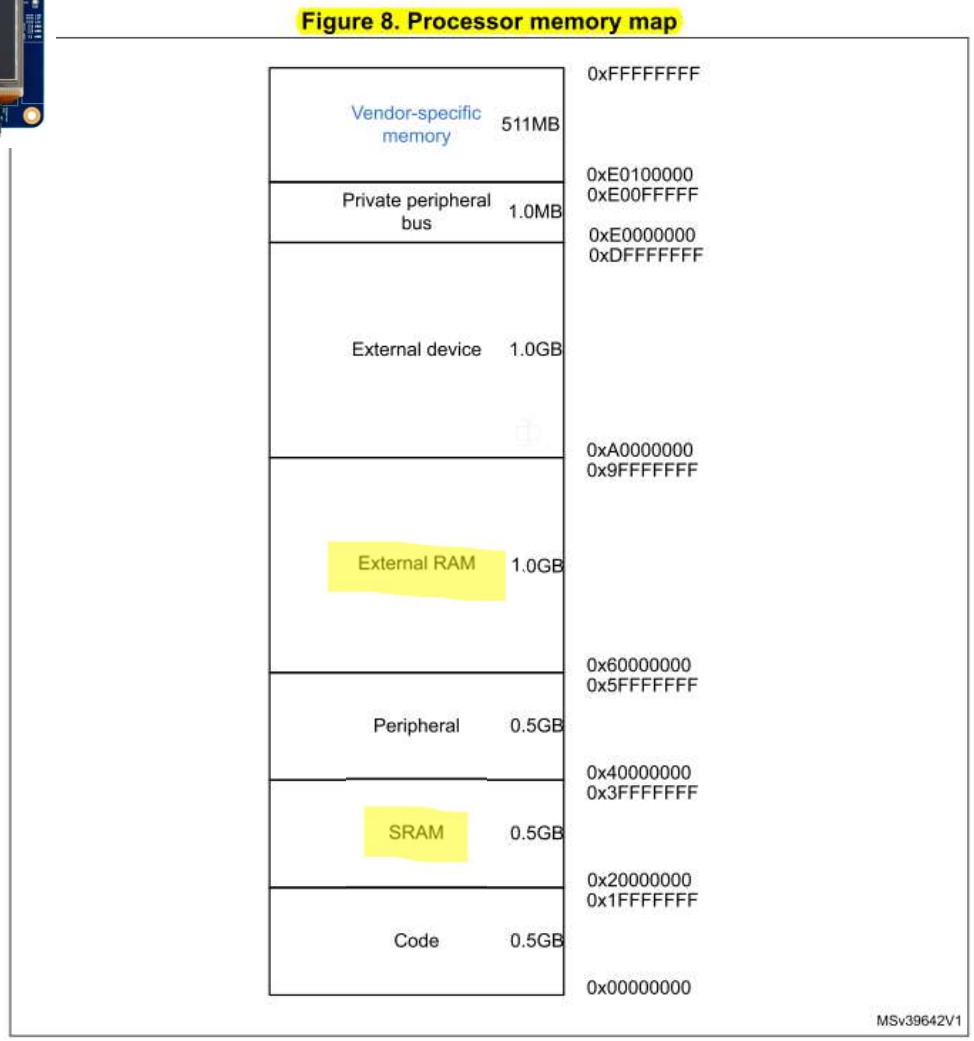
STM32H750XB



Harvardska arhitektura predpomnilnikov



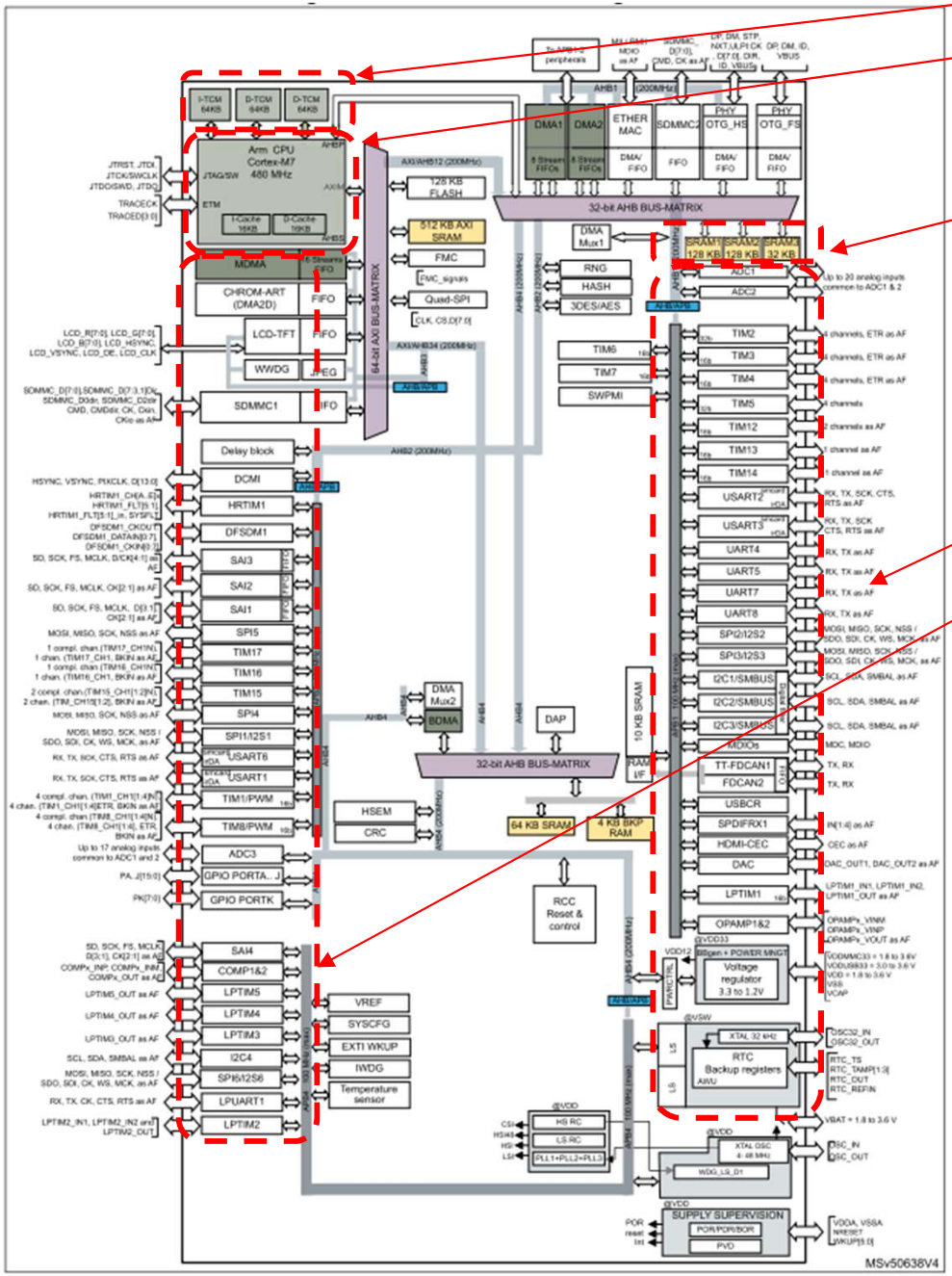
Shema pomnilniškega prostora

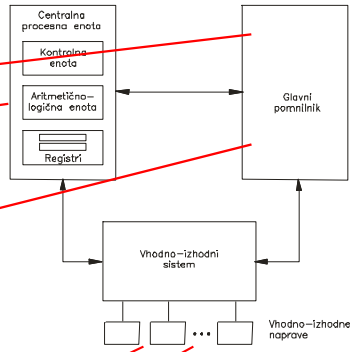


Princetonska arhitektura glavnega pomnilnika

```
MEMORY
{
 FLASH (rx)    : ORIGIN = 0x08000000, LENGTH = 128K
 DTCMRAM (xrw) : ORIGIN = 0x20000000, LENGTH = 128K
 RAM_D1 (xrw)  : ORIGIN = 0x24000000, LENGTH = 512K
 RAM_D2 (xrw)  : ORIGIN = 0x30000000, LENGTH = 288K
 RAM_D3 (xrw)  : ORIGIN = 0x38000000, LENGTH = 64K
 ITCMRAM (xrw) : ORIGIN = 0x00000000, LENGTH = 64K
}
```

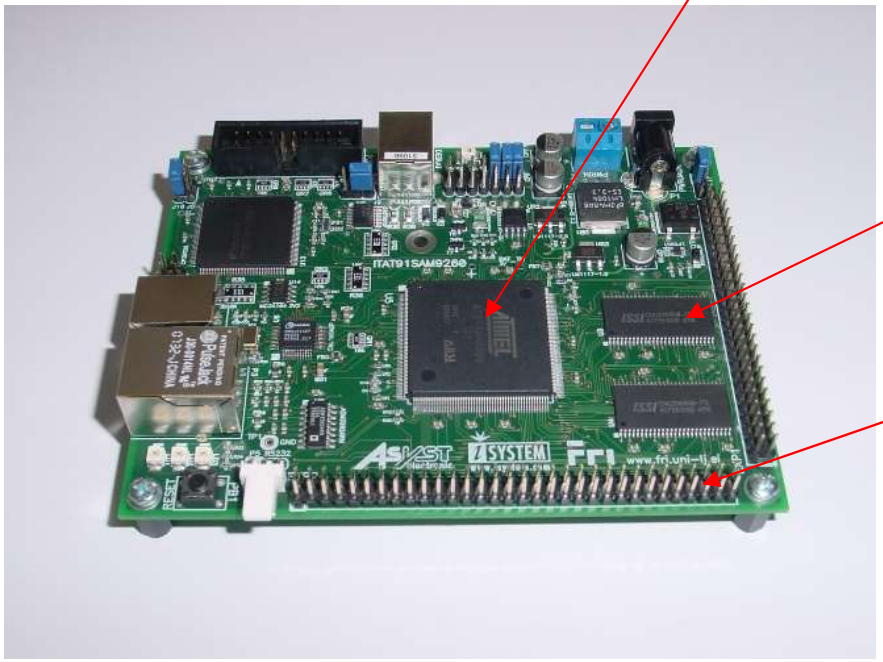
STM32H750XB

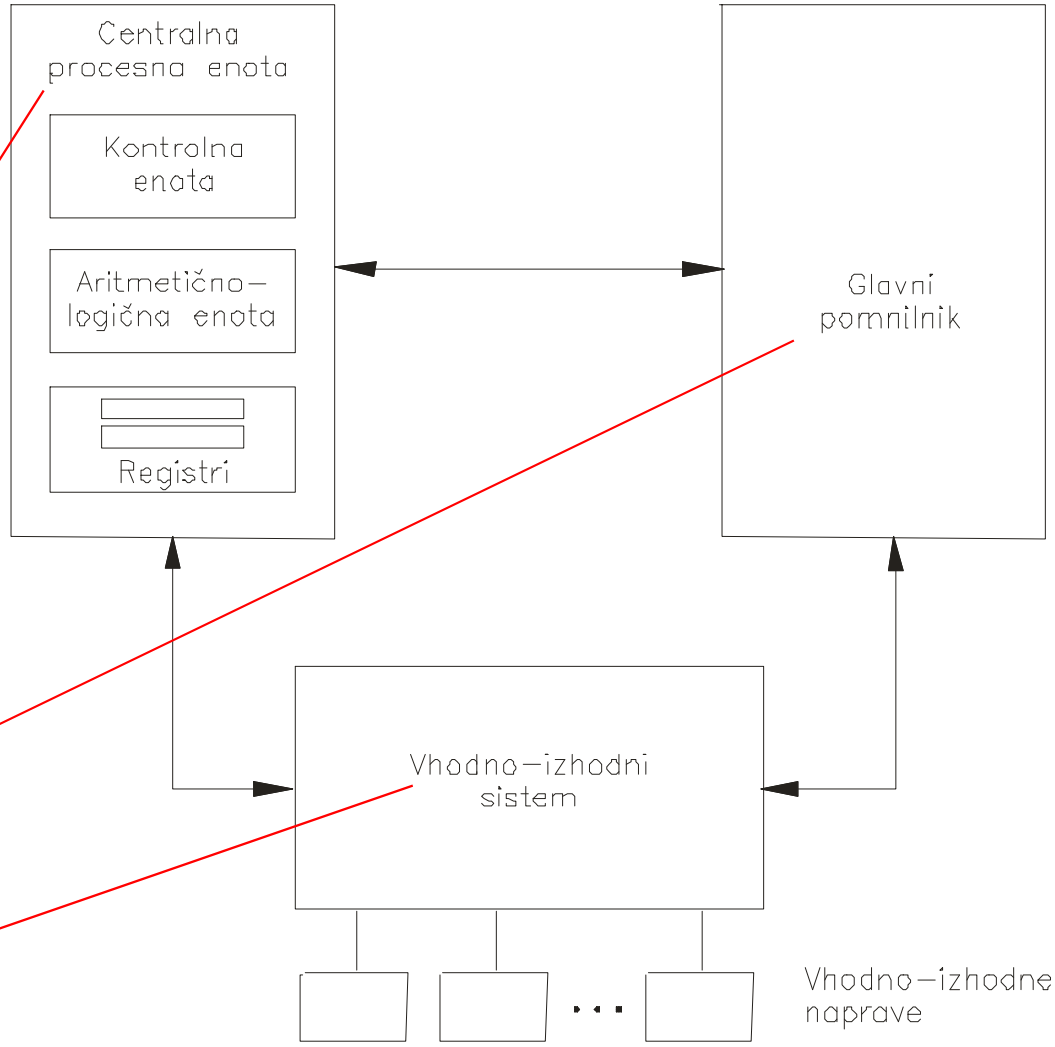




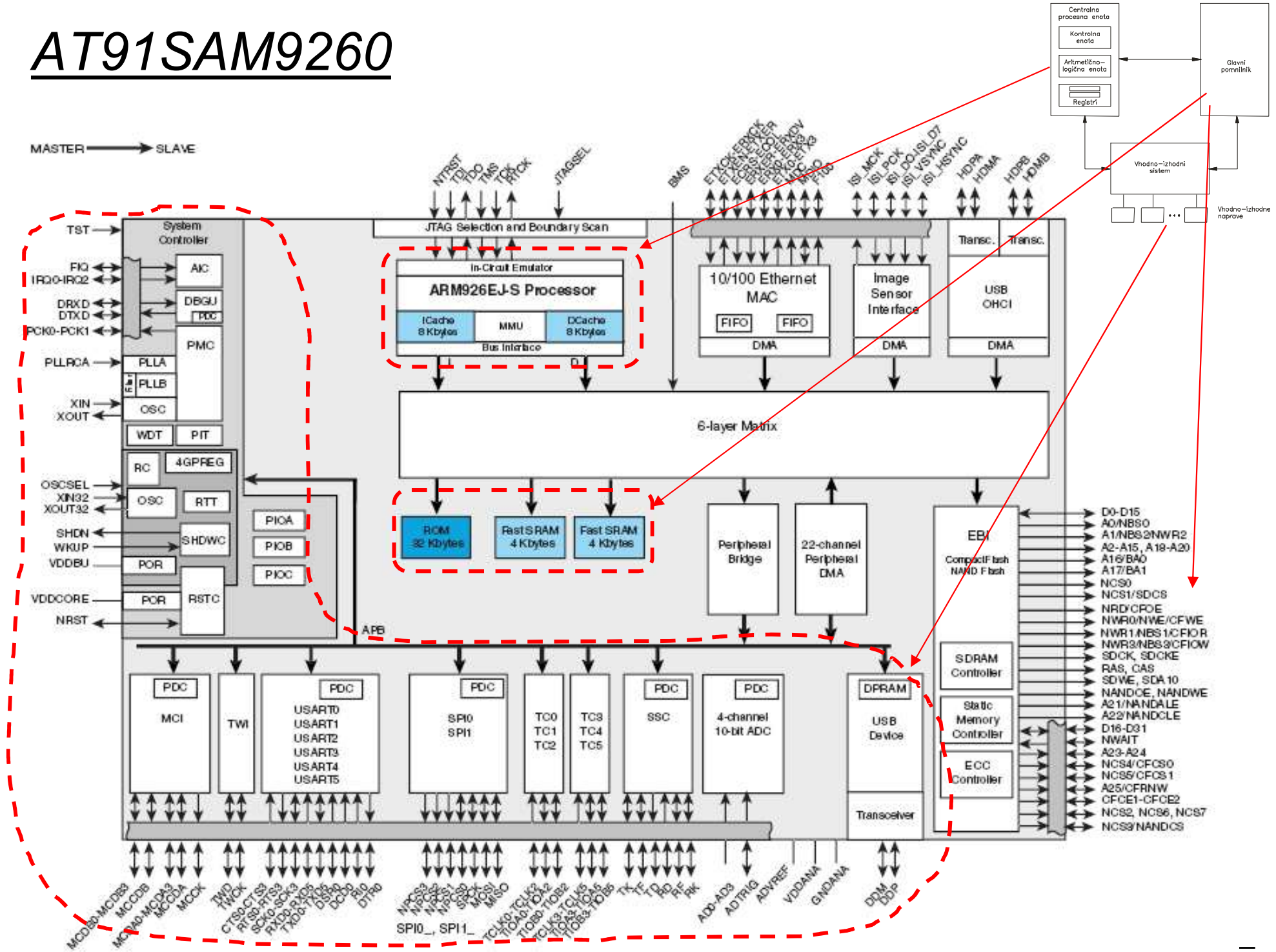
Osnovni model računalnika

FRI SMS



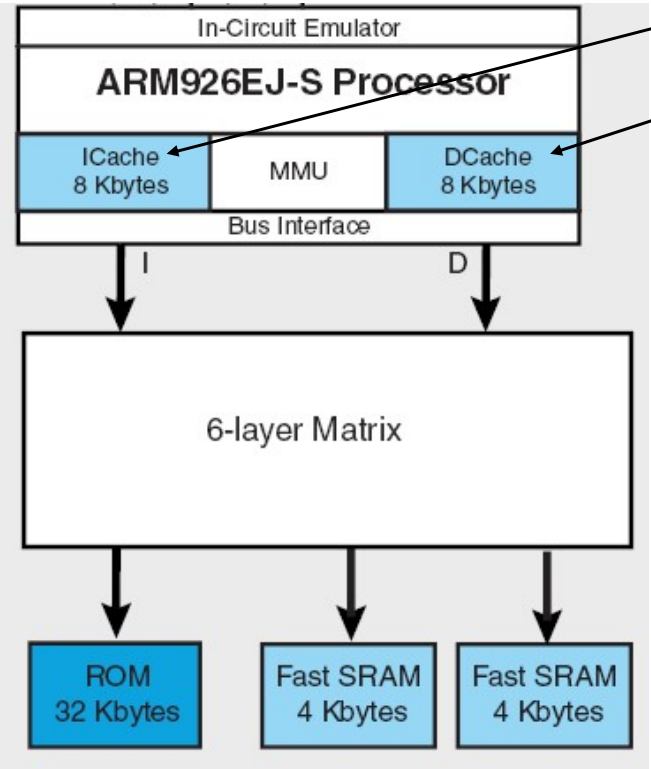


AT91SAM9260

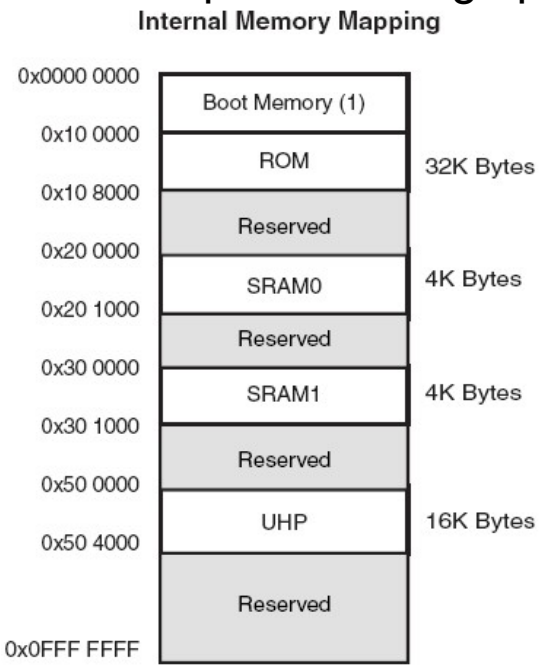


AT91SAM9260

Ločena predpomnilnika za ukaze in podatke.



Shema pomnilniškega prostora

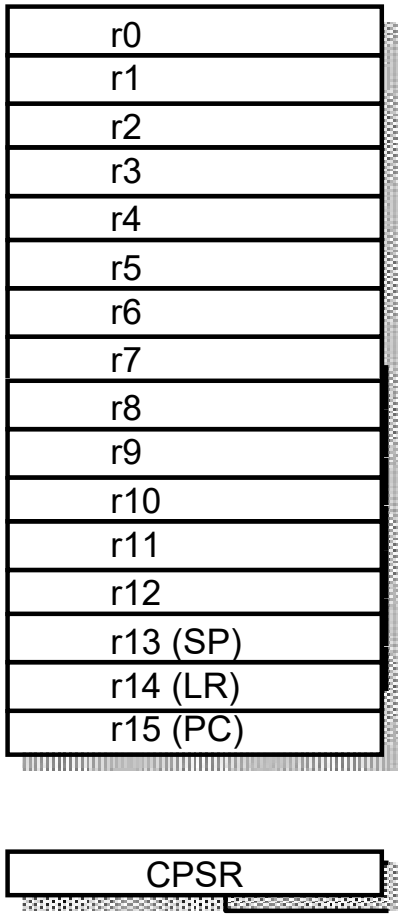


1) SRAM0 lahko preslikamo na naslove 0x00000000 – 0x00001000. To storimo, ker je pomnilnik na teh naslovih potreben ob zagonu.

ARM programski model

- **Programski model sestavlja 16 registrov ter statusni register CPSR (Current Program Status Register)**
- **Več** *načinov delovanja*, **vsak ima nekaj svojih registrov. Vseh registrov je v resnici 36**
- **Kateri registri so vidni je odvisno od načina delovanja procesorja (*processor mode*)**
- **Načine delovanja delimo v dve skupini:**
  - privilegirani (dovoljena bralni in pisalni dostop do CPSR)
  - neprivilegirani (dovoljen le bralni dostop do CPSR)

Programski model – uporabniški način



Uporabniški način (*user mode*):

- edini neprivilegirani način
- v tem načinu se izvajajo uporabniški programi

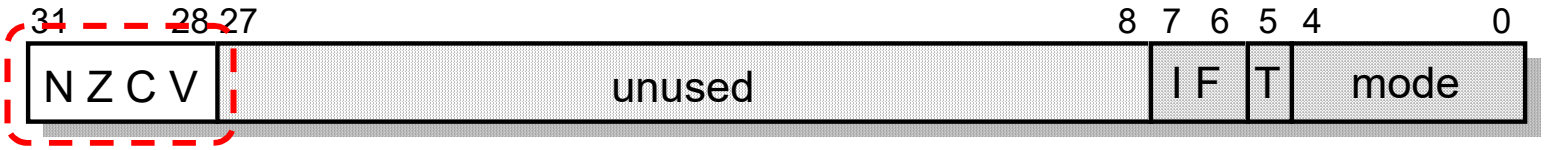
Programsko je vidnih 17 32-bitnih registrov:

r0 – r15 ter CPSR

Vidni registri:

- r0-r12: splošnonamenski (ortogonalni) registri
- r13(sp): skladovni kazalec (*Stack Pointer*)
- r14(lr): povratni naslov (*Link Register*)
- r15(pc): programski števec (*Program Counter*)
- CPSR: statusni register (*Current Program Status Register*)

Register CPSR



- zastavice (**N,Z,V,C**)
- maskirna bita za prekinitve (I, F)
- bit T določa nabor ukazov:
  T=0 : ARM arhitektura, procesor izvaja 32-bitni ARM nabor ukazov
  T=1: Thumb arhitektura, procesor izvaja 16-bitni Thumb nabor ukazov
- spodnjih 5 bitov določa način delovanja procesorja
- v uporabniškem (neprivilegiranem) načinu lahko CPSR beremo; ukazi lahko spreminjajo le zastavice.

Zastavice (lahko) ukazi spreminjajo glede na rezultat ALE:

N = 0: bit 31 rezultata je 0,	**N** = 1: bit 31 rezultata je 1 (*Negative*)
Z = 1: rezultat je 0,	**Z** = 0: rezultat je različen od nič (*Zero*)
C = 1: rezultat je povzročil prenos,	**C** = 0: rezultat ni povzr. Prenosa (*Carry*)
V = 1: rezultat je povzročil preliv,	**V** = 0: rezultat ni povzr. Preliva (*oVerflow*)

Programiranje v zbirniku

Zbirni jezik	Opis ukaza	Strojni jezik
adr r0, stev1	R0 ← nasl. stev1	0xE24F0014
ldr r1, [r0]	R1 ← M[R0]	0xE5901000
adr r0, stev2	R0 ← nasl. stev2	0xE24F0018
ldr r2, [r0]	R2 ← M[R0]	0xE5902000
add r3, r2, r1	R3 ← R1 + R2	0xE0823001
adr r0, rez	R0 ← nasl. rez	0xE24F0020
str r3, [r0]	M[R0] ← R3	0xE5803000

- **V zbirniku simbolično opisujemo:**
  - ukaze (z mnemoniki),
  - registre,
  - naslove
  - konstante

- **Programerju tako ni treba:**
  - poznati strojnih ukazov in njihove tvorbe
  - računati odmikov ter naslovov

Prevajalnik za zbirnik (*assembler*) :
- prevede simbolično predstavitev ukazov v ustrezne strojne ukaze,
- izračuna dejanske naslove ter
- ustvari pomnilniško sliko programa

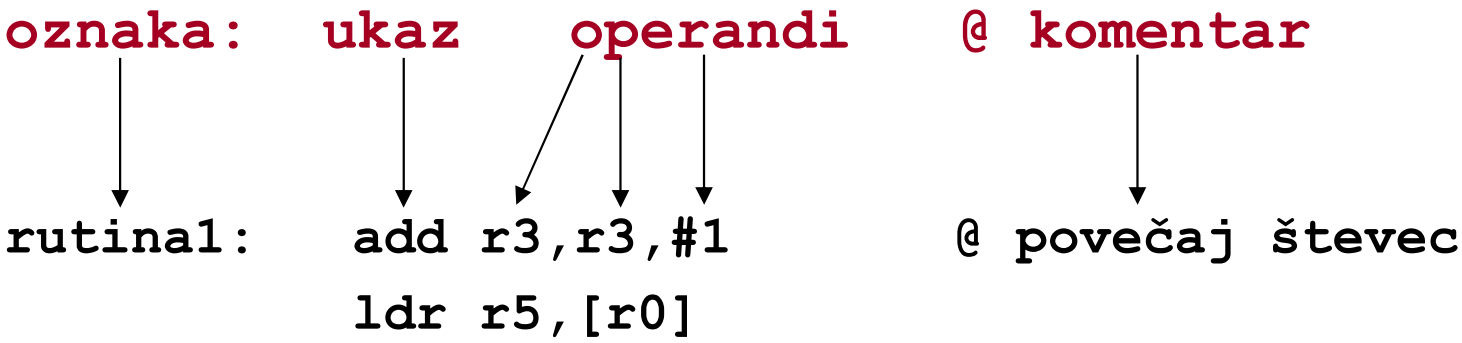
- **Program v strojnem jeziku ni prenosljiv:**
  - namenjen je izvajanju le na določeni vrsti mikroprocesorja

- **Zbirnik (*assembly language*) je „nizkonivojski“ programski jezik**

Programiranje v zbirniku

- **Vsaka vrstica programa v zbirniku predstavlja običajno en ukaz v strojnem jeziku**

- **Vrstica je sestavljena iz štirih stolpcev:**



- **Stolpce ločimo s tabulatorji, dovoljeni so tudi presledki**

Ukazi

- **Vsi ukazi so 32-bitni**

```
add r3, r2, r1 ⟹ 0xE0823001=0b1110…0001
```

- **Rezultat je 32-biten. Izjema je le množenje**

```
R1 + R2        ⟹ R3
```

- **Aritmetično-logični ukazi so 3-operandni**

```
add r3,r3,#1
```

- **Load/store arhitektura**

```
ldr r1, stev1        @ prenos v registre
ldr r2, stev2        @ prenos v registre
add r3, r2, r1       @ vsota registrov
str r3,rez           @ vsota v pomnilnik
```

Operandi

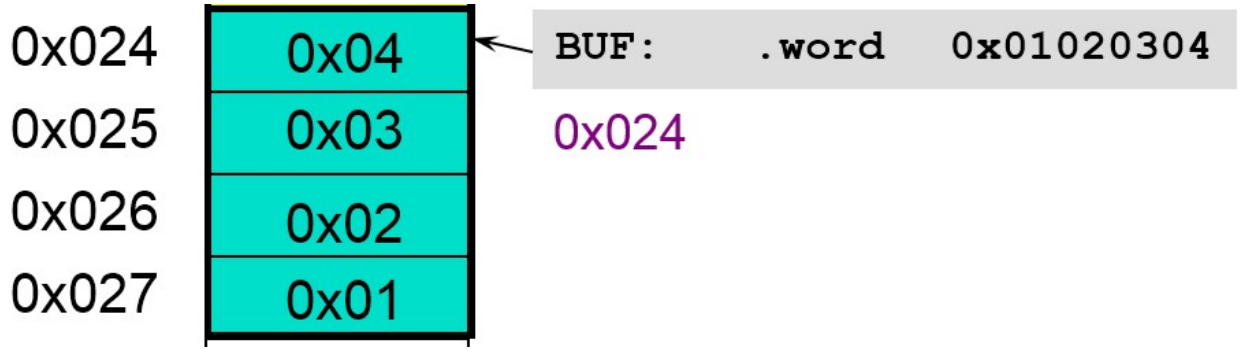
- **8, 16, 32-bitni ter predznačeni ali nepredznačeni pomnilniški operandi**

- **Obvezna poravnanost ukazov in operandov (16,32bitnih):**
  - 16-bitni poravnani na sodih naslovih
  - 32-bitni poravnani na naslovih, deljivih s 4

- **V CPE se vse izvaja 32-bitno (razširitev ničle ali predznaka)**

```
0xFF  ⟹  0x000000FF
```

- **Uporablja se pravilo tankega konca**



Oznake (labele)

Oznaka je nam razumljivo **simbolično poimenovanje :**

- **pomnilniških lokacij** ali
- **vrstic** v programu

```
            .text

stev1:      .word 64
stev2:      .word 0x10
rez:        .space 4

            .align
            .global __start
__start:

            ldr r1, stev1
            ldr r2, stev2
            add r3, r2, r1
            str r3, rez

__end:      b __end
```

Oznake običajno uporabljamo na dva načina:

- s poimenovanjem pomnilniških lokacij

  dobimo „spremenljivke“

```
STEV1:      .word   0x12345678
STEV2:      .byte   1,2,3,4
REZ:        .space  4
```

- za poimenovanje ukazov (vrstic), na katere se sklicujemo pri skokih.

```
            mov r4,#10
LOOP:       subs r4, r4, #1
            ...
            bne LOOP
```

Psevdoukazi - ukazi prevajalniku

Psevdoukazi:

- so navodila prevajalniku
- običajno so označeni s piko pred ukazom
- niso strojni ukazi za CPE, temveč ukazi prevajalniku
- CPE jih v končnem programu ne vidi

Psevdoukaze uporabljamo za:

- določanje vrste pomnilniških odsekov
- poravnavo vsebine
- rezervacijo pomnilnika za „spremenljivke“
- rezervacijo prostora v pomnilniku
- določanje začetne vsebine pomnilnika
- ustavljanje prevajanja
- rezervacijo in inicializacijo pomnilnika

```
.text .data
.align
.space
.space
.(h)word, .byte,…
.end
.fill
```

Ustvarjanje pomnilniške slike

Psevdoukaza za določanje pomnilniške slike sta:

```
.text
.data
```

S tema psevdoukazoma določimo, kje v pomnilniku bodo program in podatki.

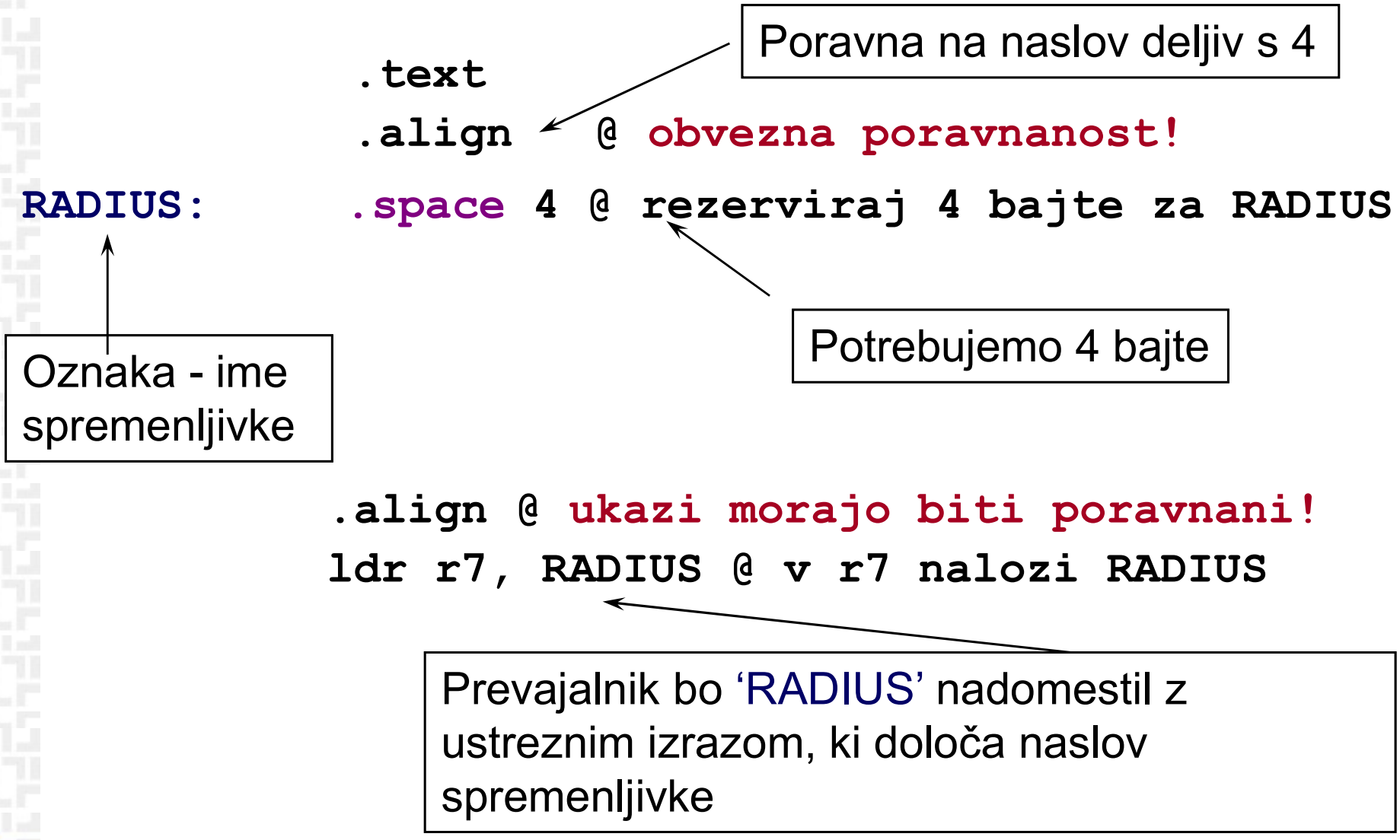
Tako za ukaze kot spremenljivke bomo v simulatorju uporabljali segment .text

Pri delu s ploščami pa bosta uporabljeni oba segmenta:

- **.text se shrani v Flash pomnilnik**
- **.data se shrani v Flash in ob zagonu prenese v RAM pomnilnik**

Rezervacija pomnilnika za spremenljivke

Za spremenljivke moramo v pomnilniku rezervirati določen prostor.



Rezervacija prostora v pomnilniku

Oznake omogočajo boljši pregled nad pomnilnikom:
– pomnilniškim lokacijam dajemo imena in ne uporabljamo absolutnih naslovov (preglednost programa)

```
BUFFER:     .space 40     @rezerviraj 40 bajtov
BUFFER2:    .space 10     @rezerviraj 10 bajtov
BUFFER3:    .space 20     @rezerviraj 20 bajtov

;poravnanost? Če so v rezerviranih blokih bajti, ni
težav, sicer je (morda) potrebno uporabiti .align
```

- **oznaka BUFFER ustreza naslovu, od katerega naprej se rezervira 40B prostora.**
- **oznaka BUFFER2 ustreza naslovu, od katerega naprej se rezervira 10B prostora. Ta naslov ja za 40 večji kot BUFFER.**
- **oznaka BUFFER3 ustreza naslovu, od katerega naprej se rezervira 20B prostora. Ta naslov ja za 10 večji kot BUFFER2.**

```
.fill 8,1,0      @number,size in bytes,value
```

Rezervacija prostora z zač. vrednostmi

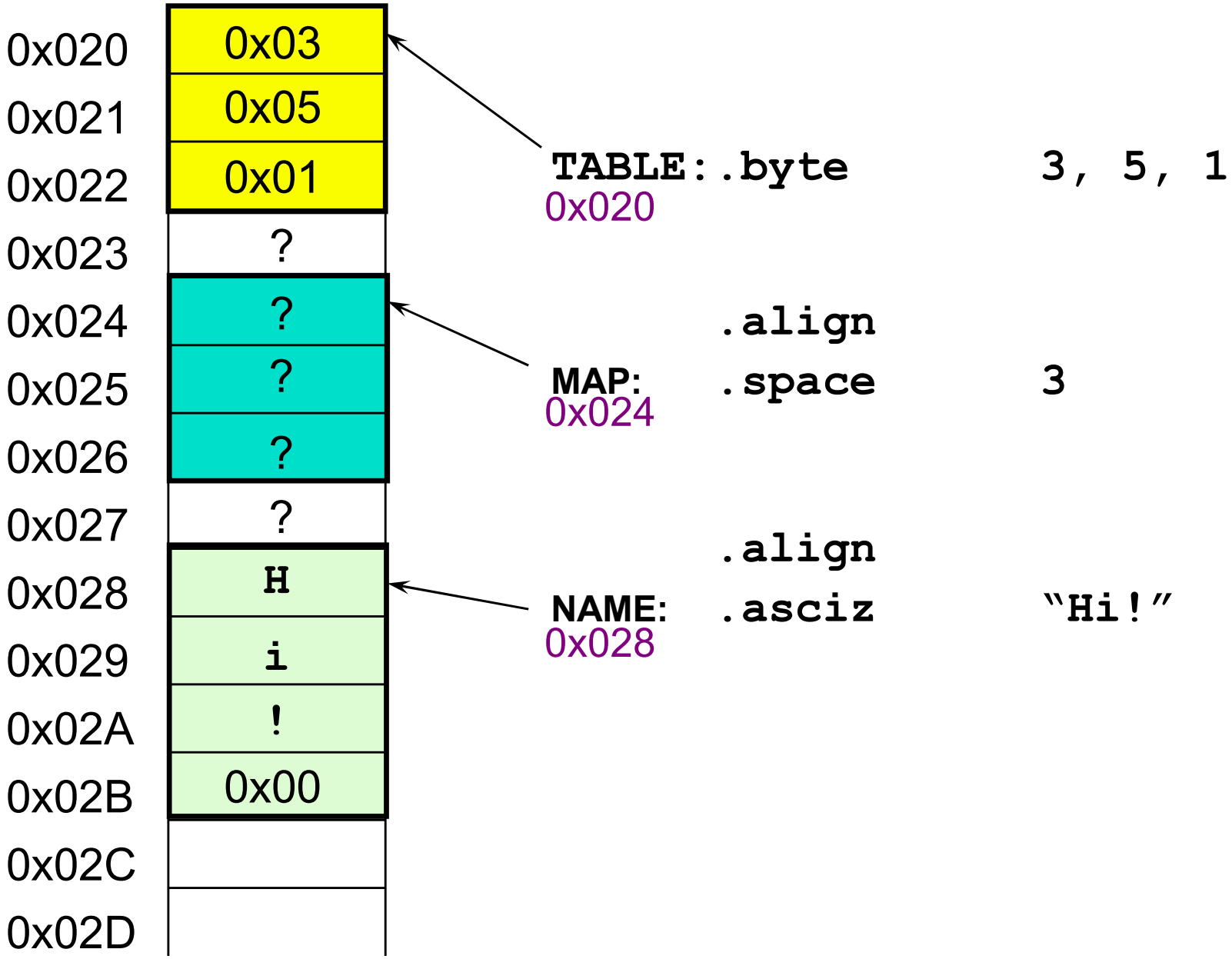
Večkrat želimo, da ima spremenljivka neko začetno vrednost.

```
niz1:   .asciz          "Dober dan"
niz2:   .ascii          "Lep dan"
        .align
stev1:  .word           512,1,65537,123456789
stev2:  .hword          1,512,65534
stev3:  .hword          0x7fe
Stev4:  .byte           1, 2, 3
        .align
naslov: .word           niz1
```

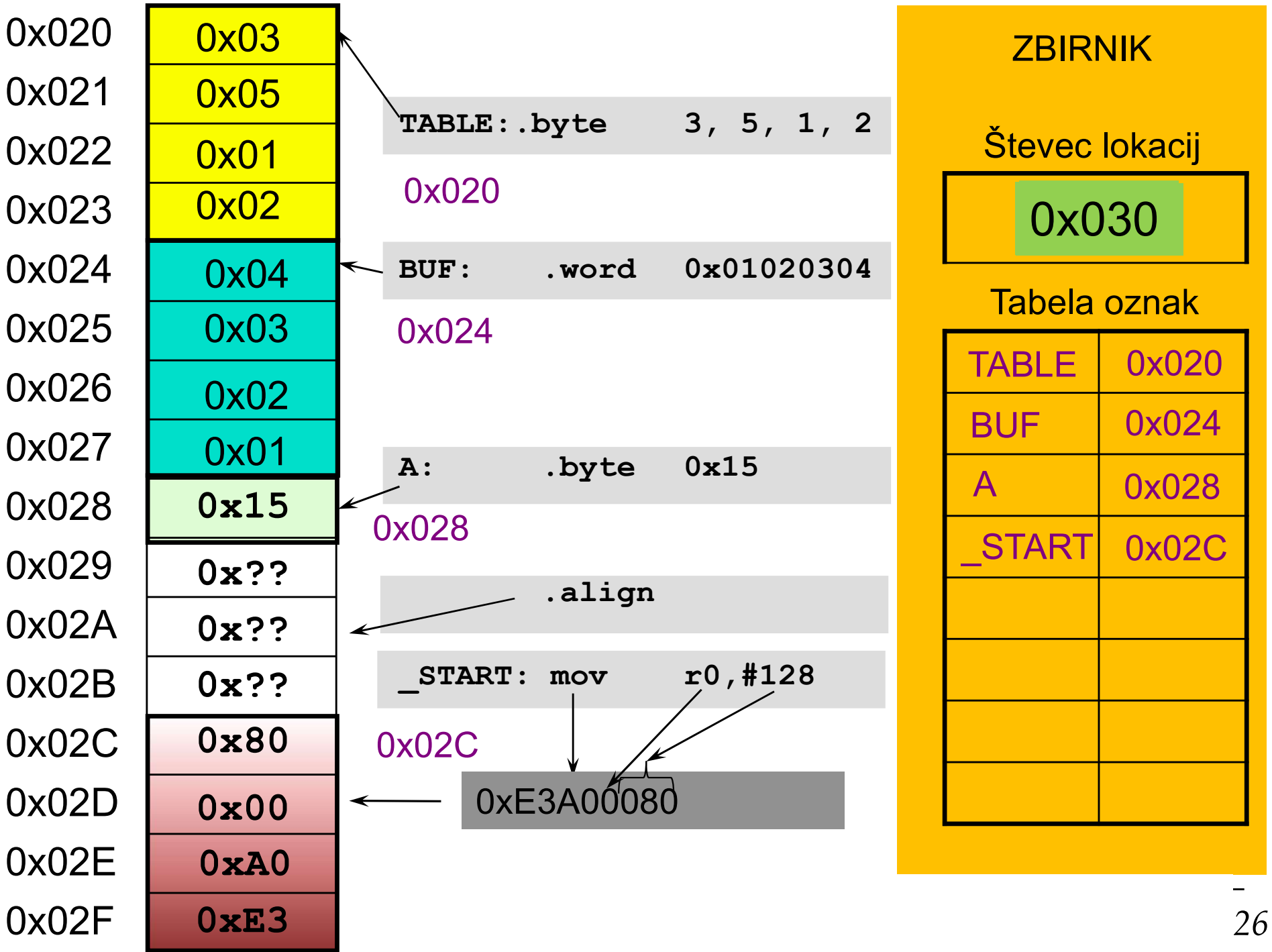
- **„spremenljivke“, inicializirane na ta način, lahko kasneje v programu spremenimo (ker so le naslovi pomnilniških lokacij)**
- **če želimo, da je oznaka vidna tudi v drugih datotekah projekta, uporabimo psevdoukaz .global, npr:**

```
.global niz1, niz2
```

Povzetek – psevdoukazi



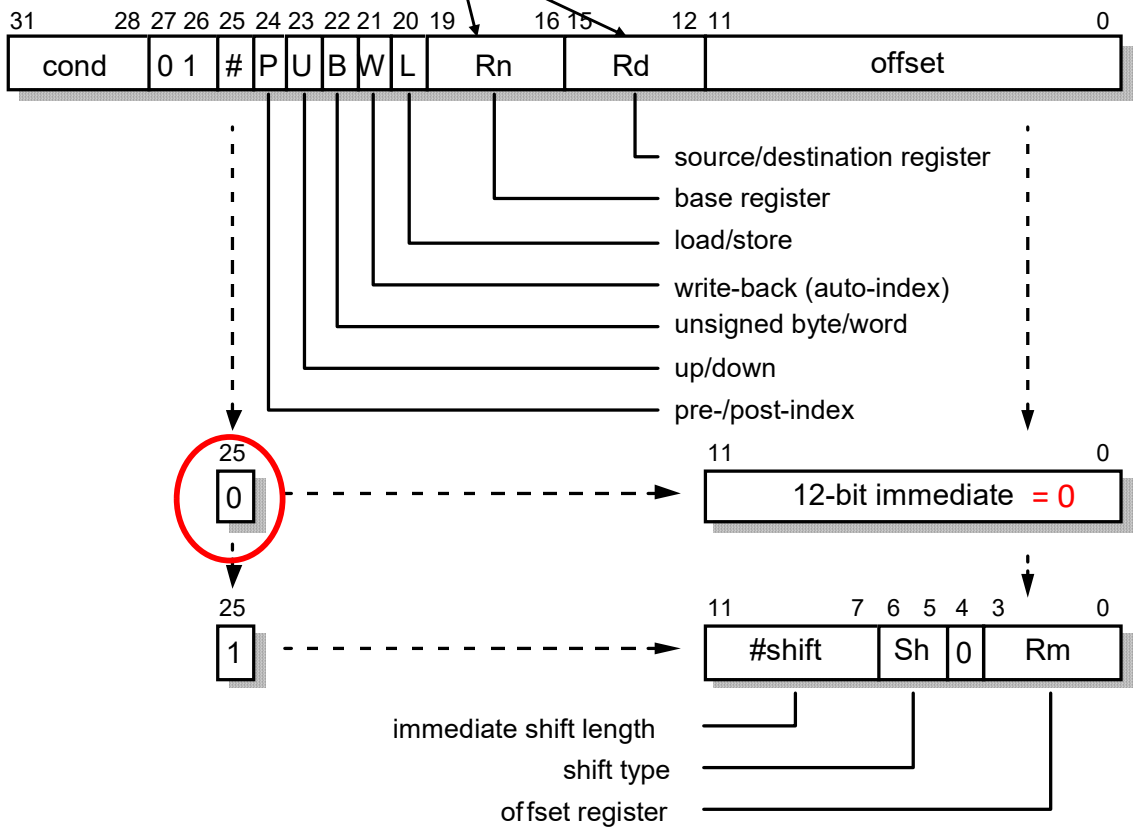
Povzetek – prevajanje (psevdoukazi,ukazi)



Load/store – načini naslavljanja

1. Posredno naslavljanje brez odmika

```
ldr r0,[r1]; r0<-mem32[r1]
```




```
ldr r0,[r1]; r0<-mem32[r1]
str r0,[r1]; mem32[r1]<-r0
strb r0,[r1]; mem8[r1]<-r0

ldr(s)b r0,[r1]; r0<-mem8[r1]*
ldr(s)h r0,[r1]; r0<-mem16[r1]*
strh r0,[r1]; mem16[r1]<-r0*

*format ukaza je drugačen
```

Naslov je določen z baznim registrom (Rn).

Load/store – posredno naslavljanje brez odmika

a) Naslov spremenljivke najprej naložimo v bazni register z:

```
adr r0, stev1
```

b) Nato uporabimo ukaz load/store oblike

```
ldr r1, [r0]   @ r1 <- mem32[r0]
str r5, [r0]   @ mem32[r0] <- r5
```

Opomba:

adr ni pravi ukaz. Prevajalnik ga nadomesti z ALE ukazom, ki izračuna naslov spremenljivke s pomočjo PC in konstante.

Primer:

```
adr r0, stev1 prevajalnik nadomesti npr. s sub r0, pc, #2c
```

Load/store – načini naslavljanja

2. Posredno naslavljanje – bazno naslavljanje s takojšnjim odmikom
(preindex with immediate offset): *32b ARM ISA*

```
ldr r0,[r1, #n12]; r0<-mem32[r1+n12]
str r0,[r1, #n12]; mem32[r1+n12]<-r0
strb r0,[r1, #n12]; mem8[r1+n12]<-r0[b0..b7]

ldr(s)b r0,[r1, #n8]; r0<-mem8[r1+n8]
ldr(s)h r0,[r1, #n8]; r0<-mem16[r1+n8]
strh r0,[r1, #n8]; mem16[r1+n8]<-r0[b0..b15]

n12 – 12-bitni predznačen odmik
n8 – 8-bitni predznačen odmik

Zgledi:
ldr r1, [r0, #4]          @ r1 <- mem32[r0 + 4]
ldr r5, [r0, #-20]        @ r5 <- mem32[r0 - 20]
                          @ v r0 mora biti ustrezen naslov!!!
strb r7, [r2,#10]         @ mem8[r2 + 10] <- r7[b0..b7]
                          @ v r2 mora biti ustrezen naslov!!!
```

Naslov je vsota baznega registra in predznačenega odmika

Load/store – posredno naslavljanje s takojšnjim odmikom

Če so **spremenljivke in program** v naslovnem prostoru **dovolj blizu**,
se kot bazni register pogosto uporablja programski števec (PC).

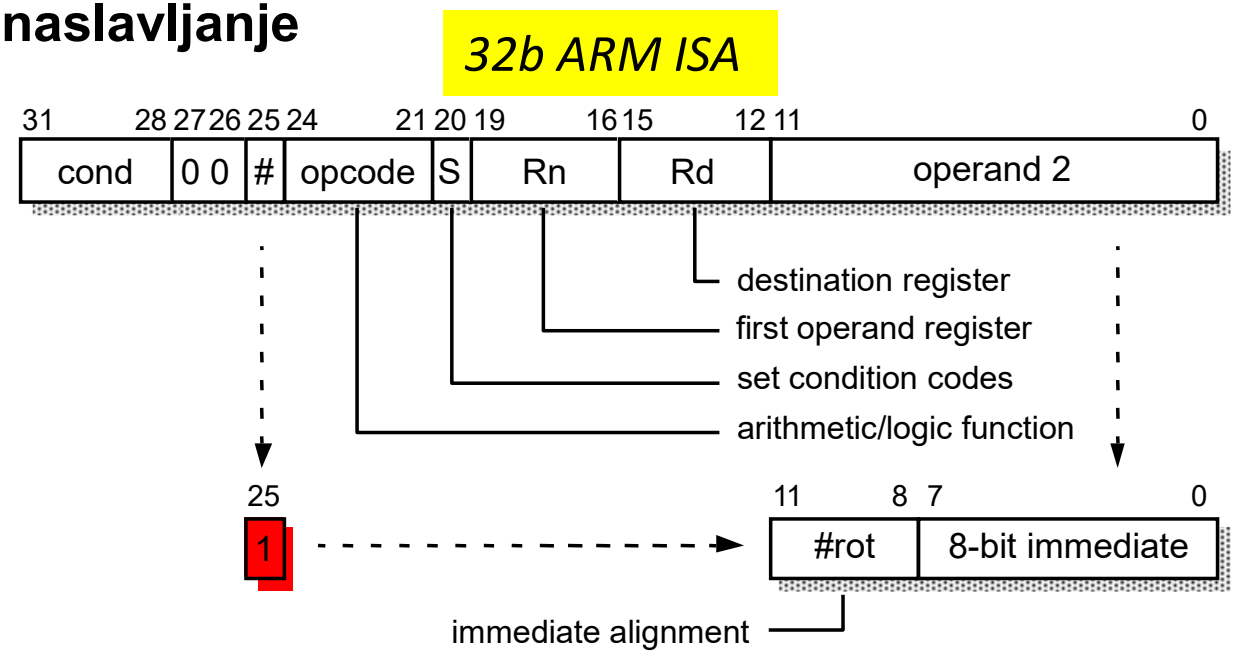
Zgled:

```
        .text
spr1:   .word 123

        .align
.global __start
__start:
        ldr r1, spr1

__end:  b __end
```

Ukaz `ldr r1, spr1` se prevede v `ldr r1, [pc, -0x000C]`.
Gre torej za bazno naslavljanje s takojšnjim odmikom. Kot bazni register se uporabi PC, odmik pa se izračuna pri prevajanju.

Aritmetično-logični ukazi (takojšnje naslavljanje)

3. Takojšnje naslavljanje



$$\text{Takojšnji operand} = (0..255) * 2^{2*(0..12)}$$

32-bitni takojšnji operand tvorimo z rotiranjem bitov 0-7 za sodo število mest znotraj 32-bitne vsebine. Takojšnji operand torej ni poljuben. Tvori ga prevajalnik, če ga ne more, nas opozori.

```
mov r1, #3
add r2, r7, #32
sub r4, r5,#1
```

Aritmetično-logični ukazi (takojšnje naslavljanje)

Takojšnji operand je del ukaza, torej mora biti v času prevajanja iz zbirnega v strojni jezik že znan. Zato takojšnjih operandov ne moremo spreminjati – so konstante. Poleg tega je polje za takojšnje operande v ukazu razmeroma kratko. Zato konstante niso poljubna 32-bitna števila.

Zgled:

```
mov r1, #3              @ r1 ← 3
add r2, r7, #0x20       @ r2 ← r7 + 32
sub r4, r5,#1           @ r4 ← r5 - 1
```

Takojšnji operand je nepredznačeno 8-bitno število, ki je lahko rotirano za 2*#rot bitov v levo.

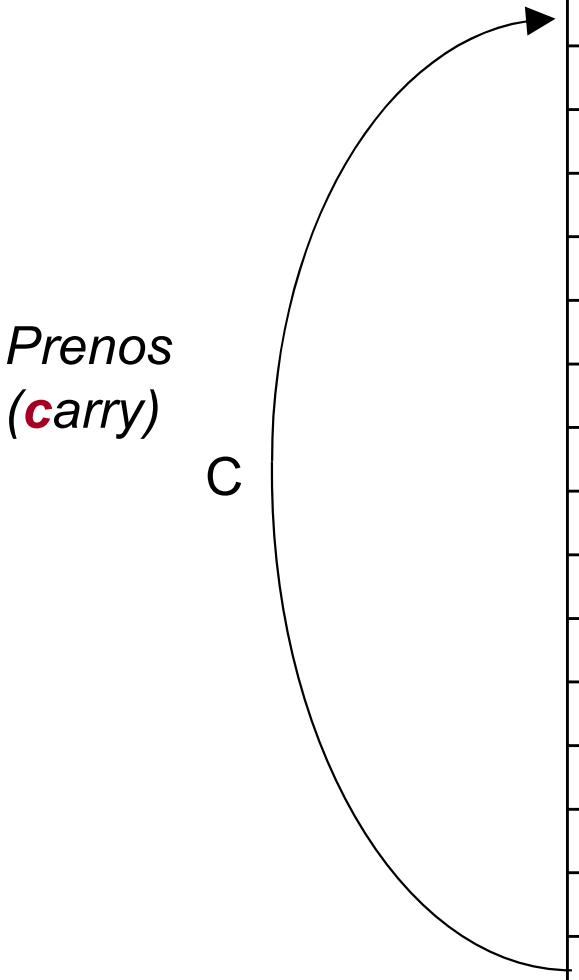
Aritmetično-logični ukazi (neposredno registrsko naslavljanje)

4. Neposredno registrsko naslavljanje

• za računanje z registri in prepisovanje vrednosti iz enega registra v drugega.

```
and r2, r7, r12
sub r4, r5, r1
mov r1, r4
```

Nepredznačena in predznačena cela števila



Dvojiški zapis	Nepredznačeno	Predznačeno
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	-8
1001	9	-7
1010	10	-6
1011	11	-5
1100	12	-4
1101	13	-3
1110	14	-2
1111	15	-1

Pri odštevanju je stanje C obratno (posebnost ARM)!

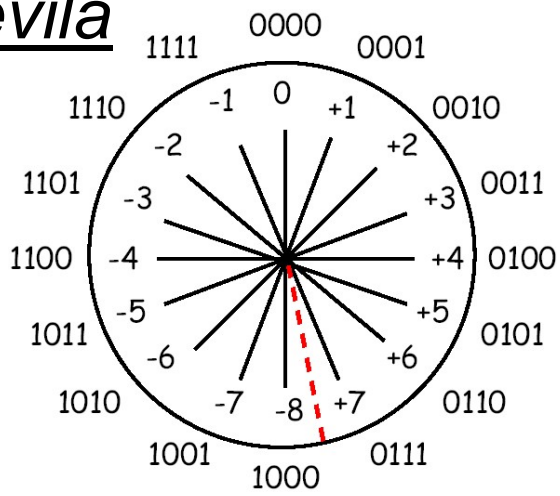
- če ne prekoračimo 0 => C=1

- če prekoračimo 0 => C=0

preliv (overflow)

$$V = A_{n-1} B_{n-1} \overline{S}_{n-1} \vee \overline{A}_{n-1} \overline{B}_{n-1} S_{n-1}$$

Nepredznačena in predznačena cela števila



preliv (overflow)

Dvojiški zapis	Nepredznačeno	Predznačeno
1000	**8**	**-8**
1001	**9**	**-7**
1010	**10**	**-6**
1011	**11**	**-5**
1100	**12**	**-4**
1101	**13**	**-3**
1110	**14**	**-2**
1111	**15**	**-1**
0000	**0**	**0**
0001	**1**	**1**
0010	**2**	**2**
0011	**3**	**3**
0100	**4**	**4**
0101	**5**	**5**
0110	**6**	**6**
0111	**7**	**7**

C ↓

V

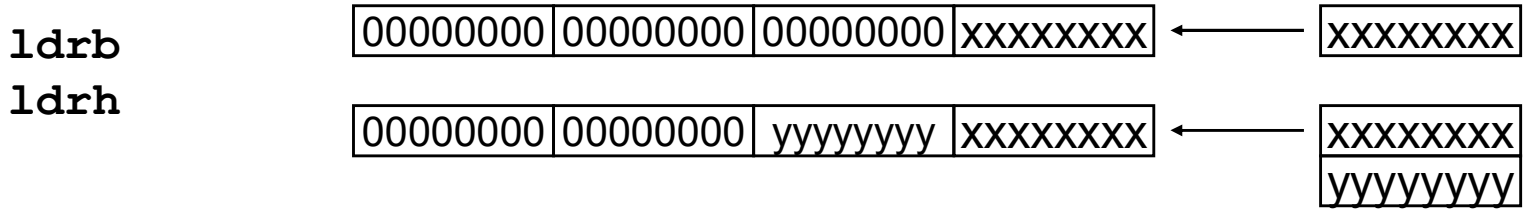
$$V = A_{n-1}B_{n-1}\overline{S}_{n-1} \vee \overline{A}_{n-1}\overline{B}_{n-1}S_{n-1}$$

Vir slike: https://www.doc.ic.ac.uk/~eedwards/compsys/arithmetic/index.html

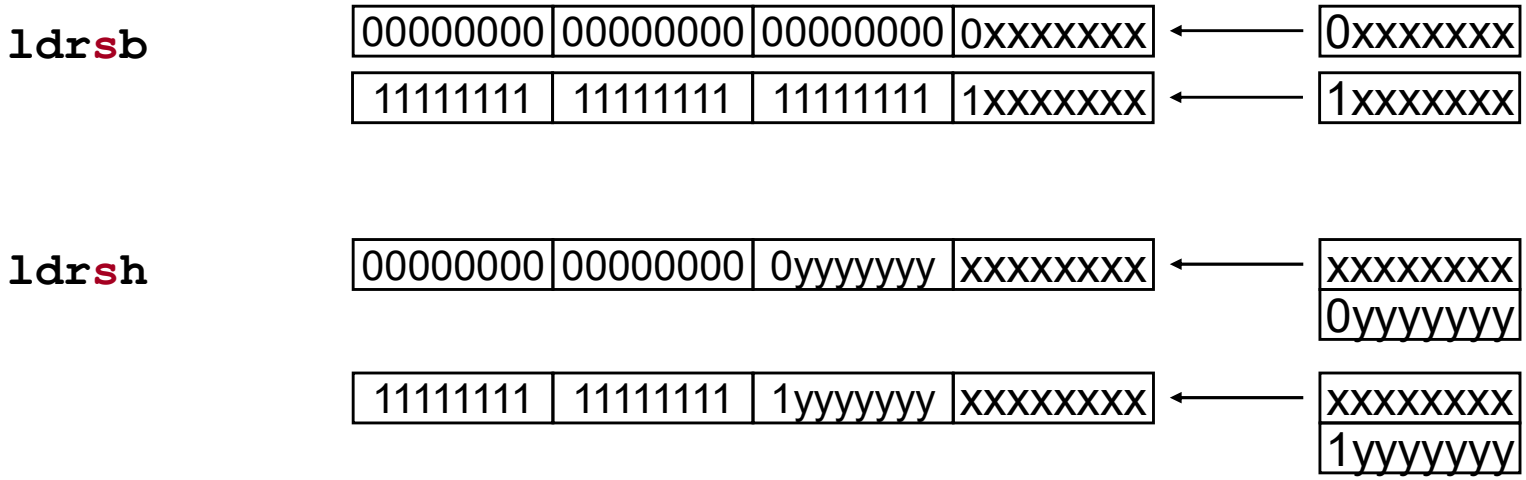
Razširitev ničle / razširitev predznaka

• pri nalaganju 8 in 16 – bitnih pomnilniških operandov je potrebno razširiti predznak ali ničlo (ker so registri in ALE operacije 32 bitni). S pomnilniškimi operandi delata samo load in store (ldr, str).

• pri nepredznačnih operandih je potrebno razširiti ničlo:



• pri predznačnih operandih je potrebno razširiti predznak:



Zastavice

- so štirje biti v registru CPSR, za vsak bit velja :
- 1 – zastavica je postavljena.
- 0 – zastavica ni postavljena

Zastavice (lahko) ukazi spreminjajo glede na rezultat ALE:

N = 0: bit 31 rezultata je 0, N=1: bit 31 rezultata je 1 (*Negative*)

Z = 1: rezultat je 0, Z=0: rezultat je različen od nič (*Zero*)

C: **+**: C = 1: rezultat je povzročil prenos, C = 0: rezultat ni povzročil prenosa (*Carry*)

-: C = **0**: rezultat je povzročil prenos, C = **1**: rezultat ni povzročil prenosa (*Carry*)

V = 1: rezultat je povzročil preliv, V = 0: rezultat ni povzročil preliva (*oVerflow*)

Pri odštevanju je stanje C obratno (posebnost ARM)!

- če ne prekoračimo 0 => C=1

- če prekoračimo 0 => C=0

Če želimo, da ALE ukaz vpliva na zastavice, mu dodamo s:

```
movs r1, #3              @ r1 ← 3
adds r2, r7, #0x20       @ r2 ← r7 + 32
subs r4, r5,#1           @ r4 ← r5 - 1
```

Primerjave

Za spreminjanje zastavic lahko uporabimo ukaze za primerjanje (spadajo med ALE ukaze):

cmp (Compare): postavi zastavice glede na rezultat Rn - Op2

cmp R1, #10 @ R1-10

cmn (Compare negated): postavi zastavice glede na rezultat Rn + Op2

cmn R1, #10 @ R1+10

Ukaza vplivata samo na zastavice, vrednosti registrov **ne spreminjata**. Ker se uporabljata zgolj za spreminjanje zastavic, jima ne dodajamo pripone S.

Primerjave nepredznačenih števil

Zgled: primerjanje dveh nepredznačenih števil:

- **opazujemo zastavici C in Z**

```
mov r1,#11
cmp r1,#10 @ C=1, Z=0

mov r1,#10
cmp r1,#10 @ C=1, Z=1

mov r1,#9
cmp r1,#10 @ C=0, Z=0
```

Torej:

			Pogoj
r1 > 10	C=1 in Z=0	Higher	HI
r1 >= 10	C=1	Higher or Same	HS
r1 = 10	Z=1	Equal	EQ
r1 < 10	C=0	Lower	LO
r1 <= 10	C=0 ali Z=1	Lower or Same	LS

Primerjave predznačenih števil

Ker gre pri primerjanju za odštevanje/seštevanje, ki je za predznačena števila enako kot za nepredznačena, tudi za primerjanje predznačenih števil uporabimo iste ukaze, opazovati pa moramo druge zastavice!

- Opazovati je potrebno zastavice V, Z in N

Zgled:

```
mov r1,#0
cmp r1,#-1 @ C=0, Z=0, V=0, N=0
```

Zastavice ne ustrezajo pogoju > za neprezdnačena števila (C=1 in Z=0)!

Pogoj **> za predznačena** števila je drugačen od pogoja **> za nepredznačena** števila. Pravilen pogoj je: N = V

Oznake pogojev

Oznaka pogoja	Pomen	Stanje zastavic, ob katerem se ukaz izvede
EQ	Equal / equals zero	Z set
NE	Not equal	Z clear
CS	Carry set	C set
CC	Carry clear	C clear
MI	Minus / negative	N set
PL	Plus / positive or zero	N clear
VS	Overflow	V set
VC	No overflow	V clear
HS	Unsigned higher or same	C set
LO	Unsigned lower	C clear
HI	Unsigned higher	C set and Z clear
LS	Unsigned lower or same	C clear or Z set
GE	Signed greater than or equal	N equals V
LT	Signed less than	N is not equal to V
GT	Signed greater than	Z clear and N equals V
LE	Signed less than or equal	Z set or N is not equal to V
Predzn. in nepredzn.	Nepredznačena	Predznačena

Skočni ukazi

Skok je ukaz tipa GOTO oznaka - pri skokih se sklicujemo na oznake. Naslov ukaza, ki stoji za oznako se zapiše v PC.

b (Branch)

```
zanka:      sub r1, r1, #1
            b zanka @ GOTO zanka
```

Zanka se bo ponavljala v nedogled. r1 se bo neprestano zmanjševal, ko bo prišel do 0, bo prišlo do prenosa, v r1 pa bo 0xffffffff.

Če želimo narediti zanko, ki se bo nehala ponavljati, ko bo r1 prišel do 0, potrebujemo ukaz tip **IF pogoj THEN GOTO oznaka**. Ukaz b se bo torej izvedel samo, če bo pogoj ustrezen.

Pogojni skoki

V zbirniku ARM je pogoj vedno določen s stanjem zastavic! Oznake pogojev smo že spoznali.

Preden uporabimo pogojni skok moramo primerno postaviti zastavice. To lahko naredimo z ukazi za primerjavo, zelo pogosto pa kar z enim izmed ostalih ALE ukazov.

Zanka, ki se ustavi, ko r1 pride do 0 bi lahko bila realizirana tako:

b (Branch)

```
zanka:      … (telo zanke)
            sub r1, r1, #1
            cmp r1, #0
            bne zanka @ IF Z=0 THEN GOTO zanka
```

Ukazu b smo dodali pripono, ki določa, ob kakšnem stanju zastavic se skok izvede. Če stanje zastavic ni ustrezno, se ukaz ne izvede!

Ker se skok izvede le ob določenem pogoju, mu pravimo **pogojni skok**.

Pogojni skoki

Ukaz cmp v prejšnjem zgledu je torej pripravil zastavico Z, ki je predstavljala pogoj za pogojni skok. Zastavico bi lahko postavili že pri zmanjševanju r1. Ukazu sub je potrebno dodati s:

b (Branch)

```
            mov r1, #10
zanka:      … (telo zanke)
            subs r1, r1, #1 @ postavi zastavice!
            bne zanka @ IF Z=0 THEN GOTO zanka
            mov r2, #10
```

Zanka se bo ponovila desetkrat. Ko bo r1 prišel do 0, bo subs zastavico Z postavil na Z=1. Pogojni skok se takrat ne bo izvršil, izvedel se bo ukaz `mov r2, #10` za pogojnim skokom. Pogojni skok torej deluje na način:

IF pogoj THEN PC ← oznaka
ELSE PC ← PC+4

Pogojni skoki

Ukazu b lahko dodamo katerokoli oznako pogoja iz tabele na prosojnici 35. Tako dobimo vse možne pogojne skoke:

Branch	Interpretation	Normal uses
B	Unconditional	Always take this branch
BEQ	Equal	Comparison equal or zero result
BNE	Not equal	Comparison not equal or non-zero result
BPL	Plus	Result positive or zero
BMI	Minus	Result minus or negative
BCC	Carry clear	Arithmetic operation did not give carry-out
BLO	Lower	Unsigned comparison gave lower
BCS	Carry set	Arithmetic operation gave carry-out
BHS	Higher or same	Unsigned comparison gave higher or same
BVC	Overflow clear	Signed integer operation; no overflow occurred
BVS	Overflow set	Signed integer operation; overflow occurred
BGT	Greater than	Signed integer comparison gave greater than
BGE	Greater or equal	Signed integer comparison gave greater or equal
BLT	Less than	Signed integer comparison gave less than
BLE	Less or equal	Signed integer comparison gave less than or equal
BHI	Higher	Unsigned comparison gave higher
BLS	Lower or same	Unsigned comparison gave lower or same

Pogojno izvajanje ukazov

Pogojni skoki so le poseben primer pogojnega izvajanja ukazov. Tudi za druge ukaze je mogoče z dodajanjem ustreznih končnic določiti, da se izvedejo le ob določenem pogoju.

```
        cmp r0, #5
        beq SKOK
        add r1,r1,r0
        sub r1,r1,r2
SKOK    ..
```

```
cmp   r0, #5
addne r1,r1,r0
subne r1,r1,r2
```

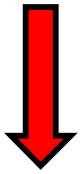
if (r1<10) then r4=r1+5
else r4=r1+8

```
        cmp r1, 10
        blo MANJ
        add r4,r1,#8
        b NAPREJ
MANJ    add r4,r1,#5
NAPREJ  ...
```

```
cmp   r1,#10
addlo r4,r1,#5
addhs r4,r1,#8
```

Pogojno izvajanje ukazov

```
if ((r0==r1) AND (r2==r3)) then r4=r4+1
```



```
cmp   r0,r1     ; postavi Z, ce je r0=r1
cmpeq r2,r3     ; primerjaj le, ce je Z=1 in
                ;   spet postavi Z, ce je r2=r3
addeq r4,r4,#1  ; sestej le, ce je Z=1 (r0=r1 in r2=r3)
```

- Večino if-then-else stavkov je mogoče implementirati s pogojnim izvajanjem!
- if-then-else stavke, ki vsebujejo AND ali OR lahko implementiramo z uporabo pogojnih primerjanj.
- Uporaba pogojnega izvajanja je pogosto bolj učinkovita kot uporaba skokov!

ARM – OR nadgradnja

Arhitektura in
programiranje v zbirniku

Spletni simulator cpulator

- [CPUlator ARMv7 System Simulator (01xz.net)]